实验六报告

【个人信息】

院系：数据科学与计算机学院　　专业、年级：17 级计算机科学与技术
学号：17341097　　姓名：廖永滨　　指导教师：凌应标

【实验题目】

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。

【实验目的】

1. 在内核实现多进程的二状态模，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。

【实验要求】

1. 实验六在实验五基础上进行，保留或扩展原有功能，实现部分新增功能。
2. 实现多进程

【实验方案】

1. 实验工具：

Notepad++：编写程序时使用的编辑器；
Subline：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

VmWare 虚拟机：创建裸机环境，生成虚拟磁盘

2. 实验基本操作方法：

a. 使用 TCC 编译命令：tcc -mt -c - o cfile.obj cfile.c >ccmsg.txt
b. TASM 汇编命令：tasm afile.asm afile.obj > amsg.txt
c. 链接命令：tlink /3 /t cfile.obj afile.obj , final.com
d. NASM 汇编指令：NASM afile.asm

3. 实验原理

中断机制基础知识，中断程序的编写规范。

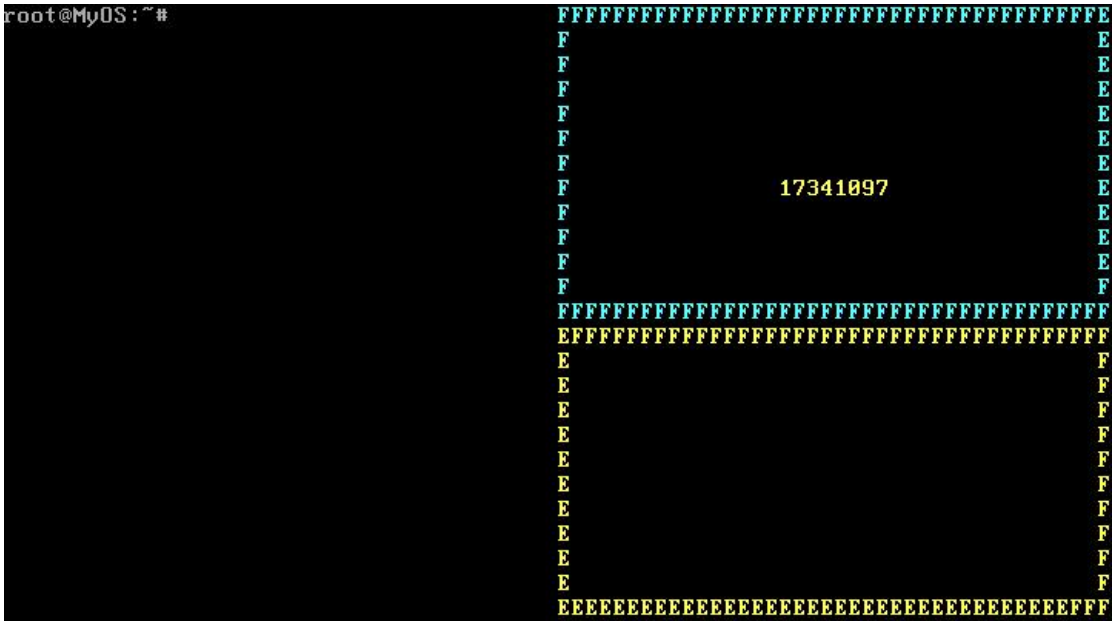
【实验过程】

1、实现的内容：

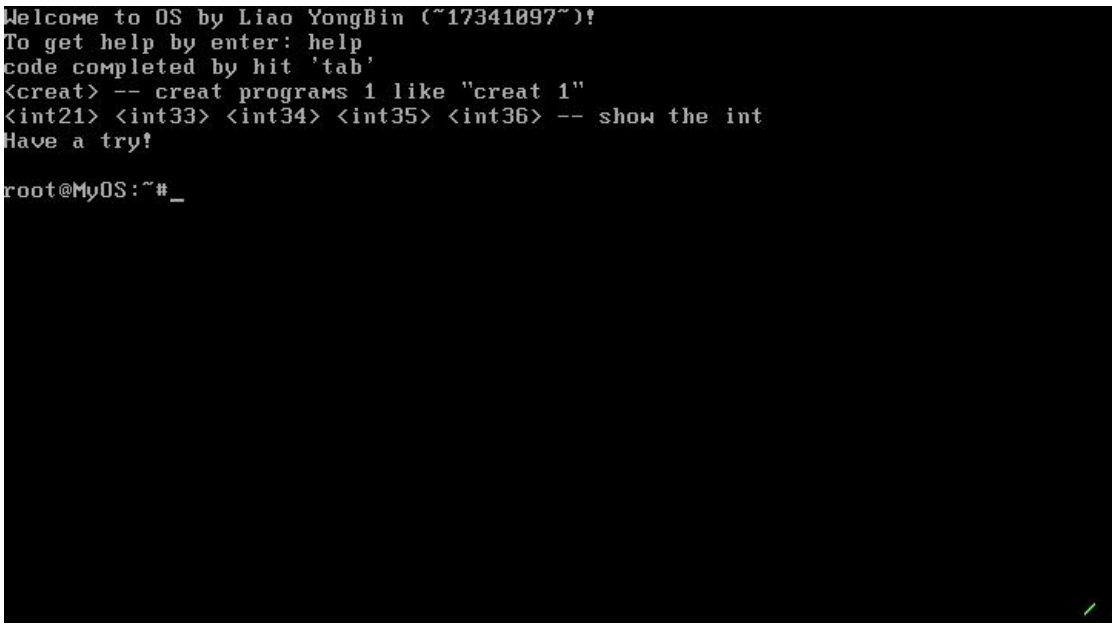
这次实验主要为在前面的基础上引入进程的概念，并以时间片轮转的方式进行进程调度。实验中用一个变量 kernel_mode 表示操作系统的运行状态，kernal_mode 等于 1 时为内核态，等于 0 时为用户态，时钟中断在进入操作系统时立刻启动，在时钟中断中，首先判断当前操作系统状态，如果为内核态则在右下角显示不断变换的字符，如果为用户态则在进程间保存和调度。进程的定义在 C 中实现，创建进程，保存进程控制块，调度算法也由 C 进行处理。当用户指定了执行的程序，比如 creat 1234,则分别将程序 1、程序 2、程序 3、程序 4 从磁盘载入到内存，然后调用 create_new_PCB 函数，创建新进程，当 4 个进程创建完毕，将 kernel_mode 设为 0，进入用户态，时钟中断将首先保存操作系统进程控制块信息,之后将在用户选择的进程间切换 - 保存当前进程控制块(save_PCB 函数)，调用轮转算法(schedule 函数)，得到下一个执行的进程，将要执行的进程的寄存器信息恢复到对应寄存器中。在轮转时，按任意键即可返回内核，此时时钟中断将结束轮转，设 kernel_mode 为 1，并恢复原保存的操作系统进程，进入内核态继续执行。

实现效果：

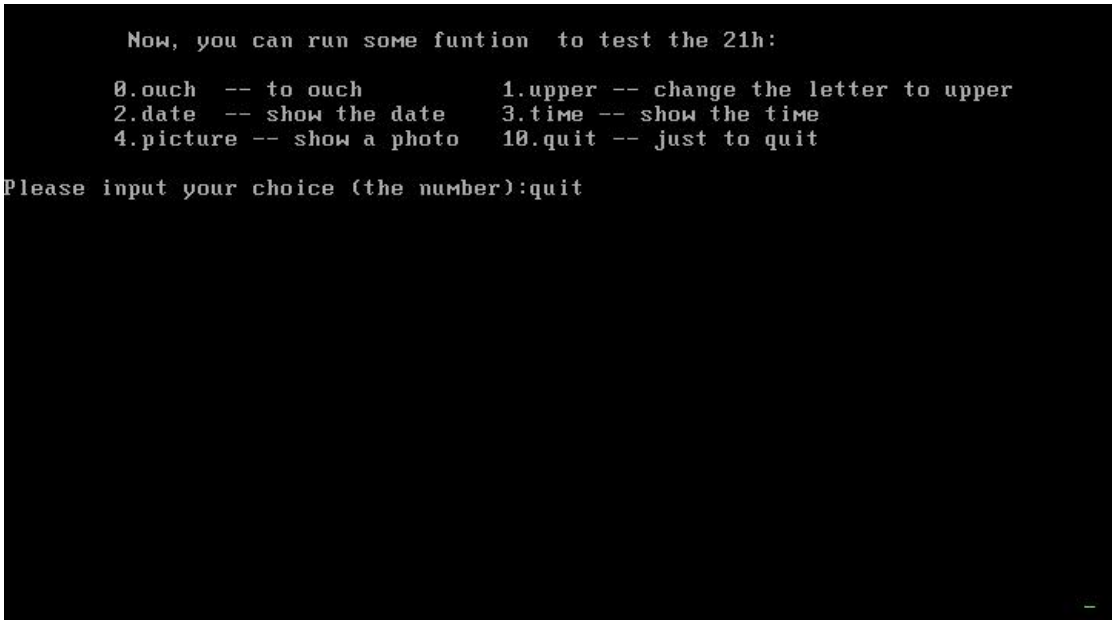
输入指令 Creat 123



输入指令 Creat 24



按任意键返回后



其余中断等等容易功能未曾改变，就连 run 指令我还留着，在上面的 ouch 我也留着

2、进程模型原理的详细说明：

（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，每个用户程序操作系统原理实验占用其中一个区，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。

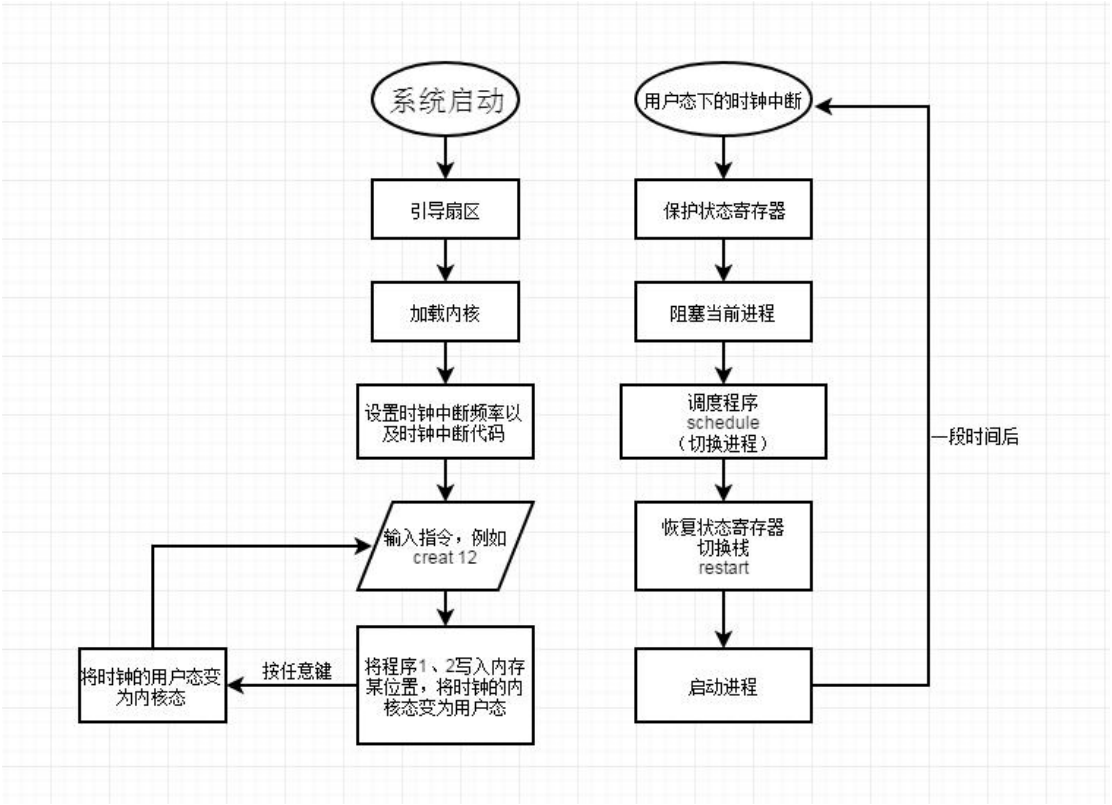
（4） 进程交替执行原理

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操作系统原理实验序，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）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。

3、程序流程图（进程模型的工作原理）



4、代码解析（实现方法）

(1)MyOS.asm：

基本无变化，新 include 了一个 process.asm 文件，用以添加进程相关代码

```
97    include process.asm
```

(2)services.asm：

此处为时间中断的核心所在，在实验五的基础上，新增用户程序态。在创建进程运行时，风火轮会不见，改换为用户进程态进行运作。

```
Timer:
    cmp word ptr [_kernal_mode], 1
    jne process_timer
    jmp kernal_timer
    
process_timer:
    .386
    push ss
    push gs
    push fs
    push es
    push ds
    .8086
    push di
    push si
    push bp
    push sp
    push dx
    push cx
    push bx
    push ax
    
    cmp word ptr [back_time], 0
    jnz time_to_go
    mov word ptr [back_time], 1
    mov word ptr [_current_process_number], 0
    mov word ptr [_kernal_mode], 1
    mov ax, 600h
    mov bx, 700h
    mov cx, 0
    mov dx, 184fh
    int 10h
    call _initial_PCB_settings
    call _PCB_Restore
    
time_to_go:
    inc word ptr [back_time]
    mov ax, cs
    mov ds, ax
    mov es, ax
    call _save_PCB
    call _schedule
    call _PCB_Restore
    iret
```

```
179  public _PCB_Restore
180  _PCB_Restore proc
181  ⟶mov ax, cs
182  ⟶mov ds, ax
183  ⟶call _get_current_process_PCB
184  ⟶mov si, ax
185  ⟶mov ss, word ptr ds:[si]
186  ⟶mov sp, word ptr ds:[si+2*7]
187  ⟶cmp word ptr [_first_time], 1
188  ⟶jnz next_time
189  ⟶mov word ptr [_first_time], 0
190  ⟶jmp start_PCB
191  ⟶
192  next_time:
193  ⟶add sp, 11*2 ⟶ ⟶ ⟶ ⟶ ⟶ ⟶
194  ⟶
195  start_PCB:
196  ⟶mov ax, 0
197  ⟶push word ptr ds:[si+2*15]
198  ⟶push word ptr ds:[si+2*14]
199  ⟶push word ptr ds:[si+2*13]
200  ⟶
201  ⟶mov ax, word ptr ds:[si+2*12]
202  ⟶mov cx, word ptr ds:[si+2*11]
203  ⟶mov dx, word ptr ds:[si+2*10]
204  ⟶mov bx, word ptr ds:[si+2*9]
205  ⟶mov bp, word ptr ds:[si+2*8]
206  ⟶mov di, word ptr ds:[si+2*5]
207  ⟶mov es, word ptr ds:[si+2*3]
208  ⟶.386
209  ⟶mov fs, word ptr ds:[si+2*2]
210  ⟶mov gs, word ptr ds:[si+2*1]
211  ⟶.8086
212  ⟶push word ptr ds:[si+2*4]
213  ⟶push word ptr ds:[si+2*6]
214  ⟶pop si
215  ⟶pop ds
216  ⟶
217  process_timer_end:
218  ⟶push ax
219  ⟶mov al, 20h
220  ⟶out 20h, al
221  ⟶out 0A0h, al
222  ⟶pop ax
223  ⟶iret
224  endp _PCB_Restore
225  ⟶
226  kernal_timer:
227  ····push ax
228  ⟶push bx
229  ⟶push cx
230  ⟶push dx
231  ⟶push bp
232  ····push es
233
```

……

后面的 kernal_timer 即为实验五的时钟中断，此处不再描述。

可以看到，每次调用时钟中断时会先判断 kernal_mode 变量的值，如果是内核态就跑风火轮，如果不是则进行用户态轮转！跑用户态时，参照上面给的程序流程图进行运行。此处最大的问题在于没有保护好用户进程的相关数据，但是解决方法也很简单，每个过程中在不涉及进程数据时要 push 和 pop，防止修改关键的地址数据等等。（此处我是参照学长的代码根据自己代码的需要进行修改的）

(3)process.asm ：

新增的进程相关的一系列汇编函数与代码。

```
;============================================================================
;--->--->--->--->--->void _set_timer()
;============================================================================
public _set_timer
_set_timer proc
    push ax
    mov al, 36h
    out 43h, al
    mov ax, 11931--->--->;频率为100Hz
    out 40h, al
    mov al, ah
    out 40h, al
    pop ax
    ret
_set_timer endp
;============================================================================
;--->--->--->--->--->void _set_clock()
;============================================================================
public _set_clock
_set_clock proc
    push es
    call near ptr _set_timer
    xor ax, ax
    mov es, ax
    mov word ptr es:[20h], offset Timer
    mov word ptr es:[22h], cs
    pop es
    ret
_set_clock endp

back_time dw 1
;============================================================================
;--->--->--->--->--->void _run_process(int start, int seg)
;============================================================================
public _run_process
_run_process proc
    push es
    --->
    mov ax, word ptr [_current_seg]
    mov es, ax
    mov bx, 100h
    mov ah, 2
    mov al, 1
    mov dl, 0
    mov dh, 1
    mov ch, 0
    mov cl, byte ptr [_sector_number]
    int 13h
    --->
    call _create_new_PCB
    --->
    pop es
    ret
_run_process endp
```

第一个函数是简单的设置中断频率函数，直接参照老师给的就行。第二个函数是设置时间中断的函数，早在风火轮那一次实验就已经解决了。而第三个函数是创建进程相关的函数，其实就是将软盘里指定位置的代码放到对应的内存空间里方便系统调度。

(4)kernal.c：

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

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

typedef struct PCB{
    Register regs;
    PCB_STATUS status;
    int ID;
} PCB;

PCB PCB_LIST[8];

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;

PCB *get_current_process_PCB() {
    return &PCB_LIST[current_process_number];
}
```

```
44
45  void save_PCB(int ax, int bx, int cx, int dx, int sp, int bp,
46      int si, int di, int ds, int es, int fs, int gs, int ss, int ip, int cs, int flags) {
47      current_process_PCB_ptr = get_current_process_PCB();
48
49      current_process_PCB_ptr->regs.ss = ss;
50      current_process_PCB_ptr->regs.gs = gs;
51      current_process_PCB_ptr->regs.fs = fs;
52      current_process_PCB_ptr->regs.es = es;
53      current_process_PCB_ptr->regs.ds = ds;
54      current_process_PCB_ptr->regs.di = di;
55      current_process_PCB_ptr->regs.si = si;
56      current_process_PCB_ptr->regs.sp = sp;
57      current_process_PCB_ptr->regs.bp = bp;
58      current_process_PCB_ptr->regs.bx = bx;
59      current_process_PCB_ptr->regs.dx = dx;
60      current_process_PCB_ptr->regs.cx = cx;
61      current_process_PCB_ptr->regs.ax = ax;
62      current_process_PCB_ptr->regs.ip = ip;
63      current_process_PCB_ptr->regs.cs = cs;
64      current_process_PCB_ptr->regs.flags = flags;
65  }
66
67  void schedule() {
68      if (current_process_PCB_ptr->status == PCB_READY) {
69          first_time = 1;
70          current_process_PCB_ptr->status = PCB_RUNNING;
71          return;
72      }
73      current_process_PCB_ptr->status = PCB_BLOCKED;
74      current_process_number++;
75      if (current_process_number >= process_number) current_process_number = 0;
76      current_process_PCB_ptr = get_current_process_PCB();
77      if (current_process_PCB_ptr->status == PCB_READY) first_time = 1;
78      current_process_PCB_ptr->status = PCB_RUNNING;
79      return;
80  }
81
82  void PCB_initial(PCB *ptr, int process_ID, int seg) {
83      ptr->ID = process_ID;
84      ptr->status = PCB_READY;
85      ptr->regs.gs = 0x0B800;
86      ptr->regs.es = seg;
87      ptr->regs.ds = seg;
88      ptr->regs.fs = seg;
89      ptr->regs.ss = seg;
```

```
90      ptr->regs.cs = seg;
91      ptr->regs.di = 0;
92      ptr->regs.si = 0;
93      ptr->regs.bp = 0;
94      ptr->regs.sp = 0x0100 - 4;
95      ptr->regs.bx = 0;
96      ptr->regs.ax = 0;
97      ptr->regs.cx = 0;
98      ptr->regs.dx = 0;
99      ptr->regs.ip = 0x0100;
100     ptr->regs.flags = 512;
101 }
102
103 void create_new_PCB() {
104     if (process_number > 8) return;
105     PCB_initial(&PCB_LIST[process_number], process_number, current_seg);
106     process_number++;
107     current_seg += 0x1000;
108 }
109
110 void initial_PCB_settings() {
111     process_number = 0;
112     current_process_number = 0;
113     current_seg = 0x1000;
114 }
```

```
char sector_number;
void create_process(char *comm) {
    int i, sum = 0, flag = 0;
    for (i = 5; i < strlen(comm); ++i) {
        if (comm[i] == ' ' || comm[i] >= '1' && comm[i] <= '4') continue;
        else {
            print("  invalid program number: ");
            printChar(comm[i]);
            print("\n\n\r");
            return;
        }
    }
    for (i = 5; i < strlen(comm); ++i) {
        if (comm[i] != ' ') flag = 1;
    }
    if (flag == 0) {
        print("  invalid input\n\n\r");
        return;
    }
    run_process(13, current_seg);
    for (i = 5; i < strlen(comm) && sum < 8; ++i) {
        if (comm[i] == ' ') continue;
        sum++;
        sector_number = comm[i] - '0' + 12;
        run_process(sector_number, current_seg);
    }
    kernal_mode = 0;
}
```

这一部分代码解释十分繁琐，基本参照老师给的样例进行实现。说实话我也看不怎么明白，我是在样例的基础上不断修修补补的，最后能成功运行的具体原因我也不太明白。

【实验心得】

这次实验，最大的收获无异于对时间中断以及进程模型的了解与实现。

在实验五的基础之上，实验六的完成还是十分十分困难，因为是完全不想管的事情。我不得不参照学长的代码才能成功攻克难关。之间也遇到了很多很多问题（因为学长的代码有 bug……）。

首先，我遇到了退不回内核态的问题。也就是回不到风火轮的状态，风火轮在轮转完进程后就不再出现。一开始解决方法很牵强，就是每次回去都强制改变 kernal_model 的值，后面发现是时间中断的代码有 bug，才彻底解决问题。

接着，我发现运行的子程序有时候不能按键退出，会卡死。这个问题困扰了我很久很久。我在观察别的同学的代码后才发现问题所在。其实就是时间中断那里漏了关键的中断发送信号。就是那个发送 EOI 到主 8529A 那一段代码漏掉了。

在这两个问题解决的基础上，实验 6 基本上就完成了。

总的来说，实验六非常困难，先是无从下手，再是参考别人代码后还 bug 不断。幸好是解决了，但是原理自己还没怎么想明白。（一开始还打算搞个背景音乐的，网上有很多相关的 MASM 代码可以考虑移植，但是我用的是 NASM，移植难度有点大，经常出现卡死后 BB 的响个不停，所以暂时放弃了，实验七我会尝试补上）

由于实验 5 的所有内容我一个没删除，所以内核很膨胀，所有实验 5 能用的指令，实验 6 中依旧能用，我会考虑在实验 7 中删掉一些。

【代码清单】

Project6

--操作系统
    --OS.COM
    --KERNAL.C
    --KLIBA.ASM
    --MYOS.ASM
    --services.asm
    --clib.asm
    --process.asm
--工具包
    --DOSBox.exe（只是快捷方式）
    --mydoc.conf (DOSBOX 的配置文件)
    --nasm.exe
    --nasmpath.bat
    --TASM.EXE
    --TCC.EXE
    --TLINK.EXT
--软盘文件
    --实验 6.flp
--引导程序
    --loading.asm
    --loading.com
--用户程序
    --a1
    --a1.asm
    --a2
    --a2.asm
    --a3
    --a3.asm
    --a4
    --a4.asm
    --process1
    --process1.asm
    --process2
    --process2.asm
    --process3
    --process3.asm
    --process4
    --process4.asm