实验六：二状态的进程模型

姓名：庞业展　　学号：16337192 数据科学与计算机学院

指导老师：凌应标

一、实验目的

1. 在内核实现多进程的二状态模型，理解简单进程的构造方法和时间片轮转调度过程。

2. 实现解释多进程的控制台命令，建立相应进程并能启动执行。

3. 至少一个进程可用于测试前一版本的系统调用，搭建完整的操作系统框架，为后续实验项目打下扎实基础。

二、实验要求

保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：

1. 在 c 程序中定义进程表，进程数量至少 4 个。

2. 内核一次性加载多个用户程序运行时，采用时间片轮转调度进程运行，用户程序的输出各占 1/4 屏幕区域，信息输出有动感，以便观察程序是否在执行。

3. 在原型中保证原有的系统调用服务可用。再编写 1 个用户程序，展示系统调用服务还能工作

三、实验设备

硬件：操作系统为 windows10 的笔记本电脑

虚拟机配置：无操作系统，102MB 硬盘，4MB 内存，启动时连接软盘

实验工具:TCC+TASM+Tlink; WinHex; Notepad++; VMware Workstation 14 Player; Dosbox 0.74; NASM

四、程序设计

进程交替执行

1. 进程表

我们在 C 模块中建立了一个重要的数据结构：进程表和进程控制块 PCB。它包括进程标识（内存单元模拟）和逻辑 CPU 模拟（逻辑 CPU 轮流映射到物理 CPU，实现多道程序的并发执行）。然后声明了大小为 5 的数组 PCBlist 用来存放 5 个 PCB（1 个内核和 4 个用户程序），以及一个指针变量指向当前进程。

```
43  typedef struct PCB
44  {
45      int ax;
46      int bx;
47      int cx;
48      int dx;
49      int si;
50      int di;
51      int bp;
52      int es;
53      int ds;
54      int ss;
55      int sp;
56      int ip;
57      int cs;
58      int flags;
59      int status;
60  };
61  struct PCB PCBlist[5];
62  struct PCB *CurrentProc;
```

2.时钟中断

本次实验我们需要采用时钟中断打断执行中的用户程序实现 CPU 在进程之间交替，就是说在每次时钟中断处理时，将 CPU 控制权从当前用户程序交接给另一个用户程序。要知道中断发生时谁在执行（CurrentProc），还要把被中断的用户程序的 CPU 寄存器信息保存到对应的 PCB 中，以后才能恢复到 CPU 中保证程序继续正确执行。中断返回时，CPU 控制权交给另一个用户程序。

设置时钟中断的程序也就是将 timer 中断服务程序的段地址和偏移地址送到时钟中断向量表中去。

```
418 clock_vector dw 0,0
419 ;设置时钟中断 08h
420 set_clock_int proc
421     cli
422     push es
423     push ax
424     xor ax,ax
425     mov es,ax
426     ;save the vector
427     mov ax,word ptr es:[20h]
428     mov word ptr [clock_vector],ax
429     mov ax,word ptr es:[22h]
430     mov word ptr [clock_vector+2],ax
431     ;fill the vector
432     mov word ptr es:[20h],offset Timer
433     mov ax,cs
434     mov word ptr es:[22h],ax
435     pop ax
436     pop es
437     sti
438     ret
439 set_clock_int endp
```

我们重写了一个跳转函数，从内核跳转到 Timer 过程进行多个用户程序的轮换，首先保存内核的 ss 和 sp（在这次实验中我们把内核也当作一个进程（CurrentProc=0），可以通过时间片轮转与四个用户程序切换，当用户程序切换一定次数时，把 CurrentProc 置 0，返回内核），然后手动模拟时钟中断从内核开始进行时间片轮转，轮转期间不再进入内核。最后把 ss 和 sp 恢复，回到内核，恢复时钟中断。

```
232     mov word ptr [kernelss],ss
233     mov word ptr [kernelsp],sp
234     call SetTimer
235     call set_clock_int
236     ;;;;;;;;;;;;;;
237     ;手动模拟时钟中断从内核开始进行时间片轮转
238     ;时间片轮转期间 不轮转到内核 直到轮转结束
239     mov ax,512  ;
240     push ax     ;将flag压入栈
241     push cs     ;将cs压入栈
242     call Timer  ;利用call将ip压栈
243     ;;;;;;;;;;;;;;
244     mov ss,word ptr [kernelss]
245     mov sp,word ptr [kernelsp]
246     call re_clock_int
```

SetTimer 函数，设置计时器，使时钟每秒约 20 次中断，在多个用户程序分享 CPU 时间的时候，通过每秒 20 次的时钟中断，我们就可以观察到在多个用户程序在 cpu 中的时候，程序几乎是并发执行的，达到了一种分时的效果。

```
;设置计时器，时钟每秒20次中断（50ms一次）：
;*************************************************************
;*                      SetTimer
;*************************************************************
; 设置计时器函数
SetTimer proc
    cli
    push ax
    mov al,34h          ; 设控制字值
    out 43h,al              ; 写控制字到控制字寄存器
    mov ax,60000            ; 约每秒20次中断（50ms一次）
    out 40h,al              ; 写计数器0的低字节
    mov al,ah               ; AL=AH
    out 40h,al              ; 写计数器0的高字节
    pop ax
    sti
    ret
SetTimer endp
```

Timer 函数是时钟中断服务程序，首先在内核中进行现场保护工作，然后利用 call 指令跳转到 save 进行当前用户程序的现场保护工作，接着调用 c 中的调度函数决定下一个在 CPU 中执行的用户程序，最后利用 restart 函数作现场恢复工作，恢复当前已经被调度完成的程序的寄存器相关内容进行执行。

```
;时钟中断处理程序参考程序：
;*******************************************************
;*                          Timer
;*******************************************************
; 时钟中断处理程序
                ; 使用当前进程的栈
Timer proc
    call save
    call near ptr _Scheduler
    jmp restart

Timer endp
```

我们用 C 实现了进程调度函数 Scheduler（），采用时间片轮转法，依次运行 pcb_num 个进程(不包括内核)，current_pcb 为当前运行进程的编号。中断一定次数（time_cnt）后自动返回内核（current_pcb=0），并且将指针 CurrentProc 设置成当前运行进程的结构体地址，为了能够让汇编中访问进程表中的内容。

```
/*****************************************************************
/*                    Scheduler 进程调度函数：时间片轮转
/*****************************************************************
void Scheduler()
{
    if(pcb_num)current_pcb++;
    if(current_pcb==pcb_num+1)current_pcb=1;

    /*中断一定次数后结束所有用户程序,返回内核*/
    /*规定0号pcb为内核pcb*/
    time_cnt--;
    if(time_cnt==0)current_pcb=0;
    CurrentProc=&PCBlist[current_pcb];
    return;
}
```

3.现场保护（save 过程）

内核一个专门的程序 save，负责保护被中断的进程的现场（记录 PCB 中的 16 个寄存器值），将这些寄存器的值转移至当前进程的 PCB 中。为了及时保护中断现场，必须在中断处理函数的最开始处，通过 save 过程立即保存被中断程序的所有上下文寄存器中的当前值。下面的基本框架基本上是老师 ppt 上面的参考代码，然后我们改了对应的 PCB 进程表的指针地址，使内核可以访问到 c 中的结构体中数据。

```
;中断现场保护
public save
save proc
    push ds
    push cs
    pop ds
    pop word ptr [ds_save]
    pop word ptr [ret_save]
    mov word ptr[si_save],si
    mov si,word ptr [_CurrentProc]
    add si,22
    pop word ptr [si]
    add si,2
    pop word ptr [si]
    add si,2
    pop word ptr [si]
    mov word ptr [si-6],sp
    mov word ptr [si-8], ss
    mov si,ds
    mov ss,si
    mov sp,word ptr [_CurrentProc]
    add sp,18
    push word ptr[ds_save]
    push es
    push bp
    push di
    push word ptr[si_save]
    push dx
    push cx
    push bx
    push ax
    mov sp,word ptr[kernelsp]
    mov ax,word ptr [ret_save]
    jmp ax
save endp
```

4.进程切换（restart 过程）

用内核函数 restart 来恢复下一进程原来被中断时的上下文，并切换到下一进程运行。如果使用我们的临时（对应于下一进程的）PCB 栈，也可以用指令 IRET 完成进程切换，但是却无法进行栈切换。因为在执行 IRET 指令之后，执行权已经转到新进程，无法执行栈切换的内核代码；而如果在执行 IRET 指令之前执行栈切换（设置新进程的 SS 和 SP 的值），则 IRET 指令就无法正确执行，因为 IRET 必须使用 PCB 栈才能完成自己的任务。所以我们使用 IRET 指令，在用户进程的栈中保存 IP、CS 和 FLAGS 时，必须将 IP、CS 和 FLAGS 放回用户进程栈中。

```
lds_low dw ?
lds_high dw ?
restart  proc
    mov word ptr[kernelsp],sp
    mov sp,word ptr[_CurrentProc]
    pop ax
    pop bx
    pop cx
    pop dx
    pop si
    pop di
    pop bp
    pop es
    mov word ptr[lds_low],bx
    pop word ptr[lds_high]
    mov bx,sp
    mov bx,word ptr[bx]
    mov ss,bx
    mov bx,sp
    add bx,2
    mov sp,word ptr[bx]
    push word ptr[bx+6]
    push word ptr[bx+4]
    push word ptr[bx+2]
    lds bx,dword ptr[lds_low]
    push ax
    mov al,20h              ; AL = EOI
    out 20h,al              ; 发送EOI到主8529A
    out 0A0h,al             ; 发送EOI到从8529A
    pop ax
    iret                    ; 从中断返回
restart  endp
```

增加的系统调用

这部分的思想就像老师的 ppt 上说的那样，在内核中用一个系统调用的总入口，用来获取参数和分析功能号，再根据功能号产生分支结构，根据系统调用号决定选择对应的分支完成相应的服务；每个分支实现一种系统调用功能，用汇编语言实现。部分代码已经在上次实验的报告中陈述，故不在此赘述。

1. 设置中断向量表

2. 中断服务程序

首先关中断，压栈保护。然后计算目标系统调用偏移量=跳转表偏移量+ah*2 来调用目标系统调用。最后向 8259A 发送 EOI，出栈，开中断。

3. 功能 1（ah=0）：判断一个数的奇偶性

用户程序把需要判断的参数传进 cx 寄存器中，然后把全局变量_number 设为 cx 里的值，调用 C 中的函数 C_evenodd（）进行判断。

4. 功能 2（ah=1）：显示学号字符串

本功能的设计思想与实验一中的显示姓名学号差不多，所以不在此赘述。

5. 功能 3（ah=2）：两个个位数相加并显示结果

汇编部分：

```
;系统调用2，ah=2
;两个只有个位的正数相加
vector2 proc
    push cx
    mov al,cl
    add al,ch
    mov byte ptr [_sum],al
    mov byte ptr [_num1],cl
    mov byte ptr [_num2],ch
    call near ptr _C_sum
    pop cx
    ret
vector2 endp
```

C 部分：

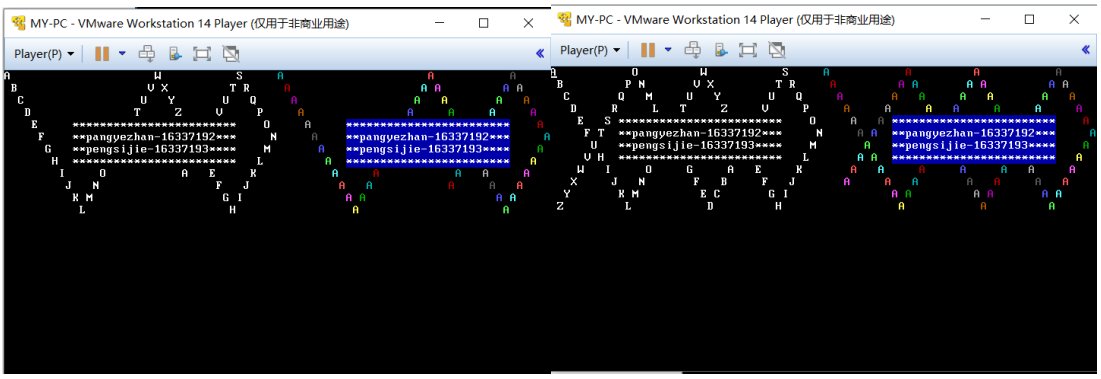
```
int sum=0;
int num1,num2;
char result[]=" + =  ";
C_sum()
{
    result[0]=num1+'0';
    result[2]=num2+'0';
    if(sum>=10)
    {
        result[4]=sum/10+'0';
        result[5]=sum%10+'0';
        for(int_i=0;result[int_i]!='\0';++int_i)
        show_color_char(result[int_i],13,34+int_i,5);
    }
    else
    {
        result[4]=sum+'0';
        for(int_i=0;result[int_i]!='\0';++int_i)
        show_color_char(result[int_i],14,34+int_i,5);
    }
}
```

系统调用测试程序（test2）

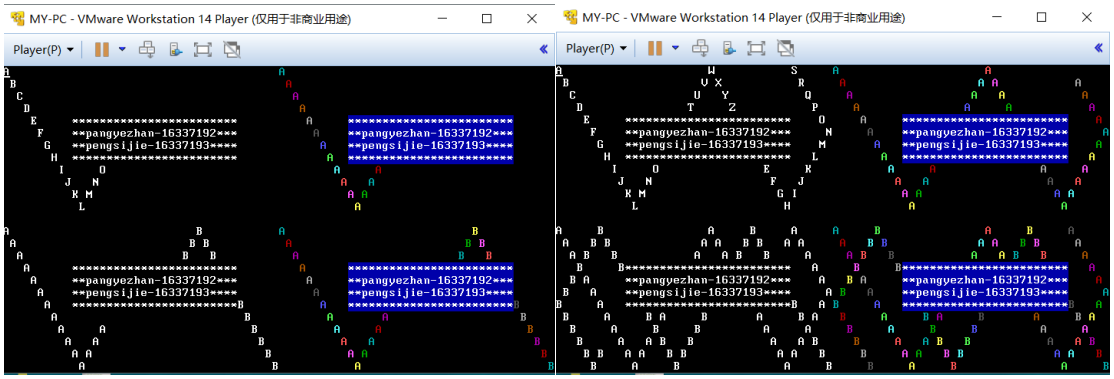
```
5     ;判断一个数是奇数还是偶数
6     mov ah,0
7     mov cx,3    ;这个数是3
8     int 21h
9     mov ah,0
10    mov cx,4     ;这个数是4
11    int 21h
12    ;在0ah行，20h列显示颜色为'3f'的信息
13    mov ah,1
14    mov dx,0a20h
15    mov cl,3fh
16    int 21h
17    ;计算3+4
18    mov ah,2
19    mov cl,3
20    mov ch,4
21    int 21h
22    ;计算5+8
23    mov ah,2
24    mov cl,5
25    mov ch,8
26    int 21h
```

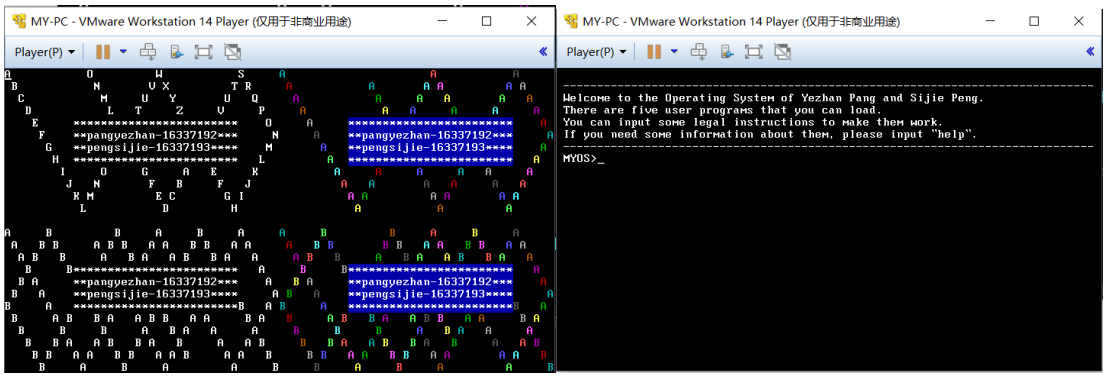
五、实验结果

1. 两个程序运行：run 1 2

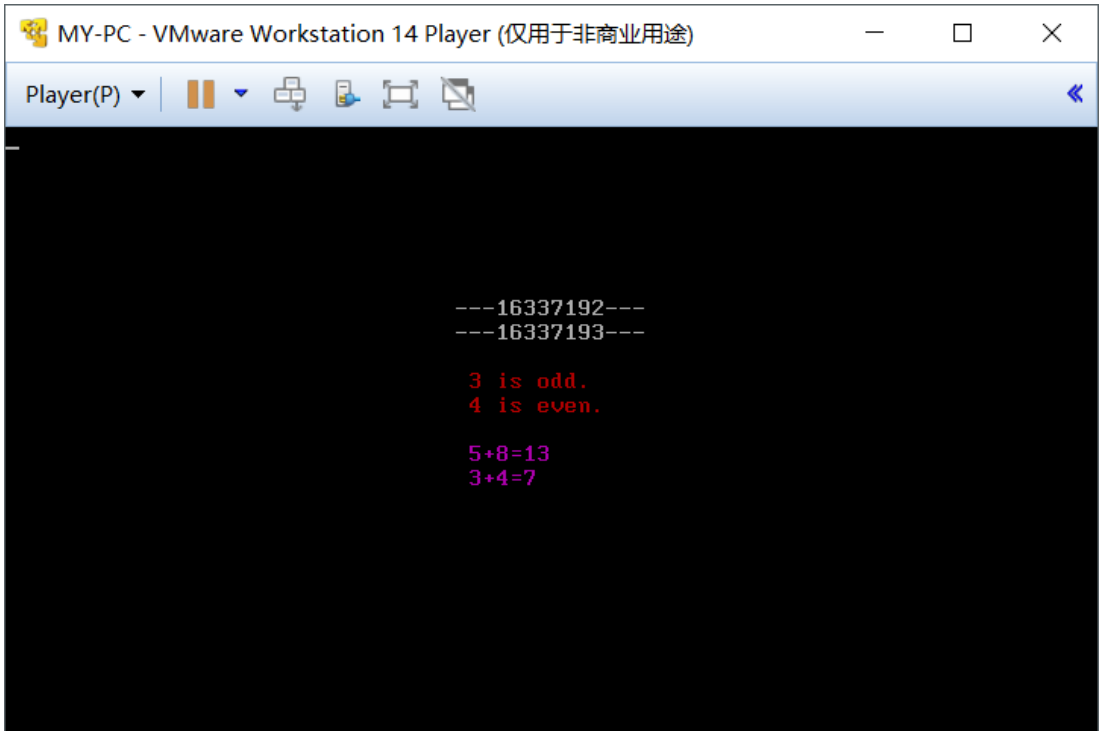


2. 多个程序运行：run1234



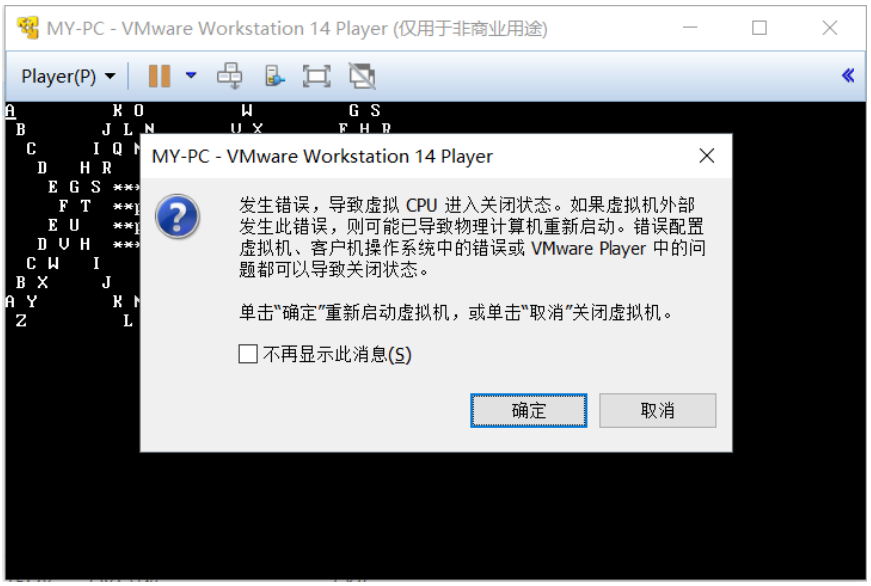


3. 系统调用测试程序：test



六、实验中出现的问题

1. 用户程序与进程切换的逻辑错误：我们在用户程序中设置了一个计数器，程序显示一定数量字符后自动返回内核。同时我们在内核里也设置了一个计数器，当进程切换一定次数时自动返回内核。而当内核中的中断计数变量小于该计数器的值时就会出现 bug。如下图所示。原因就是进程显示了一定数量的字符后，已经自动返回内核，而中断计数变量还未为 0，仍然在进行切换。



解决方法是在让进程切换的次数小于进程显示的字符数。于是我们进行了如下修改：

```
112     time_cnt=30*pcb_num;
```

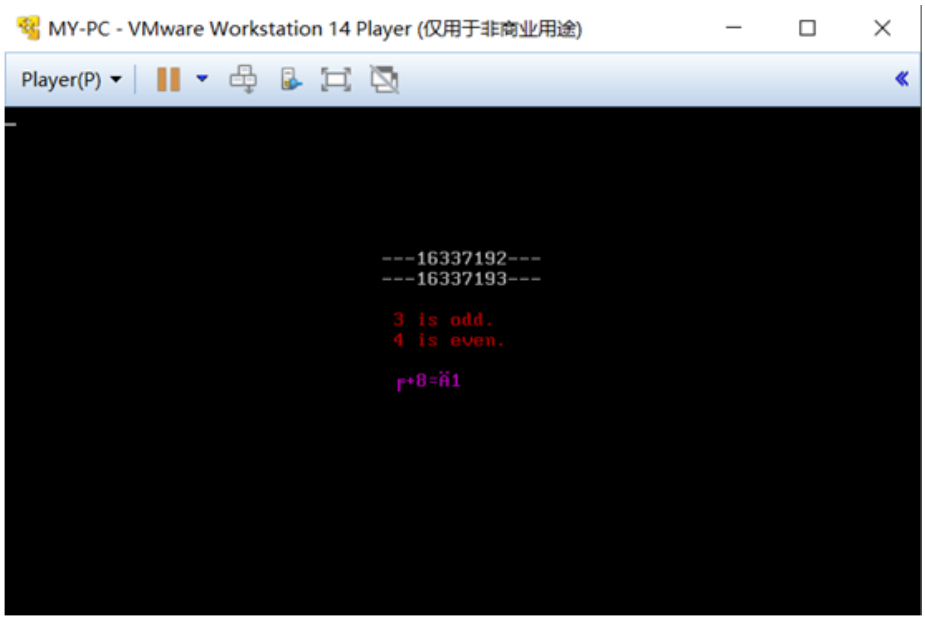
如此一来，进程切换的次数一定会小于进程显示的总字符数。

2. 系统调用功能 3 显示出现乱码。

原本的设计是在用户程序中利用 bx 和 cx 寄存器传进两个参数，然后调用 C 中的函数 C_sum 来实现加法器功能并显示结果。

```
583 ;系统调用2, ah=2
584 ;两个只有个位的正数相加
585 vector2 proc
586     push bx
587     push cx
588     mov ax,bx
589     add ax,cx
590     mov word ptr [_sum],ax
591     mov word ptr [_num1],bx
592     mov word ptr [_num2],cx
593     call near ptr _C_sum
594     pop cx
595     pop bx
596     ret
597 vector2 endp
```

出现乱码：



修改：

```
583 ;系统调用2，ah=2
584 ;两个只有个位的正数相加
585 vector2 proc
586     push cx
587     mov al,cl
588     add al,ch
589     mov byte ptr [_sum],al
590     mov byte ptr [_num1],cl
591     mov byte ptr [_num2],ch
592     call near ptr _C_sum
593     pop cx
594     ret
595 vector2 endp
```

```
23      mov ah,2
24      mov cl,5
25      mov ch,8
26      int 21h
```

3.将内核的段地址 cs 修改为了 800h，将用户程序的 cs 改为了用户程序加载的段地址。修改了引导扇区里面的代码：

```
mov bx,OffSetOfKernal
mov ah,2
mov al,20
mov dl,0
mov dh,0
mov ch,0
mov cl,11
int 13H ;
jmp OffSetOfKernal
```

```
mov bx,OffSetOfKernal
mov ah,2
mov al,20
mov dl,0
mov dh,0
mov ch,0
mov cl,11
int 13H ;
jmp 800h:100h
```

如此一来，内核的 ds 和 cs 就一致了，不需要再用 add ax，800h 这行代码。以前的实验中,都是利用改变 ip 的值来实现跳转到用户程序，因此 cs 一直都为内核的 cs，导致用户程序所加载的地方是固定的，必须由用户程序决定。在本次实验中,采用的是将 cs 改为用户程序所加载的段地址,ip 改为用户程序所加载到的偏移量这种方法来实现跳转用户程

序。因此用户程序不再需要手动修改 ds 值,只要根据运行时的 cs 值来决定 ds 寄存器即可。因此用户程序所加载到的地方由内核来决定。

七、实验心得

庞业展：

这次实验的主要内容是实现二状态进程模型。进程模型的大致框架如下，首先系统启动后读取引导扇区的数据，加载内核程序，内核程序中会加载用户程序并初始化时钟，并设置时钟中断处理程序。当时钟中断发生时，保存状态寄存器的值，暂停当前的进程，根据调度程序挑选下一个进程，恢复选中进程的 CPU 寄存器，最后启动选中的进程，如此循环往复，实现进程之间的切换。

这次实验的代码很大程度上参考了老师提供的代码例如 save，restart 和 timer。但是也不可能是生搬硬套就可以的，例如老师代码里的 CurrentProc 这个变量，我一开始是声明为 int 型变量，结果在编译时就会报错，后来参考了老师提供的原型后声明为指向当前进程 PCB 的指针才解决。还有下面的 mov ax，1193182/20 用 TCC+TASM 编译也会报错，因为 ax 是 16 位寄存器，不能存放精度过高的小数，所以我把它改成了 60000，就可以编译通过。

```
SetTimer:
    mov al,34h                          ;设控制字值
    out 43h,al                          ;写控制字到控制字寄存器
    mov ax,1193182/20       ;每秒20次中断（50ms一次）
    out 40h,al                          ;写计数器0的低字节
    mov al,ah                           ;AL=AH
    out 40h,al                          ;写计数器0的高字节
    ret
```

于此同时我们重写了实验四/五的 jmp 函数和 load 函数（jmp2 和 load2），具体的修改请看代码。

我们在实验中用了老师所说的“滚雪球”的方法，先按照要求实现进程轮转，把上次实验的利用时钟中断画框和键盘中断 OUCH！还有自己写的中断服务程序都去掉了，然后逐步增加和完善操作系统的功能，的确 debug 的难度和复杂度都下降不少（缺点就是最后懒得再加上去了。。。）把理论运用于实际，这对于我们来说也是一个不小的收获。

中断发生的时候，cpu 就会把 flags,cs,ip 顺序压入当前被中断的进程堆栈中，这是为了保存 CPU 当前执行的代码段的地址和下一条指令的地址以便返回的时候继续执行。而 iret 指令在返回的时候又会自动把 ip,cs,flags 从栈顶弹出去。而且本次实验和之前的实验不同的就是，之前实验的用户程序一直和内核共用一个 cs，只是偏移地址不一样而已，但是在这次的实验中，我们在用户程序中设置了独立的 cs：ip，ss 和 ds。然后就是更加了解了 ss 和 sp 的使用，ss 和 sp 对应的是栈顶元素的段地址和偏移量，且始终指向栈顶元素，并且每往栈中增加一个元素 sp 指针的值就要减 2 因为字占两个内存单元，栈指针在调用 c 函数的传参时候，也是一样的，每隔 2 个单位就是下一个参数。然后在各种跳转的时候就要特别注意这两个寄存器的使用，因为只要 ss 和 sp 指向不准确，程序就会出现各种各样的问题，这个在之前的实验中就已经遇到过很多次了。

总之一些逻辑上的问题加以理论以及一些相关的资料是能够花一些时间让人搞清楚其中的相关联系的，在这次的实验中，更加了解了寄存器 cs ip 和 ss sp 在程序调度过程中的重要性，相信在之后的实验中也不免会遇到关于此类的问题。我一般在解决问题的时候喜欢拿一张草稿纸仔细列出其中的因果关系，先干什么后干什么，然后等到差不多理顺了之后才会开始动手实践，我觉得这个在解决很多方面的问题都十分的有效。