

实验六：二态进程模型内核

19335174施天予

一、实验题目

二态进程模型内核

二、实验目的

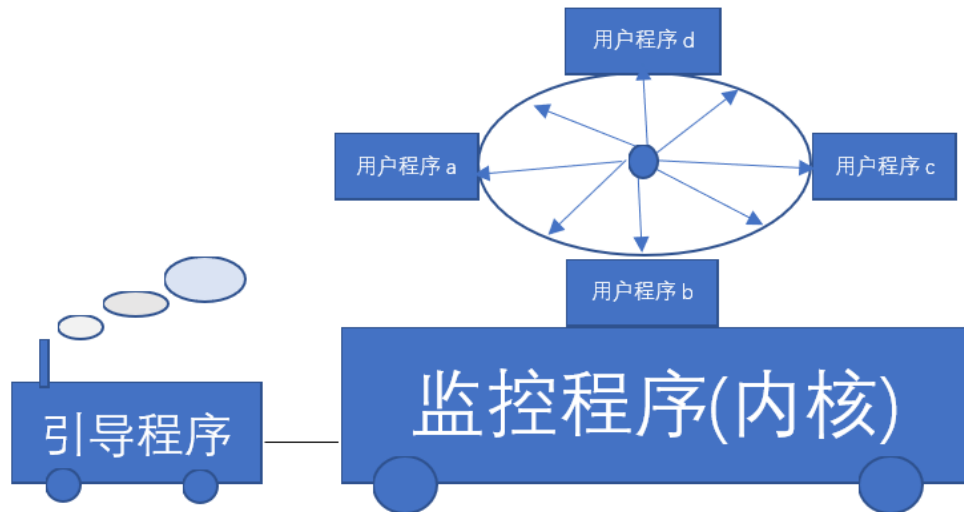
- 1、学习多道程序与CPU分时技术
- 2、掌握操作系统内核的二态进程模型设计与实现方法
- 3、掌握进程表示方法
- 4、掌握时间片轮转调度的实现

三、实验要求

- 1、了解操作系统内核的二态进程模型。
- 2、扩展实验五的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。
- 3、修改时钟中断处理程序，调用时间片轮转调度算法。
- 4、设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行。
- 5、修改save()和restart()两个汇编过程，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现场。
- 6、编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

四、实验内容

- 1、修改实验5的内核代码，内核执行期间，关中断，实现内核不可再入。
- 2、修改实验5的内核代码，定义进程控制块，包括进程号、程序名、进程内存地址信息、CPU寄存器保存区、进程状态等必要数据项。
- 3、扩展实验五的内核程序，增加一条命令可同时执行多个用户程序，内核加载这些程序，创建多个进程，实现分时运行。
- 4、修改时钟中断处理程序，保留无敌风火轮显示，而且增加调用进程调度过程。
- 5、内核增加进程调度过程:每次调度，将当前进程转入就绪状态，选择下一个进程运行，如此反复轮流运行。
- 6、修改save()和restart()两个汇编过程，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。
- 7、实验5的内核其他功能，如果不必要，可暂时取消服务。



五、实验方案

实验环境

- 1、实验运行环境：Windows10
- 2、虚拟机软件：VirtualBox和DOSBox
- 3、NASM编译器
- 4、TCC+TASM+TLINK混合编译

实验工具

- 1、编程语言：NASM, TASM, C
- 2、文本编辑器：Notepad++
- 3、软盘编辑软件：WinHex

实验思想

二状态的进程模型

1、进程模型就是实现多道程序和分时系统的一个理想的方案。多个用户程序实现并发执行。在进程模型中，操作系统可以知道有几个用户程序在内存运行，每个用户程序执行的代码和数据放在什么位置，入口位置和当前执行的指令位置，哪个用户程序可执行或不可执行，各个程序运行期间使用的计算机资源情况等等。

2、二状态进程模型分为执行和等待两种状态。目前进程的用户程序都是COM格式的，是最简单的可执行程序。进程仅涉及一个内存区、CPU、显示屏这几种资源，所以进程模型很简单，只要描述这几个资源。以后扩展进程模型解决键盘输入、进程通信、多进程、文件操作 等问题。

初级进程

1、现在的用户程序都很小，只要简单地将内存划分为多个小区，每个用户程序占用其中一个区，就相当于每个用户拥有独立的内存。

2、根据我们的硬件环境，CPU可访问1M内存，我们规定MYOS加载在第一个64K中，用户程序从第二个64K内存开始分配，每个进程64K，作为示范，我们实现的MYOS进程模型只有两个用户程序，大家可以简单地扩展，让MYOS中容纳更多的进程。

3、对于键盘，我们先放后解决，即规定用户程序没有键盘输入要求，我们将在后继的关于终端的实验中解决。

4、对于显示器，我们可以参考内存划分的方法，将25行80列的显示区划分为多个区域，在进程运行后，操作系统的显示信息是很少的，所以我们就将显示区分为4个区域，用户程序如果要显示信息，规定在其中一个区域显示。当然，理想的解决方案是用户程序分别拥有一个独立的显示器，这个方案会在后续的多终端实验中解决。

5、文件资源和其它系统软资源，则会通过扩展进程模型的数据结构来实现，相关内容将安排在文件系统实验和其它一些相关实验中解决。

进程表

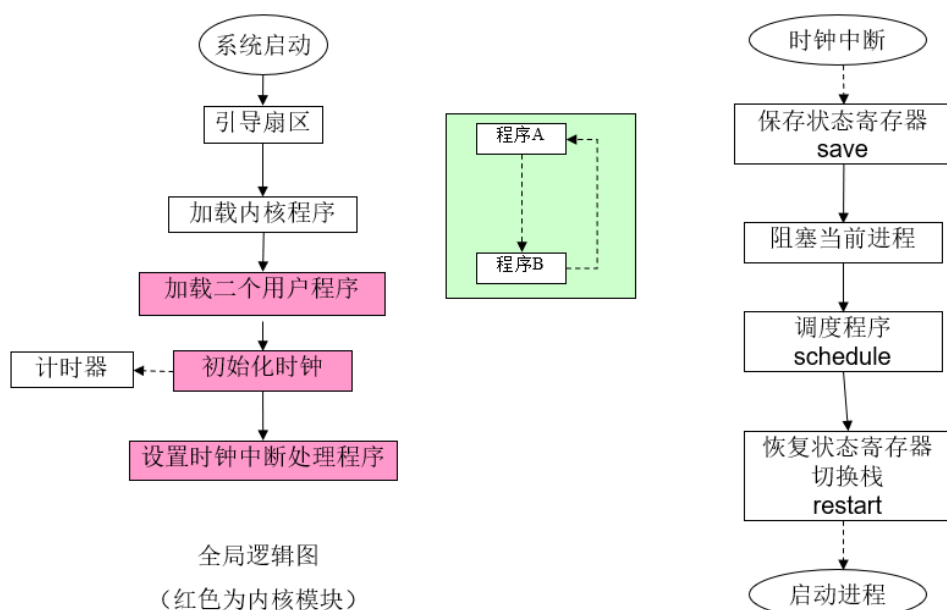
初级的进程模型可以理解为一个CPU模拟为多个逻辑独立的CPU。每个进程具有一个独立的逻辑CPU。同一计算机内并发执行多个不同的用户程序，MYOS要保证独立的用户程序之间不会互相干扰。为此，内核中建立一个重要的数据结构：进程表和进程控制块PCB。现在的PCB它包括进程标识、逻辑CPU和内存区描述。逻辑CPU包含8086CPU的所有寄存器AX、BX、CX、DX、BP、SP、DI、SI、CS、DS、ES、SS、IP、FLAG，用内存单元模拟。逻辑CPU轮流映射到物理CPU，实现多道程序的并发执行。可以使用汇编语言模块描述PCB，但在C语言模块中描述PCB更为方便。

进程交替执行原理

1、在以前的原型操作系统顺序执行用户程序，内存中不会同时有两个用户程序，所以CPU控制权交接问题简单，操作系统加载了一个用户到内存中，然后将控制权交接给用户程序，用户程序执行完再将控制权交接回操作系统，一次性完成用户程序的执行过程。

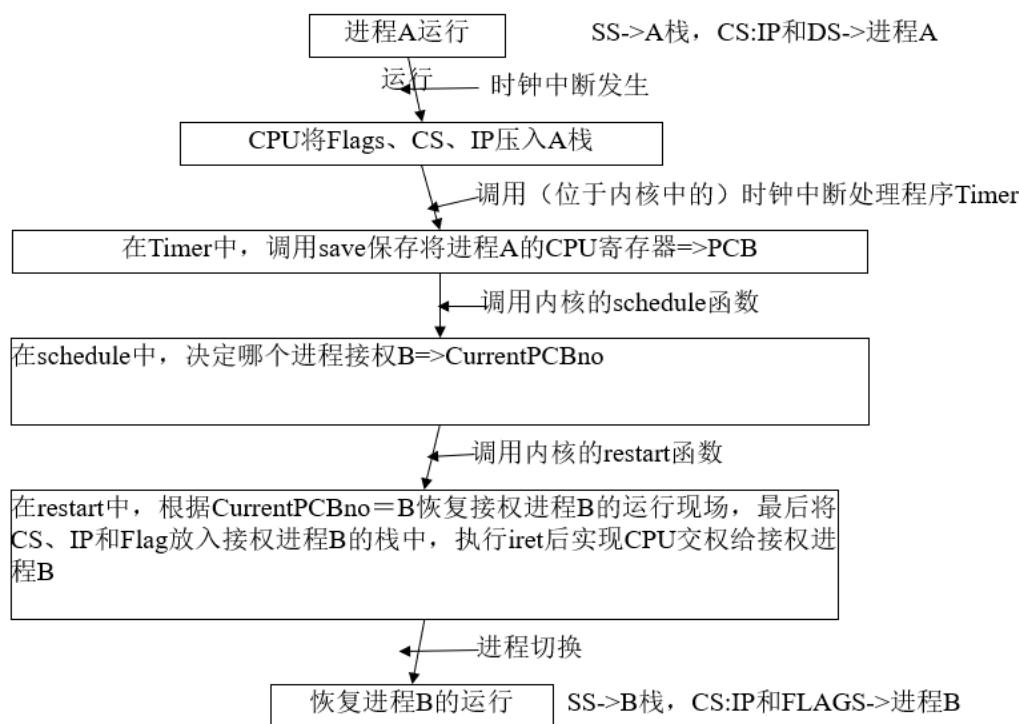
2、采用时钟中断打断执行中的用户程序实现CPU在进程之间交替。

3、简单起见，我们让两个用户的程序均匀地推进，就可以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。



内核

- 1、利用时钟中断实现用户程序轮流执行。
- 2、在系统启动时，将加载两个用户程序A和B，并建立相应的PCB。
- 3、修改时钟中断服务程序。每次发生时钟中断，中断服务程序就让A换B或B换A。要知道中断发生时谁在执行，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。



save过程

- 1、save是一个非常关键的过程，保护现场不能有丝毫差错，否则再次运行被中断的进程可能出错。
- 2、涉及到二种不同的栈：应用程序栈、进程表栈、内核栈。其中的进程表栈，只是我们为了保存和恢复进程的上下文寄存器值，而临时设置的一个伪局部栈，不是正常的程序栈。
- 3、在时钟中断发生时，实模式下的CPU会将FLAGS、CS、IP先后压入当前被中断程序（进程）的堆栈中，接着跳转到（位于kernel内）时钟中断处理程序（Timer函数）执行。注意，此时并没有改变堆栈（的SS和SP），换句话说，我们内核里的中断处理函数，在刚开始时，使用的是被中断进程的堆栈。
- 4、为了及时保护中断现场，必须在中断处理函数的最开始处，立即保存被中断程序的所有上下文寄存器中的当前值。不能先进行栈切换，再来保存寄存器。因为切换栈所需的若干指令，会破坏寄存器的当前值。这正是我们在中断处理函数的开始处，安排代码保存寄存器的内容。
- 5、我们PCB中的16个寄存器值，内核一个专门的程序save，负责保护被中断的进程的现场，将这些寄存器的值转移至当前进程的PCB中。

restart过程

- 1、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。这里面最棘手的问题是SS的切换。
- 2、使用标准的中断返回指令IRET和原进程的栈，可以恢复（出栈）IP、CS和FLAGS，并返回到被中断的原进程执行，不需要进行栈切换。

3、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。

4、解决办法有三个，一个是所有程序，包括内核和各个应用程序进程，都使用共同的栈。即它们共享一个（大栈段）SS，但是可以有各自不同区段的SP，可以做到互不干扰，也能够用IRET进行进程切换。第二种方法，是不使用IRET指令，而是改用RETF指令，但必须自己恢复FLAGS和SS。第三种方法，使用IRET指令，在用户进程的栈中保存IP、CS和FLAGS，但必须将IP、CS和FLAGS 放回用户进程栈中，这也是我们程序所采用的方案。

模块结构

扇区号	扇区数（大小）	内容
1	1 (512 B)	引导程序
2	1 (512 B)	用户程序1
3	1 (512 B)	用户程序2
4	1 (512 B)	用户程序3
5	1 (512 B)	用户程序4
6~7	2 (893 B)	混合编译用户程序
8~14	7 (3474 B)	操作系统内核

命令功能

命令	功能
help	显示支持的命令及其功能
cls	清屏
run-	并发运行任意数量的用户程序，如“run-1234”
int22h	显示“INT22H”
test	运行混合编译用户程序（输入输出）
ls	显示文件列表
reboot	重启
poweroff	退出操作系统并关机

六、实验过程

PCB数据结构的设计

切换进程时，把当前进程的cpu状态保存到进程控制块PCB里面。内核中有一个PCB数组(5个元素，4个用户进程，1个系统进程)，用一个指针指向当前正在运行的进程控制块。

```
typedef struct {
    int ax;
    int bx;
    int cx;
    int dx;
    int di;
    int bp;
    int es;
    int ds;
    int si;
    int ss;
    int sp;
    int ip;
    int cs;
    int flags;
}cpuRegisters;

typedef struct {
    cpuRegisters cpu;
    int pid;
    char name[10];
    int state; /*三种状态: 0进程空闲, 1进程运行, 2进程阻塞*/
}PCB;

PCB pcbList[5]; /*创建一个最多有5个进程的PCB列表, 最后一个为内核进程*/
PCB *curPCB = &pcbList[4]; /*指向当前运行的PCB块*/
PCB *kerPCB = &pcbList[4];
int NumOfProcess = 0;
int MaxNumOfProcess = 5;
```

save与restart的改进

save和restart的实现与实验5大同小异，主要是数据结构改为使用PCB。

save函数用来中断处理时保护现场。操作系统进入中断后，立即调用save把当前状态下cpu所有寄存器，标志寄存器。中断返回点都保存到内存的0-1023某处。并且save结束后，es、ds、cs、ss寄存器都是内核的，即进入了内核空间。

```
public _save
_save proc
    push ds
    push cs
    pop ds
    push si

    mov si,word ptr [_curPCB] ; 保存_curPCB的内容
    pop word ptr [si+16] ; 保存si
    pop word ptr [si+14] ; 保存ds

    lea si,ret_save ; 保存_save的返回点
    pop word ptr [si]
```

```

    mov si,word ptr [_curPCB]
    pop word ptr [si+22]      ; 保存ip
    pop word ptr [si+24]      ; 保存cs
    pop word ptr [si+26]      ; 保存flags

    mov [si+18],ss
    mov [si+20],sp

    mov si,ds
    mov ss,si

    mov sp,word ptr [_curPCB]
    add sp,14

    push es
    push bp
    push di
    push dx
    push cx
    push bx
    push ax

    mov si,word ptr [_kerPCB]
    mov sp,[si+20]
    mov ax,cs
    mov es,ax

    lea si,ret_save
    mov ax,[si]
    jmp ax
_save endp

```

restart函数用于在中断响应结束前恢复现场，末尾要用iret返回。

```

public _restart
_restart proc
    mov si,word ptr [_kerPCB]
    mov [si+20],sp
    mov sp,word ptr [_curPCB]
    pop ax
    pop bx
    pop cx
    pop dx
    pop di
    pop bp
    pop es

    lea si,ds_save
    pop word ptr [si]
    lea si,si_save
    pop word ptr [si]

    lea si,bx_save
    mov [si],bx      ; 保护bx，用它给ss赋值

    pop bx            ; 恢复栈

```

```

mov ss,bx
mov bx,sp
mov sp,[bx]

add bx,2
push word ptr [bx+4]      ; flags,cs,ip压栈
push word ptr [bx+2]
push word ptr [bx]

push ax
push word ptr [si]        ; 把bx压栈保存
lea si,ds_save
mov ax,[si]
lea si,si_save
mov bx,[si]
mov ds,ax
mov si,bx

pop bx
pop ax
iret
_restart endp

```

多进程框架

初始化PCB

InitPCB()函数用于初始化PCB列表。操作系统启动时将所有进程设置为“空闲”状态，并且设置内核进程的状态为运行态。

```

/*初始化PCB列表*/
void InitPCB() {
    int i;
    for (i = 0; i < MaxNumOfProcess; i++)
        pcbList[i].state = 0;
    kerPCB->state = 1;
    kerPCB->pid = MaxNumOfProcess - 1;
    strcpy(kerPCB->name, "kernel");
}

```

进程调度

schedule()函数用于进程调度，由时钟中断响应调用。schedule()首先判断用户进程的个数，如果为0，就直接切换到内核进程。如果不为0，则从PCB数组里找到下一个阻塞的进程，然后切换。

```

/*进程调度*/
void schedule() {
    int i = (curPCB->pid + 1) % MaxNumOfProcess;
    if (NumOfProcess == 0) {
        curPCB = &pcbList[MaxNumOfProcess - 1];
        curPCB->state = 1;
        return;
    }
    while (i != curPCB->pid) {
        if (pcbList[i].state == 2) {
            curPCB->state = 2;

```



```

        pcbList[i].state = 1;
        curPCB = &pcbList[i];
        return;
    }
    i = (i + 1) % MaxNumOfProcess;
}
}

```

进程创建

fork()函数用于创建进程。要创建进程，首先要找到一个空闲的PCB块，然后对各个寄存器的值初始化，并且从软盘中加载程序到内存指定位置运行，最后把PCB块的state置为2(阻塞)，等待时间片结束切换运行。

```

/*进程创建，成功返回进程号pid，否则返回-1*/
int fork(int cs, int ip, int id, int num) {
    int i;
    for (i = 0; i < MaxNumOfProcess; i++)/*寻找空闲的PCB*/
        if (pcbList[i].state == 0)
            break;
    if (i == MaxNumOfProcess) {/*没有空闲PCB，进程创建失败*/
        printf("Fork failed\r\n");
        return 0;
    }
    pcbList[i].cpu.cs = cs;
    pcbList[i].cpu.ds = cs;
    pcbList[i].cpu.es = cs;
    pcbList[i].cpu.ip = ip;
    pcbList[i].cpu.ss = cs;
    pcbList[i].cpu.sp = InitStack(cs);/*初始化进程栈*/
    pcbList[i].cpu.flags = 512;
    pcbList[i].pid = i;
    pcbList[i].name[0] = '0' + i;
    pcbList[i].name[1] = 0;
    loadReady(cs);/*加载代码段前缀到指定的段*/
    loadProgram(cs, ip, id, num);/*加载用户程序到内存指定位置*/
    pcbList[i].state = 2;
    NumOfProcess += 1;
    return i;
}

```

在fork()函数中调用了汇编代码部分的几个函数。

InitStack()函数用于创建用户进程时，初始化用户进程的栈，即压一个0x0000进去。用于用户进程结束后返回代码前缀段以返回操作系统。调用时，有一个参数，即用户进程所在的段地址(ss)，并且返回栈顶位置(sp)。

```

; 初始化用户进程的栈
public _InitStack
_InitStack proc near
    push bp
    mov bp,sp
    push bx
    mov ax,[bp+4]          ; 获取用户程序段地址
    mov ss,ax
    mov sp,0

```

```

    xor ax,ax
    push ax                ; 用户栈压0x0000
    mov ax,sp
    mov bx,cs
    mov ss,bx              ; 恢复到内核栈
    sub bp,2
    mov sp,bp
    pop bx
    pop bp
    ret
_InitStack endp

```

loadReady()函数用于加载代码段前缀到指定的段，便于用户程序结束后返回内核对应位置。

```

; 加载代码段前缀到指定的段
public _loadReady
_loadReady proc
    push bp
    mov bp,sp

    push cx
    push es
    push si
    push di

    mov ax,[bp+4]
    mov es,ax              ; 段地址
    xor di,di
    mov ax,cs
    mov ds,ax
    lea si,testbegin
    lea cx,testend
    sub cx,si              ; 循环次数
    rep movsb              ; 循环写

    pop di
    pop si
    pop es
    pop cx
    pop bp
    ret

testbegin:
    int 20h                ; 用int 20h返回
testend:nop
_loadReady endp

```

loadProgram()函数用于加载用户程序到指定位置。与之前跳转用户程序不同的是，原来的用户程序单个执行，所以段地址可以相同，但现在我们要实现进程并发执行，所以要设定不同的段地址。我给这个函数设定了4个参数：cs段地址，ip偏移地址，id起始扇区号，num扇区数，这样跳转功能更为强大实用。

```

; 加载用户程序 4个参数(cs段地址, ip偏移地址, id起始扇区号, num扇区数)
public _loadProgram
_loadProgram proc
    push bp

```

```

mov bp,sp

push ax
push bx
push cx
push dx
push es

mov ax,[bp+4]          ; 段地址, 存放数据的内存基地址
mov es,ax              ; 设置段地址 (不能直接mov es,段地址)
mov bx,[bp+6]          ; 偏移地址; 存放数据的内存偏移地址
mov ah,2               ; 功能号2
mov al,[bp+10]         ; 扇区数
mov dl,0               ; 驱动器号 ; 软盘为0, 硬盘和U盘为80H
mov dh,0               ; 磁头号, 起始编号为0
mov ch,0               ; 柱面号, 起始编号为0
mov cl,[bp+8]          ; 起始扇区号 ; 起始编号为1
int 13H                ; 调用读磁盘BIOS的13h功能
;用户程序已加载到指定内存区域

pop es
pop dx
pop cx
pop bx
pop ax
pop bp
ret
_loadProgram endp

```

进程结束

endProcess()函数用于结束进程。首先把要结束进程的PCB块的state置为0(空闲), 并且把进程的个数减一, 然后开中断, 让cpu可以响应时钟中断, 最后用while(1)进行死循环阻塞, 直到一个时间片结束。endProcess由int 20h中断调用。

```

/*结束进程*/
void endProcess() {
    NumOfProcess--;
    curPCB->state = 0;
    setIF();/*开中断*/
    while (1);/*阻塞至一个时间片结束*/
}

```

修改中断向量表

与之前实验相同, 我们在内核汇编部分修改4个中断向量。

```

; 修改中断向量表
xor ax, ax
mov es, ax
mov word ptr es:[8*4], offset Timer
mov word ptr es:[8*4+2], cs

mov word ptr es:[32*4], offset int20h
mov word ptr es:[32*4+2], cs

```

```

mov word ptr es:[33*4], offset int21h
mov word ptr es:[33*4+2], cs

mov word ptr es:[34*4], offset int22hprint
mov word ptr es:[34*4+2], cs

```

时钟中断

与之前实验大体没有区别。保留了风火轮的设计，并在这里调用schedule()实现进程调度。我在时钟中断中加入了两个新函数time和date，用于实时显示时间和日期，这两个新功能在后面会有具体描述。

```

; 时钟中断处理程序
Timer:
    call _save                ; save保护中断现场

    lea bx, count             ; 判断count值是否为0，为0显示下一个"风火轮"
    mov al, [bx]
    dec al
    mov [bx], al
    cmp al, 0
    jnz exit
    mov al, Delay
    mov [bx], al
    lea bx, wheel
    mov al, [bx]
    xor ah, ah
    inc ax
    cmp ax, 4
    jnz next
    mov ax, 1
next:
    mov [bx], al
    add bx, ax                ; [bx]是要显示的字符，加上偏移量获取
    push es
    mov ax, 0b800h
    mov es, ax
    mov al, [bx]
    mov ah, 0Fh               ; 黑底白字
    mov bx, ((24*80+79)*2)    ; 24行79列
    mov es:[bx], ax
    pop es
exit:
    call _date                ; 显示日期
    call _time                ; 显示时间
    call near ptr _schedule   ; 进程调度

    mov al, 20h               ; AL = EOI
    out 20h, al               ; 发送EOI到主8529A
    out 0A0h, al              ; 发送EOI到从8529A

    jmp _restart               ; restart恢复现场

Delay equ 5
count label byte
    db Delay
wheel label byte
    db 1

```

```
db '|'
db '/'
db '\'
```

INT 20H

从用户程序返回内核程序。要先将之前保存的内核PCB取出，然后用endProcess()函数返回内核。

```
; int20h 用户程序返回
int20h:
    mov ax,cs
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov si,word ptr [_kerPCB] ; 保存到内核PCB
    mov sp,[si+20]
    call near ptr _endProcess ; 结束进程
```

内核函数修改

这是内核部分RunCom()函数的修改，用于执行相应的命令。我们将所有用户程序的调用都用fork()函数实现，设定段地址和偏移地址，加载到内存指定位置运行。我这里设计了一个任意数量任意用户程序都能并发运行的功能，既可运行单个用户程序，也可多个用户程序并发运行。如“run-3”运行单个用户程序3，“run-14”并发运行用户程序1和4，“run-123”并发运行用户程序1、2和3，“run-1234”并发运行全部4个用户程序。需要注意的是，run-命令因为是让用户程序并发执行，所以要给每个用户程序设定不同的段地址。最后要在所有进程结束前用一个while循环阻塞，实现同步。

```
void RunCom(char *command) {
    .....
    else if(command[0]=='r'&&command[1]=='u'&&command[2]=='n'&&command[3]=='-')
    {
        while (command[i] == '1' || command[i] == '2' || command[i] == '3' ||
command[i] == '4') {
            if (command[i] == '1') fork(0x2000, 0x100, 2, 1);
            if (command[i] == '2') fork(0x3000, 0x100, 3, 1);
            if (command[i] == '3') fork(0x4000, 0x100, 4, 1);
            if (command[i] == '4') fork(0x5000, 0x100, 5, 1);
            i++;
        }
    }
    .....
    while (NumOfProcess != 0);/*阻塞，直至进程都完成*/
}
```

新功能：实时显示时间和日期

在看老师的参考代码时我发现了显示时间和日期这个有趣的功能。但参考代码将这个功能放入内核主程序中，只是输入相应命令时调用显示一下，并且没有秒钟部分，于是我上网搜集资料将这个功能进行了改进，实现了秒钟的部分，并且把它放入时钟中断中，这样就可以实时显示时间和日期了。

time函数用于显示时间。int 1Ah中断的02h号功能可以读取时间，ch读取时钟，cl读取分钟，dh读取秒钟。将每个字符一位位读取，放入到一个字符串中。最后用int 10h中断的13h号功能输出字符串，即可显示时间。

```
public _time
```

```

_time proc
    push ds
    push es

    mov ax,cs
    mov ds,ax          ; DS = CS
    mov ax,cs
    mov es,ax          ; ES = CS
    mov di,offset _ttime

    mov ah,02h          ; 读取时钟
    int 1Ah
    mov al,ch
    mov ah,0
    mov bl,16
    div bl
    add al,30h
    mov [di],al

    mov ah,02h
    int 1Ah
    mov al,ch
    and al,0fh
    add al,30h
    mov [di+1],al
    mov byte ptr [di+2],':'

    mov ah,02h          ; 读取分钟
    int 1Ah
    mov al,cl
    mov ah,0
    mov bl,16
    div bl
    add al,30h
    mov [di+3],al

    mov ah,02h
    int 1Ah
    mov al,cl
    and al,0fh
    add al,30h
    mov [di+4],al
    mov byte ptr [di+5],':'

    mov ah,02h          ; 读取秒钟
    int 1Ah
    mov al,dh
    mov ah,0
    mov bl,16
    div bl
    add al,30h
    mov [di+6],al

    mov ah,02h
    int 1Ah
    mov al,dh
    and al,0fh
    add al,30h

```

```

mov [di+7],al
mov byte ptr [di+8],' '

mov ax,cs
mov ds,ax          ; DS = CS
mov es,ax          ; ES = CS
mov bp,offset _ttime ; BP=当前串的偏移地址
mov ax,ds          ; ES:BP = 串地址
mov es,ax          ; 置ES=DS
mov cx,9           ; CX = 串长 (=9)
mov ax,1300h       ; AH = 13h (功能号)、AL = 00h (光标置于起始位置)
mov bx,0007h       ; 页号为0(BH = 0) 黑底白字(BL = 07h)
mov dh,24          ; 行号=24
mov dl,70          ; 列号=70
int 10h            ; BIOS的10h功能: 显示一行字符

pop es
pop ds
ret
_time endp

```

date函数用于显示日期。int 1Ah中断的04h号功能可以读取日期，ch读取世纪，cl读取年份，dh读取月份，dl读取天数。将每个字符一位位读取，放入到一个字符串中。最后用int 10h中断的13h号功能输出字符串，即可显示日期。

```

public _date
_date proc
    push ds
    push es

    mov ax,cs
    mov ds,ax          ; DS = CS
    mov ax,cs
    mov es,ax          ; ES = CS
    mov si,offset _ddate

    mov ah,04h         ; 读取世纪
    int 1Ah
    mov al,ch
    mov ah,0
    mov bl,16
    div bl
    add al,30h
    mov [si],al

    mov ah,04h
    int 1Ah
    mov al,ch
    and al,0fh
    add al,30h
    mov [si+1],al

    mov ah,04h         ; 读取年份
    int 1Ah
    mov al,cl
    mov ah,0
    mov bl,16

```

```

div bl
add al,30h
mov [si+2],al

mov ah,04h
int 1Ah
mov al,cl
and al,0fh
add al,30h
mov [si+3],al
mov byte ptr [si+4], '.'

mov ah,04h           ; 读取月份
int 1Ah
mov al,dh
mov ah,0
mov bl,16
div bl
add al,30h
mov [si+5],al

mov ah,04h
int 1Ah
mov al,dh
and al,0fh
add al,30h
mov [si+6],al
mov byte ptr [si+7], '.'

mov ah,04h           ; 读取天数
int 1Ah
mov al,d1
mov ah,0
mov bl,16
div bl
add al,30h
mov [si+8],al

mov ah,04h
int 1Ah
mov al,d1
and al,0fh
add al,30h
mov [si+9],al
mov byte ptr [si+10], ' '

mov ax,cs
mov ds,ax           ; DS = CS
mov es,ax           ; ES = CS
mov bp,offset _ddate ; BP=当前串的偏移地址
mov ax,ds           ; ES:BP = 串地址
mov es,ax           ; 置ES=DS
mov cx,11           ; CX = 串长(=11)
mov ax,1300h        ; AH = 13h (功能号)、AL = 00h (光标置于起始位置)
mov bx,0007h        ; 页号为0(BH = 0) 黑底白字(BL = 07h)
mov dh,24           ; 行号=24
mov dl,59           ; 列号=59
int 10h             ; BIOS的10h功能: 显示一行字符

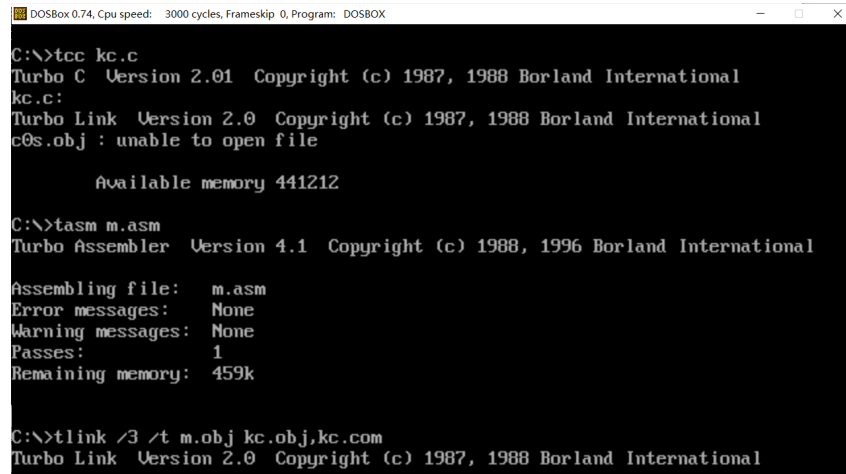
```



```
pop es
pop ds
ret
_date endp
```

七、实验结果

内核混合编译



```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

C:\>tcc kc.c
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
kc.c:
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
c0s.obj: unable to open file


        Available memory 441212

C:\>tasm m.asm
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file:    m.asm
Error messages:     None
Warning messages:   None
Passes:             1
Remaining memory:   459k

C:\>tlink /3 /t m.obj kc.obj,kc.com
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
```

操作系统界面



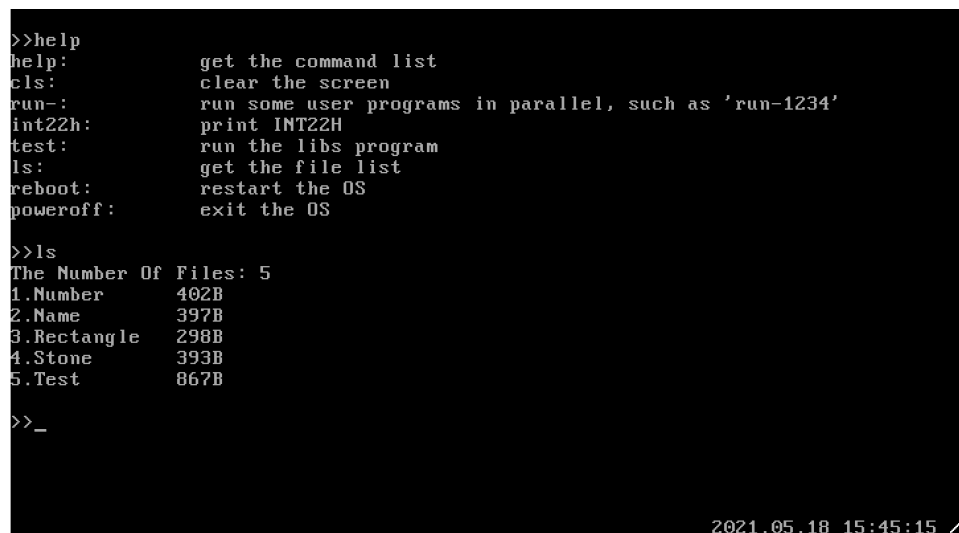
```
*****
*                                     *
*                               Welcome to my OS!                             *
*                                     *
*                               STY 19335174                                   *
*                                     *
*                               Enjoy yourself!                               *
*                                     *
*****

Enter help to get the command list

>>_

2021.05.14 10:55:47 ✓
```

帮助菜单和文件列表



```
>>help
help:      get the command list
cls:       clear the screen
run-:      run some user programs in parallel, such as 'run-1234'
int22h:    print INT22H
test:      run the libs program
ls:        get the file list
reboot:    restart the OS
poweroff:  exit the OS

>>ls
The Number Of Files: 5
1.Number    402B
2.Name      397B
3.Rectangle 298B
4.Stone     393B
5.Test     867B

>>_

2021.05.18 15:45:15 ✓
```

用户程序1

```
193351193319335174      193319335174
193319331933517474  19335119335174
1919331933517451741933517419335174
193319335174335174335174  19335174
193319335174193351745174  19335174
193319335174  1933517474  19335174
19331933517474  19335174  19335174
1933193351745174191933517419335174
1933193351743351193319335174335174
1933193351741933193351193351745174
1919335174  19193351741933517474
19335174      19335174  19335174
```

2021.05.14 10:57:09 ✓

用户程序2

```
>>
```

```
STSTY      STYTY STSTY
STSTY      STYTYTYTSTSTY
STYTYSTY   STYTYSTYTYSTY
STY STSTY  STYTY STYTY STSTY
STY STSTY  STYTY STSTYTY STSTY
STY STSTY STYTY STSTSTYTY STSTY
STY STSTSTYTY STSTY STYTY STSTY
STY STSTYTY STSTY STYTY STSTY
STY STYTY STSTY STYTY STY
STSTYTYSTY STYTYTY
STYTYTSTY  STYTYTY
STY STSTY  STYTY
```

2021.05.14 10:58:29 ✓

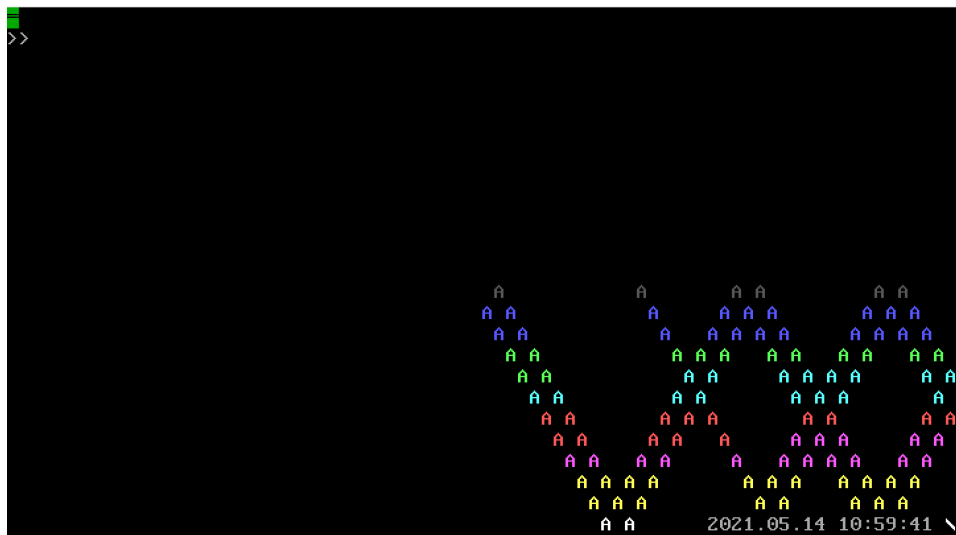
用户程序3

```
>>
```

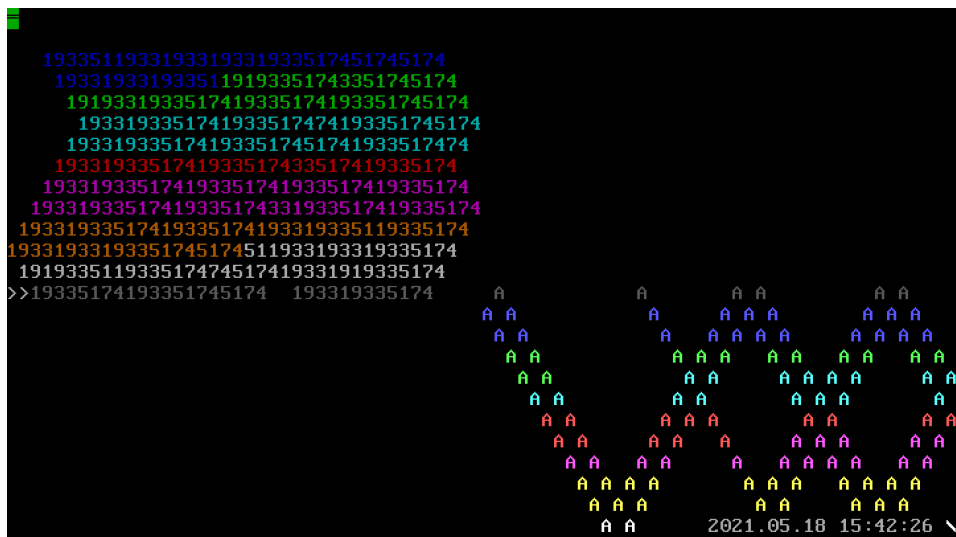
```
19335174STY
```

2021.05.14 10:59:02 ✓

用户程序4



2个用户程序并行



3个用户程序并行



4个用户程序并行

```
1933511933193319331933517451745174
1933193319335119193351743351745174 A
19193319335174193351741STY51745174 A
19331933517419335174741STY51745174A
>> 1933193351741933517451741STY517474 A
193319335174193351743351741STY5174 A
19331933517419335174193351741STY5174 A
1933193351741933517433193351741STY5174
193319335174193351741933193351193STY74 A
19331933193351745174511933193319335STY A
19193351193351747451741933191933517STY
19335STY193351745174 S193319335174 STY A
STY STYTY STSTY STSTY STYTY STY A A A A A A
STSTY STY STY STY STY STSTY STY STY A A A A A A
STY STSTY STYTY STY STSTY A A A A A A A A
STYTY STY STY STYTY STY A A A A A A A A
STY STY STYTY STSTY STY STY STYTY A A A A A A
STY STSTY STY STY STYTY STSTY STY A A A A A A
STY STY STYTY STY STY STY STY A A A A A A
STY STY STY STSTY STY STY STY STY A A A A A A
STSTY STSTY STYTY STSTY STSTY A A A A A A
STY STY STY STY STY A A 2021.05.18 18:54:19 \
```

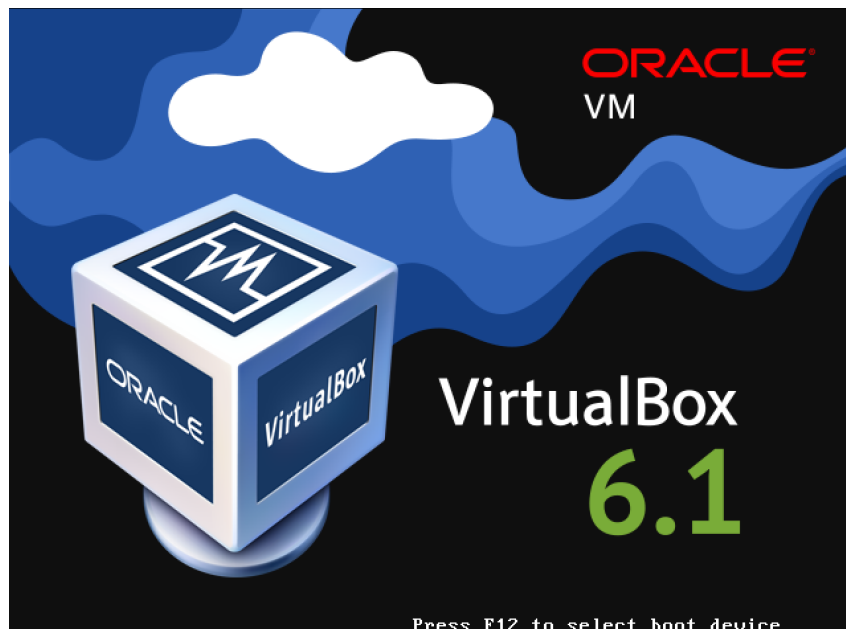
用户程序5 (libs program)

```
ch=a
str=shitianyu
a=7
a
shitianyu
ch=a,a=7,str=shitianyu
>>
2021.05.14 11:00:35 i
```

INT 22H

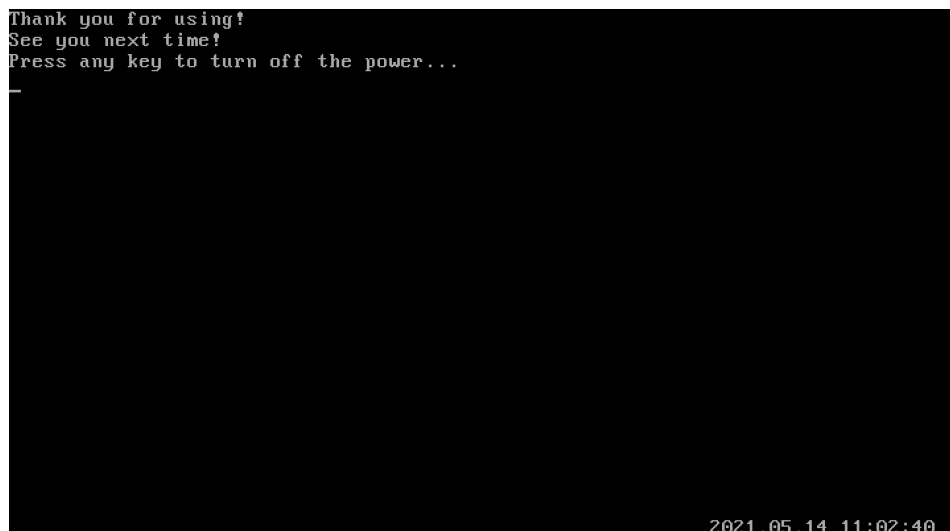
```
>>int22h
>>_
INT22H
2021.05.14 11:01:03 ✓
```

重新启动 (int 21h的4号功能)



关闭电源 (int 21h的3号功能)

按下任意按键后关闭电源!



八、实验总结

这次实验相比上次实验要轻松一些，因为艰难的save和restart已经写好了，只要稍作修改就能使用，但因为要构建整个多进程的框架，所以我也遇到了很多问题，导致我花费了大量时间。

实现多进程的思路十分清晰：save过程负责将所有寄存器的值保存到存储器中的PCB内，schedule过程负责选择下一个就绪进程，restart过程负责把被调度的下一个进程 PCB 中的值恢复到对应的寄存器中，最后返回到被调度进程即可。虽然这些看起来很简单，但实际操作起来其实并不容易。由于寄存器的保护、恢复必须非常严谨，不能出现任何差错。

进程的调度我一开始想不好如何实现。我开始太局限于用进程的状态来切换进程，想用进程编号和进程控制块的数组下标对应来实现，却总是在调度时出错。后来我想了可以用一个while循环找到阻塞队列中的第一个进程，将它设为运行态，并把之前进程从运行态改为阻塞态，修改curPCB为当前运行的进程，就能实现并发执行。需要注意的是，从汇编中调用C的函数，要使用call near ptr _schedule，我一开始傻乎乎直接call _schedule，结果虚拟机直接死机，风火轮都消失了orz。

好不容易终于实现了多进程并发执行，我却又发现我在用户程序结束后无法正常返回内核了。这里也有许多要注意的点。首先，要在内核主程序中添加一个while (NumOfProcess != 0)的循环进行阻塞，因为进程的结束有先后之分，这样才能实现进程同步，不会出错。其次，就是endProcess中也要添加一个while(1)循环，等待一个时间片结束。最重要的是，要在把用户程序加载到内核指定位置前先保存好内

核中的寄存器，这样在使用int 20h时才能正确返回内核对应的位置。

后来我发现我的程序又出现了一个奇怪的错误。当我执行完“run-1234”后，再执行一遍“run-1234”，按道理应该是能继续实现4个用户程序并行的，但是此时却只有用户程序1在执行。并且显示“Fork failed!”即说明我当时没有空闲的进程块！然而我之前每个用户程序都能正常退出，按理说应该没有问题。后来我只好在内核主函数里加了一个将所有PCB的state清零的for循环，终于成功完成了！

但这次实验最后我还有一个小bug没有解决。就是多进程并行的时候，如果我的用户程序3输入的顺序不是最后一个输入的，那么字符串“19335174STY”就不会显示。比如“run-1243”能正常显示，“run-1234”就不能正常显示。我检查了好久也不知道是什么原因，不过还好这个小bug也不影响最终的结果...

本次实验我新加了功能实时显示时间和日期。这个功能其实我在实验4时就想添加，当时在阅读《X86汇编语言：从实模式到保护模式》时我参照了它的代码，在DOSBox中正确实现了，但在虚拟机中去无法显示。终于，在这次实验中我参考了老师的代码，发现确实比我之前写得精妙许多。于是我取其精华，再补充修改，终于实现了这个功能，也算是圆梦了~

俗话说得好，“细节决定成败。”虽然本次实验整体的思路十分清晰，但到一个个小细节如何实现时却是非常复杂而困难的，特别是对栈的操作，需要格外小心谨慎。我也在一次次挫折与失败中学习，从而不断收获成长。老师说操作系统的很多原理与生活息息相关，我也觉得确实如此，这也让我更加体会到操作系统的精妙之处。

总而言之，这次实验花费了我很多时间，但也让我学到了很多知识，受益匪浅。多进程的框架已经建立，后面的实验也会建立在这个基础上越来越难。希望我也能迎难而上，再接再厉，勇往直前！

九、参考文献

《汇编语言（第3版）》

《X86汇编语言：从实模式到保护模式》

<https://blog.csdn.net/zang141588761/article/details/52325106>

<https://wenku.baidu.com/view/6e5e3767f5335a8102d22077.html>

<https://blog.csdn.net/q6475005/article/details/8202775>