中山大学数据科学与计算机学院
操作系统实验课程

实 验 报 告

教　　师　凌应标

学　　号　17341035

姓　　名　傅畅

实验名称　实验四（异步事件编程）

实 验 四
异 步 事 件 编 程

姓名：傅畅 **学号：**17341038

邮箱：fuch8@mail2.sysu.edu.cn

实验时间：周五（3-4 节）

目录

一、 实验要求

（一）利用时钟中断，在右下角轮流显示转轮

操作系统工作期间，利用时钟中断，在屏幕 24 行 79 列位置轮流显示’|’、’/’和’\’，适当控制显示速度，以方便观察效果。

（二）编写键盘中断响应程序

编写键盘中断响应程序，原有的你设计的用户程序运行时，键盘事件会做出有事反应：当键盘有按键时，屏幕适当位置显示”OUCH! OUCH!”。

（三）编写软中断服务程序

在内核中，对 33 号、34 号、35 号和 36 号中断编写中断服务程序，分别在屏幕 1/4 区域内显示一些个性化信息。再编写一个汇编语言的程序，作为用户程序，利用 int 33、int 34、int 35 和 int 36 产生中断调用你这 4 个服务程序。

二、 实验配置

（一）实验支撑环境

- 硬件：个人计算机
- 主机操作系统：Linux 4.18.0-16-generic 17-Ubuntu
- 虚拟机软件: Bochs 2.6.9

三、 x86 保护模式学习

（一）使用选择子访存

保护模式下，访问的每一段内存，都需要提前跟 CPU 声明；描述一段内存无非需要它的

1) 基地址
2) 长度
3) 扩展方式（向高地址还是向低地址）
4) 之后会用于维护访存秩序的特权级
5) 之后会与 Cache 有联系的“是否存在与主存”
6) 指明默认操作数大小。
7) 用途的粗分类

所以才有了描述符那样奇怪而繁杂的标志位，阅读到后面的章节，我才对这些奇怪的标志位有了进一步的认识

这些选择子同样需要按表的方式组织，CPU 中有一个专门的寄存器 gdtr，指向段描述子表 gdt 的首地址；段寄存器中则只需要记录用于检索该段的选择子在表中起始位置相对 GDT 的偏移量，就足够描述一个段了。　　x86 中共有 4 个控制寄存器,cr0 3。涉及到保护模式开启的另一个标志位是 cr0 的 PE 位。将该位设为 1，便进入了保护模式。

（二）分页机制

x86 提供页式访存机制。通过页目录–页表两层的结构，将线性（逻辑）地址映射到物理地址。

一张页目录和一张页表和一张页都是 4kb，4B/项 *1024 项。页目录中的每一项记录页表的物理首地址，页表每一项的都记录一张 4kb 页的物理首地址。

每个页目录/页表项的高 20 位表示物理地址的高 12 位 (因为页都是按 4kb 分配的)；低 12 位就用来记录和

- 高速缓存 Cache
- 页读写策略，
- 是否全局页/页表

有关的的信息。页目录需要通过 cr3 来指向其物理首地址，并把 cr0 中的 PG 位置一，来正式开启分页机制。

（三）中断门

门是一 x86 中一类用来处理不同特权级代码之间进行控制转移的一种（保护的）格式。本质上它还是一个描述符，只是不同于代码段和数据段而已。

调用门，中断门，任务门，陷阱门他们都非常相似，他们都必须包含 16 位目标代码段的选择子，32 位段内偏移量和特权级。

类似于 GDT，每个中断例程都通过中断门来调用，这些 256 个中断门组成一个 1KB 的表（每项 4B），由 IDTR 来指向中断向量表的首地址以及偏移量

之后每个对应中断发生时，都通过“中断向量—中断门—目标代码选择子”来实现调用中断例程。

四、 实验代码设计

（一）从软盘到使用硬盘启动

由于使用空间比较大，从实验四开始，我决定使用硬盘作为主要存储方式，bochs 的配置修改如下

```
ata0-master: type=disk, mode=flat, translation=auto, path="h.img", cylinders=2,
    heads=16, spt=64, biosdetect=auto, model="Generic 1234"

boot:disk
```

代码 1: bochs 硬盘配置参数

为了方便地制作硬盘镜像，我编写了批处理写二进制文件的脚本。在 linux 下，使用 cat 命令可以将二进制文件追加至指定文件末尾；至于‘0’字符可以由 linux 中的特殊 0 字符设备/dev/zero 提供，即：从/dev/zero 读取任意长度的都是 0 的字符，然后追加到指定文件末

```
fil=h.img
make
cat cmbr.bin > $fil # clear and cat in

cat ccore.bin >> $fil
len=$[10*512-$(stat -c %s "$fil")]
dd if=/dev/zero count=$len bs=1 | cat>> $fil

cat c1.bin >> $fil
len=$[2*16*64*512-$(stat -c %s "$fil")]
dd if=/dev/zero count=$len bs=1 | cat>> $fil
```

代码 2: make diskimg.sh

（二）**mbr.asm**

4.2.1 加载内核与用户程序

1) 这部分内容有两个程序，一个是读取一个硬盘扇区,到目的物理地址；

2) 另一个是根据程序程序头来确定程序长度并整个地读完程序。

3) 受别人代码的启发，我规定被加载程序头保留 2 个双字，分别表示目标程序长度与目标程序期望被加载的线性地址

```
Load_program:;以下加载程序,
            ;栈中的第一个参数为被加载程序的目的物理地址，第二个参数为程序在硬盘
                中的起始扇区
```

```
3       pushad
4               mov edi,[esp+40]
5
6               mov eax,[esp+36]
7               mov ebx,edi                             ;起始地址
8               call read_hard_disk_0                   ;以下读取程序的起始部分（一个扇
                    区）
9
10              ;以下判断整个程序有多大
11              mov eax,[edi]                           ;核心程序尺寸
12              xor edx,edx
13              mov ecx,512                             ;512字节每扇区
14              div ecx
15
16              or edx,edx
17              jnz @1                                  ;未除尽，因此结果比实际扇区数少1
18              dec eax                                 ;已经读了一个扇区，扇区总数减1
19      @1:
20              or eax,eax                              ;考虑实际长度≤512个字节的情况
21              jz endLoad_program                                  ;EAX=0 ?
22
23              ;读取剩余的扇区
24              mov ecx,eax                             ;32位模式下的LOOP使用ECX
25              mov eax,[esp+36]
26              inc eax                                 ;从下一个逻辑扇区接着读
27      @2:
28              call read_hard_disk_0
29              inc eax
30              loop @2                                 ;循环读，直到读完整个内核
31
32          endLoad_program:
33          popad
34          ret
```

代码 3: Load_progam

其中，read_hard_disk 将逻辑扇区号的一个扇区加载到目标地址

```
1   read_hard_disk_0:
2               ;从硬盘读取一个逻辑扇区
3               ;EAX=逻辑扇区号
4               ;DS:EBX=目标缓冲区地址
5               ;返回：EBX=EBX+512
6       push eax
7       push ecx
8       push edx
9
10      push eax
```

```
11
12      mov dx,0x1f2
13      mov al,1
14      out dx,al                       ;读取的扇区数
15
16      inc dx                          ;0x1f3
17      pop eax
18      out dx,al                       ;LBA地址7~0
19
20      inc dx                          ;0x1f4
21      mov cl,8
22      shr eax,cl
23      out dx,al                       ;LBA地址15~8
24
25      inc dx                          ;0x1f5
26      shr eax,cl
27      out dx,al                       ;LBA地址23~16
28
29      inc dx                          ;0x1f6
30      shr eax,cl
31      or al,0xe0                      ;第一硬盘  LBA地址27~24
32      out dx,al
33
34      inc dx                          ;0x1f7
35      mov al,0x20                     ;读命令
36      out dx,al
37
38      .waits:
39      in al,dx
40      and al,0x88
41      cmp al,0x08
42      jnz .waits                      ;不忙，且硬盘已准备好数据传输
43
44      mov ecx,256                     ;总共要读取的字数
45      mov dx,0x1f0
46      .readw:
47      in ax,dx
48      mov [ebx],ax
49      add ebx,2
50      loop .readw
51
52      pop edx
53      pop ecx
54      pop eax
55
56      ret
```

代码 4: read_hard_disk_0

4.2.2 安装 GDT

1) 跳进保护模式前，需要手动配置代码段和栈段和选择子

2) 为了方便，我这次实验中只使用两个选择子，一个表示数据段，一个表示代码段，两者的段界限都是 4GB。以后的 ds，ss 都使用同一选择子进行访存

3) 设置完毕之后需要使用 lgdt，sgdt, 命令直接保存或修改 gdtr，然后打开控制寄存器 cr0 的 PE 位，开启保护模式

```
1        ;计算GDT所在的逻辑段地址
2        mov eax,[cs:pgdt+0x02]                ;GDT的32位物理地址
3        xor edx,edx
4        mov ebx,16
5        div ebx                               ;分解成16位逻辑地址
6
7        mov ds,eax                            ;令DS指向该段以进行操作
8        mov ebx,edx                           ;段内起始偏移地址
9
10       ;跳过0#号描述符的槽位
11       ;创建1#描述符，保护模式下的代码段描述符
12       mov dword [ebx+0x08],0x0000ffff       ;基地址为0，界限0xFFFFF，DPL=00
13       mov dword [ebx+0x0c],0x00cf9800       ;4KB粒度，代码段描述符，向上扩展
14
15       ;创建2#描述符，保护模式下的数据段和堆栈段描述符
16       mov dword [ebx+0x10],0x0000ffff       ;基地址为0，界限0xFFFFF，DPL=00
17       mov dword [ebx+0x14],0x00cf9200       ;4KB粒度，数据段描述符，向上扩展
18
19       ;初始化描述符表寄存器GDTR
20        mov word [cs: pgdt],23                ;描述符表的界限
21
22        lgdt [cs: pgdt]
23
24        in al,0x92                           ;南桥芯片内的端口
25        or al,0000_0010B
26        out 0x92,al                          ;打开A20
27
28        cli                                  ;中断机制尚未工作
29
30       mov eax,cr0
31        or eax,1
32        mov cr0,eax                          ;设置PE位
```

```
33
34          ;以下进入保护模式... ...
35          jmp  dword  0x0008:flush                ;16位的描述符选择子：32位偏移
36                                                  ;清流水线并串行化处理器
```

代码 5: 安装 gdt

4.2.3 开启分页

1) 本次使用分页，我只是使用一张目录和一张页表，因为程序只使用到物理低端 1MB 的地址

2) 初始建立页表的时候，必须有恒等映射作为过渡，否则刚刚开启页表后的访存操作会由于对错误的线性地址操作而是失效

3) 同时手动修改部分寄存器，使其线性地址指向高端（物理地址不变）

```
1           pge:
2           ;准备打开分页机制.
3
4           ;创建系统内核的页目录表PDT
5           mov  ebx,0x00020000                 ;页目录表PDT的物理地址
6
7           ;在页目录内创建指向页目录表自己的目录项
8           mov  dword  [ebx+4092],0x00020003
9
10          mov  edx,0x00021003
11          ;在页目录内创建与线性地址0x00000000对应的目录项
12          mov  [ebx+0x000],edx                ;写入目录项（页表的物理地址和属性）
13                                              ;此目录项仅用于过渡。
14          ;在页目录内创建与线性地址0x80000000对应的目录项
15          mov  [ebx+0x800],edx                ;写入目录项（页表的物理地址和属性）
16
17          ;创建与上面那个目录项相对应的页表，初始化页表项
18          mov  ebx,0x00021000                 ;页表的物理地址
19          xor  eax,eax                        ;起始页的物理地址
20          xor  esi,esi
21  .b1:
22          mov  edx,eax
23          or  edx,0x00000003
24          mov  [ebx+esi*4],edx                ;登记页的物理地址
25          add  eax,0x1000                     ;下一个相邻页的物理地址
26          inc  esi
27          cmp  esi,256                        ;仅低端1MB内存对应的页才是有效的
28          jl  .b1
```

代码 6: 手动完善页目录和页表

安装号页目录和页表之后，修改 cr3 和 cr0 寄存器就可以正式开启分页了

```
        ;令CR3寄存器指向页目录，并正式开启页功能
        mov eax,0x00020000              ;PCD=PWT=0
        mov cr3,eax

        ;将GDT的线性地址映射到从0x80000000开始的相同位置
        sgdt [pgdt]
        mov ebx,[pgdt+2]
        add dword [pgdt+2],0x80000000   ;GDTR也用的是线性地址
        lgdt [pgdt]

        mov eax,cr0
        or eax,0x80000000
        mov cr0,eax                     ;开启分页机制
```

代码 7: 开启 paging

（三）**core.asm**

core 是内核代码段，主要负责中断处理的配置并跳转到用户程序，其内容包括

1) 安装 IDT，包括键盘、时钟中断和 4 个软中断
2) 初始化 8259A 芯片，包括其中的我重点使用的时钟中断硬件 RTC
3) 编写中断处理例程

4.3.1 安装 idt

本次中断我都使用中断门来实现，特权级都设为 0。

除了实验要求的 6 个中断，其余 250 个中断（包括异常中断处理）我都设置为空处理的例程。

```
        mov eax,general_exception_handler   ;门代码在段内偏移地址
        mov bx,flat_4gb_code_seg_sel        ;门代码所在段的选择子
        mov cx,0x8e00                       ;32位中断门，0特权级
        call flat_4gb_code_seg_sel:make_gate_descriptor

        mov ebx,idt_linear_address          ;中断描述符表的线性地址
        xor esi,esi
.idt0:
        mov [ebx+esi*8],eax
```

```
10          mov [ebx+esi*8+4],edx
11          inc esi
12          cmp esi,19                          ;安装前20个异常中断处理过程
13          jle .idt0
14
15          ;其余为保留或硬件使用的中断向量
16          mov eax,general_interrupt_handler   ;门代码在段内偏移地址
17          mov bx,flat_4gb_code_seg_sel        ;门代码所在段的选择子
18          mov cx,0x8e00                       ;32位中断门，0特权级
19          call flat_4gb_code_seg_sel:make_gate_descriptor
20
21          mov ebx,idt_linear_address          ;中断描述符表的线性地址
22  .idt1:
23          mov [ebx+esi*8],eax
24          mov [ebx+esi*8+4],edx
25          inc esi
26          cmp esi,255                         ;安装普通的中断处理过程
27          jle .idt1
28
29          ;设置实时时钟中断处理过程
30          mov eax,rtm_0x70_interrupt_handle   ;门代码在段内偏移地址
31          mov bx,flat_4gb_code_seg_sel        ;门代码所在段的选择子
32          mov cx,0x8e00                       ;32位中断门，0特权级
33          call flat_4gb_code_seg_sel:make_gate_descriptor
34
35          mov ebx,idt_linear_address          ;中断描述符表的线性地址
36          mov [ebx+0x70*8],eax
37          mov [ebx+0x70*8+4],edx
38
39         ; set the keyboard interruption
40          mov eax, keyboard_interrupt_handle
41          mov bx, flat_4gb_code_seg_sel
42          mov cx, 0x8e00
43          call flat_4gb_code_seg_sel:make_gate_descriptor
44
45          mov ebx, idt_linear_address
46          mov [ebx+0x21*8], eax
47          mov [ebx+0x21*8+4], edx
48
49          ; set personal interrupt1
50          mov eax, personal1_interrupt_handle
51          mov bx, flat_4gb_code_seg_sel
52          mov cx, 0x8e00
53          call flat_4gb_code_seg_sel:make_gate_descriptor
54
55          mov ebx, idt_linear_address
56          mov [ebx+0x11*8], eax
```

```
57          mov [ebx+0x11*8+4] , edx
58
59          ; set personal interrupt2
60          mov eax, personal2_interrupt_handle
61          mov bx, flat_4gb_code_seg_sel
62          mov cx, 0x8e00
63          call flat_4gb_code_seg_sel:make_gate_descriptor
64
65          mov ebx, idt_linear_address
66          mov [ebx+0x12*8], eax
67          mov [ebx+0x12*8+4] , edx
68          ; set personal interrupt3
69          mov eax, personal3_interrupt_handle
70          mov bx, flat_4gb_code_seg_sel
71          mov cx, 0x8e00
72          call flat_4gb_code_seg_sel:make_gate_descriptor
73
74          mov ebx, idt_linear_address
75          mov [ebx+0x13*8], eax
76          mov [ebx+0x13*8+4] , edx
77          ; set personal interrupt4
78          mov eax, personal4_interrupt_handle
79          mov bx, flat_4gb_code_seg_sel
80          mov cx, 0x8e00
81          call flat_4gb_code_seg_sel:make_gate_descriptor
82
83          mov ebx, idt_linear_address
84          mov [ebx+0x14*8], eax
85          mov [ebx+0x14*8+4] , edx
86          ;准备开放中断
87          mov word [pidt],256*8-1             ;IDT的界限
88          mov dword [pidt+2],idt_linear_address
89          lidt [pidt]                         ;加载中断描述符表寄存器IDTR
```

代码 8: 安装 IDT

4.3.2 初始化 8259A

```
1           ;设置8259A中断控制器
2           mov al,0x11
3           out 0x20,al                         ;ICW1：边沿触发/级联方式
4           mov al,0x20
5           out 0x21,al                         ;ICW2:起始中断向量
6           mov al,0x04
7           out 0x21,al                         ;ICW3:从片级联到IR2
8           mov al,0x01
```

```
9         out 0x21,al                          ;ICW4:非总线缓冲，全嵌套，正常EOI
10
11
12        mov al,0x11
13        out 0xa0,al                          ;ICW1：边沿触发/级联方式
14        mov al,0x70
15        out 0xa1,al                          ;ICW2:起始中断向量
16        mov al,0x04
17        out 0xa1,al                          ;ICW3:从片级联到IR2
18        mov al,0x01
19        out 0xa1,al                          ;ICW4:非总线缓冲，全嵌套，正常EOI
20
21
22        ;设置和时钟中断相关的硬件
23        mov al,0x0b                          ;RTC寄存器B
24        or al,0x80                           ;阻断NMI
25        out 0x70,al
26        mov al,0x12                          ;设置寄存器B，禁止周期性中断，开放更
27        out 0x71,al                          ;新结束后中断，BCD码，24小时制
28
29        in al,0xa1                           ;读8259从片的IMR寄存器
30        and al,0xfe                          ;清除bit 0(此位连接RTC)
31        out 0xa1,al                          ;写回此寄存器
32
33        mov al,0x0c
34        out 0x70,al
35        in al,0x71                           ;读RTC寄存器C，复位未决的中断状态
36
37        sti                                  ;开放硬件中断
```

代码 9: 初始化 8259A

4.3.3 中断处理例程编写

时钟中断处理过程，我先向 8259A 发送 End Of Interrupt 信号并读取寄存器 C 之后，便可以执行其他代码了。

整个处理程序除了按中断显示风火轮，还重新显示当前的时间；后者只需要向 0x70 端口发送 0x80,0x82, 0x84 就可以从 0x71 读取秒、分、时了。

向 0xb8000 写入输出信息，只需要利用平坦 4gb 选择子即可直接写入。

```
1         rtm_0x70_interrupt_handle:                    ;实时时钟中断处理过程
2
3         pushad
4
```

```
5           mov al,0x20                             ;中断结束命令EOI
6           out 0xa0,al                             ;向8259A从片发送
7           out 0x20,al                             ;向8259A主片发送
8
9           mov al,0x0c                             ;寄存器C的索引。且开放NMI
10          out 0x70,al
11          in al,0x71                              ;读一下RTC的寄存器C，否则只发生一次
                中断
12                                                  ;此处不考虑闹钟和周期性中断的情况
13          ;转动风火轮 ，并在右下角显示
14
15          xor ebx, ebx
16          mov bx , [curcyc]
17        ; shr bx , 10
18          and bx , 0x3
19          add ebx , message_cyc
20          mov cl , [ebx]
21          mov ch , 0x7
22          mov [VideoSite+0x0f9e],cx           ; (24*80+79)*2
23
24          mov bx , [curcyc]
25          inc bx
26          mov [curcyc], bx
27
28
29        mov ebx, timestrLim ; timestr+len-1, the last is '\0'
30        mov byte [ebx],0
31                            ;       显示当前时间
32                            ;       按照秒、分、时的顺序从后往前，从低到高位构造
                                  时间字符串
33
34        xor al, al
35        or al, 0x80
36        out 0x70, al
37        in al, 0x71
38        mov cl, al
39        and cl, 0x0f
40        add cl, '0'
41        dec ebx
42        mov [ebx], cl
43        shr al, 4
44        add al, '0'
45        dec ebx
46        mov [ebx], al
47        dec ebx
48        mov byte [ebx], ':'
49
```

```
50          mov al, 2
51          or al, 0x80
52          out 0x70, al
53          in al, 0x71
54          mov cl, al
55          and cl, 0x0f
56          add cl, '0'
57          dec ebx
58          mov [ebx], cl
59          shr al, 4
60          add al, '0'
61          dec ebx
62          mov [ebx], al
63          dec ebx
64          mov byte [ebx], ':'
65
66          mov al, 4
67          or al, 0x80
68          out 0x70, al
69          in al, 0x71
70          mov cl, al
71          and cl, 0x0f
72          add cl, '0'
73          dec ebx
74          mov [ebx], cl
75          shr al, 4
76          add al, '0'
77          dec ebx
78          mov [ebx], al
79
80          push ebx
81          mov ax , 0x7c6 ;        显示位置定位(24, 70)
82          shl eax, 16
83          mov ax , 0x2
84          push eax
85          call flat_4gb_code_seg_sel:simple_puts
86          add esp ,8
87            popad
88
89            iretd
```

代码 10: 时钟中断处理

键盘中断显示的 Ouch! 和时钟中断类似，同样需要记得读取端口以清空阻塞并发送 EOI

```
1           keyboard_interrupt_handle:          ;键盘中断处理例程
```

```
        ;通过判断Scan Set 1 code的最高位，判断这次中断是按下还是弹起
pushad
mov al, 0x20
out 0xa0, al
out 0x20, al

in al, 0x60                    ; 一定要把端口里的数给读出来，不然下次中断不会被触发

mov ch, al
xor ch , 0x80
shr ch , 7                     ; 最高位为0时按下，此时颜色代码为0x4
        ; 最高位为1时弹起，此时颜色代码0x0
shl ch , 2
mov cl, 'O'
mov [VideoSite+1998], cx       ; (12*8+39)*2
mov cl, 'u'
mov [VideoSite+2000], cx
mov cl, 'c'
mov [VideoSite+2002], cx
mov cl, 'h'
mov [VideoSite+2004], cx
mov cl, '!'
mov [VideoSite+2006], cx

popad
iretd
```

代码 11: 键盘中断处理例程

接下来就是要实现四个软中断例程。第一个我用于显示一个个人信息的字符串；第二个到第四个我用于执行实验一中的弹跳小球，只不过通过修改左上框的基地址分别在三个象限各执行一段时间，然后结束。

```
personal1_interrupt_handle:                ; 第一个自定义中断例程， 放在0x11处，用于显示
    pushad
    mov al, 0x20                           ; 发送EOI
    out 0xa0, al
    out 0x20, al

    push id_info
    xor eax, eax
    mov ax, 1*80+0                         ; 先压入字符串地址，再压入坐标颜色
    shl eax, 16
    mov ax, 0x09
    push eax
```

```
13      call flat_4gb_code_seg_sel:simple_puts
14      add esp , 8
15
16      popad
17      iretd
18  personal2_interrupt_handle:                 ; 第二个中断历程，在程序起始时持续显
        示弹跳小球
19      push eax
20      mov al, 0x20
21      out 0xa0, al
22      out 0x20, al
23
24
25      push 0x0
26      push 0x0                ; BaseX Y      该历程只需要压入弹跳框的左上角基地址
27      call flat_4gb_code_seg_sel:block_stone
28      add esp, 8
29      pop eax
30      iretd
31  personal3_interrupt_handle:                 ;中断例程3 4 同2，改换基地址再运行几
        次
32      push eax
33      mov al, 0x20
34      out 0xa0, al
35      out 0x20, al
36
37      push 0
38      push 40             ; BaseX Y
39      call flat_4gb_code_seg_sel:block_stone
40      add esp, 8
41      pop eax
42      iretd
43  personal4_interrupt_handle:
44              push eax
45      mov al, 0x20
46      out 0xa0, al
47      out 0x20, al
48
49      push 12
50      push 40             ; BaseX Y
51      call flat_4gb_code_seg_sel:block_stone
52      add esp, 8
53      pop eax
54      iretd
55  ;————————————————
```

代码 12: 四个软中断例程

其中的 block_stone 代码如下

```
block_stone:
pushad
mov eax, [esp+44]
mov [BaseX], al
mov eax, [esp+40]
mov [BaseY], al
mov ecx , ShowTime                    ; 限定运动次数
                                       ; 以下过程同实验一
.show:
        push ecx

        mov eax, 0x0
        mov ebx, 0x0
        mov ecx, 0x0
        mov edx, 0x0
        mov al, [posx]
        mov cl, [BaseX]
        add al, cl
        mov bl, [posy]

        mov cl, [BaseY]
        add bl, cl
        mov cx, 0x50
        mul cx


        add ax, bx
        shl eax, 1
        mov ebx, VideoSite
        add ebx, eax

        mov byte [ebx], '*'
        mov cl, [esp]
        and cl, 0x7
        inc cl
        mov byte [ebx+0x1], cl

        mov ecx, [delay]
.sleeploop:
        loop .sleeploop

        mov byte[ebx],0x0
        mov byte[ebx+0x1],0x0
.slide:
        mov dl, [posx]
        mov dh, [posy]
```

```
47              mov al, [dir]
48              xor ebx, ebx
49              mov bl, al
50              mov al, [delx+ebx]
51              mov ah, [dely+ebx]
52
53              add dl, al
54              add dh, ah
55  ; add bl, '0'
56  ; mov [VideoSite+4], bl
57  ; mov byte [VideoSite+5], 0x07
58  ; sub bl, '0'
59
60  ; add al, '0'
61  ; mov [VideoSite+6], al
62  ; mov byte [VideoSite+7], 0x07
63  ; sub al, '0'
64              mov cl, bl
65              cmp dl, 0xff
66              jne .Endjudge1
67                      xor cl, 0x02
68                      mov [dir], cl
69                      jmp near .slide
70              .Endjudge1:
71
72              cmp dl, LimX
73              jne .Endjudge2
74                      xor cl, 0x02
75                      mov [dir], cl
76                      jmp near .slide
77              .Endjudge2:
78
79              cmp dh, 0xff
80              jne .Endjudge3
81                      xor cl, 0x01
82                      mov [dir], cl
83                      jmp near .slide
84              .Endjudge3:
85
86              cmp dh, LimY
87              jne .Endjudge4
88                      xor cl, 0x01
89                      mov [dir], cl
90                      jmp near .slide
91              .Endjudge4:
92
93              mov [dir],cl
```

```
94              mov [posx], dl
95              mov [posy], dh
96
97              pop ecx
98              dec ecx
99              cmp ecx , 0x0
100             jne .show
101
102     popad
103     retf
```

代码 13: stone_v3

中断处理程序 1 中的简单字符串输出程序如下

```
1   simple_puts:
2   pushad                  ; 简单的输出字符串，不涉及光标移动
3                             ; arg1 is string pointer
4                             ; arg2 的低16位表示颜色，高16为表示显示的启示位置，即x
                                *80+y (col,xy)
5
6   mov ebx , [esp+0x28] ;from 40
7   xor eax , eax
8   mov ax  , bx
9   shr ebx , 15            ; shr 16  ,, shl 1
10
11  mov ebp , [esp+0x2c]  ; from 44
12  .enumchar:
13          mov cl,[ebp]
14          cmp cl, 0x0   ; 字符串默认以0结尾，
15          je .endenum
16          mov [VideoSite+ebx], cl
17          inc ebx
18          mov [VideoSite+ebx], al
19          inc ebx
20          inc ebp
21          jmp .enumchar
22  .endenum:
23  popad
24  retf
```

代码 14: simple print string

（四）**user.asm**

4.4.1 用户程序软中断调用

```
1   [bits 32]
```

```
2       user0_length dd user0_end-user0_start
3       user0_entry  dd user0_start
4   [section user0 vstart=0x80040500]
5   user0_start:
6       int 0x12                                    ; 依次调用自定义的软中断
7       int 0x13
8       int 0x14
9       int 0x11
10
11      mov cl, '#'
12      mov ch, 0x07
13  userloop:
14      mov ebx, 0x800b8004
15      mov [ebx],cx                            ; 主过程不断反色地显示一个‘#’字符,
16                                              ; 观察其与时钟中断的并行程度
17      xor ch, 0x1
18      jmp userloop
19  user0_end:
```

代码 15: user0 asm

五、 实验结果及分析

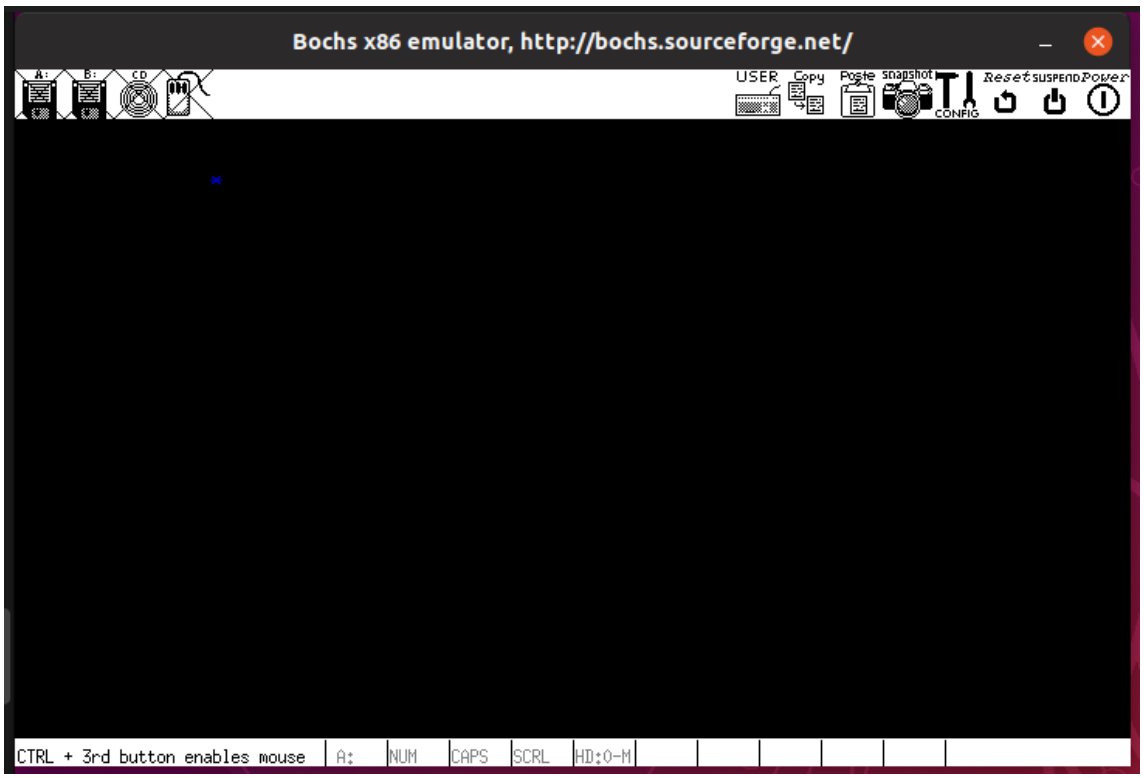

图 1: 软中断 0x12，在左上角显示弹球

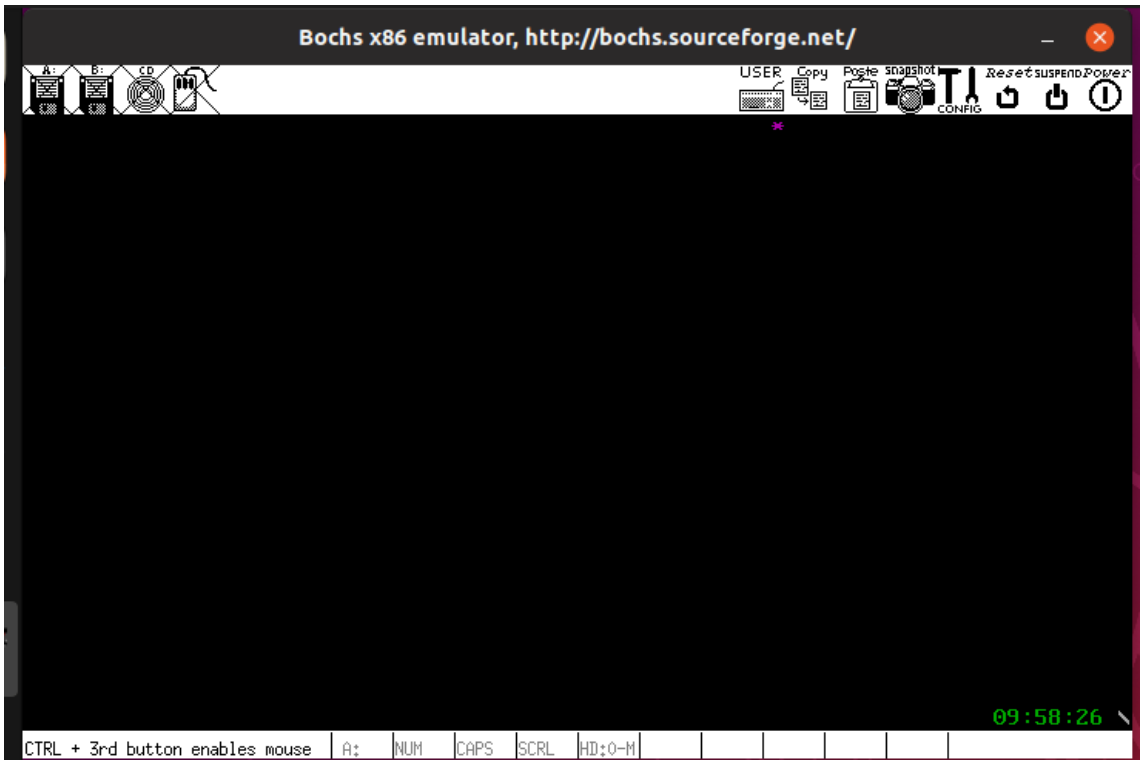

图 2: 软中断 0x13，在右上角显示弹球

图 3: 软中断 0x14，在右下角显示弹球

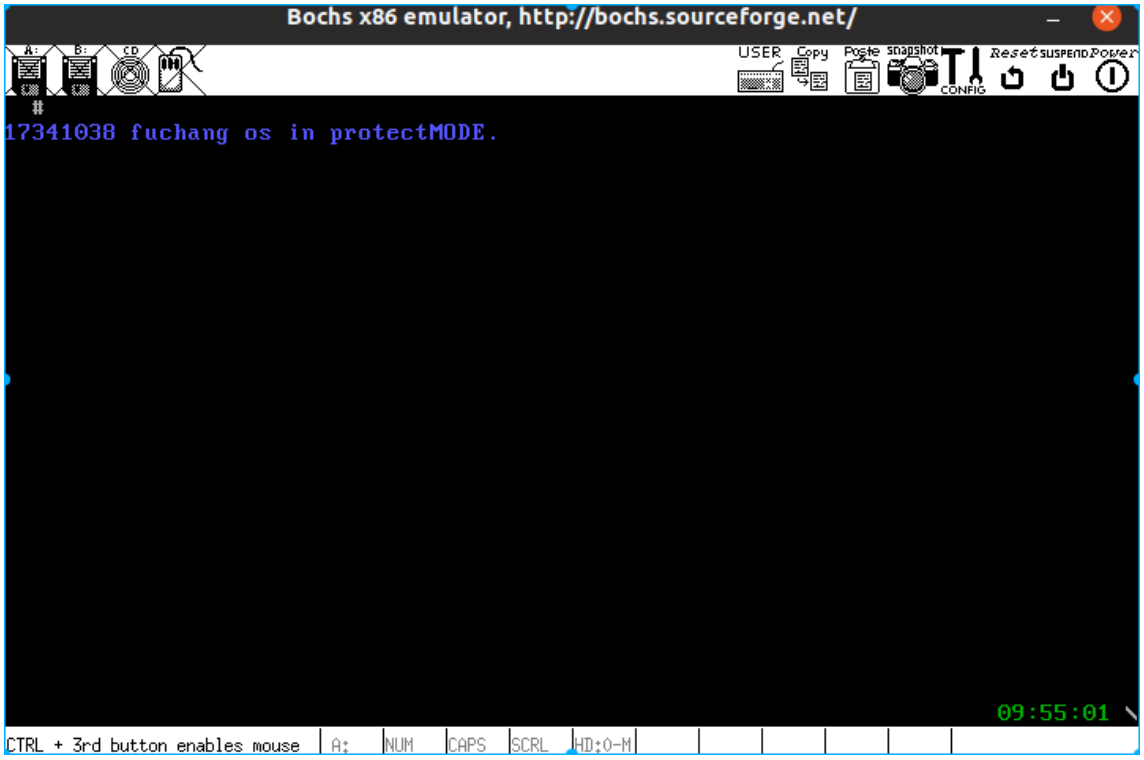

图 4: 软中断 0x11，此时第二行显示个人信息；同时主程序开始执行，显示一个白色的‘#’号

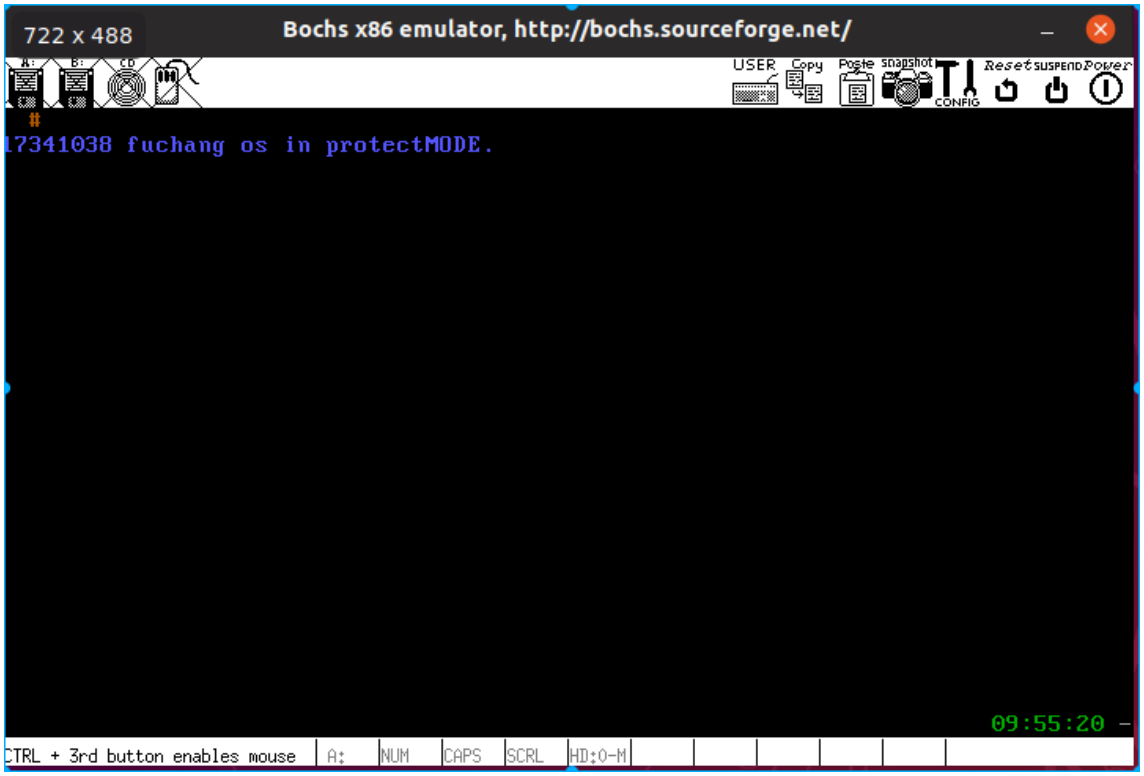

图 5: 主程序中的‘#’号变成了棕色；并且可见右下角的转轮正常转动

图 6: 按下键盘任意键，显示了 ouch！

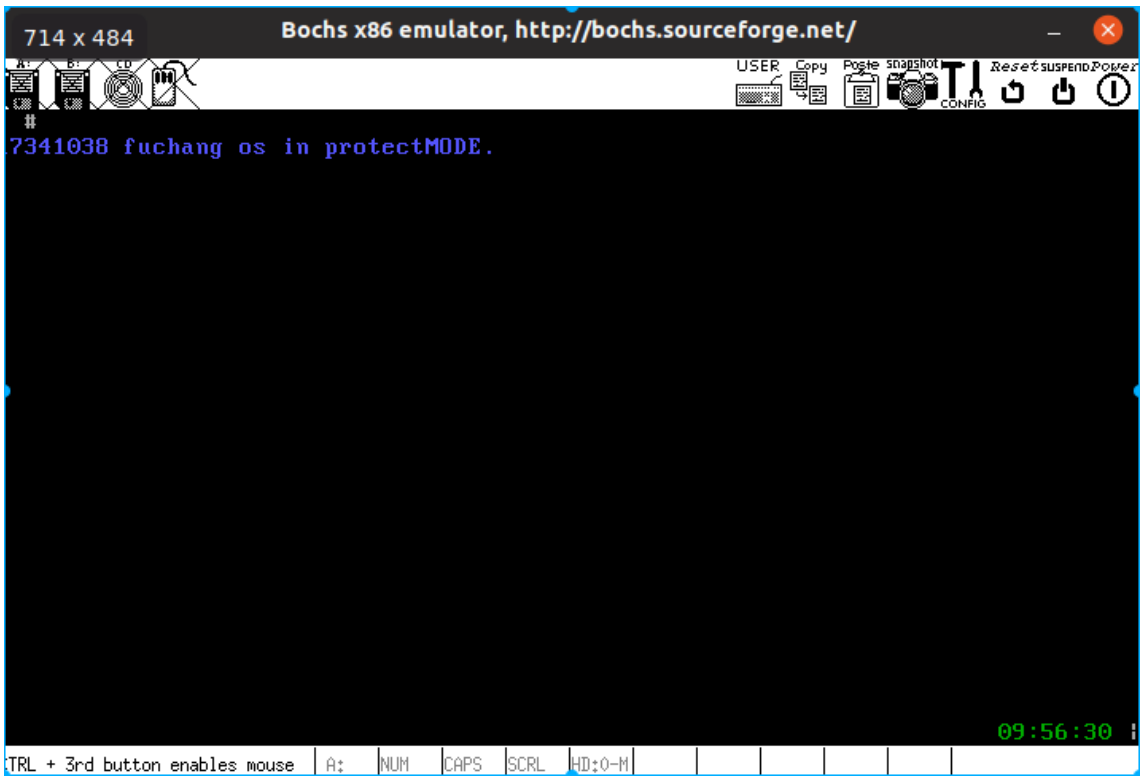

图 7: 松开键盘，ouch 消失；同时主程序的‘#’继续反色，右下角的风火轮继续转动

六、 实验总结

经过实验三的 terminal 的锻炼之后，我在实验四挑战进入保护模式得多亏有别人帮助。很多较实模式新的 x86 体系结构对我来说，非常需要仔细研究学习。我这次实验一改以往“先摸索再查书”的撞南墙做法。先沉下心研究手册和书上的说明、范例的代码。之前自己盲目摸索，其实很多坑是不必要自己去死磕上几个小时去学会的。对于我这种新手来说，认真模仿观察别人本身就是一个学习创新的过程。

尽管本次实验我模仿的东西很多，但是其实自己还是遇到了许多书上没有提到的问题。但相比起实验三，这次实验暂时我没有使用 C 语言实现一些比较复杂的内容，不过，在结下来的实验中，各种复杂的数据结构 (如“PCB”）以及复杂的协议，可能就得跳入 C 语言了。

同时我在这过程中改掉了以往全部写完再调试的不良习惯。其实对于这些大型的工程代码，聚沙成塔式地一步步添加小模块–编译调试–添加小模块–编译调试的这一循环往复地构建代码的方式，其实更加高效，因为模块测试是无论如何逃不开的。“自顶向下”地设计程序，但“自底向上”才是编写程序的好顺序。