

# 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	2024 年 10 月 9 日
实验名称	中断与异常	实验周次	第 4 周
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5 班
王浚杰	2022302181143	网络空间安全	5 班
程序	2022302181131	信息安全	4 班
周业营	2022302181145	信息安全	5 班

# Contents

<b>1 实验目的及实验内容</b>	<b>3</b>
<b>2 实验环境</b>	<b>3</b>
<b>3 实验步骤</b>	<b>4</b>
3.1 调试 8259A 的编程基本例程 . . . . .	4
3.1.1 pmtest9a.asm 代码分析 . . . . .	4
3.1.2 pmtest9a.asm 代码调试 . . . . .	6
3.2 调试时钟中断例程 . . . . .	7
3.2.1 pmtest9.asm 代码分析 . . . . .	7
3.2.2 pmtest9.asm 代码调试 . . . . .	9
3.3 实现自定义中断向量 . . . . .	10
3.3.1 代码编写 . . . . .	10
3.3.2 结果展示 . . . . .	11
<b>4 实验过程分析</b>	<b>12</b>
4.1 将 jmp \$删除 . . . . .	12
4.2 将 si 寄存器保护起来 . . . . .	13
<b>5 实验结果总结</b>	<b>14</b>
5.1 自定义中断向量运行结果展示 . . . . .	14
5.2 什么是中断，什么是异常 . . . . .	15
5.2.1 中断 . . . . .	15
5.2.2 中断的处理机制 . . . . .	16
5.2.3 异常 . . . . .	17
5.3 8259A 的工作原理是怎样的，怎么给这些中断号的处理向量初始化 . . . . .	17
5.3.1 外部中断 . . . . .	18
5.3.2 工作原理 . . . . .	18
5.3.3 中断号的向量初始化 . . . . .	19
5.4 如何建立 IDT，如何实现一个自定义的中断 . . . . .	20
5.4.1 建立 IDT . . . . .	20
5.4.2 实现自定义中断 . . . . .	20
5.5 如何控制时钟中断，为什么时间中断的时候没有看到 int 指令 . . . . .	20
5.6 简单解释一下 IOPL 的作用与基本机理 . . . . .	21
5.6.1 IOPL 的作用 . . . . .	21
5.6.2 IOPL 的基本机理 . . . . .	21
5.7 实验改进意见 . . . . .	21
<b>6 各人实验贡献与体会（每人各自撰写）</b>	<b>22</b>
<b>7 教师评语</b>	<b>23</b>

# 1 实验目的及实验内容

(本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析)

## 实验目标：

1. 理解中断与异常机制的实现机理
2. 对应章节：第三章 3.4 节《中断和异常》
3. 了解 3.5 节《保护模式下的 I/O》

## 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

## 需要回答的问题：

1. 什么是中断，什么是异常
2. 8259A 的工作原理是怎样的？怎么给这些中断号的处理向量初始化值？
3. 如何建立 IDT，如何实现一个自定义的中断
4. 如何控制时钟中断，为什么时钟中断时候，没有看到 int 的指令？
5. 简要解释 IOPL 的作用与基本机理

# 2 实验环境

(本次实验所使用的器件、仪器设备等的情况；具体实验步骤)

WSL 2  
Ubuntu 20.04  
nasm 2.14.02  
VSCode  
Bochs 2.7

### 3 实验步骤

(本次实验的具体实验步骤)

#### 3.1 调试 8259A 的编程基本例程

##### 3.1.1 pmtest9a.asm 代码分析

pmtest9a.asm 在之前代码的基础上新增了建立 IDT 和初始化 8259A 两部分代码。保护模式下的中断向量表由中断描述符表 IDT 来实现。IDT 的作用是将每一个中断向量和一个描述符对应起来。IDT 中的描述符可以是中断门描述符、陷阱门描述符、任务门描述符三种门描述符之一。

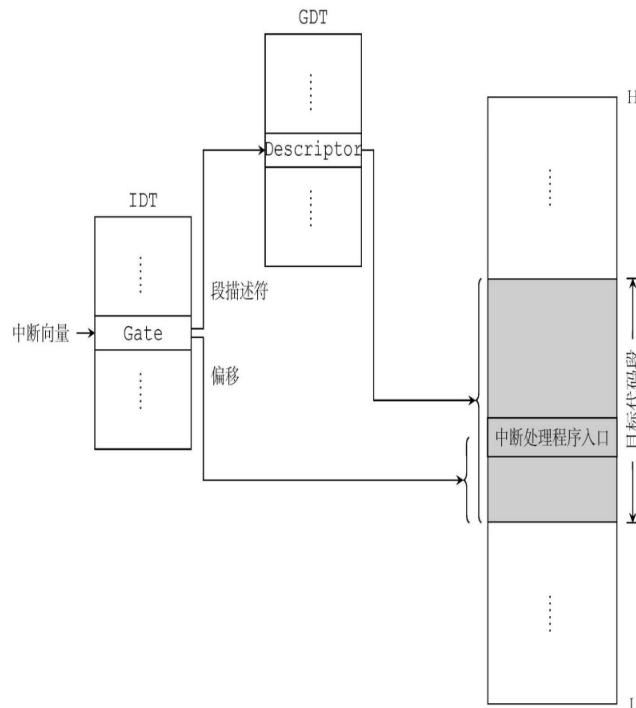


Figure 1: IDT 寻址方式

建立一个 IDT 的过程与之前的实验中建立 GDT、LDT 的过程基本一致。

```
1 ; IDT
2 [SECTION .idt]
3 ALIGN 32
4 [BITS 32]
5 LABEL_IDT:
6 ; 门          目标选择子,           偏移, DCount, 属性
7 %rep 32
8     Gate      SelectorCode32, SpuriousHandler,      0, DA_386IGate
9 %endrep
10 .020h:    Gate      SelectorCode32,     ClockHandler,      0, DA_386IGate
11 %rep 95
12     Gate      SelectorCode32, SpuriousHandler,      0, DA_386IGate
13 %endrep
14 .080h:    Gate      SelectorCode32,   UserIntHandler,      0, DA_386IGate
```

```

15
16     IdtLen      equ $ - LABEL_IDT
17     IdtPtr       dw  IdtLen - 1 ; 段界限
18     dd  0          ; 基地址
19 ; END of [SECTION .idt]
20
21 ...
22
23 ; 为加载 IDTR 作准备
24 xor eax, eax
25 mov ax, ds
26 shl eax, 4
27 add eax, LABEL_IDT      ; eax <- idt 基地址
28 mov dword [IdtPtr + 2], eax ; [IdtPtr + 2] <- idt 基地址
29
30 ; 保存 IDTR
31 sidt [_SavedIDTR]
32
33 ...
34
35 ; 加载 IDTR
36 lidt [IdtPtr]

```

这里利用了 NASM 的%rep 预处理指令将每一个描述符都设置为指向 SelectorCode32:SpuriousHandler 的中断门。SpuriousHandler 的功能是在屏幕的右上角打印红色的字符 “!” 然后进入死循环。

```

1 _SpuriousHandler:
2 SpuriousHandler equ _SpuriousHandler - $$

3     mov ah, 0Ch           ; 0000: 黑底    1100: 红字
4     mov al, '!'
5     mov [gs:((80 * 0 + 75) * 2)], ax      ; 屏幕第 0 行，第 75 列。
6     jmp $
7     iretd

```

初始化保护模式下的 8259A:

```

1 ; Init8259A
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

---

```

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.

```

```

16      call    io_delay
17
18      mov al, 004h      ; IR2 对应从 8259
19      out 021h, al      ; 主 8259, ICW3.
20      call    io_delay
21
22      mov al, 002h      ; 对应主 8259 的 IR2
23      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
-----
```

### 3.1.2 pmtest9a.asm 代码调试

代码执行效果如图所示：

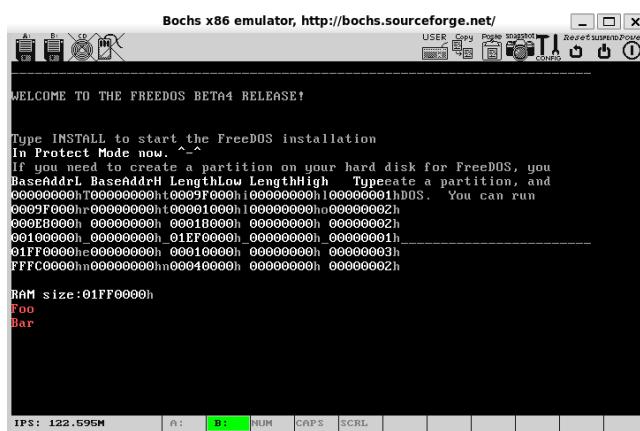


Figure 2: pmtest9a.asm 原始执行结果

在代码的执行过程中，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，我们推测是由于在保护模式下中断处理程序并没有来

得及执行，所以处于实模式的系统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“!”。在调试的过程中，先将断点设置在了 SpuriousHandler 函数中，但在运行的过程中并未触发断点，推测是中断处理程序并未得到调用和执行。

在对 pmtest9a 做出以下修改之后，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 “!”。但是，pmtest9a.asm 中并没有调用 Init8259A，所以该程序中 8259A 的状态并不是我们人为设置的，而是默认的全嵌套方式。我们不能确定最终打印出 “!” 是否是通过时钟中断产生的。

```

1      call      PagingDemo      ; 演示改变页目录的效果
2
3      sti
4      jmp  $
5
6      ; 到此停止
7      jmp  SelectorCode16:0

```

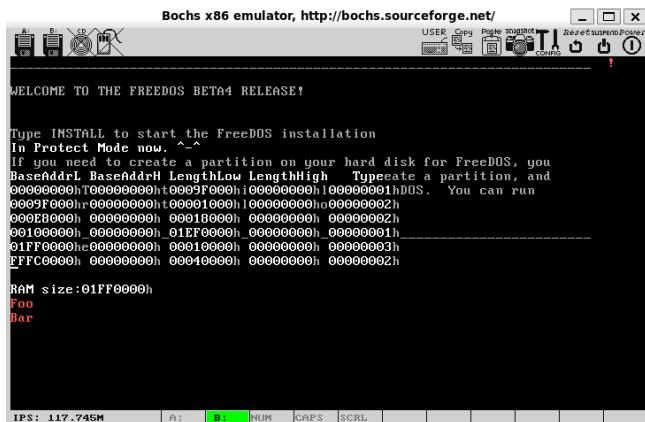


Figure 3: pmtest9a 修改之后的执行结果

## 3.2 调试时钟中断例程

### 3.2.1 pmtest9.asm 代码分析

相较于 pmtest9a.asm，pmtest9.asm 修改或新增了以下内容：

```

1 LABEL_IDT:
2 ; 门                      目标选择子,          偏移, DCount, 属性
3 %rep 32
4     Gate    SelectorCode32, SpuriousHandler,      0, DA_386IGate
5 %endrep
6 .020h:    Gate    SelectorCode32,     ClockHandler,      0, DA_386IGate
7 %rep 95
8     Gate    SelectorCode32, SpuriousHandler,      0, DA_386IGate
9 %endrep
10    Gate    SelectorCode32,   UserIntHandler,      0, DA_386IGate

```

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 ASCII 值自增，来起到改变字符显示时钟变化的功能。

```

1 _ClockHandler:
2 ClockHandler    equ _ClockHandler - $$ 
3     inc byte [gs:((80 * 0 + 70) * 2)]      ; 屏幕第 0 行，第 70 列。
4     mov al, 20h
5     out 20h, al                      ; 发送 EOI
6     iretd
7
8 _UserIntHandler:
9 UserIntHandler   equ _UserIntHandler - $$ 
10    mov ah, 0Ch                  ; 0000: 黑底      1100: 红字
11    mov al, 'I'
12    mov [gs:((80 * 0 + 70) * 2)], ax      ; 屏幕第 0 行，第 70 列。
13    iretd

```

保存 IDTR 和中断屏蔽寄存器 (IMREG) 值。

```

1 ; 保存 IDTR
2 sidt [_SavedIDTR]
3
4 ; 保存中断屏蔽寄存器 (IMREG) 值
5 in al, 21h
6 mov [_SavedIMREG], al

```

设置好实模式下的中断处理机制。

```

1 ; SetRealmode8259A
-----+
3 SetRealmode8259A:
4     mov ax, SelectorData
5     mov fs, ax
6
7     mov al, 017h
8     out 020h, al      ; 主 8259, ICW1.
9     call io_delay
10
11    mov al, 008h      ; IRQ0 对应中断向量 0x8
12    out 021h, al      ; 主 8259, ICW2.
13    call io_delay
14
15    mov al, 001h
16    out 021h, al      ; 主 8259, ICW4.
17    call io_delay
18
19    mov al, [fs:SavedIMREG] ; 恢复中断屏蔽寄存器 (IMREG) 的原值
20    out 021h, al
21    call io_delay
22
23    ret
; SetRealmode8259A
-----+

```

加入 sti 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。

```
1      call    Init8259A
2
3      int 080h
4      sti
5      jmp $
```

### 3.2.2 pmtest9.asm 代码调试

直接运行代码，得到以下运行结果：

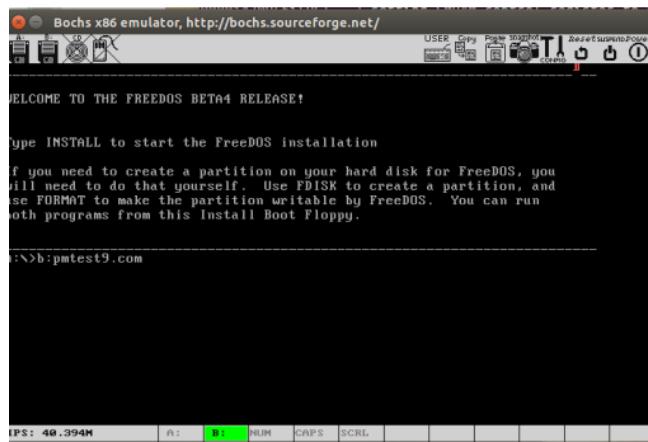


Figure 4: pmtest9.asm 运行结果

可以看到，第 0 行第 70 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：

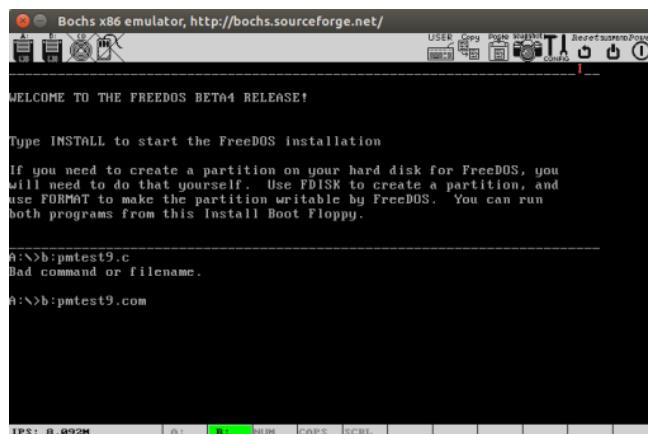


Figure 5: 停在 UserIntHandler 的断点处

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：

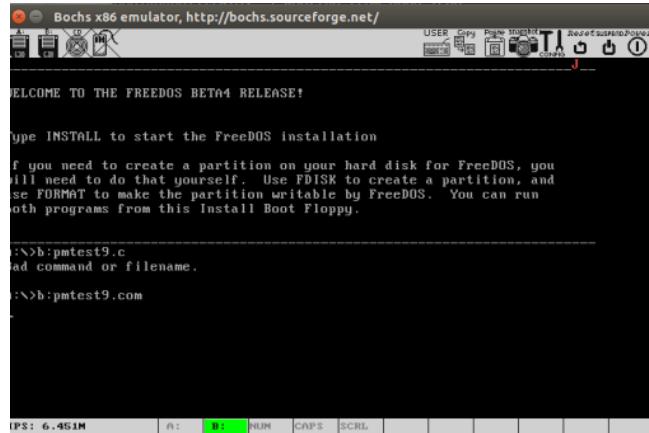


Figure 6: 停在 ClockHandler 的断点处

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。

### 3.3 实现自定义中断向量

#### 3.3.1 代码编写

我们将实现一个键盘中断向量，功能为每次按下键盘后，打印的字符向左移动一位

- 编写中断处理程序

```

1 _KeyboardHandler:
2 KeyboardHandler equ _KeyboardHandler - $$ 
3     in al, 60h ; 获取键盘扫描码
4
5 ; inc byte [gs:((80 * 0 + 75) * 2)] ; 屏幕第 0 行，第 75 列。
6
7     lea di, [80 * 0 + si] ; 计算新位置
8     shl di, 1
9     mov byte [gs:di], ' '
10
11    dec si
12    mov ah, 0Ch
13    mov al, '!'
14    lea di, [80 * 0 + si] ; 计算新位置
15    shl di, 1
16    mov [gs:di], ax
17    mov al, 20h
18    out 20h, al
19    iretd

```

- 添加 IDT 表项

```

1 .020h:      Gate      SelectorCode32,      ClockHandler,      0,
2           DA_386IGate
3 .021h:      Gate      SelectorCode32,      KeyboardHandler,   0,
4           DA_386IGate

```

我们给键盘中断使用 21h 这个中断向量

- 初始化 8259A，打开对应中断因为使用的是 21h 向量，所以 OCW1 的值应该为 11111101b

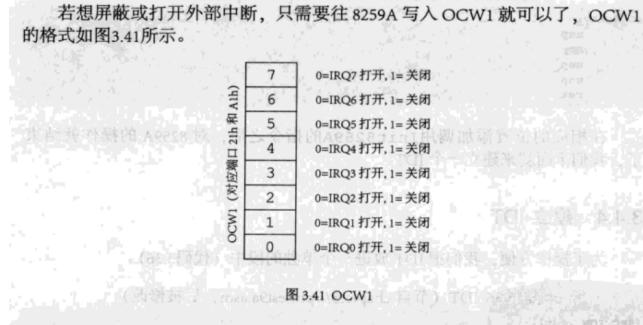


Figure 7: OCW1

对应的代码修改为位置为：

```
;mov    al, 1111111b ; 屏蔽主8259所有中断
mov al, 11111101b ; 开启keyboard中断
out 021h, al ; 主8259, OCW1.
call    io_delay

mov al, 1111111b ; 屏蔽从8259所有中断
out 0A1h, al ; 从8259, OCW1.
call    io_delay
```

Figure 8: 8259A 修改部分

### 3.3.2 结果展示

最开始的时候!在屏幕的右上方 多次按键盘后!'产生了明显的向左偏移

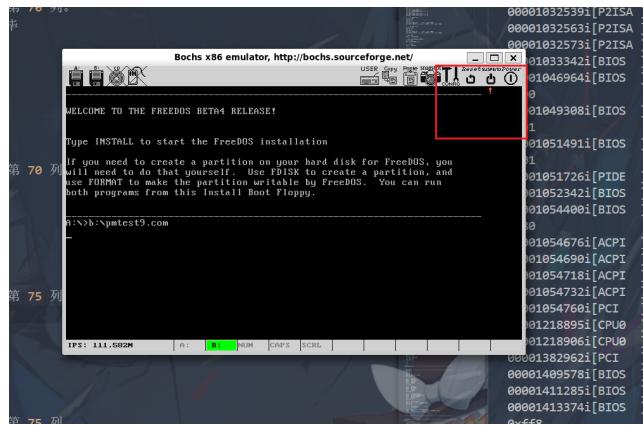


Figure 9: 按键盘前

自定义中断向量过程中，产生了许多问题，进行了一些代码调试，详见实验过程分析部分

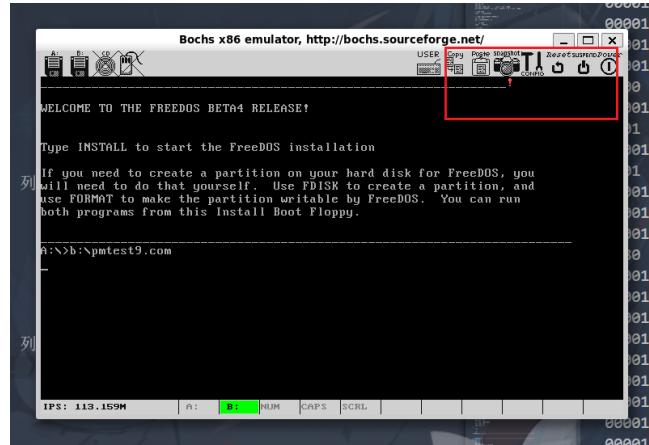


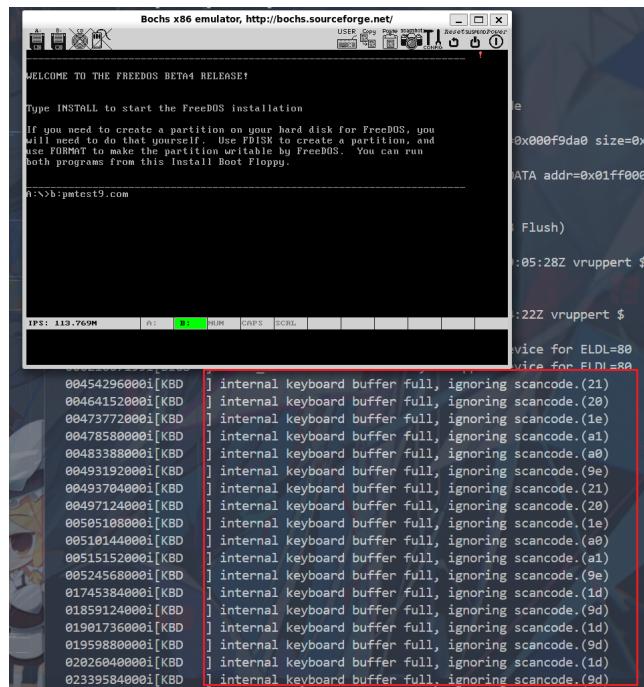
Figure 10: 多次按键盘后

## 4 实验过程分析

本部分主要为键盘中断向量调试过程中所遇到的问题及其解决

### 4.1 将 jmp \$删除

实验过程中，我们发现，无论怎么按下键盘，程序都毫无反应，甚至产生了键盘缓冲区已满，无法再接收更多的键盘输入的情况，如下图：经过调试，我们发现程序停在了一个 jmp \$位置，



即 jmp \$，通过定位，我们发现是 SpuriousHandler 中的该代码导致不能正常响应中断，于是将该行代码删除

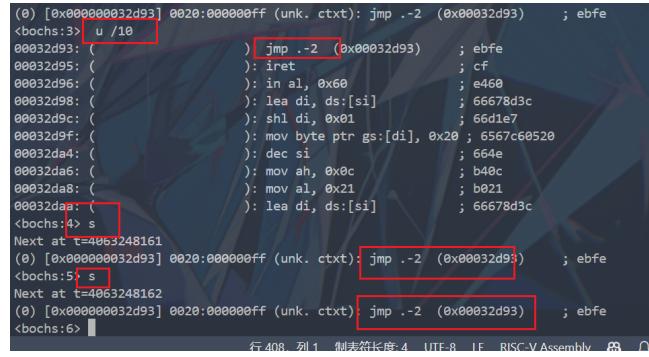


Figure 11: 程序执行陷入死循环

```

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

```

## 4.2 将 si 寄存器保护起来

我们需要寄存器来控制每次的移动，开始编译时 nasm 在编译汇编代码时出现了错误，提示“invalid effective address”（无效的有效地址）。这个问题通常出现在对内存地址的引用格式不正确，比如：

```
1     mov di, [gs:((80 * 0 + si) * 2)] ; 计算新位置
```

这部分代码格式不正确，因为 nasm 不允许直接在内存操作数中进行复杂的算术表达式。因此，需要改进对有效地址的引用。

我们已经修复了 jmp \$的问题，但按下键盘后还是和上图一样，字符没有正确的移动，经过研究发现，原来的 keyboard 中断代码为

```

1 _KeyboardHandler:
2 KeyboardHandler equ _KeyboardHandler - $$ 
3     in al, 60h
4
5     ; inc byte [gs:((80 * 0 + 75) * 2)] ; 屏幕第 0 行, 第 75 列。
6     mov si 75
7     lea di, [80 * 0 + si] ; 计算新位置
8     shl di, 1
9     mov byte [gs:di], ' '
10
11    dec si
12    mov ah,0Ch
13    mov al, '!'
14    lea di, [80 * 0 + si] ; 计算新位置
15    shl di, 1
16    mov [gs:di], ax
17    mov al,20h
18    out 20h,al
19    iretd

```

每次 iretd 之后，重新 call 到 init8259A，si 寄存器的值都从 75 开始，因此按下键盘后，显示的位置并不会改变。因此我们将 si 寄存器的赋值改到了 call init 8259A 之前  
最终我们成功解决了代码问题，实现了自定义键盘中断向量功能。

```
LABEL_SEG_CODE32:  
    mov ax, SelectorData  
    mov ds, ax          ; 数据段选择子  
    mov es, ax  
    mov ax, SelectorVideo  
    mov gs, ax          ; 视频段选择子  
  
    mov ax, SelectorStack  
    mov ss, ax          ; 堆栈段选择子  
    mov esp, TopOfStack  
  
    mov si, 75  
    call    Init8259A  
  
    int 080h  
    sti  
    jmp $  
  
    ; [下面显示一个字符串]  
    push   szPMMessge  
    call    DispStr  
    add esp, 4  
  
    push   szMemChkTitle  
    call    DispStr
```

## 5 实验结果总结

(对实验结果进行分析，完成思考题目，并提出实验的改进意见)

### 5.1 自定义中断向量运行结果展示

我们实现的自定义中断为键盘中断向量，功能为每次按下键盘后，打印的字符向左移动一位。  
如下图，最开始的时候'!'在屏幕的右上方。

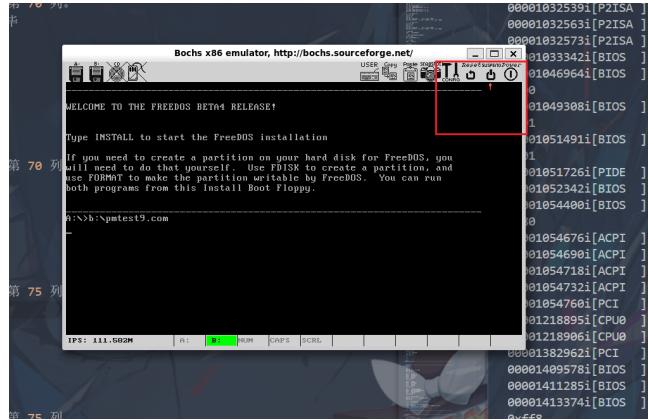


Figure 12: 按键盘前

多次按键盘后'!' 产生了明显的向左偏移

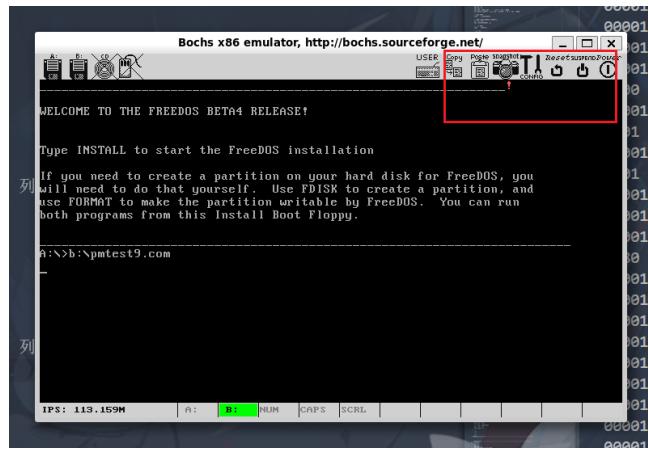


Figure 13: 多次按键盘后

## 5.2 什么是中断，什么是异常

### 5.2.1 中断

中断，又称为外部中断，是计算机系统中用于处理异步事件的一种重要机制，它允许外部设备或软件在不影响 CPU 主要工作的情况下，及时通知处理器去处理某个重要事件。中断机制使得处理器能够更高效地管理系统资源，而无需不停地轮询检查设备状态。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。

中断产生的方式可以分为两种，分别是硬件设备产生的中断和软件产生的中断。

- **硬件中断：**硬件设备（如键盘、网卡、定时器等）发出的中断请求，通知处理器设备需要处理。每个硬件中断通常有唯一的中断请求号（IRQ）。
- **软件中断：**由程序或操作系统显式发出的中断信号，通常通过执行 int 指令来触发。软件中断常用于系统调用，允许用户程序请求操作系统执行某些特定的任务。

中断根据是否可以被屏蔽又分为可屏蔽中断和不可屏蔽中断。

- **屏蔽中断**: 通过设置中断控制器或处理器中的特定标志位，可以屏蔽某些中断，使得它们在特定时间段内不被处理。例如，当 CPU 正在处理某个重要任务时，可以暂时屏蔽低优先级的中断。
- **不可屏蔽中断 (NMI)**: 非屏蔽中断是一种不能被屏蔽或忽略的中断，通常用于处理紧急或关键的硬件故障（如电源故障）。非屏蔽中断的优先级极高，处理器必须立即响应。

### 5.2.2 中断的处理机制

最近的一次对中断的使用是在实验 3 中，通过使用 int 15h 得到了计算机内存信息。但是在做实验的过程中我们也发现了所有中断的操作都是在实模式下进行的，而在转入保护模式之后就没有以 int 的形式使用中断了。查阅资料之后我们发现，在转入保护模式之后中断机制发生了改变，不再与实模式下的中断处理机制相同。所以在转入保护模式之前我们需要关中断，在最后跳回实模式之后才重新开中断。下面就分别解释在两种不同模式下的中断处理机制。

#### 实模式下的中断处理机制

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中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：

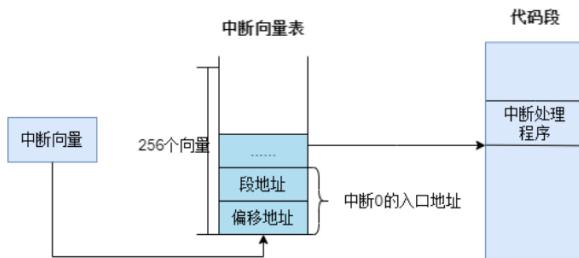


Figure 14: 实模式下的中断处理

#### 保护模式下的中断处理机制

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。中断门和陷阱门的结构如下图所示：

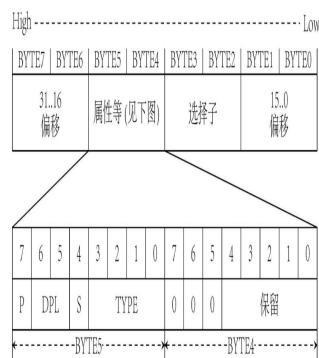


Figure 15: 中断门和陷阱门结构

其中灰色部分保留，不使用。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。具体的处理机制如下图：虽然 IDT 中可以有中断门、陷阱门或者任务门。但任务门在有些操作系统中根本就没有用到，教材中也并未做过多的解释，我们也不再关注。而中断

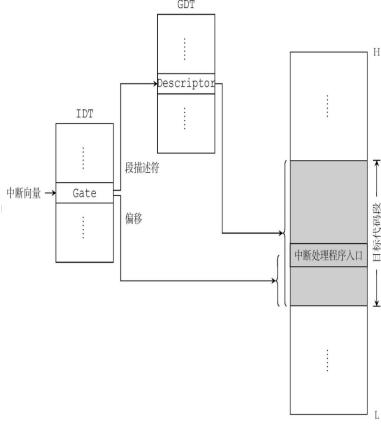


Figure 16: 保护模式下的中断处理

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。

### 5.2.3 异常

异常是在计算机系统中，当处理器执行指令时，遇到某些错误或特殊情况（例如非法操作、硬件故障、算术错误等），处理器自动触发的一种特殊中断。异常通常是由 CPU 内部产生的，而不是由外部设备引起的。异常通常是同步事件，即异常发生的位置和时间与程序的执行是密切相关的。

异常根据其触发的原因和处理方式，可以分为以下几种类型：

- **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。

## 5.3 8259A 的工作原理是怎样的，怎么给这些中断号的处理向量初始化值

8259A 的内部结构如下图所示：

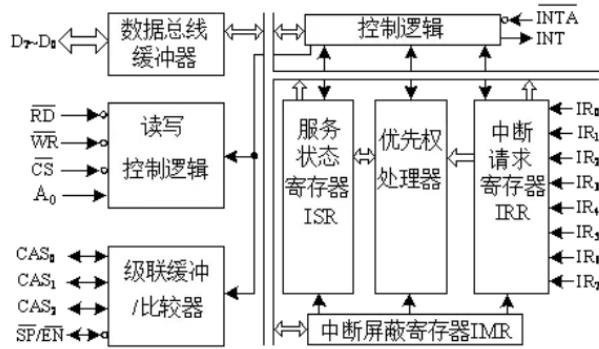


Figure 17: 8259A 内部结构

### 5.3.1 外部中断

中断产生的原因有两种，一种是外部中断，也就是由硬件产生的中断，另一种是由指令产生的中断。

对于外部中断，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，因为它与 IF 是否被设置无关。NMI 中断对应的中断向量号为 2。

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。可以认为它是中断机制中所有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。

与 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。

在 BIOS 初始化的时候，IRQ0 ~ IRQ7 被设置为对应向量号 08h ~ 0Fh，而通过中断向量表我们知道，在保护模式下向量号 08h-0Fh 已经被占用了，所以需要重新设置主从 8259A。通过向相应的端口写入特定的 ICW (InitializationCommandWord) 来实现。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。

ICW 共有 4 个，每一个都是具有特定格式的字节。初始化过程：

- 往端口 20h (主片) 或 A0h (从片) 写入 ICW1。
- 往端口 21h (主片) 或 A1h (从片) 写入 ICW2。
- 往端口 21h (主片) 或 A1h (从片) 写入 ICW3。
- 往端口 21h (主片) 或 A1h (从片) 写入 ICW4。这 4 步的顺序是不能颠倒的。

4 个 ICW 的格式如下图所示，可以看到，在写入 ICW2 时涉及与中断向量号的对应

### 5.3.2 工作原理

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 (OCW1)，决定是否将其丢弃还是接受。

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 CPU 正忙着执行某条指令时，还有可能

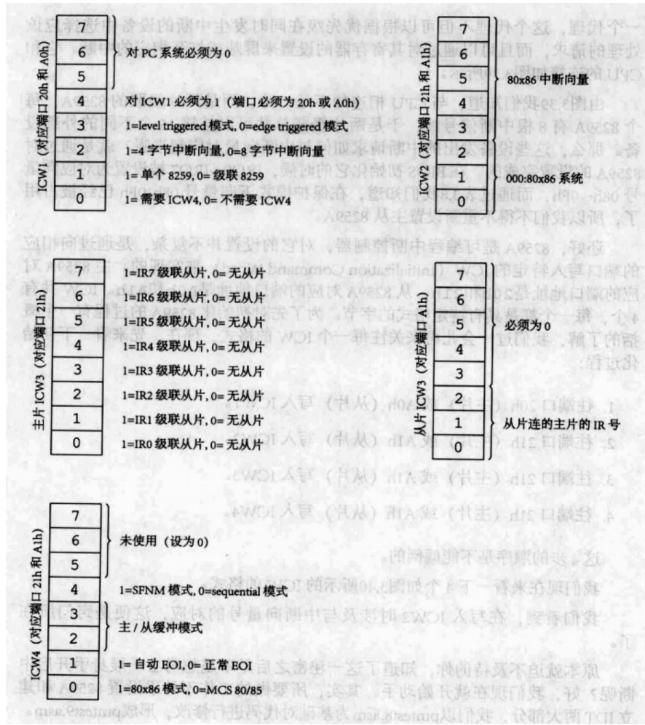


Figure 18: 8259A 内部结构

有其它的 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 的相应位置会置 1。

当 CPU 执行完一条指令时，会检查一下 INTR 管脚是否有信号，如果有信号，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 中相应位置复零，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。这样，CPU 还会发送第二个 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 的数据线段。

### 5.3.3 中断号的向量初始化

要给这些中断号的处理向量初始化值，需要对 8259A 进行初始化编程，即向它写入四个初始化命令字  $ICW_1 - ICW_4$ 。这些命令字可以指定 8259A 的基本工作参数，如中断向量码的起始地址，工作模式，级联方式等。具体的编程步骤如下

- 向 8259A 的命令寄存器写入  $ICW_1$ ，指定 8259A 是否级联，是否有  $ICW_4$  等
- 向 8259A 的数据寄存器写入  $ICW_2$ ，指定中断向量码的高 5 位（低 3 位由 8259A 自动提供）
- 如果有级联，则向 8259A 的数据寄存器写入  $ICW_3$ ，指定主片和从片之间的连接方式。
- 如果有  $ICW_4$ ，则向 8259A 的数据寄存器写入  $ICW_4$ ，指定特殊全嵌套模式，缓冲模式，自动结束模式等。

## 5.4 如何建立 IDT，如何实现一个自定义的中断

### 5.4.1 建立 IDT

- 分配内存：首先，你需要在内存中分配一块空间来存储 IDT。这可以通过使用操作系统提供的内存分配函数或手动分配内存来完成。
- 定义 IDT 条目：IDT 是由一系列的描述符（Descriptor）组成的表格。每个描述符对应一个中断或异常类型，并包含了相应的处理程序的地址和其他相关信息。
- 设置 IDT 条目：将中断处理程序的地址和其他相关信息设置到 IDT 的相应条目中。
- 加载 IDT：将 IDT 的地址加载到 IDTR 寄存器中，以告诉处理器 IDT 的位置和大小。

### 5.4.2 实现自定义中断

- 定义中断处理程序：首先需要定义要执行的中断处理程序。这可以是你自己编写的汇编或 C 代码，用于处理特定中断类型的操作。
- 初始化 IDT：IDT 是一个存储中断处理程序地址的表格。你需要在内存中分配一块空间来存储 IDT，并将其初始化为全零。
- 设置中断处理程序地址：将你定义的中断处理程序的地址设置到 IDT 的相应条目中。每个中断类型都有一个唯一的中断向量值，用于索引 IDT 的相应条目。
- 加载 IDT：通过将 IDT 的地址加载到 IDTR 寄存器中，将 IDT 告诉处理器。
- 启用中断：根据你的操作系统或应用程序的需求，启用或禁用特定的中断。

## 5.5 如何控制时钟中断，为什么时间中断的时候没有看到 int 指令

可以通过修改 8259A 的状态来控制时钟中断 从主从 8259A 结构中可以看到，时钟中断处在

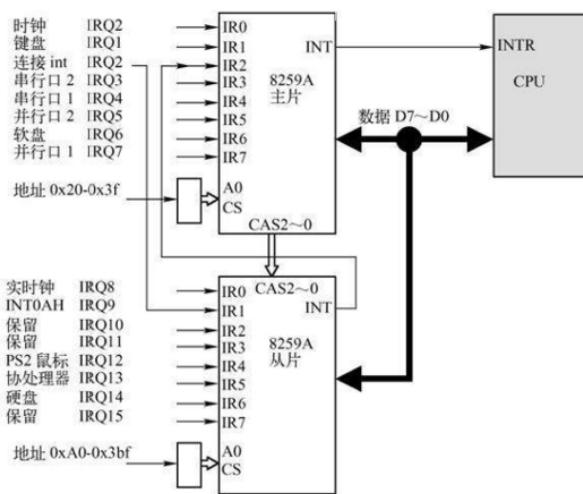


Figure 19: 8259A 内部结构

8259A 主片的 IR0 端口。通过 OCW1 可以设置主片的 IR0 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。

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。因此时钟中断时，没有看到 int 指令

## 5.6 简单解释一下 IOPL 的作用与基本机理

### 5.6.1 IOPL 的作用

IOPL 是 I/O 保护机制的关键之一，位于寄存器 eflags 的 12、13 位。指令 in、ins、out、outs、



Figure 20: 8259A 内部结构

cli、sti 只有在 CPL <= IOPL 的时候才能执行。前面说的指令被称为 I/O 敏感指令，如果低特权级的指令试图访问这些 I/O 敏感指令将会导致常规保护错误

### 5.6.2 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。

I/O 位图与 IOPL 是或的关系，即只用满足一个条件即可。

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

## 5.7 实验改进意见

- 希望能够提供更详细的实验指导：在每个实验步骤中，提供更详细的指导和说明，比如预期结果等，这样能帮助我们更好地完成实验。
- 希望能够提供常见错误类型及纠错指南，帮助同学们在遇到问题的时候能够自己解决问题。

- 希望老师能讲解汇编代码的一些关键部分，同学们自己阅读汇编码容易忽略一些代码细节问题。

## 6 各人实验贡献与体会（每人各自撰写）

- 黄东威：独立完成实验，完成 8259A 和时钟中断例程的调试，完成和撰写自定义的中断例程部分实验报告，回答了思考题四、五。

个人体会：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，我编写了保护模式下自定义中断（键盘中断和）详见 3.3 实现自定义中断向量。基于自定义中断过程中的调试问题我编写了实验过程分析。在实验过程中，我和组员之间积极讨论，对原理有了更深刻的理解。

- 王浚杰：独立完成实验，完成 8259A 和时钟中断例程的调试，补充实验遇到的问题和解决。

个人体会：在此次实验中，通过对 8259A 中断控制器、IDT 以及时间中断的调试与编程，我深入理解了保护模式下中断处理的实现过程。在处理硬件中断的过程中，我分别实践了时间中断和键盘中断，掌握了如何正确初始化 8259A，并通过设置 8259A 的 ICW（初始化命令字）来指定 IRQ0 或 IRQ1 对应的中断向量号。同时我学会了如何配置 IDT 表项，关联中断向量与中断处理程序，并通过 cli 指令开启中断，使系统能够有效响应硬件中断。

- 程序：独立完成实验，完成 8259A 和时钟中断例程的调试，补充部分改进建议。

个人体会：本次实验涉及对中断机制的实现理解。在之前的汇编语言课上，我们主要采用 int + 中断向量号的方式来触发中断，并通过中断向量表来自定义中断处理程序。而本次实验带我们认识了保护模式下的中断机制实现，与实模式下不同，主要涉及到 8259A 和 IDT 两部分知识，并通过实现一个自定义的中断向量加深对中断机制的理解，对后面实现自己的操作系统打下基础。此外，实验还介绍了中断的特权检验和 IO 指令的保护，让我们对保护模式的保护机制有了更深的理解。

- 周业营：独立完成实验，完成 8259A 和时钟中断例程的调试并撰写调试部分实验报告，回答了思考题一、二、三。

个人体会：本次实验主要探究了异常与中断的实现。在大一学习 x86 汇编语言的课程中，就已经接触过了实模式下的中断，知道了使用中断向量表来实现中断处理程序的定位、跳转执行的机制，而在这次实验中我们重点关注了保护模式下的中断处理机制。在正式接触代码之前，还需要我们自己熟悉 8259A 的引脚排布、ICW 的主从设置、OCW 的格式等内容才能开始调试代码。在对 pmtest9a.asm 进行调试的过程中，学习了 IDT 的建立以及保护模式下中断调用；在对 pmtest9.asm 进行调试的过程中，亲身接触和学习了现代 PC 中使用 8259A 来实现中断的机制，理解了以时钟中断为例的中断定义和触发执行的机理，通过对比保护模式和实模式下对中断处理机制的差异也具体回答了之前做实验的过程中对于切换两种模式时需要开关中断的疑问，让我加深了对操作系统不同工作模式的理解。

## 7 教师评语

(实验报告的考评：依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性等五个维度综合考评)

分数	评语
85-100	<ul style="list-style-type: none"><li>• 实验内容完整或者有超出课程实验大纲的内容；</li><li>• 实验步骤详尽，能够体现完整的实验过程；</li><li>• 实验结果正确且实验数据分析得当；</li><li>• 实验心得与思考全面并且有自己的独立思考；</li><li>• 实验报告文档规范、排版整齐。</li></ul>
75-84	<ul style="list-style-type: none"><li>• 实验内容较为完整；</li><li>• 实验步骤较为详尽，能够体现实验过程；</li><li>• 实验结果正确且实验数据分析较为得当；</li><li>• 实验心得与思考全面；</li><li>• 实验报告文档规范、排版较为整齐。</li></ul>
60-74	<ul style="list-style-type: none"><li>• 实验内容有缺失；</li><li>• 实验步骤不够详尽，不能够体现完整的实验过程；</li><li>• 实验结果部分正确；</li><li>• 实验心得与思考无或者不够深入；</li><li>• 实验报告文档规范性有待增强。</li></ul>
60 以下	<ul style="list-style-type: none"><li>• 实验内容严重缺失、实验态度不够端正；</li><li>• 实验步骤不够详尽，不能够体现完整的实验过程；</li><li>• 实验结果部分正确；</li><li>• 实验心得与思考无或者不够深入；</li><li>• 实验报告文档规范性有待增强。</li></ul>

---

## 教师评分（请填写好姓名、学号）

姓名	学号	分数

教师签名：

年       月       日