实验六：实现时间片轮转的二态进程模型

学院名称：	数据科学与计算机学院
专业（班级）：	18计科教学3班
学生姓名:	夏一溥
学号:	18340178
时间:	2020 年 7 月 3 日
实验四：	实现时间片轮转的二态进程模型

实验内容

- [x] 内核中定义程序控制块，包括进程号、内存地址、CPU寄存器、进程状态。（与老师给出的做法不同，我是用宏在汇编中申请空间作为PCB表使用）
- [x] 增加一条命令使得操作系统可同时执行多个用户程序。内核预先加载并创建多个进程，再实现分时运行，并使用二态进程模型。
- [x] 保留风火轮显示，并在时钟调用中增加修改后的save()和restart()两个过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的运行。

实验过程

实验环境:

- 操作系统： Ubuntu 18.04.4 LTS
- 汇编编译器： NASM
- 调试工具： Bochs
- c语言编译器： GCC

设计思路:

Part 1： 定义程序控制块

在定义时钟中断处理程序的汇编段中定义进程控制块，使用了宏:

```
%macro prog_contr_block 1       ; 参数：段值
    %1_AX dw 0
    %1_CX dw 0
    %1_DX dw 0
    %1_BX dw 0
    %1_SP dw 0
    %1_BP dw 0
    %1_SI dw 0
    %1_DI dw 0
    %1_DS dw 0
    %1_ES dw 0
    %1_FS dw 0
```

```
    %1_GS dw 0
    %1_SS dw 0
    %1_IP dw 0
    %1_CS dw 0
    %1_FG dw 0
    %1_ID db 0
    %1_STATE db 0                       ;0-初始 1-就绪 2-运行
%endmacro
pcbSize equ pcb1 - pcb0
pcb0:
    _AX dw 0
    _CX dw 0
    _DX dw 0
    _BX dw 0
    _SP dw 0
    _BP dw 0
    _SI dw 0
    _DI dw 0
    _DS dw 0
    _ES dw 0
    _FS dw 0
    _GS dw 0
    _SS dw 0
    _IP dw 0
    _CS dw 0
    _FG dw 0
    _ID db 0
    _STATE db 0
pcb1:
prog_contr_block _1
prog_contr_block _2
prog_contr_block _3
prog_contr_block _4
prog_contr_block _5
prog_contr_block _6
prog_contr_block _7
prog_contr_block _8
prog_contr_block _9
prog_contr_block _10
```

其中，`pcb0` 内核各信息的存储块，`pcb1` 后为进程控制块。进程控制块的长度通过

```
pcbSize equ pcb1 - pcb0
```

获得。

PCB表的声明放在程序段的末尾，这样使得后续内存地址方便扩展。进程创建后通过

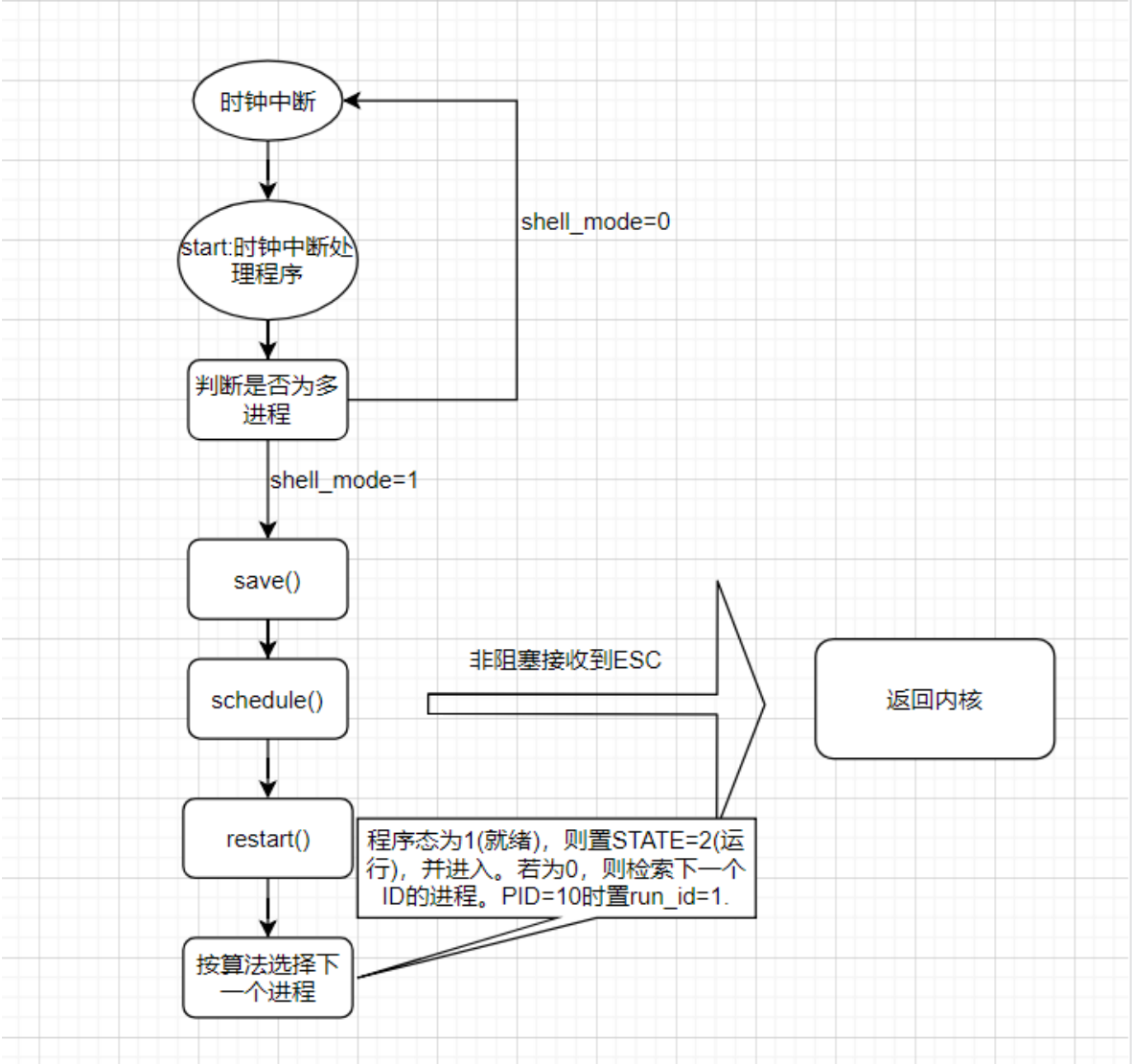
```
push word[cs:_FG+di]            ; 新进程flags
push word[cs:_CS+di]            ; 新进程cs
push word[cs:_IP+di]            ; 新进程ip
...
iret
```

确保栈为`stack/*/flags/cs/ip` 通过`iret`进入对应的进程段。

Part 2： 使用时间片轮转的二态进程模型的实现

概述

流程图如下：



从内核进入多进程模式

这一部分的实现我想了很久，最开始考虑逐一加载需要运行的多个进程，但这样实际是将操作系统作为了Prog1，并不符合底层逻辑。

我想讲所有的将要运行的进程统一加载到内存但并不运行，这时想法可行。

```
    for(int i=1;i<5;i++){   //挂载Prog 1, 2, 3, 4
                u16 sec = 1 + (i - 1)*2;
                _preload(sec,addr[i],i);
            }

            shell_mode=1;
            pause();
            shell_mode=0;
            _CLS();
```

做法是先通过`int 13H`挂载所有进程到内存，同时我设置了多进程模式的开关`shell_mode`，当其为1时多进程模式打开，等待下一次时钟中断就自行跳转到对应进程的内存。

SAVE 过程

通过维护栈来完成：

```
    ;Stack: */fg/cs/ip
    push sp
    ;Stack: */fg/cs/ip/sp/
    push di
    push ax
    ;Stack: */fg/cs/ip/sp/di/ax
    push ds
    push cs
    pop ds
    ;Stack: */fg/cs/ip/sp/di/ax/ds
    mov ax,pcbSize
    mul word[cs:run_now]
    mov di,ax
    mov ax,ss
    mov word[ds:_SS+di],ax
    mov ax,gs
    mov word[ds:_GS+di],ax
    mov ax,fs
    mov word[ds:_FS+di],ax
    mov ax,es
    mov word[ds:_ES+di],ax
    mov ax,si
    mov word[ds:_SI+di],ax
    mov ax,bp
    mov word[ds:_BP+di],ax
    mov ax,bx
    mov word[ds:_BX+di],ax
    mov ax,cx
    mov word[ds:_CX+di],ax
    mov ax,dx
    mov word[ds:_DX+di],ax
    ;Stack: */fg/cs/ip/sp/di/ax/ds
    pop ax
    ;Stack: */fg/cs/ip/sp/di/ax/
    mov word[ds:_DS+di],ax
    pop ax
    mov word[ds:_AX+di],ax
    pop ax
    mov word[ds:_DI+di],ax
    pop ax
    mov word[ds:_SP+di],ax
    ;Stack: */fg/cs/ip/
    pop ax
    mov word[ds:_IP+di],ax
    pop ax
    mov word[ds:_CS+di],ax
    pop ax
    mov word[ds:_FG+di],ax
    ;Stack: */
```

此时，所有寄存器的值已经被存储在对应程序控制块中，将原本栈中fg，cs，ip弹出，需要注意在后续中恢复sp。

Schedule 过程

在二态进程模型中，每个进程有两个状态--就绪态和运行态。Schedule过程通过判断进程的状态来调度获得下一次时钟中断进入的进程块。

```
schedule:
    pusha
    mov ax,pcbSize
    mul word[cs:run_now]
    mov di,ax                                   ;di作为程序号的偏移量

    mov byte[cs:_STATE+di], 1
    ...
    judge_num:
        inc word[cs:run_now]
        add di, pcbSize
        cmp word[cs:run_now], 10                ;总共最多10个进程
        jna change_state
        mov word[cs:run_now], 1
        mov di, pcbSize
    change_state:
        cmp byte[cs:_STATE+di], 1
        jne judge_num
        mov byte[cs:_STATE+di], 2
    quit_schedule:
    popa

;restart()
```

同时，Schedule 过程还要负责多进程的退出，与应用程序的`ctr+'z'`区别，这里非阻塞检索`Esc`,当检索到时，回到内核，并将所有进程的状态置0：

```
shutdown:
    pusha
    mov cx, 10
    mov si, pcb0+pcbSize
    loop1:
        mov byte[cs:si+_STATE], 0
        add si, 34
        loop loop1
    popa
    ret
```

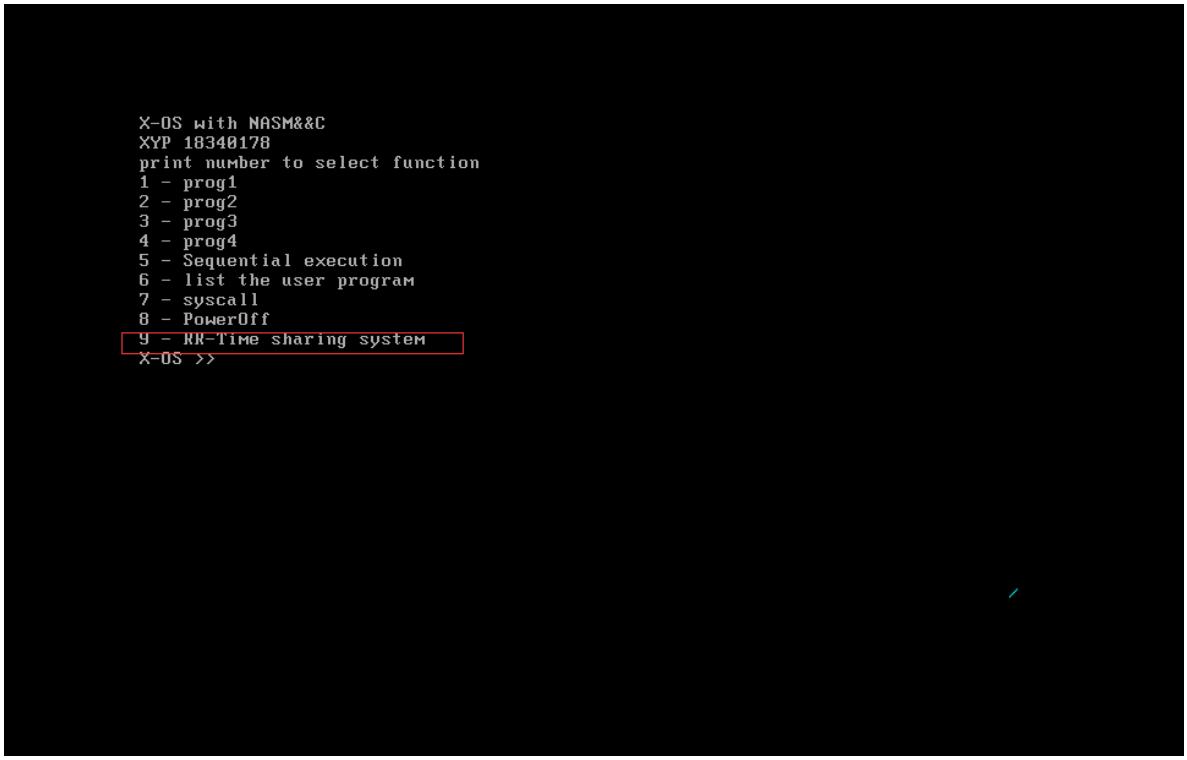
restart 过程

这个过程进行的操作是将进程控制块的内容覆盖当前的寄存器内容，值得注意的是最后跳转的部分，需要保证栈内容为`*/fg/cs/ip/`：

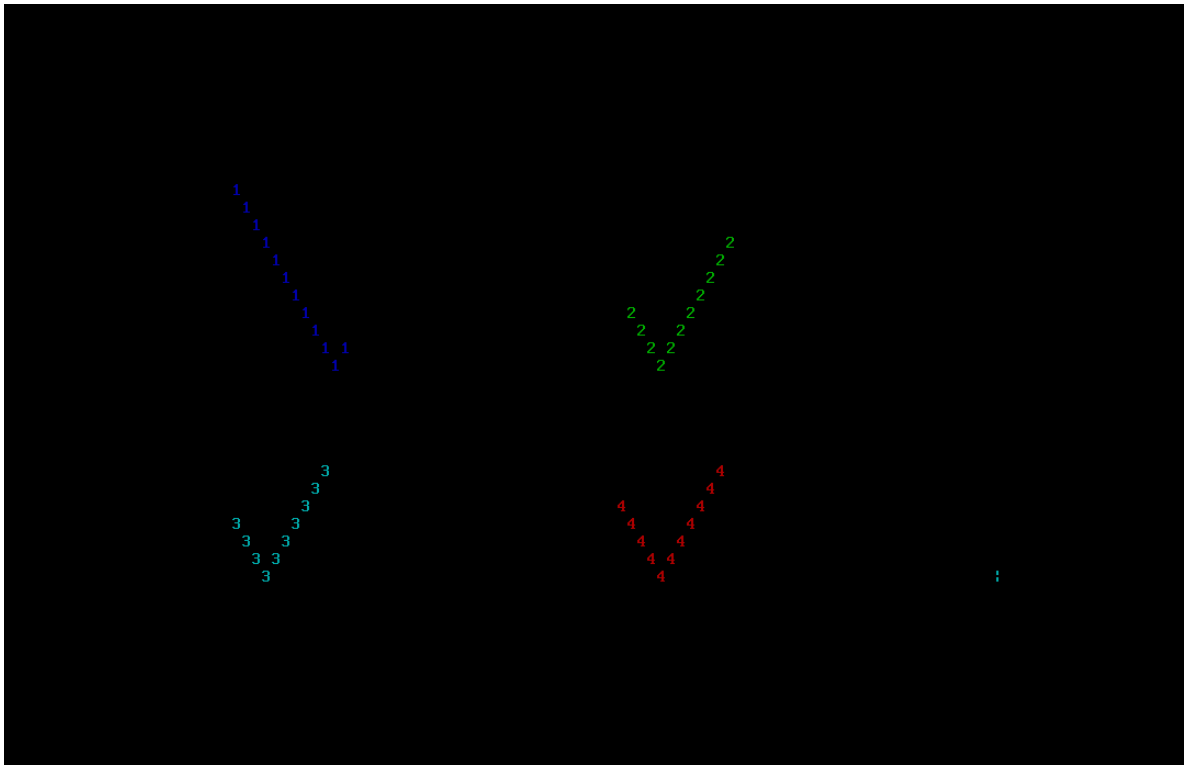
```
    push word[cs:_FG+di]            ; 新进程flags
    push word[cs:_CS+di]            ; 新进程cs
    push word[cs:_IP+di]            ; 新进程ip

    ;Stack: */fg/cs/ip/
    push word[cs:_DI+di]
    pop di

    iret
```

实验结果：

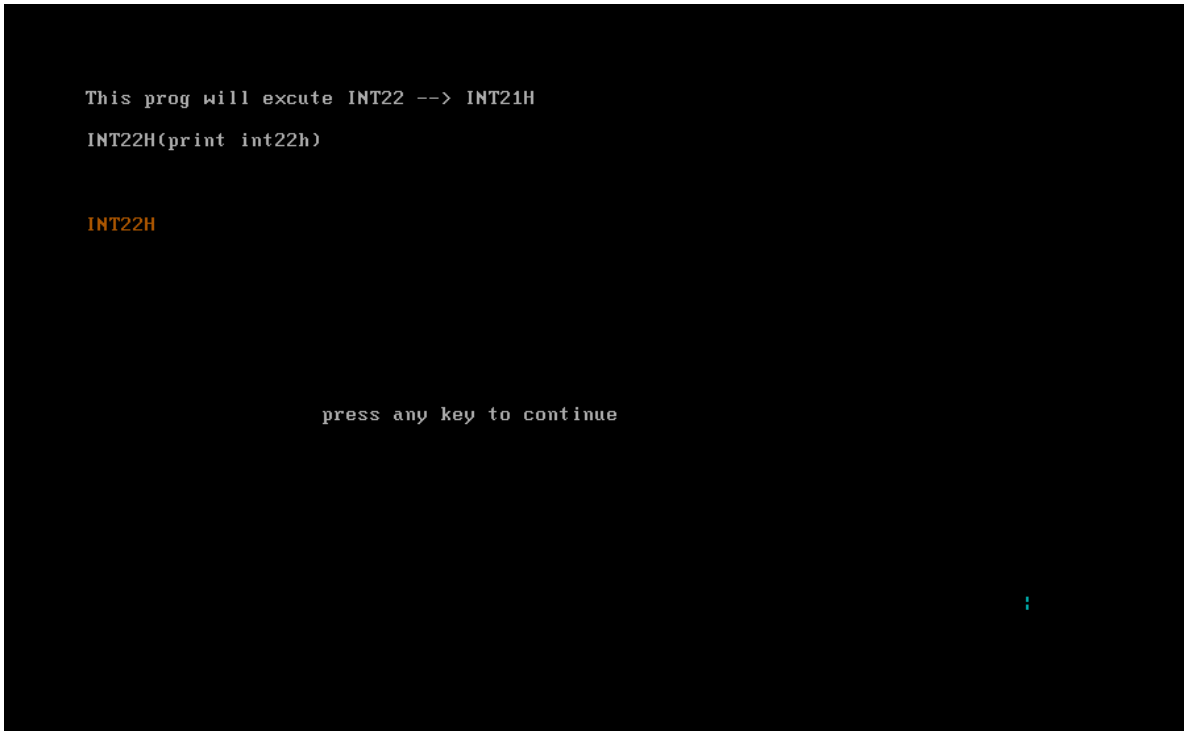
选择：



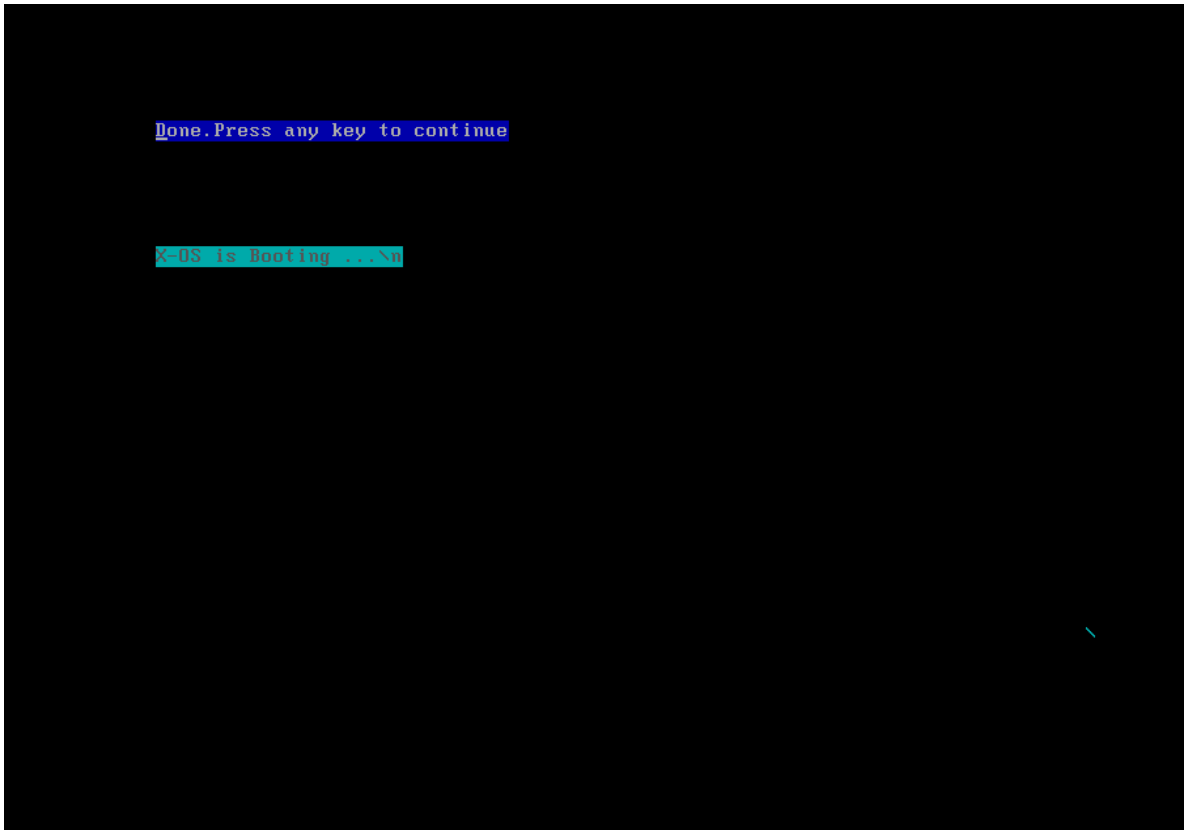
运行：



原系统调用正常：


This prog will excute INT22 --> INT21H
INT22H(print int22h)
INT22H
press any key to continue


INT21H AH=0(show Time)
2019-3-10 17:24:53
press any key to continue



附有视频 ex.mp4

纠错与反思：

1. **DS寄存器与GS寄存器的维护**

这次实验的难点在于调度过程的实现，我最开始盲目使用实验五完成的SAVE()过程与RESTART ()过程，却在调试过程中遇到了很多意外。我的操作系统在进入内核后，输入任意字符就会导致卡死，所以发现应该是_getchar()的过程中`int 16H`产生的问题，通过Bochs发现是DS寄存器与GS寄存器未维护导致。

2. **汇编loop语句中偏移地址做加法涉及的寄存器不允许进行计算，写法需要改变**

```
byte[cs:si+_STATE] ;right
byte[cs:_STATE+si] ;不允许对si运算
```

3. **bochs 调试时钟中断**

通过

```
info ivt
```

可以获取所有中断的地址，在对应地址设置断点即可。

实验总结

实验做下来最有体会就是汇编语言的逐渐熟练，bochs功能的用法逐渐熟悉。在初期摸爬滚打只能靠肉眼Debug与现在可以设置断点来观察运行到某行命令时的内存值与寄存器值简直效率太低。

同时我觉得内核实现的最重要的一个方面，是自己的栈的维护，务必要保证的是每次执行所有中断处理程序，保证栈的变化是合理的，严格记住自己的入栈出栈操作，否则debug只能在bochs中一步一步调用print-stack观测。

Reference

1. 《从实模式到保护模式》