

《操作系统实验》

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 : 杨志成

学 号 : 16337281

时 间 : 2018 年 3 月 17 日

目录

一，实验目的-----	3
二，实验要求-----	3
三，实验方案-----	4
(1) 基础原理-----	4
(2) 实验工具与环境-----	7
(3) 程序流程-----	8
(4) 程序模块功能-----	9
(5) 代码文档组成-----	9
(6) 实现效果-----	10
四，实验过程及实验结果-----	12
五，实验总结-----	15

成绩：

实验五：中断机制编程技术

一 实验目的

- 1, 在内核实现多进程的三状态模, 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

- (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) 基础原理

- 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。
 - 多个用户程序并发执行
 - 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
 - 执行和等待
 - 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

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，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
- 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，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

什么是进程表？

- 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，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
- 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，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

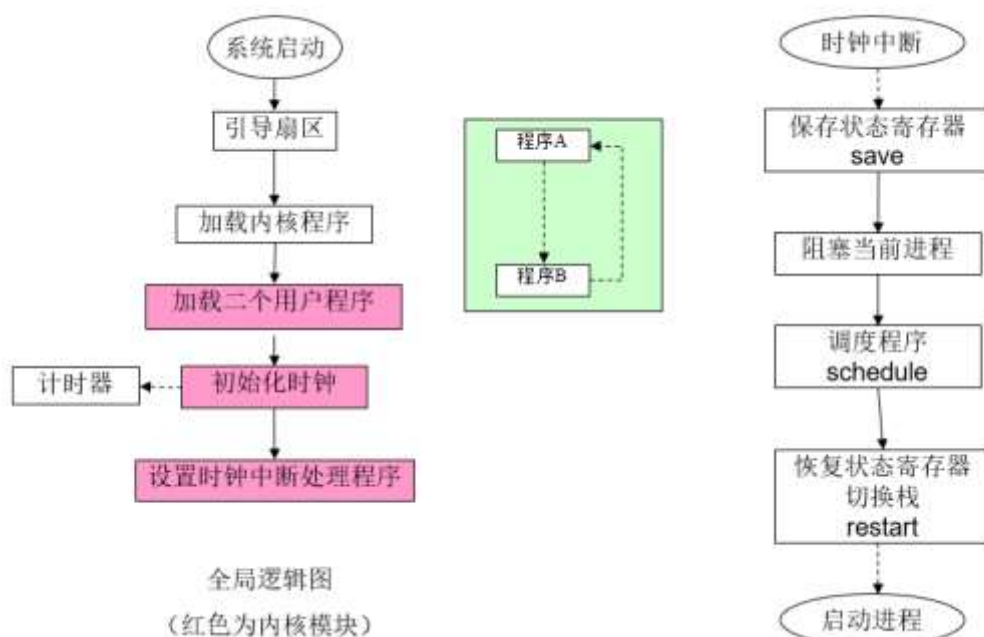
■ 进程交替执行原理：

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

■ 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

实现进程模型 的系统框架



内核的更改：

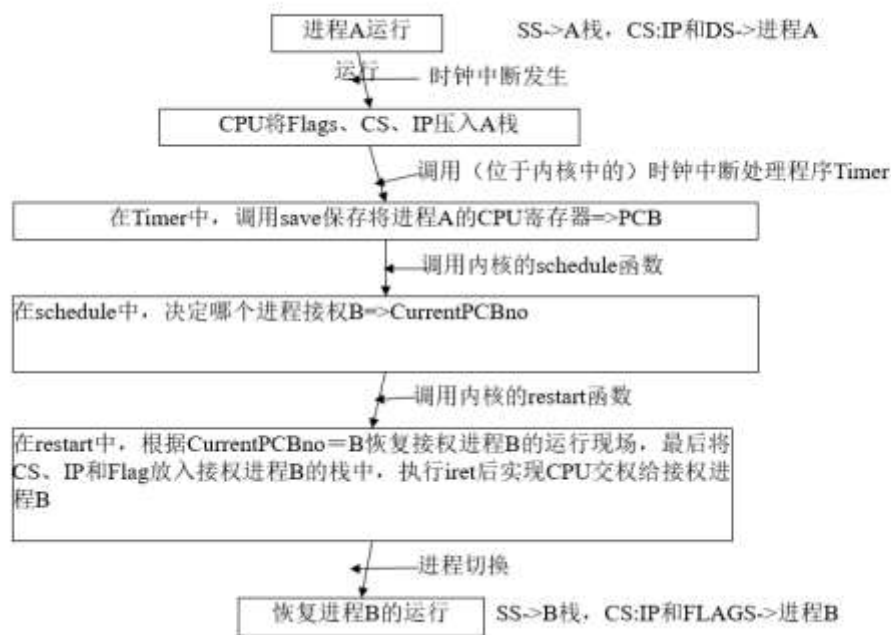
■ 利用时钟中断实现用户程序轮流执行

■ 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

■ 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。



保护现场的 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。
- 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
- 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

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。

还原现场 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。

2) 实验工具环境

实验支撑环境

硬件: 个人计算机

主机操作系统: Windows/Linux/Mac OS/其它

虚拟机软件: VMware/VirtualPC/Bochs/其它

PC 虚拟机裸机/DOS 虚拟机/其它

实验开发工具

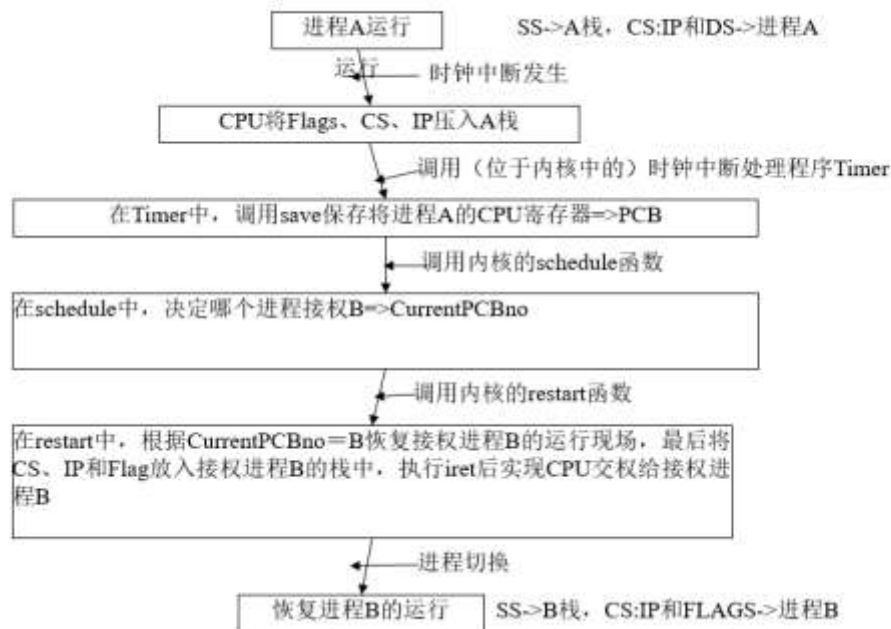
汇编语言工具: x86 汇编语言

高级语言工具: 标准 c 语言

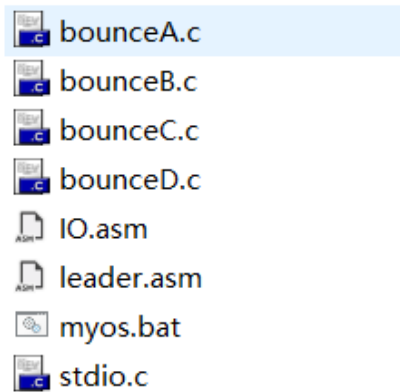
磁盘映像文件浏览编辑工具

调试工具: Bochs

(3) 程序流程示意图:



(4)代码文档组成：



- leader.asm 引导程序
- IO.asm 汇编语言库
- stdio.c c 语言基本输入输出库
- bounceA/B/C/D.c 弹跳小球的用户程序

(5) 代码模块：

save 函数的代码：

```
ds_save dw 0
ret_save dw 0
si_save dw 0
kernelsp dw 0

save:
    ;push flags
    ;push cs
    ;push ip
    ;push address
    push ds
    push cs
    pop ds      ;让 ds = cs
    pop word[ds_save] ;保存 ds
    pop word[ret_save] ;保存返回地址
    mov word[si_save],si ;保存当前 si
    mov si,word[_CurrentProc] ;获取当前 PCB 地址
    ;先保存通用寄存器和某些段寄存器
    mov word[si],ax
    mov word[si+1*4],bx
    mov word[si+2*4],cx
```



```

mov word[si+3*4],dx
mov word[si+5*4],di
mov word[si+6*4],bp
mov word[si+7*4],es
mov word[si+9*4],ss
;对 ds 及其他寄存器操作
push word[ds_save]; push ds
pop word[si+8*4] ; pop ds
pop word[si+11*4] ; store ip
pop word[si+12*4] ; store cs
pop word[si+13*4] ; store falgs
mov word[si+10*4],sp ;保存当前 sp
mov ax,word[si+4*15]
cmp ax,0 ;判断是否为内核的 PCB
jnz skipPCB ;如果不是内核 PCB
mov word[kernelsp],sp ;保存内核的 sp 至指定地址
skipPCB: ;跨段操作
mov ax,word[si_save] ;
mov word[si+4*4],ax ;保存 si
;调整段寄存器
mov ax,cs
mov ds,ax
mov ss,ax
mov es,ax
mov ax,word[kernelsp] ;获取当前内核的 sp
mov sp,ax ;保存 sp
mov ax,word[ret_save] ;跳转回原来的位置
jmp ax

```

详解：

在操作系统调用 int 08h 时，压栈顺序是 flags，cs，ip，在进入 int 08h 后，我们在 int 08h 中的操作如下：

```

INT_08H:
cli
call save
push 0
call _int_08h_override
push 0
call _Schedule
jmp restart

```

由于首先调用了 call 函数，因此程序会先压栈一个当前的返回地址，所以我们才有了以下代码

```

;push flags
;push cs

```

```

;push ip
;push address
push ds
push cs
pop ds      ;让 ds = cs
pop word[ds_save] ;保存 ds
pop word[ret_save] ;保存返回地址
mov word[si_save],si ;保存当前 si
mov si,word[_CurrentProc] ;获取当前 PCB 地址

```

在保护完通用寄存器，及其他一般寄存器后：

；先保存通用寄存器和某些段寄存器

```

mov word[si],ax
mov word[si+1*4],bx
mov word[si+2*4],cx
mov word[si+3*4],dx
mov word[si+5*4],di
mov word[si+6*4],bp
mov word[si+7*4],es
mov word[si+9*4],ss

```

我们会发现要获取当前的 ip, flags, cs 等寄存器，只需要利用 int 的压栈即可：

```

push word[ds_save]; push ds
pop word[si+8*4] ; pop ds
pop word[si+11*4] ; store ip
pop word[si+12*4] ; store cs
pop word[si+13*4] ; store falgs
mov word[si+10*4],sp ;保存当前 sp

```

我们在接下来的步骤中，要讨论保存内核的 sp 寄存器问题，因为每次调用 int 08h 时，我们都会跳进内核的中，为了正常运行，需要保存内核 sp 到正确的地址，如果不是内核进程，我们需要跳过这个阶段：

```

mov ax,word[si+4*15] ;将当前 PCB 中的 id 赋予 ax 寄存器
cmp ax,0 ;判断是否为内核的 PCB
jnz skipPCB ;如果不是内核 PCB
mov word[kernelsp],sp ;保存内核的 sp 至指定地址

```

在程序的最后，我们先保存 si 到 PCB 表，然后更新段寄存器；然后获取并保存内核的 sp

```

mov ax,word[si_save] ;
mov word[si+4*4],ax ;保存 si
;调整段寄存器
mov ax,cs
mov ds,ax
mov ss,ax
mov es,ax

```

```

mov ax,word[kernel$] ;获取当前内核的 sp
mov sp,ax             ;保存 sp
mov ax,word[ret_save] ;跳转回原来的位置
jmp ax

```

restart 模块：

```

restart:
    mov si,word[_CurrentProc];获取当前 PCB 表
    ;还原通用寄存器以及 di,bp,es,ss
    mov ax,word[si]
    mov bx,word[si+1*4]
    mov cx,word[si+2*4]
    mov dx,word[si+3*4]
    mov di,word[si+5*4]
    mov bp,word[si+6*4]
    mov es,word[si+7*4]
    mov ss,word[si+9*4]
    ;再次保存当前 sp: 即内核 sp
    mov word[kernel$],sp
    mov sp,word[si+10*4] ;还原 PCB 表中的 sp
    push word[si+13*4] ;push falgs
    push word[si+12*4] ;push cs
    push word[si+11*4] ;push ip
    push word[si+8*4]  ;push ds
    mov si,[si+4*4]    ;pop si
    push ax ;保护 ax
    mov al,20h
    out 20h,al
    out 0A0h,al
    pop ax
    pop ds ;获取 ds 寄存器的值
    sti
    iret

```

相对于 save 模块，该模块更像一个 save 模块的逆过程。

一开始，我们先将当前 PCB 表的地址保存到 si 处，然后不断的还原通用寄存器以及一些其他寄存器；

之后，我们千万要记得再次保存 sp，因为此时我们依然在内核中，因此需要再次保存当前的 sp 寄存器，以免发生意外 !!! 在这个动作后，我们才可以从 PCB 表中还原 sp 寄存器。

接下来我们需要将进行几次 push 操作，这个操作与 int 中断指令中的 push 操作正好相反，目的是为了我们接下来调用 iret 更加方便，直接调到 PCB 指向的进程。

```

push word[si+13*4] ;push falgs
push word[si+12*4] ;push cs
push word[si+11*4] ;push ip

```

最后再将 ds 还原，因为我们之前的操作都用到了 ds 寄存器，进行数据寻址，所以该寄存器需要最后还原。

Schedule 函数：

```
void Schedule()
{
    Cur_num = CurrentProc->id;
    while(1)
    {
        Cur_num++;
        if(Cur_num>4) Cur_num = 0;
        if(PCBlist[Cur_num].status == 1)
        {
            CurrentProc = &PCBlist[Cur_num];
            return;
        }
    }
}
```

通过判断当前指向的 PCB 进程 Cur_num 来进行操作；判断当前 PCB 的 status 是否为 1；其中 1 为运行态，0 为阻塞态，这是一个循环，每次选择新的 PCB，其中 PCBlist【0】存放了内核的进程控制块。

Initial 模块（初始化第一个进程控制块 PCBlist[0]）：

```
void initialPCB()
{
    PCBlist[0].ax = 0;
    PCBlist[0].bx = 0;
    PCBlist[0].cx = 0;
    PCBlist[0].dx = 0;
    PCBlist[0].si = 0;
    PCBlist[0].di = 0;
    PCBlist[0].bp = 0;
    PCBlist[0].es = 0;
    PCBlist[0].ds = 0;
    PCBlist[0].cs = 0;
    PCBlist[0].ss = 0;
    PCBlist[0].sp = 0;
    PCBlist[0].ip = 0;
    PCBlist[0].flags = 512;
    PCBlist[0].status = 1;
    PCBlist[0].id = 0;
}
```

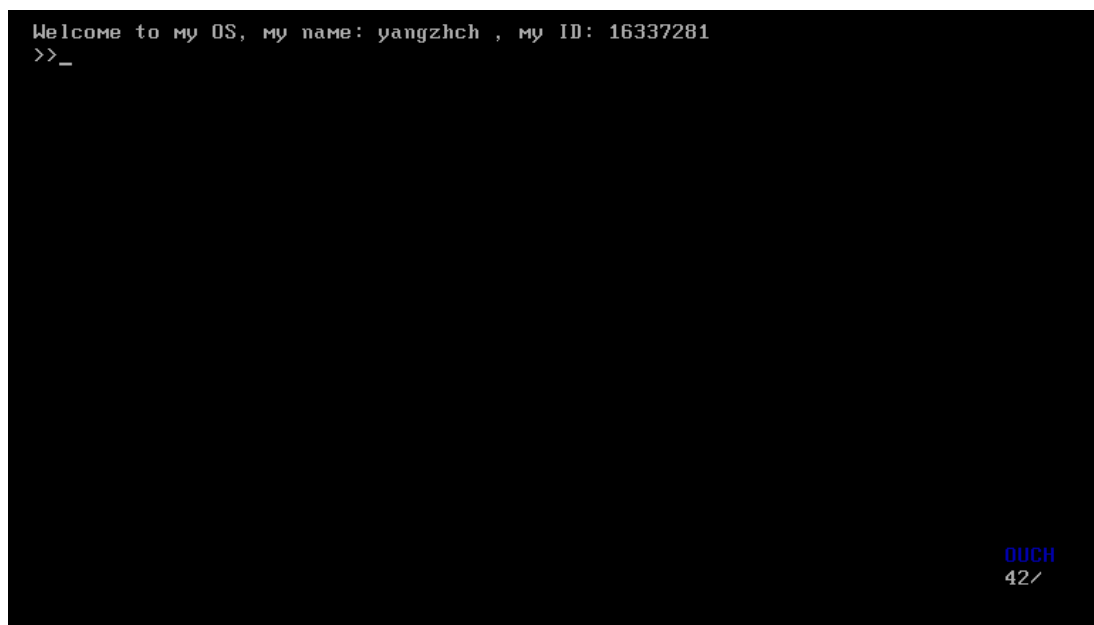
PushPCB 模块：(添加新的进程)

```
void pushPCB(int index,int offset)
{
    //int temp=0x6000;
    PCBlist[index].cs = index/3*0x1000+0x1000;//temp-index*0x1000;
    PCBlist[index].es = PCBlist[index].cs;
    PCBlist[index].ds = PCBlist[index].cs;
    PCBlist[index].ss = PCBlist[index].cs;
    PCBlist[index].flags=512;
    PCBlist[index].status=1;
    //
    PCBlist[index].ax = 0;
    PCBlist[index].bx = 0;
    PCBlist[index].cx = 0;
    PCBlist[index].dx = 0;
    PCBlist[index].si = 0;
    PCBlist[index].di = 0;
    PCBlist[index].bp = 0;
    PCBlist[index].ip = offset;
    PCBlist[index].sp = offset-4;
    PCBlist[index].id = index;
}
```

(6) 实现效果：

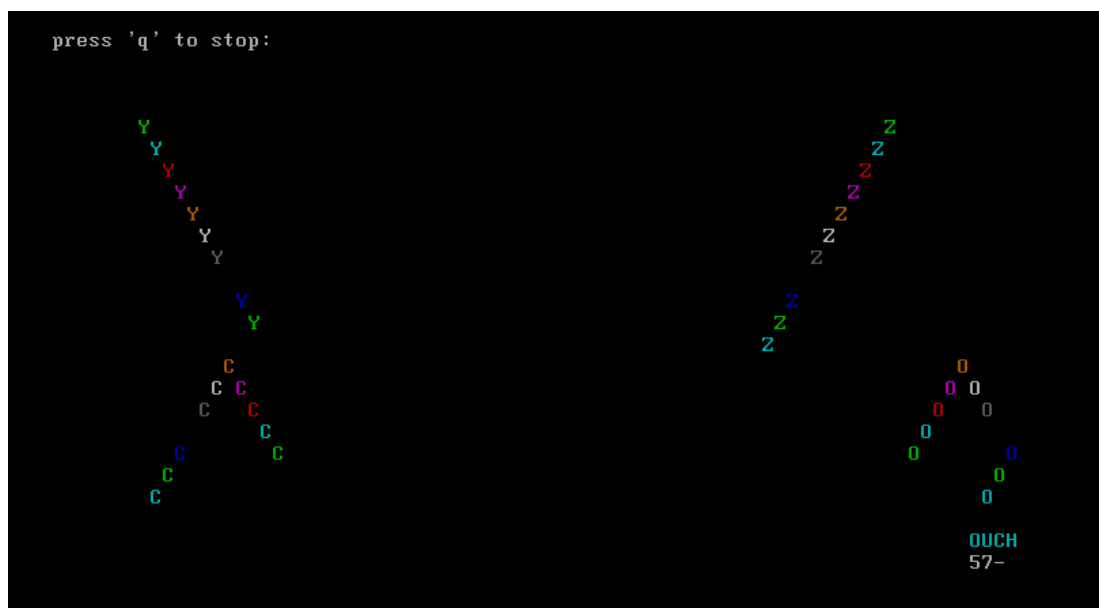
如图所示，在进入内核时，我们的程序是正确的，以前的时钟中断和键盘终端都可以正确运行

```
Welcome to my OS, my name: yangzhch , my ID: 16337281
>>_
```



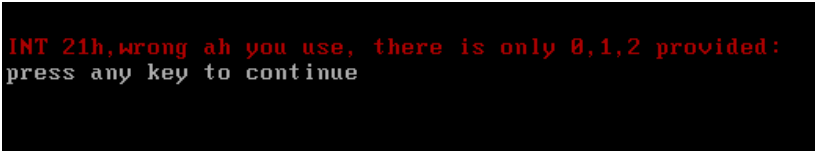
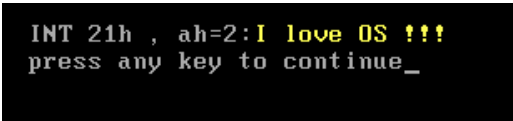
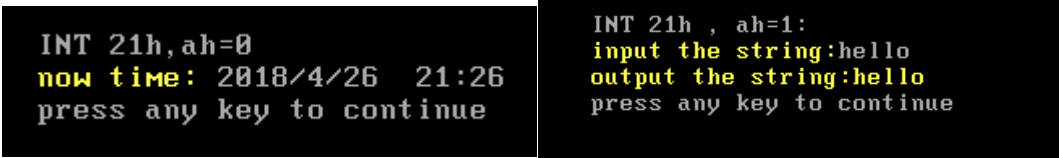
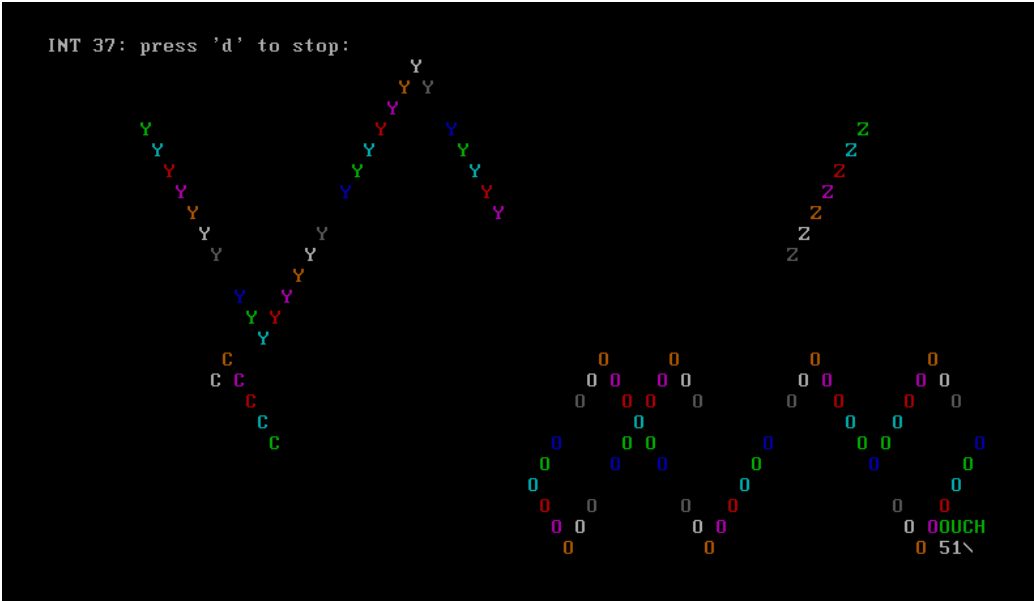
在加载四个子程序后：

```
press 'q' to stop:
```



可以看出，程序正常跳动，在本次实验中，我的程序可以通过控制按下 a,b,c,d 来控制四个小球的运行，按下 q 可以退出用户程序返回内核。

在运行测试程序时，也可以正常运行：



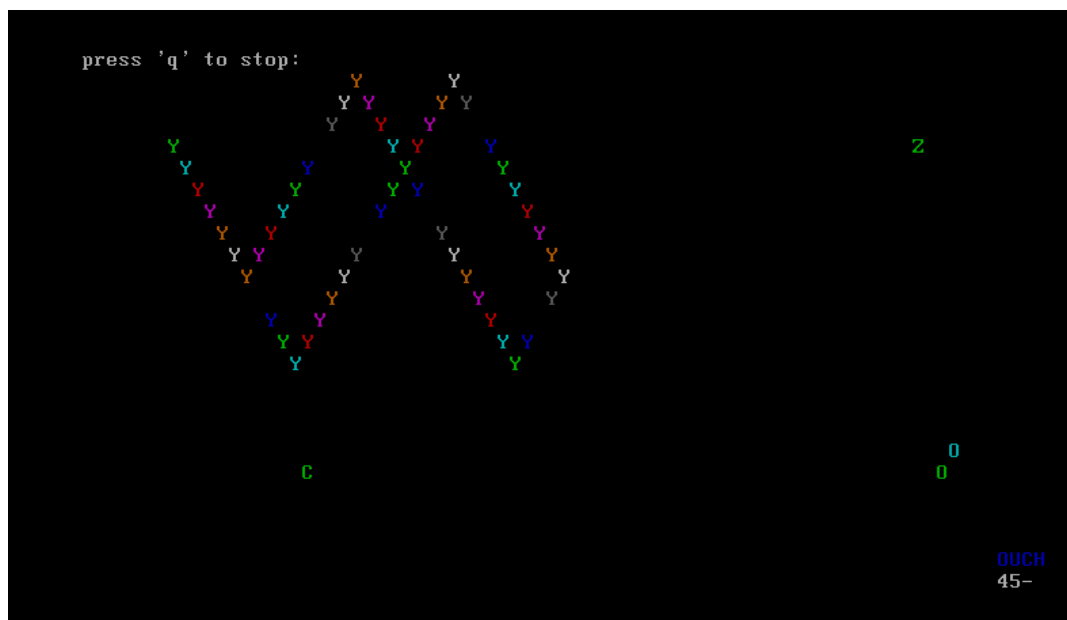
可见，所有系统中断运行正常

五 实验过程及其总结

在本次实验中，由于我是用的 nasm+gcc 实现的代码，因此老师给的原型 1, 2 的代码只能用做参考，不可完全复制，本次实验中，我主要是参考了原型一的少量代码，结合网上的代码，与同学讨论 debug 了很久，才实现了正确的 save 和 restart 代码，这是一个十分艰难但充满挑战性的过程。本次实验虽然代码量并不是很大，但是想要完成这样的一个任务确实不易。接下来分享一些我在本次实验当中遇到的问题以及解决思路：

问题 1: 有关建立新的 PCB 表时 sp 的初始化问题:

这是一个耐人寻味的问题，就目前而言有两种管理 sp 的方法：一种是将 sp 初始化到特定的位置，比如 3000h，这样 sp 堆栈就会固定咋 3000h 中。可是这种做法有一个缺陷，那就是我们不可以同一个段中放入两个或者多个应用程序，不然容易发生 sp 的冲突，导致我们在 save 过程和 restart 过程中发生地址冲突。冲突效果如下：



它会导致虽然我们的进程在不断的切换，但是由于四个程序在同一个段中共享了同一个 sp 寄存器，就导致堆栈的混淆调用，因此看上去只有一个程序在运行，找出这个问题的所在，花了我一个下午的时间！！！！

解决方案： 解决方案也有两种，

一种是一个用户程序独享一个段，这种跨段方式就保证了每个用户程序的sp虽然相同，但是段不同，因此不会共享同一个堆栈。

另一种解决方案是将 `sp` 初始化在每个程序的开头:

```
PCBlist[index].sp = offset-4;
```

其中 offset 是指当前代码偏移量，这种方法必须保证堆栈前面有足够的空间，否则会发生爆栈错误。但是这种方法的有点是我们可以同一个段中加载多个用户程序。

问题 2：有关内核 sp 的保存问题：

在我欣赏老师给出的代码以及代码原型 1，2 中，我发现均保存了内核中的 sp，这一点一开始让我很是疑惑，我觉得是没有必要的，与室友讨论后我才发现，因为每次调用 int 08h 时，我们会回到内核中，并且还会调用 Schdule 函数，为了保证不出错，我们必须不断的保存当前的内核 sp，不然容易出错，这虽然只是我的思考，但是却加深了我对 save 和 restart 的理解。

问题 3：创建新 PCBlist 的初始化问题

一开始我认为 PCBlist 的寄存器可以直接初始化为 0，但显然大错特错，首先段寄存器必须与 cs 寄存器保持一致，其次就是 ip 和 sp 这两者的初始化如下：

```
PCBlist[index].ip = offset;  
PCBlist[index].sp = offset-4;
```

此外，我们的 falgs 也应全部初始化为 1，因为这是一个必须全部置位的寄存器 Status 也必须为 1，因为 1 是运行态。

```
PCBlist[index].flags=512;  
PCBlist[index].status=1;
```

问题 4：有关 cli 和 sti 的坑

虽然老师上课时就一直强调了这个问题，但是老师比较聪明，故意把 restart 代码中的 sti 这一行删掉了。我一开始也没有意识到这个问题。因此在运行我的操作系统时，我发现我的代码只在进程表中切换了一次，这让我十分迷茫！我还以为是虚拟机坏了，后来我才意识到应该在代码的末尾添加一个 sti

```
INT_08H:  
cli  
call save  
push 0  
call _int_08h_override  
push 0  
call _Schedule  
jmp restart
```

在 restart 的最后两行：

```
sti  
iret
```

这样就保证了在进入时钟中断前 cli，出来后 sti，就完成了进程的切换。

问题 5：如何在 save 函数中只保存内核中的 sp？

一开始，我每次进入 save 函数中，我都会保存 sp 的值到 kernesp 中，但是这种方法显然不对，我意识到，kernesp 必须只能保存内核进程中的 sp，但是我们应该怎样获取/得知，当前进程是不是内核进程呢？

我通过一下几行代码实现了这个问题，一开始在老师写的进程控制块中，并没有 id 这个选项，但我修改了一下进程控制块，使得 id 指向自己的进程控制块序号。在 save 函数中

我通过获得这个序号并将其与 0 比较（因为 0 号进程就是内核进程），然后判断是否需要保存它的 sp，个人认为这种做法比较巧妙。

```
mov ax,word[si+4*15]
cmp ax,0 ;判断是否为内核的 PCB
jnz skipPCB ;如果不是内核 PCB
mov word[kernelsp],sp ;保存内核的 sp 至指定地址
skipPCB: ;跨段操作
```

问题 6：如何在内核中控制进程的运行和阻塞？

我想在内核中实现对进程的控制，我想出了这样的一种方法，通过判断键盘缓冲区的字符来决定，是否运行该程序，比如在我的 ball 四个用户级进程程序中，我输入 a 则运行/阻塞 a，输入 b 则运行/阻塞 b 进程……，最后在内核中通过判断是否输入了 p 来结束所有用户进程。

代码如下：

```
char t = isinput();
switch(t)
{
    case 'q':
        stop_pro();
        clear();
        break;
    case 'a':
        if(PCBlist[1].status == 0) PCBlist[1].status = 1;
        else PCBlist[1].status = 0;
        break;
    case 'b':
        if(PCBlist[2].status == 0) PCBlist[2].status = 1;
        else PCBlist[2].status = 0;
        break;
    case 'c':
        if(PCBlist[3].status == 0) PCBlist[3].status = 1;
        else PCBlist[3].status = 0;
        break;
    case 'd':
        if(PCBlist[4].status == 0) PCBlist[4].status = 1;
        else PCBlist[4].status = 0;
        break;
}
```

每次判断键盘输入时，我都会检查当前该字母指向进程的 status，然后对其取反，这样就实现了随叫随停，随叫随到的功能，最后按下 p，结束所有用户级进程

六 实验感想

本次实验可能代码量不大，但是却并不好写，因为 save 和 resart 函数比较难以实现，所以我花了很多的功夫。在这次的实验中，我在网上找到了很多的资料，通过分析这些资料（源码），我得出了 save 和 restart 程序的大致写法，可以见得网络对我们的重要性。

此外，有关一些十分隐藏的 bug 我们找不出来的时候，要善于询问同学，这次我就是有一个 bug 找不出来，问同学才问出来的。

本次实验的代码量大概 100 行，但是确实是十分的重要的，因为这次的代码，是我们以后多进程，多线程的基础，PCB 模块的实现是整个操作系统的基础。用户级与内核线程的分离是必须要走的路，虽然这条路不那么平坦。

其实这次实验对我的困难主要在于，我用的 nasm+gcc 的思路，然而老师的代码全是 tcc 的 16 位代码，这让我很是棘手，不能直接使用老师的 save 和 restart 函数，这虽然消耗了我很多时间，但是我对 save 和 restart 函数的理解却十分的深刻，可以说投入多少就可以收获多少。操作系统这门课的确很有意义。