《操作系统原理实验》

实验报告

（实验六）

学 院 名 称 ： 数据科学与计算机学院

专业（班级）： 16 计科 2 班

学 生 姓 名 ： 朱志儒

学 号 ： 16337341

时 间 ： 2018 年 4 月 26 日

实 验 六 ： 进程模型

一． 实验目的

1、 在内核实现多进程的二状态,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的基础。

二． 实验要求

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． 实验方案

1、 虚拟机配置

使用Vmware Workstation配置虚拟机，虚拟机的配置：核心数为1的处理器、4MB的内存、10MB的磁盘、1.44MB的软盘。

2、 软件工具与作用

Notepad++：编写程序时使用的编辑器；

16位编辑器WinHex：可以以16进制的方式打开并编辑任意文件；

TAMS汇编工具：可以将汇编代码编译成对应的二进制代码；

NAMS汇编工具：可以将汇编代码编译成对应的二进制代码；

TCC编译器：可以将c代码编译成对应的二进制代码；

TLINK链接器：将多个.obj文件链接成.com文件；

Bochs：可调试操作系统

WinImage：可以创建虚拟软盘。

3、 基础原理

（1） 二状态的进程模型

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，就是实现多个用户程序并发执行。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。

二状态进程模型有两个状态，即执行态和等待态。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。

（2） 初级进程

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。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。

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。

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，我们就将显示区分为4个区域。如果用户程序要显示信息，就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，这个方案会在关于终端的实验中提供。

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。

（3） 进程表

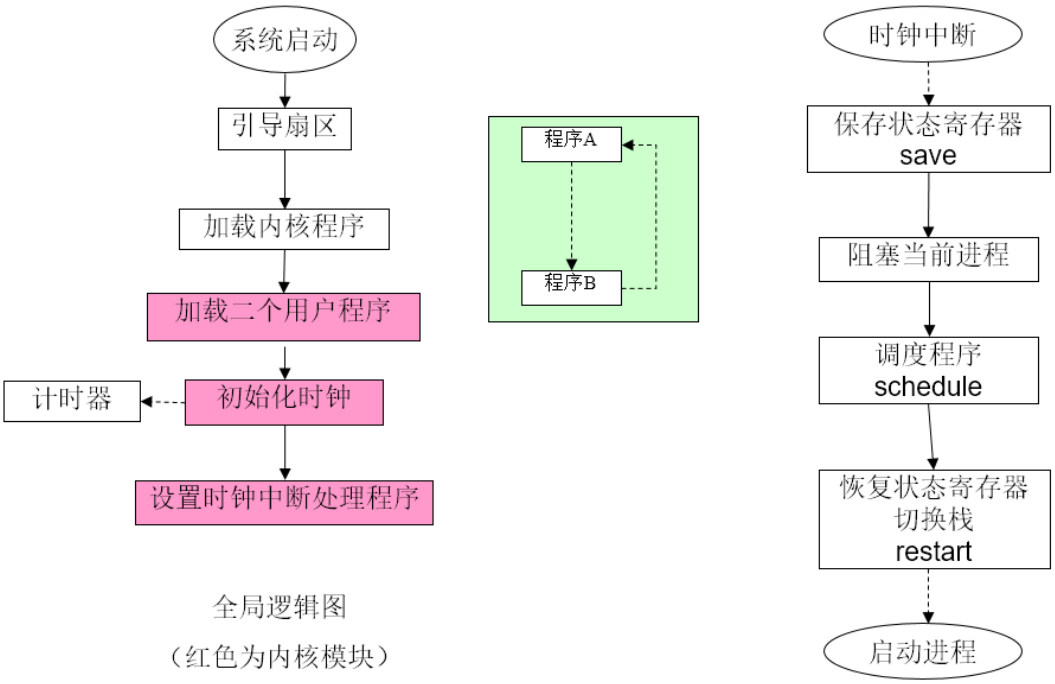
初级的进程模型可以理解为将一个CPU模拟为多个逻辑独立的CPU。每个进程具有一个独立的逻辑CPU。同一计算机内并发执行多个不同的用户程序，MYOS要保证独立的用户程序之间不会互相干扰。为此，内核中建立一个重要的数据结构：进程表和进程控制块PCB。现在的PCB包括进程标识和逻辑CPU模拟。逻辑CPU中包含8086CPU的所有寄存器，即AX、BX、CX、DX、BP、SP、DI、SI、CS、DS、ES、SS、IP、FLAG，这些用内存单元模拟。逻辑CPU轮流映射到物理CPU，实现多道程序的并发执行。可选择在汇编语言或是C语言中描述PCB。

（4） 进程交替执行原理

在以前的原型操作系统中顺序执行用户程序，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。而在本次实验中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。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。

（5） 实现进程模型的系统框架



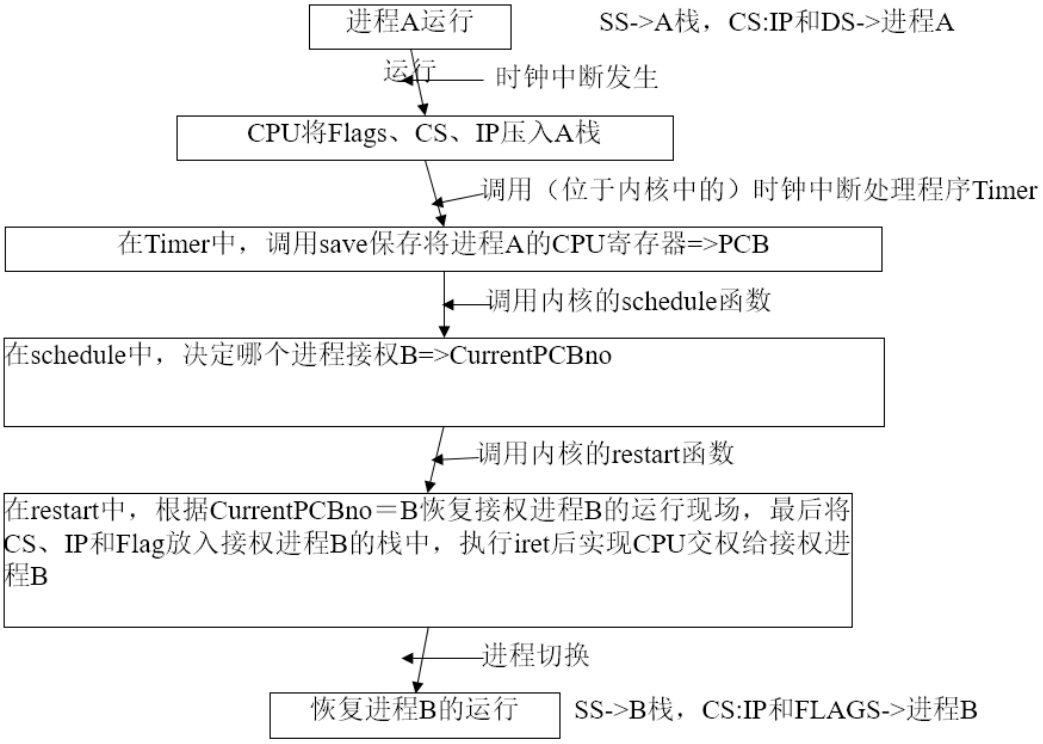
全局逻辑图

（红色为内核模块）

（6） 内核

利用时钟中断实现用户程序轮流执行。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。修改时钟中断服务程序，即每次发生时钟中断，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。

（7） 时钟中断服务程序



（8） 现场保护：SAVE过程

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。

Save过程涉及到几种不同的栈：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。

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。

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。

对于PCB中的16个寄存器，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。

（9） 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，这里面最棘手的问题是SS的切换。

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。

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。

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。

4、 方案思想

受老师给的案例的启发，我就决定将整个系统分为两种模式，即用户模式和内核模式。在内核模式下，操作系统通过时钟中断在屏幕右下角动态显示‘|’，‘\’，‘—’，‘/’。在用户模式下，操作系统通过时钟中断打断执行中的用户程序实现CPU在进程之间交替。

首先，编写一个名为PCB.h的头文件，该文件中声明进程的各种状态、进程的上下文、进程队列和进程控制块。其中进程的上下文结构用于保存进程的各种寄存器数据，进程控制块保存进程的上下文、进程状态和进程ID，进程队列保存可并发执行的进程。在PCB.h头文件中，还编写了获取当前进程在进程队列中的位置函数get_current_process_PCB()、保存进程控制块数据的函数save_PCB()、调度程序schedule()、初始化进程控制块的函数PCB_initial()、创建新的进程控制块函数create_new_PCB()、初始化所有有关设置进程控制块参数的函数initial_PCB_settings()。

然后，在kliba.asm文件中编写创建并加载进程函数run_process()、设置时钟频率函数set_timer()、装载时钟中断函数set_clock()和时钟中断程序。

接着，在kernal.c文件中#include "PCB.h"并编写解析用户指令并创建进程函数create_process(char *comm)。

最后，在MyOS.asm文件中，在调用cmain程序之前调用set_clock()函数来装载时钟中断。

操作系统工作方式:

在内核中，响应用户的指令输入，当用户输入指令r + 进程号时，create_process(char *comm)函数将解析指令并通过run_process()函数创

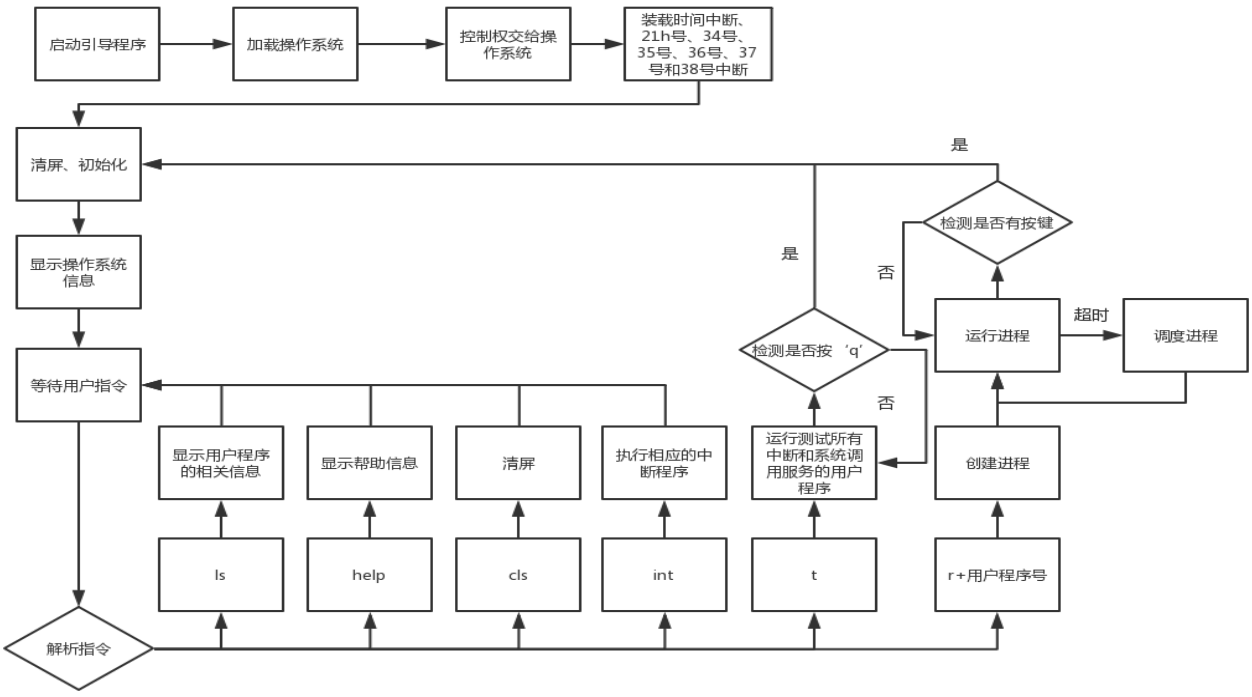
建进程并将进程装入预定的内存地址，加载完所有的进程后，将当前模式变为用户模式，然后等待时钟中断。

在时钟中断中，先判断当前模式。若是用户模式，则进入用户时钟中断；若是内核模式，则进入内核时钟中断。

在内核模式下的时钟中断与实验四&五的相差不大，所以就不再重复说明。

在用户模式下的时钟中断中，首先压栈保存各种寄存器的值，之后会将这些值作为参数传给save_PCB()函数。压栈后判断是否有按键，若用户有按键，则会将当前模式改为内核模式并返回内核;若用户没有按键,则会调用save_PCB()函数，该函数会将寄存器的值保存在进程的进程控制块中。接着将进行调度，切换进程，将当前进程号改为进程队列中的下一个进程号。接着，通过这个进程号定位当前进程在进程队列中的位置,然后根据之前保存在进程控制块的寄存器值恢复所有寄存器的值，最后从时间中断返回到用户进程中。

5、 程序流程



6、 算法与数据结果

数据结构：

（1） 进程状态：PCB_READY、PCB_RUNNING、PCB_BLOKED、PCB_EXIT。

（2） 进程上下文：Register结构包含所有寄存器ss，gs，fs，es，ds，di，si，sp，bp，bx，dx，cx，ax，ip，cs，flag。

（3） 进程队列：PCB_LIST[MAX_PCB_NUMBER];

（4） 进程控制块：PCB结构包含进程上下文regs、进程状态status、进程ID。

7、 程序关键模块

（1）在PCB.h头文件中声明名为PCB_STATUS的枚举类型来表示进程的各种状态，如新建态（PCB_READY）、运行态（PCB_RUNNING）、阻塞态（PCB_BLOCKED）和终止态（PCB_EXIT）。代码如下：

```
typedef enum PCB_STATUS{PCB_READY, PCB_EXIT, PCB_RUNNING, PCB_BLOCKED} PCB_STATUS;
```

（2）在PCB.h头文件中声明一个名为Register的结构，用于存储进程的上下文，即各寄存器的值。代码如下：

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

（3）在PCB.h头文件中声明一个名为PCB的进程控制块，用户存储进程的进程状态、进程ID和进程上下文，代码如下：

```
typedef struct PCB{
    Register regs;
```

```
    PCB_STATUS status;
    int ID;
} PCB;
```

（4）在PCB.h头文件中分别声明进程队列PCB_LIST、指向当前进程的指针current_process_PCB_ptr、判断是否第一次运行的变量first_time、判断是否为内核模式的变量kernal_mode、进程总数process_number、加载进程至内存的段地址current_seg和当前进程号current_process_number。代码如下：

```
PCB PCB_LIST[MAX_PCB_NUMBER];
PCB *current_process_PCB_ptr;
int first_time;
int kernal_mode = 1;
int process_number = 0;
int current_seg = 0x1000;
int current_process_number = 0;
```

（5）在PCB.h头文件中的获取当前进程在进程队列中的位置函数get_current_process_PCB()，代码如下：

```
PCB *get_current_process_PCB() {
    //获取当前进程在进程队列中的位置
    return &PCB_LIST[current_process_number];}
```

（6）在PCB.h头文件中的保存进程控制块数据的函数save_PCB()，代码如下：

```
void save_PCB(int ax, int bx, int cx, int dx, int sp, int bp, int si, int di, int ds, int es, int fs, int gs, int ss, int ip, int cs, int flags)
{
    //获取当前进程在进程队列中的位置
    current_process_PCB_ptr = get_current_process_PCB();
    //保存当前进程的上下文
    current_process_PCB_ptr->regs.ss = ss;
    current_process_PCB_ptr->regs.gs = gs;
    current_process_PCB_ptr->regs.fs = fs;
    current_process_PCB_ptr->regs.es = es;
    current_process_PCB_ptr->regs.ds = ds;
```

```
    current_process_PCB_ptr->regs.di = di;
    current_process_PCB_ptr->regs.si = si;
    current_process_PCB_ptr->regs.sp = sp;
    current_process_PCB_ptr->regs.bp = bp;
    current_process_PCB_ptr->regs.bx = bx;
    current_process_PCB_ptr->regs.dx = dx;
    current_process_PCB_ptr->regs.cx = cx;
    current_process_PCB_ptr->regs.ax = ax;
    current_process_PCB_ptr->regs.ip = ip;
    current_process_PCB_ptr->regs.cs = cs;
    current_process_PCB_ptr->regs.flags = flags;}
```

（7）在PCB.h头文件中的调度程序schedule()，代码如下：

```
void schedule() {
    if (current_process_PCB_ptr->status == PCB_READY) {
    //如果当前进程是新建态，则将first_time置为1，将该进程变为运行态。
        first_time = 1;
        current_process_PCB_ptr->status = PCB_RUNNING;
        return;}

    current_process_PCB_ptr->status = PCB_BLOCKED;
    //如果当前进程不是新建态，则将该进程变为阻塞态。

    current_process_number++;
    //将当前进程变为进程队列中的下一个进程

    if (current_process_number >= process_number)
        current_process_number = 0;
    //若当前进程号大于进程总数则将当前进程变为进程队列中的第一个进程

    current_process_PCB_ptr = get_current_process_PCB();
    //获取当前进程在进程队列中的位置

    if (current_process_PCB_ptr->status == PCB_READY) first_time = 1;
    //如果当前进程是新建态，则将first_time置为1，将该进程变为运行态。
    current_process_PCB_ptr->status = PCB_RUNNING;
    return;}
```

（8）在PCB.h头文件中的初始化进程控制块的函数PCB_initial()，代码如下：

```
void PCB_initial(PCB *ptr, int process_ID, int seg) {
    ptr->ID = process_ID;          //初始化进程ID
    ptr->status = PCB_READY;       //进程状态为新建态
    ptr->regs.gs = 0x0B800;        //gs寄存器初始化为0xB800
    ptr->regs.es = seg;       //es, ds, fs, ss, cs分别初始化为参数seg
    ptr->regs.ds = seg;
    ptr->regs.fs = seg;
    ptr->regs.ss = seg;
    ptr->regs.cs = seg;
    ptr->regs.di = 0;
    ptr->regs.si = 0;
    ptr->regs.bp = 0;
    ptr->regs.sp = 0x0100 - 4;    //sp寄存器初始化为0x100-4
    ptr->regs.bx = 0;             //通用寄存器初始化为0
    ptr->regs.ax = 0;
    ptr->regs.cx = 0;
    ptr->regs.dx = 0;
    ptr->regs.ip = 0x0100;        //ip初始化为0x100
    ptr->regs.flags = 512;}       //初始化标志寄存器
```

（9）在PCB.h头文件中的创建新的进程控制块函数create_new_PCB()，代码如下：

```
void create_new_PCB() {
    if (process_number > MAX_PCB_NUMBER) return;
    //若进程总数大于进程队列的最大长度，则不创建新的进程控制块

    PCB_initial(&PCB_LIST[process_number], process_number,
                current_seg);
    //创建、初始化新的进程控制块并将其加入进程队列

    process_number++;
    //进程总数加1
    current_seg += 0x1000;}       //内存段地址增加0x1000
```

（10）在PCB.h头文件中的初始化所有有关设置进程控制块的参数的函数initial_PCB_settings()，代码如下：

```
void initial_PCB_settings() {
    process_number = 0;           //进程总数初始化为0
    current_process_number = 0;   //初始化当前进程号
    current_seg = 0x1000;}        //初始化内存段地址
```

（11）在kliba.asm文件中的设置时钟频率的函数set_time()，代码如下：

```
public _set_timer
_set_timer proc
    push ax
    mov al, 36h         ; 计数器0的控制字为00110110
    out 43h, al         ; 将控制字传给计数器0的端口43h
    mov ax, 11931       ; 将时钟频率设为100Hz
    out 40h, al
    mov al, ah
    out 40h, al
    pop ax
    ret
_set_timer endp
```

（12）在kliba.asm文件中的设置时钟中断函数set_clock()，代码如下：

```
public _set_clock
_set_clock proc
    push es
    call near ptr _set_timer                ; 设置时钟频率
    xor ax, ax
    mov es, ax                              ; 将es寄存器置为0
    mov word ptr es:[20h], offset Timer     ;设置时钟中断向量的偏移地址
    mov word ptr es:[22h], cs               ; 设置时钟中断向量的段地址
    pop es
    ret
_set_clock endp
```

（13）在kliba.asm文件中的时钟中断程序Timer分为两个部分，一个为用户模式下的时钟中断，另一个为内核模式下的时钟中断。

判断是否为内核模式代码如下：

```
Timer:
    cmp word ptr [_kernal_mode], 1    ; 判断当前是用户模式还是内核模式
    jne process_timer                 ; 进入用户模式下的时钟中断
    jmp kernal_timer                  ; 进入内核模式下的时钟中断
```

用户模式下的时钟中断代码如下：

```
process_timer:
    ; push所有寄存器的值作为参数传给save_PCB()函数，保存进程的上下文
    .386
    push ss
    push gs
    push fs
    .8086
    push es
    push ds
    push di
    push si
    push bp
    push sp
    push dx
    push cx
    push bx
    push ax
    cmp word ptr [back_time], 0       ; 判断用户程序是否到达退出时间
    jnz time_to_go                    ; 未达到则继续运行
    mov word ptr [back_time], 1       ; 达到则重置back_time
    mov word ptr [_kernal_mode], 1    ; 进入内核模式
    add sp, 11*2                      ; 恢复堆栈
    push 512                          ; 设置flag寄存器
    push 800h                         ; 设置cs寄存器
    push 100h                         ; 设置ip寄存器
    iret                              ; 按该顺序压栈，中断返回时进入内核
time_to_go:
    inc word ptr [back_time]          ; 增加用户程序执行的时间
    mov ax, cs
    mov ds, ax
    mov es, ax
```

```
    call _save_PCB                  ; 保存进程控制块
    call _schedule                  ; 进程调度
store_PCB:
    mov ax, cs
    mov ds, ax
    call _get_current_process_PCB   ; 获取当前进程在进程队列中的位置
    mov si, ax
    mov ss, word ptr ds:[si]        ; 恢复 ss 寄存器
    mov sp, word ptr ds:[si+2*7]    ; 恢复 sp
    cmp word ptr [_first_time], 1   ; 判断进程是否第一次运行
    jnz next_time                   ; 不是，则平衡堆栈
    mov word ptr [_first_time], 0   ; 是，则将first_time置0
    jmp start_PCB                   ; 跳过平衡堆栈的操作
next_time:
    add sp, 11*2                    ; 平衡堆栈，恢复进入时钟中断前栈顶
start_PCB:
    mov ax, 0
    push word ptr ds:[si+2*15]      ; 恢复 flag寄存器
    push word ptr ds:[si+2*14]      ; 恢复 cs寄存器
    push word ptr ds:[si+2*13]      ; 恢复 ip寄存器
    ; 按此顺序压栈，模拟中断进入操作
    mov ax, word ptr ds:[si+2*12]   ; 恢复 ax寄存器
    mov cx, word ptr ds:[si+2*11]   ; 恢复 cx寄存器
    mov dx, word ptr ds:[si+2*10]   ; 恢复 dx寄存器
    mov bx, word ptr ds:[si+2*9]    ; 恢复 bx寄存器
    mov bp, word ptr ds:[si+2*8]    ; 恢复 bp寄存器
    mov di, word ptr ds:[si+2*5]    ; 恢复 di寄存器
    mov es, word ptr ds:[si+2*3]    ; 恢复 es寄存器
    .386
    mov fs, word ptr ds:[si+2*2]    ; 恢复 fs寄存器
    mov gs, word ptr ds:[si+2*1]    ; 恢复 gs寄存器
    .8086
    push word ptr ds:[si+2*4]       ; push ds寄存器
    push word ptr ds:[si+2*6]       ; push si寄存器
    pop si                          ; 恢复 si
    pop ds                          ; 恢复 ds
process_timer_end:
    push ax
    mov al, 20h                     ; 发送中断处理结束消息给中断控制器
    out 20h, al                     ; 发送EOI到主8529A
    out 0A0h, al                    ; 发送EOI到从8529A
    pop ax
    iret
```

内核模式下的时钟中断代码如下：

```
kernal_timer:
    push es
    push ds
    dec byte ptr es:[cccount]          ;递减计数变量
    jnz fin                            ; >0 跳转
    inc byte ptr es:[tmp]              ;自增tmp
    cmp byte ptr es:[tmp], 1           ;根据tmp选择显示内容
    jz ch1                             ;1显示‘/’
    cmp byte ptr es:[tmp], 2           ;2显示‘|’
    jz ch2
    cmp byte ptr es:[tmp], 3           ;3显示‘\’
    jz ch3
    cmp byte ptr es:[tmp], 4           ;4显示‘-’
    jz ch4
ch1:
    mov bl, '/'                        ; 显示‘/’
    jmp showch
ch2:
    mov bl, '|'                        ; 显示‘|’
    jmp showch
ch3:
    mov bl, '\'                        ; 显示‘\’
    jmp showch
ch4:
    mov byte ptr es:[tmp],0
    mov bl, '-'                        ; 显示‘-’
    jmp showch
showch:
    .386
    push gs
    mov ax,0B800h                      ; 文本窗口显存起始地址
    mov gs,ax                          ; GS = B800h
    mov ah,0Fh
    mov al,bl
    mov word[gs:((80 * 24 + 78) * 2)], ax    ;在窗口的右下角显示
    pop gs
    .8086
    mov byte ptr es:[cccount],8
fin:
    mov al,20h                         ; AL = EOI
    out 20h,al                         ; 发送EOI到主8529A
    out 0A0h,al                        ; 发送EOI到从8529A
```

```
    pop ds
    pop es                                  ; 恢复寄存器信息
    iret
```

（14）在kliba.asm文件中的创建进程函数run_process()，代码如下：

```
public _run_process
_run_process proc
    push es
    mov ax, word ptr [_current_seg]  ; 段地址
    mov es, ax                       ; 设置段地址
    mov bx, 100h                     ; 设置偏移量
    mov ah, 2                        ; 功能号
    mov al, 1                        ; 扇区数
    mov dl, 0                        ; 驱动器号，软盘为0
    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, byte ptr [_sector_number]; 起始扇区号，起始编号为1
    int 13h                          ; BIOS的13h功能调用
    call _create_new_PCB             ;为加载的用户程序创建新的进程控制块
    pop es
    ret
_run_process endp
```

（15）在kernal.c文件中#include "PCB.h"头文件，编写根据用户输入创建进程的函数create_process(char *comm)，代码如下：

```
void create_process(char *comm) {
    int i, sum = 0, flag = 0;
    for (i = 1; i < strlen(comm); ++i) {
    //判断用户是否输入了无效用户程序号
       if (comm[i] == ' ' || comm[i] >= '1' && comm[i] <= '4') continue;
        else {
            //若是无效用户程序号，则显示提醒信息
            print("  invalid program number: ");
            printChar(comm[i]);
            print("\n\n\r");
            return;}}
    for (i = 1; i < strlen(comm); ++i)
        if (comm[i] != ' ') flag = 1;
    if (flag == 0) {
        //判断用户是否没有输入用户程序号
```

```
        print("  invalid input\n\n\r");
        return;}
    run_process(10, current_seg);
    for (i = 1; i < strlen(comm) && sum < MAX_PCB_NUMBER; ++i) {
        if (comm[i] == ' ') continue;//忽略用户输入的空格
        sum++;                          //创建的进程总数加1
        sector_number = comm[i] - '0' + 10; //确定用户程序所在的扇区号
        run_process(sector_number, current_seg);} //创建进程
    kernal_mode = 0;}                   //进入用户模式
```

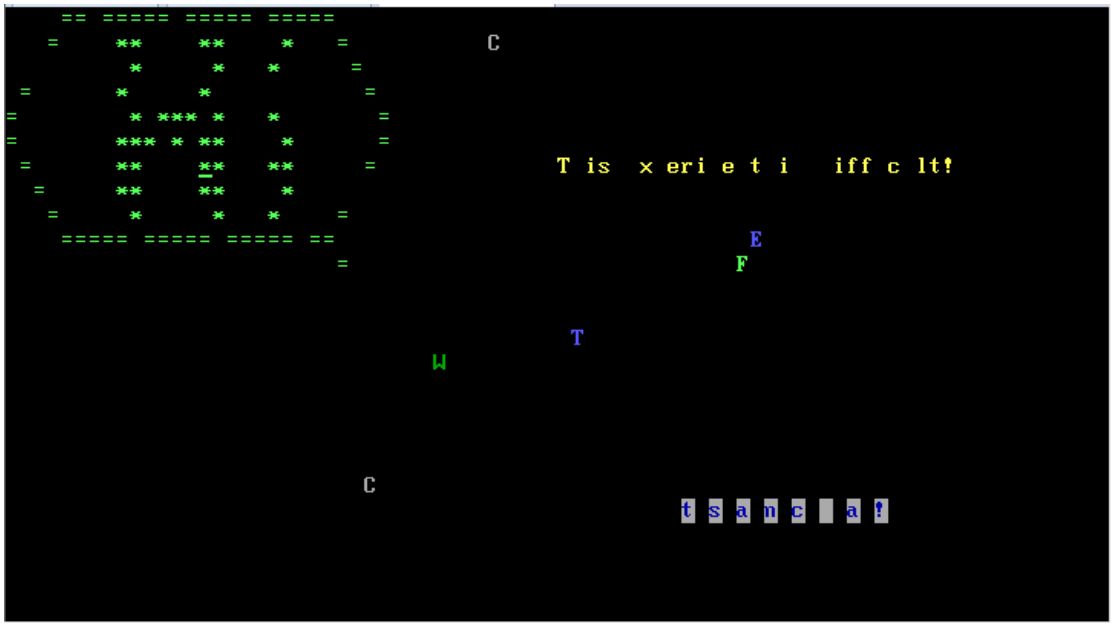
四． 实验过程和结果

1、 进入操作系统后，由于是内核模式，右下角会有一个动态显示的‘|’，‘\’，‘–’，‘/’。

2、 输入指令r 1234后，操作系统会创建进程1、2、3、4（进程中也测试了int 34、int 35、int 36、int 37中断功能），并进入用户模式，所创建的进程并发执行，效果如图所示。

3、 按下任意按键后，进入内核模式返回操作系统。

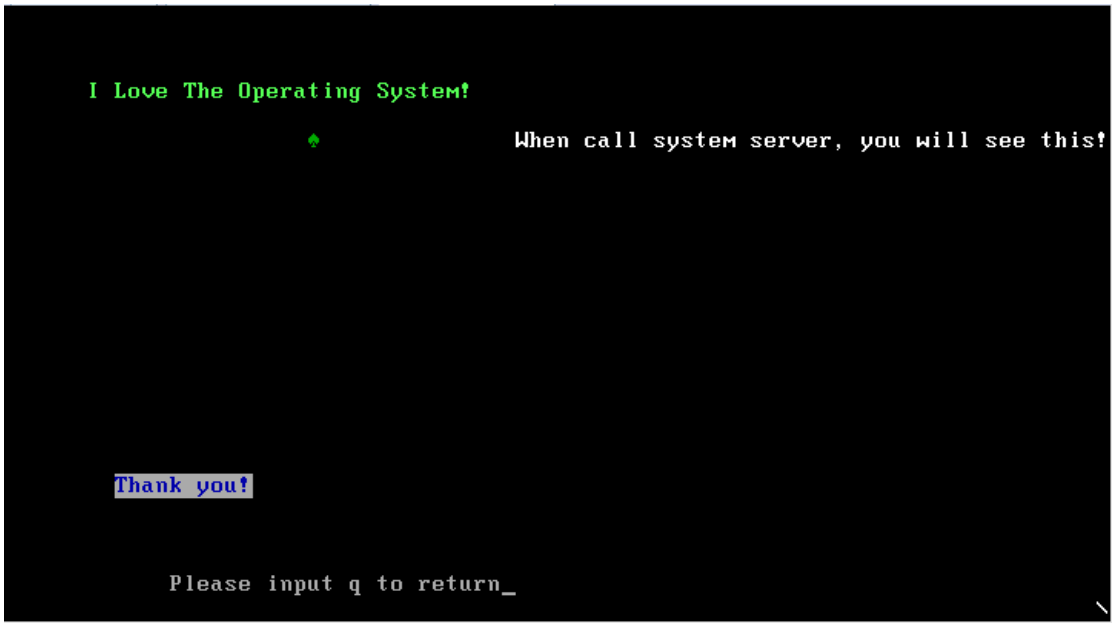
4、 输入指令r 1234123后操作系统会创建进程1、2、3、4、5、6、7，并进入用户模式所创建的进程并发执行，效果如图所示。



5、 按下任意按键后，进入内核模式返回操作系统。

6、 输入指令t后，进入测试所有的中断程序，首先显示int 34、int 35、int 36、int 37这些中断的内容，效果如图所示。

7、 调用38号中断停几秒后显示系统调用1、2、3号的服务内容，再停几秒后即可按‘q’返回操作系统。效果如图所示。



五． 实验总结

总结:

这次实验我的收获很大。明白了一成时间写代码，九成时间debug的道理。

做完这次实验后我才真正了解到多进程并发执行时保存与恢复进程上下文的重要性，如果没有正确保存和恢复寄存器的值的话，就会出现各种神奇的BUG，例如，弹球不会连续移动，而是跳跃式的移动，它会从左上角突然跳至屏幕中央；或是原本弹球是一个字符，但切换一次进程后，弹球就成了一个乱码；这些BUG仅仅增添了一些乐趣，最致命的BUG就是前一个进程执行的好好的，但切换至下一个进程后，整个程序就停住了，弹球不再动，按键也没有响应。这些BUG的出现都是没有正确保存或恢复进程上下文的缘故。

实验中遇到的问题及解决方案：

（1）无法加载并执行一个进程，且在并发执行多个进程时，无法执行第一个进程。例如，当用户输入指令r 1后，内核并不会加载或执行进程1，而是卡在内核界面，当用户再按下一个按键后会返回内核；当用户输入r 1234指令后，内核只会加载并执行进程2、3、4，而不会加载或执行进程1。

出现这个问题的原因在于，加载一个进程，将模式变为用户模式，进入用户时钟中断后，将内核的所有寄存器压栈，在调用save_PCB()函数保存寄存器值时，当前进程是刚刚加载的进程，那些寄存器值保存的位置就是这个刚创建的进程控制块中，这就会覆盖原本的新进程的寄存器值，这样就会出现无法执行第一个进程的问题。

为解决这个问题，我之前想过重写时钟中断，但重写时钟中断过于麻烦，我就想可不可以在加载用户想要的进程之前装载一个无用的进程，内核寄存器值覆盖的是无用进程控制块中的寄存器值，这样就不会影响其他的进程。这样的修改也是十分简单的，只需在kernal.c文件中的create_process(char *comm)函数中加上一句run_process(10, current_seg)，再修改一下调度程序schedule()即可。测试后发现成功解决该问题。NICE!

（2）从一个进程切换至另一个进程后，进程停止了，弹球不再移动，按键也没有任何响应。

这个问题的出现很明显是因为保护或恢复寄存器值过程出现了错误。由于可能出现问题的是两个过程，所以我先考虑保存寄存器值过程，再考虑恢复寄存器值过程。为了更好的找出错误，我决定只执行一个进程，调度时也只执行这个进程，这样就方便观察保护和恢复寄存器值过程中的变化。

在保存寄存器值过程中，先将各个寄存器压栈，即以ss，gs，fs，es，ds，di，si，bp，sp，dx，cx，bx，ax这一顺序压栈，这些寄存器值将作为参数传给save_PCB()函数，该函数以ax，bx，cx，dx，sp，bp，si，di，ds，es，fs，gs，ss，ip，cs，flags这一顺序接受参数。使用bochs调试查看压栈后的堆栈、寄存器，如图所示。

```
<bochs:38> print-stack
Stack address size 2
| STACK 0x17f40 [0x0034]
| STACK 0x17f42 [0xff00]
| STACK 0x17f44 [0x0f57]
| STACK 0x17f46 [0x0000]
| STACK 0x17f48 [0xff4a]
| STACK 0x17f4a [0xff80]
| STACK 0x17f4c [0x0000]
| STACK 0x17f4e [0xff88]
| STACK 0x17f50 [0x0040]
| STACK 0x17f52 [0x0800]
| STACK 0x17f54 [0x0000]
| STACK 0x17f56 [0x0000]
| STACK 0x17f58 [0x0800]
| STACK 0x17f5a [0xe869]
| STACK 0x17f5c [0xf000]
| STACK 0x17f5e [0x0246]
```

堆栈图

```
<bochs:39> r
rax: 00000000_00000034 rcx: 00000000_00090f57
rdx: 00000000_00000000 rbx: 00000000_0000ff00
rsp: 00000000_0000ff40 rbp: 00000000_0000ff80
rsi: 00000000_000e0000 rdi: 00000000_0000ff88
r8 : 00000000_00000000 r9 : 00000000_00000000
r10: 00000000_00000000 r11: 00000000_00000000
r12: 00000000_00000000 r13: 00000000_00000000
r14: 00000000_00000000 r15: 00000000_00000000
rip: 00000000_00000240
```

寄存器图1

```
<bochs:40> sreg
es:0x0800, dh=0x00009300, dl=0x8000ffff, valid=1
        Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0800, dh=0x00009300, dl=0x8000ffff, valid=3
        Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0800, dh=0x00009300, dl=0x8000ffff, valid=7
        Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0040, dh=0x00009300, dl=0x0400ffff, valid=3
        Data segment, base=0x00000400, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x00000000000f9a37, limit=0x30
idtr:base=0x0000000000000000, limit=0x3ff
```

寄存器图2

对比之后可知寄存器的压栈没有出现问题，运行save_PCB()后，使用bochs查看寄存器值是否存储正确，如图所示。

```
<bochs:36> xp /128bx 0x8fa6
[bochs]:
0x0000000000008fa6 <bogus+       0>:    0x00    0x08    0x00    0x00    0x00    0x00    0x00    0x08
0x0000000000008fae <bogus+       8>:    0x40    0x00    0x88    0xff    0x00    0x00    0x4a    0xff
0x0000000000008fb6 <bogus+      16>:    0x80    0xff    0x00    0xff    0x00    0x00    0x57    0x0f
0x0000000000008fbe <bogus+      24>:    0x28    0x00    0x69    0xe8    0x00    0xf0    0x46    0x02
```

内存图

对比之后发现寄存器值存储正确，看来问题可能是出在恢复寄存器这个过程。继续执行，直到恢复完所有的寄存器，使用bochs查看寄存器值是否正确，如图。

```
rax: 00000000_0000b800 rcx: 00000000_00090010
rdx: 00000000_00001430 rbx: 00000000_00000071
rsp: 00000000_000000dc rbp: 00000000_00000000
rsi: 00000000_000e0000 rdi: 00000000_00000000
r8 : 00000000_00000000 r9 : 00000000_00000000
r10: 00000000_00000000 r11: 00000000_00000000
r12: 00000000_00000000 r13: 00000000_00000000
r14: 00000000_00000000 r15: 00000000_00000000
rip: 00000000_00000240
```

寄存器图3

```
<bochs:300> sreg
es:0x3000, dh=0x00009303, dl=0x0000ffff, valid=1
        Data segment, base=0x00030000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0800, dh=0x00009300, dl=0x8000ffff, valid=3
        Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
ss:0x3000, dh=0x00009303, dl=0x0000ffff, valid=1
        Data segment, base=0x00030000, limit=0x0000ffff, Read/Write, Accessed
ds:0x3000, dh=0x00009303, dl=0x0000ffff, valid=7
        Data segment, base=0x00030000, limit=0x0000ffff, Read/Write, Accessed
fs:0x3000, dh=0x00009303, dl=0x0000ffff, valid=1
        Data segment, base=0x00030000, limit=0x0000ffff, Read/Write, Accessed
gs:0xb800, dh=0x0000930b, dl=0x8000ffff, valid=1
        Data segment, base=0x000b8000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x00000000000f9a37, limit=0x30
idtr:base=0x0000000000000000, limit=0x3ff
```

寄存器图4

对比后发现，寄存器的值全都改变了，说明问题确实出现在恢复寄存器过程中。

为解决这个问题，我重写了寄存器恢复过程，根据名为Register的结构中声明的寄存器顺序恢复相应的寄存器值，经过几番调试和修改，最终解决了这个问题。

（3）这个实验的最初的版本不能退出用户进程，不能返回内核，但是内核和用户进程可以并发执行，也就是说在用户进程执行的期间可以输入指令并执行指令，指令可以是r + 进程号，执行这条指令后会在原来的基础上新增几个弹球在屏幕上运动。我觉得很神奇，但这样不符合我的期望，于是对代码进行一次魔改。再次测试时，解决了内核和用户进程并发执行的问题，并实现了按任意键返回内核的功能，但是出现了不能运行第一个用户进程的问题。最后，我就使用上述的解决方法解决了这个问题。

做完实验后，我想了想，内核和用户进程并发执行的原因在于进入时钟中断后，错将内核当做进程并加入了进程队列，经过调度，使得内核和用户进程并发执行。

六． 参考文献

1、 《x86 PC汇编语言，设计与接口》

2、 参考原型2