实验七：进程切换交替

实验目的：

1. 在内核实现多京城的二状态模型，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：

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，进程数量为4或更多。
2. 修改内核，可通过用户命令选择加载1~4个用户程序运行，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，展示你的所有系统调用服务还能工作。

实验方案：

1. 实验环境：

- 实验运行环境:MacOS
- 调试用虚拟机:bochs
- 虚拟机软件:VMware Fusion

2. 实验工具：

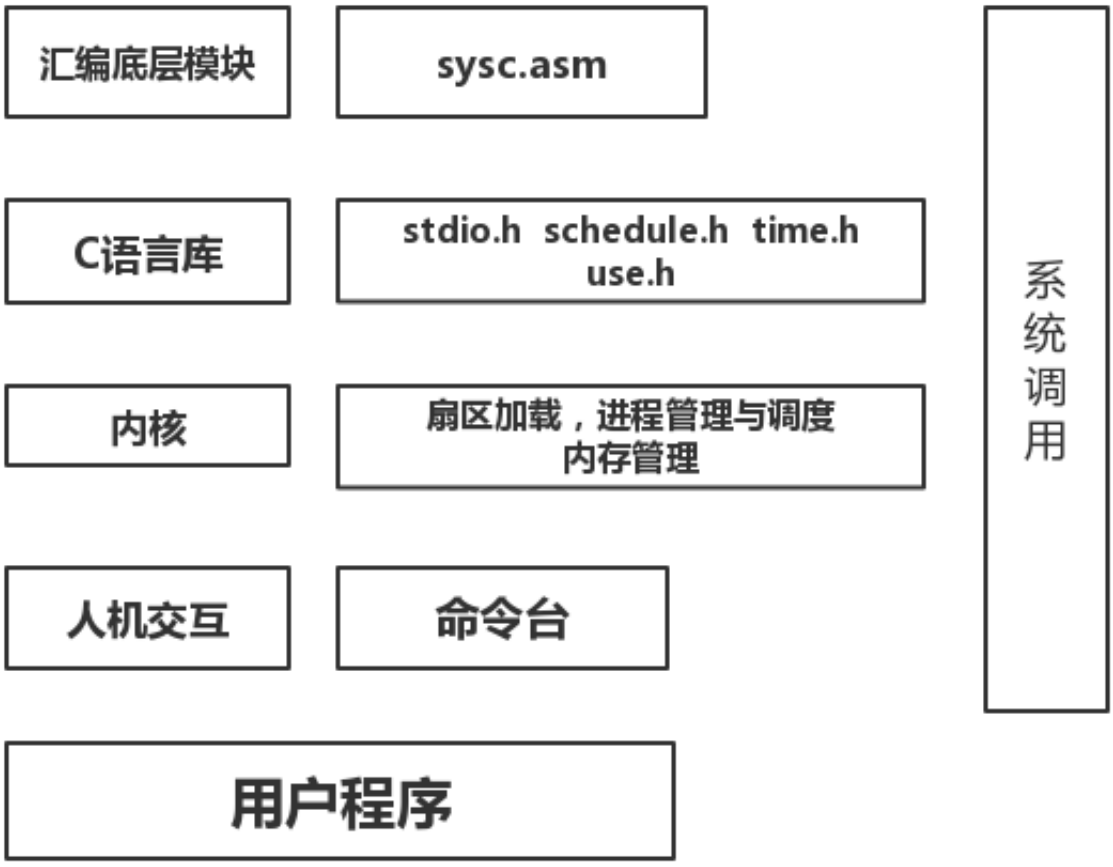
- 汇编语言:NASM汇编语言 C语言
- 编译器:gcc 4.9.2
- 链接工具:GNU ld
- 反汇编工具：ObjDump
- 构建工具:GNU make语言
- 调试工具:Bochs
- 合并bin文件到统一磁盘映像工具：dd

3. 使用了ObjDump来生成反汇编文件，

- 为了来方便debug，在编译的同时生成反汇编文件。
- 生成指令方法：

```
kernel.txt: kernel.elf
$(-melf_i386 -static -nostdlib --nmagic) -S kernel.elf > kernel.txt
```

4. 系统架构：



具体实现

1.PCB结构：

- 这次实验我们需要实现进程之间的互相切换,所以我们在切换进程之前就需要我们保存当前进程的所有寄存器以及堆栈，也就是进程的上下文。所以我们必须引入结构：PCB。
- PCB结构：

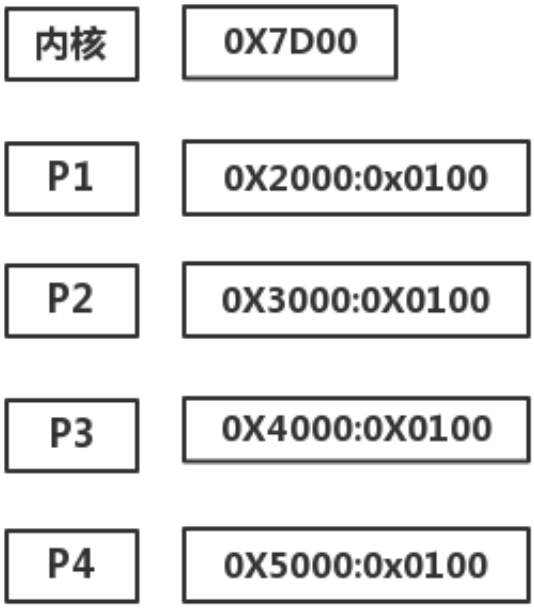
```
struct PCB{
    unsigned int eax;
    unsigned int ebx;
    unsigned int ecx;
    unsigned int edx;
    unsigned int esi;
    unsigned int edi;
    unsigned int ebp;
    int es;
    int ds;
    int ss;
    unsigned int esp;
    int ip;
    int cs;
    int eflags;
    int prg_status;
    char prg_name;
};
```

- 由于使用的是gcc和nasm的交叉编译，由于gcc使用的是32位代码，所以8个通用寄存器都是32位的寄存器，因此为了保证进程间切换时的寄存器不受污染，所以我们必须保存8个32位通用寄存器。除了通用寄存器是32位的，其他的段寄存器以及ip虽然是16位的，但是在gcc汇编中，也是将他们作为32位的。整个PCB结构是0x40个字节。

2.将进程安置在不同的段中：

- 之前的实验，为了方便，我的内核与用户程序都是安置在同一个段中，经常需要担心内核程序被用户程序覆盖。而在这次的实验中，为了避免用户程序影响内核程序，所以直接将几个用户程序安排在独立的段中。

- 内存安排：



- 由于时钟中断中，我们设立的进程PCB里面包括需要运行的程序的CS和IP，所以我们可以直接在初始化进程PCB时将他们的CS，IP进行改变即可。
- 但如果我们需要不通过时间中断进行跨段跳转时，我们可以将CS：IP放在内存中的一个双字大小的位置中，用call far指令实现段间跳转。

```
mov word [program_saved],0x100;save the program address
mov word [program_saved+2],bx
mov dword [esp_saved_inkernel],esp
call far [es:program_saved]
```

- 除了内存的代码段分配之外，考虑到了栈的大小问题，我还把每个用户程序的堆栈和内核的堆栈分开。每个用户程序的堆栈段和它自身的代码段式一样的。这样可以保证用户进程栈、内核栈以及内核进程栈的独立。

3.进程切换的具体实现.

- 首先我们必须清楚我们设立的PCBList[]他是存在内存中的一个结构数组，要想使用到我们设立的这个结构数组，我们必须设立一个全局指针变量来指向我们这个结构数组，利用这个指针在内核里面来对进程控制块进行改变。否则我们无法找到PCBList[]在内存里面的存储位置，也就无法对进程中的状态进行保存。
- 此外，我在思考PCBList[]的相关实现的时候，是这么认为的，我们PCBList[]在内存中的位置其实可以被认为是在内核中的进程栈，这个栈只有一个作用，就是帮助我们存储用户进程的各种寄存器。
- 了解了这些我们就可以明白**save**与**restart**的实现。

> save也就是将我们用户进程的各种寄存器和状态按照PCB结构定义的顺序保存在内核的进程栈中。
>
> 而restart则是将存储在内核进程栈中的通用寄存器的值以及状态赋回给寄存器，并将eflags，cs，ip按顺序压到CS段寄存器，ip寄存器，利用iret返回。

代码实现就非常的简单。

```
_save:
mov dword [esp_saved_inkernel],esp;//new added,in order to enable interupt
push ds;用户ds
push cs;
pop ds
pop word [ds_saved]
pop word [return_save]
mov dword [kernelesp_saved],esp
mov dword [esi_save],esi
mov esi,dword [_CurrentProg]
add esi,44
pop word [esi];save eip
;mov word [esi+2],0
pop word [esi+4];save cs
mov word [esi+6],0
pop word [esi+8];save eflags
mov word [esi+10],0
mov dword [esi - 4],esp
mov word [esi - 8],ss
mov si,ds
mov ss,si
mov esp,dword [_CurrentProg]; mov esp, dword ptr ds:0x9d40
add esp,36
push word 0
push word [ds_saved]
push word 0
push es
push ebp
push edi
push dword [esi_save]
push edx
push ecx
push ebx
push eax
mov esp,dword [kernelesp_saved];返回内核栈执行schedule
mov ax,word [return_save]
jmp ax
```

```
_restart:
mov dword [kernelesp_saved],esp
mov esp,dword [_CurrentProg]
pop eax
pop ebx
pop ecx
pop edx
pop esi
pop edi
pop ebp
pop es;系统数据段
pop word [temp];in order to balance
pop word [ds_saved]
pop word [temp];in order to balance
mov dword [esi_save],esi;ds is the kernel'ds
pop ss
pop word [temp];in order to balance
mov esi,esp;between ss and esp,the next one is esp
mov esp,dword [esi];把esp的存着的值给esp
push word [esi + 12];eflags
push word [esi + 8];cs
push word [esi + 4];eip
mov esi,dword [esi_save]
mov ds,word [ds_saved]
push ax
mov al,20h
out 20h,al
out 0A0H,al
pop ax
iret
```

- 而我们的**初始化PCB函数(initial())**，其实也就是将加载入内存的几个用户进程的CS，IP进行赋值，**并将用户程序的状态只为RUN态，用户程序的状态这一个结构中的成员也是决定我们操作系统采用何种状态模型的基础**,在这次实验中我们使用的是二状态模型，因此可以状态为RUN或者EXIT。 代码如下：

```
void initial_PCB(int index){
PCB_list[index - 1].cs = 0x2000+0x1000*(index - 1);//6代表内核，0代表用户程序
PCB_list[index - 1].ds = 0x2000+0x1000*(index - 1);//6代表内核，0代表用户程序
PCB_list[index - 1].ip = 0x100;
PCB_list[index - 1].prg_status = RUN;
PCB_list[index - 1].eflags = 512;
PCB_list[index - 1].prg_name = '1'+index - 1;
}
```

- 然后就是我们的**时间片轮转的调度函数(schedule())**了，由于我们在PCB中定义了进程的状态，所以在调度函数中，我们可以通过检查进程的状态来决定下一时刻的用户进程是哪一个。

```
void sys_schedule(){
    int i,j;
    if(PCB_list  == _CurrentProg ){
        i = 0;
    }
    else if(PCB_list + 1 == _CurrentProg){
        i = 1;
    }
    else if(PCB_list + 2 == _CurrentProg){
        i = 2;
    }
    else if(PCB_list + 3 == _CurrentProg){
        i = 3;
    }
    else if(PCB_list + 4 == _CurrentProg){
        i = 3;
    }
    else{
        i = 3;
    }
    for(j = 0;j < 4;j ++){
        if(i == 3){
            _CurrentProg = PCB_list;
            i = 0;
        }
        else{
            _CurrentProg ++;
            i ++;
        }
        if(_CurrentProg -> prg_status == RUN){
            return;
        }
    }
    _CurrentProg = PCB_list + 5;//返回内核
    return;
}
```

- 最后实现了多道批处理程序，我们可以自由的选择哪几个用户进程来共同执行。

```
void run(int queue[],int size){
    int j = 0;
    clearscreen();

    for(j = 0;j < size;j ++){
        initial_PCB(queue[j]);
    }
   // clearscreen();;
    thread_join();
    //Initial_Int_08h();
}
```

4.返回内核

- 在这次实验中，虽然没有要求实现从用户进程返回内核，但是由于在进程控制块中设立了状态这个结构成员，使得一个进程的结束成为了可能。不同于之前的进程按键退出，本次实验的进程退出是用定时退出。
- 我的思路是这样的，一开始我们打开时钟中断，第一次时钟中断会将我们内核的状态存到相应的进程控制块的结构数组中，当我们最后需要返回时，只需要将指针指向内核的进程控制块中即可。然后用恢复各个寄存器及状态即可。
- 在这里将结束进程的函数用中断函数调用的形式实现。

```
void sys_exit(){
    _CurrentProg -> prg_status = EXIT;
}
```

```
_SetINT38h:
    CLI
    push ax
    push ds
    mov ax,cs
    mov ds,ax
    push word 0
    call sys_exit
    pop ds
    pop ax
    iret
```

Ctime库实现

- 函数包括：

```
int Get_Hours();
int Get_Minutes();
void printTime();
```

- 实现思路：

首先在汇编中实现读取端口的函数，将时和分从端口中读取出来，然后作为返回值返回到C函数中。定义的函数如下：

```
extern int _Get_Hours_1();
extern int _Get_Hours_2();
extern int _Get_Minutes_1();
extern int _Get_Minutes_2();
```

然后在C中利用数学关系将读出来的数据进行转换，得出真正的24小时制的时分值。

实验难点和解决方案

1.堆栈统一

- 由于每个用户进程的堆栈段与内核的堆栈段是不同的，而我们使用系统调用的时候都是在内核执行的，所以，我们在运行用户进程时想要调用系统调用，或者中断的时候，必须要把用户程序堆栈和内核的堆栈进行统一。 方法如下：每次进入系统调用之前首先先将用户进程的堆栈(包括堆栈段值ss和偏移量esp进行保存），然后将save过程所保存的kernel_esp的值传给内核的esp，然后才开始执行系统调用代码。在执行完系统调用之后，再还原用户的堆栈值。

```
mov word [ss_saved_in_user],ss;存在内核中
mov ss,ax
mov dword [esp_saved_in_user],esp;in order to protect kernel's ss and turn back to k
mov esp,dword [esp_saved_inkernel]
push word 0
call sys_printheart
mov ss,word[ss_saved_in_user];还原用户ss
mov esp,dword[esp_saved_in_user]
```

2.遇到的问题：用户进程与内核同时执行

- 在这次实验中，当我的内核程序让权给用户程序执行之后，我的内核程序依旧继续执行，在改进了我的调度函数之后，仍旧会出现这种情况，我做过的尝试包括在跳入用户程序之前将内核的PCB控制块中的状态置为退出态，然后当用户进程执行完之后再次置为运行态；还试过只有用户进程的状态所有都为退出态时才会跳入内核程序执行。

- 但是在这些尝试都无法解决这个看似并行运行的bug。后来，**我参考了在多线程编程里面一个重要的函数join(),这个函数的作用就是等待所有的子进程执行完，才开始执行主进程。**
- 我的实现思路是这样的：首先是实现一个函数去计算用户进程里面正在运行的进程个数(counter())。而join函数就是一个死循环，只有当进程个数为0时，才会跳出循环，否则就会一直在循环里面等待所有用户进程的死亡。

  代码如下：

```
void thread_join(){
    while(counter()){
    }
}
int counter(){
    int i = 0;
    int count = 0;
    for(i = 0;i < PCB_NUMMER - 1;i ++){
        if(PCB_list[i].prg_status == 1){
            count ++;
        }
    }
    return count;
}
```

3. 结构指针声明出错，导致无法正确定位到PCBList

- 上面也说过，我使用了指针来指向进程控制块的结构数组，这样才能正确的进行保存和改变。
- 但是在一开始实现的时候，我并没有将指针作为全局变量，而是在一个函数里面单独声明，后来经过不断的在bochs调试之后才发现原来我的进程控制块指针指向的那块区域里的数据不是正确的，进一步退出我的指针声明的有问题，导致初始化出现问题。经过改正后程序就可以正常的运转。

4. 没有意识到寄存器在定义的PCB结构中的位置就是实际在内存中的位置

- 这会导致restart过程中，会将俩个成员变量赋给相反位置的寄存器。
- 同样适用bochs进行调试。

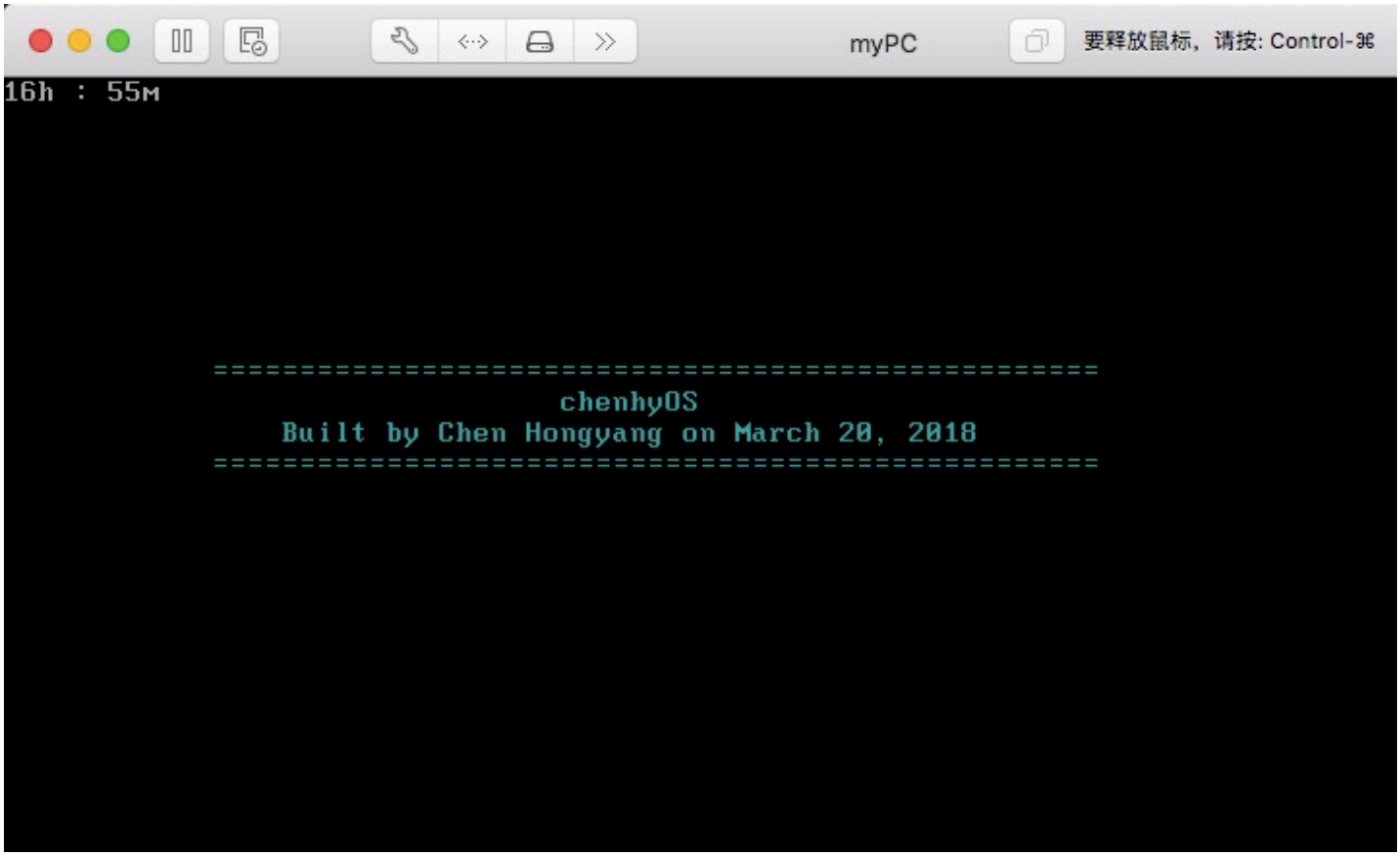
5. 进程调度过程的调试

- 这次实验，的确是要求我们利用时钟中断来进行进程的切换。但是时钟中断作为一个硬件中断，我们是无法进行调试的。
- 在实验的过程中，为了方便调试，我另外设立了一个中断，将进程切换的save函数，schedule函数，restart函数按照顺序放入那个中断中，然后调用这个中断，开启bochs调试来一个函数一个函数的调

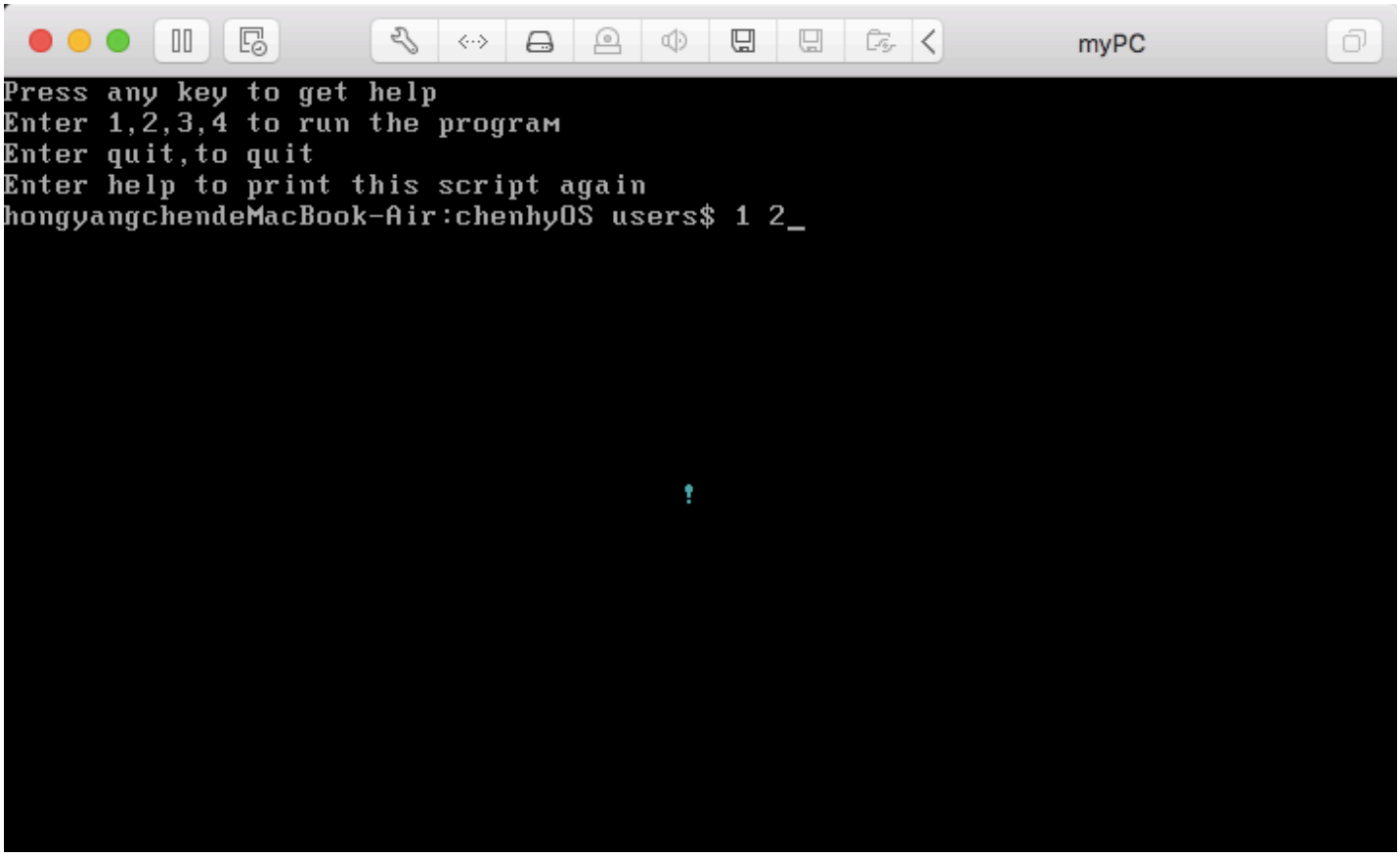
试，只要最后能够这个中断跳入用户程序进行执行，就能够说明，我们的调度函数是正确的，可以移植到时钟中断。

实验结果截图

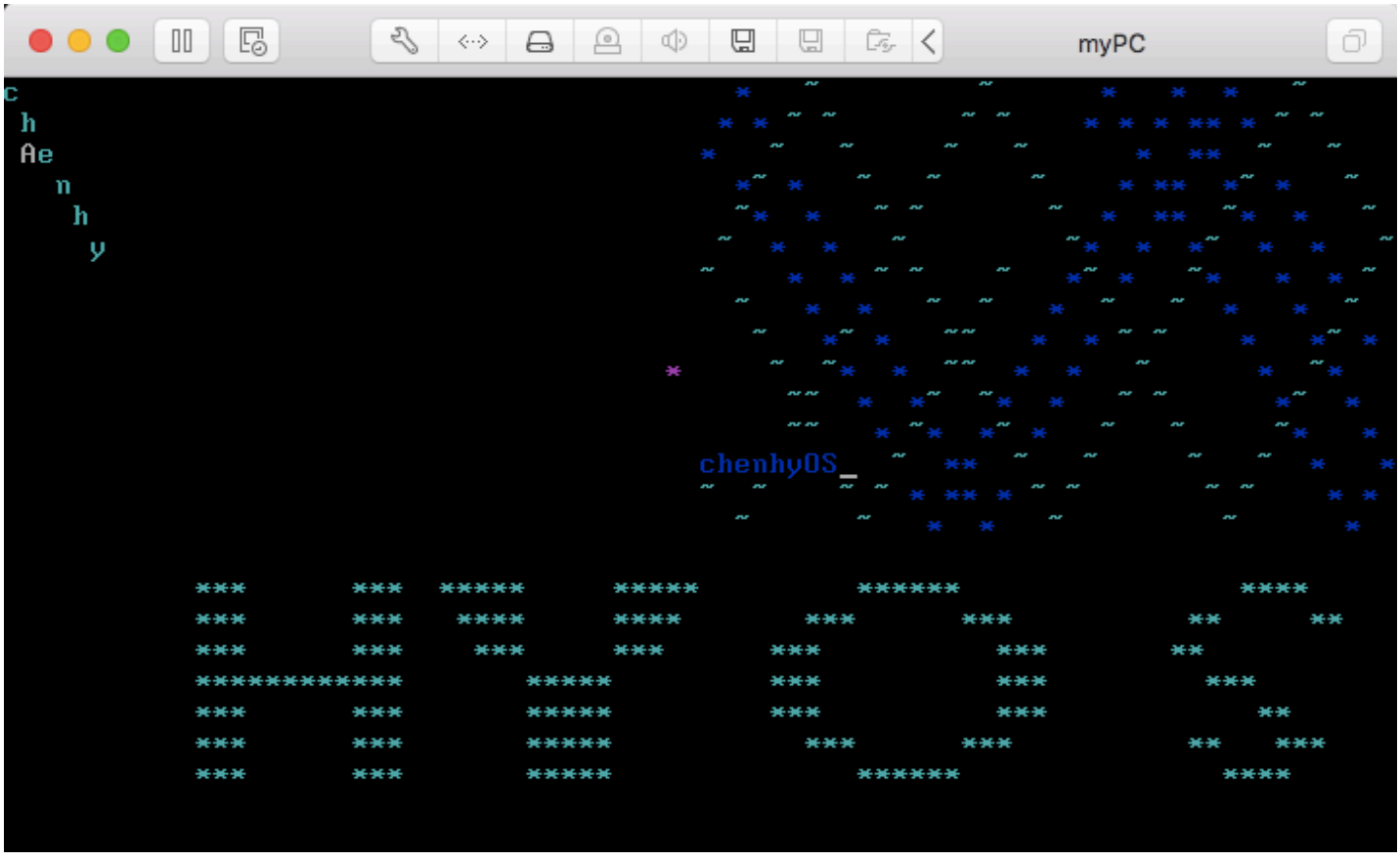
- 开机界面显示时间



- 进程选择（多道批处理）

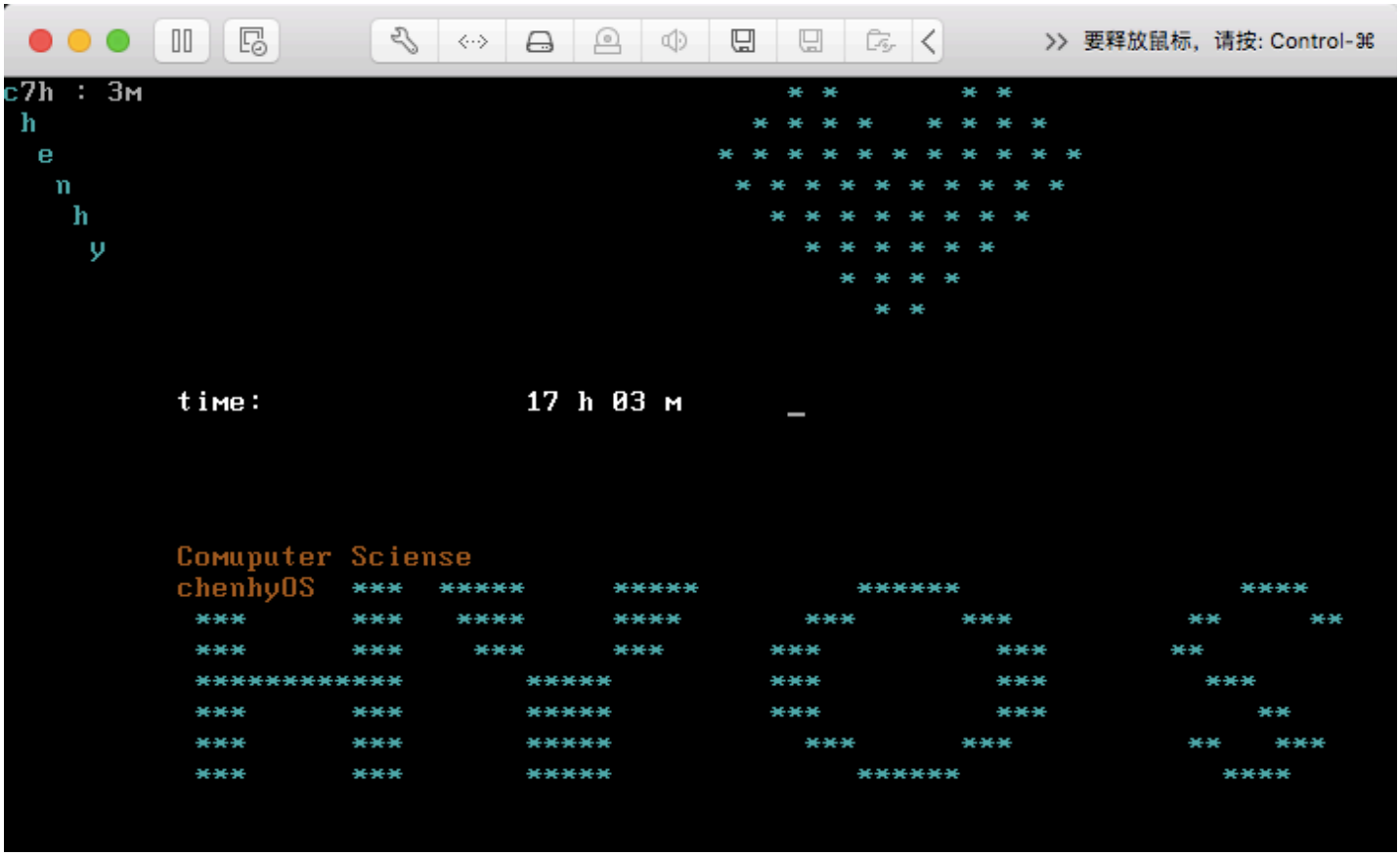


- 进程交替运行



- 系统调用正常

（33h：显示斜体字符串，34h：显示HYOS，35h：显示个人信息，36h显示爱心,37h: 显示系统时间）



实验感想

这次实验，其实跟之前的实验相比，实现的步骤其实不是很复杂。只是要把原理想清楚了再动手，这样实现起来会更加有方向。

而在写这次实验的时候，我觉得最重要的就是掌握调试的方法。如果一上来实现了整个程序调度之后就将代码放到时钟中断去运行，很大概率是不能成功。同时又因为时钟中断无法进行调试，也就是程序的堆栈变化，寄存器的存取，我们都无法看出来。所以想出一个方便的明显的调试方法对于实验的成功是很有必要的。

而在这次实验中，我选择先将调度程序放在一个可以调用的中断中，然后在内核中调用这个中断，然后在调度函数里不断观察每一个重要步骤的堆栈值和寄存器的值。只要能够保证将进程中的所有寄存器保存起来，以及将下一个进程的所有寄存器进行恢复，同时保证堆栈的平衡，基本上这个实验就能够成功。

但是在这次实验中，要想实现从用户程序回到内核程序并不是一件简单的事情，首先需要确保用户程序的状态能够正常的返回，同时调度程序还能将系统的控制权交还给内核程序。只有这样才能保证内核的继续执行。除此之外，由于用户进程关闭的比较慢，所以在我们看来，内核程序与用户程序是并发同步执行的，为了解决这个问题，我曾经试过在用户程序执行完之前增加一个输入函数作为阻塞，但是这样并不是一个合理的方法。后来学习了多线程编程的函数思想，新增了join()函数。这样的经历也让我意识到多学习一些函数实现的思想是非常有用的。

虽然这次实验基本完成了要求，但是还是有一些瑕疵。首先由于内核和用户进程的堆栈分离，导致每次调用键盘中断之后，无法定位到正确的内核堆栈值，对于这个问题，我暂时还没有特别好的解决方法。除此之外，原来的时钟中断已经完全作为调度函数的中断，原有的显示风车的功能也被砍了。但是对于这个问题，我还是有实现的思路，我们在执行运行用户程序之前可以先将显示风车的功能函数hack起来，然后在运行完用户程序返回内核之后，再恢复。这样就能做到在内核中可以运行。除此之外，我曾经尝试将写在底层的汇编函数模块按照功能划分为多个模块，但是可能没有处理好公有变量的问题，这次尝试并没有成功。

在做完了这次实验，让我对进程交替的概念理解的更加深入，让我对接下来的五状态模型也更加有信心。