

# 实验报告 实验六

姓名：王钦 学号：13349112 班级：计科二班

## 实验目的

1. 学习进程模型知识，掌握进程模型的实现方法。
2. 利用时钟中断，设计时钟中断处理进行进程交替执行
3. 扩展MyOS，实现多进程模型的原型操作系统

## 实验内容

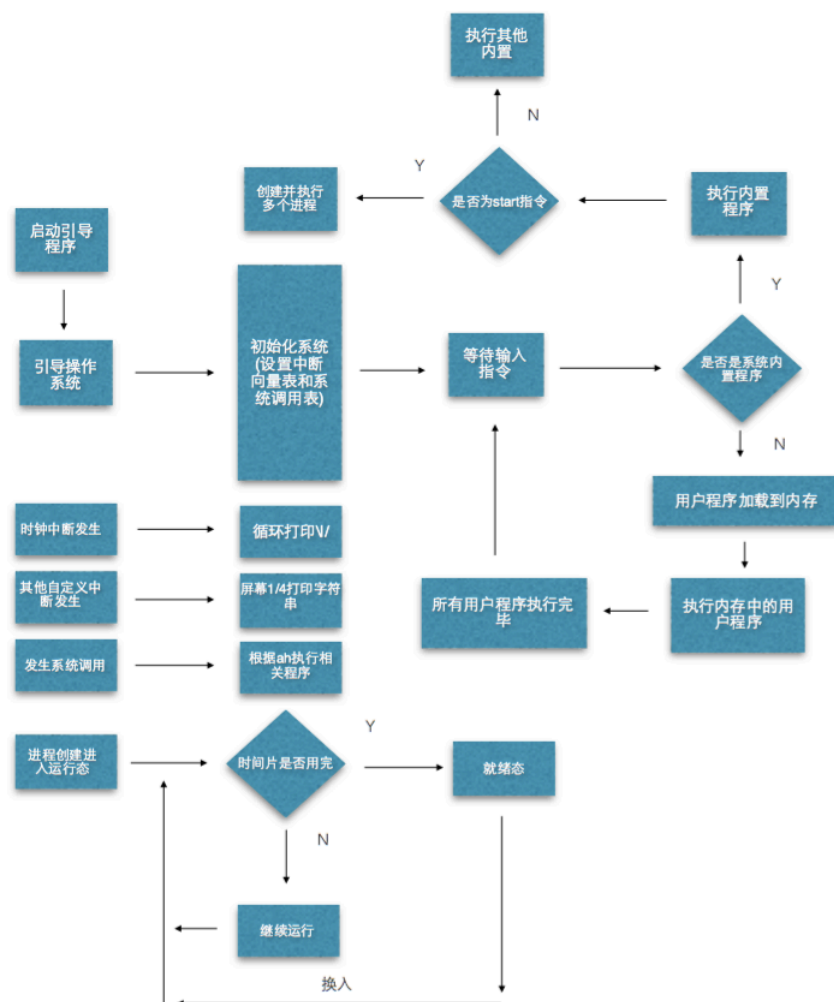
在实验五的基础上，进化你的原型操作系统，原型保留原有特征的基础上，设计满足下列要求的新原型操作系统：

- (1)内核实现简单进程模型，进程具有就绪、运行两种基本状态。建议在c程序中定义进程表，进程数量最多4个。
- (2)内核可以一次性加载最多4个用户程序。用户进程采用时间片轮转调度进程。由你设计有个性的用户程序，它们的输出各占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4个用户程序，展示系统调用服务还能工作。

## 实验平台

dd+gcc+ld+nasm+Linux+vim

### 算法流程图



## 功能一览

### 1. 系统内置功能:

terminal, 装载内核shell, 为用户提供一个与操作系统交互的工具, 开机后自动进入, 以下所有功能都在terminal中交互

date, 显示当前日期

time, 显示当前时间

asc, 显示一个字符的asc码

clear, 清除当前屏幕所有字符, 刷新屏幕

help, 显示系统帮助信息

`man` ,显示内置函数的帮助信息,比如 `man date` ,显示date的相关帮助

python , python 扩展, 类似python命令行工具, 可以使用这

个工具输入计算表达式返回计算结果，目前只支持加法减法

`start`, 开始创建并执行四个进程并且每秒18.2次的调度，分别在屏幕1/4处打印一些个性化信息( 不同配置的虚拟机动画速度不一样，建议使用vmware测试)

## 2. 用户程序:

`run`, 软盘中含有两个用户程序,输入`run 12`,可分别执行两个用户程序，当然也可以通过改变执行序列来改变执行的顺序

## 3. 自定义中断:

时钟中断：通过PTR每秒发出18.2次的信号来从8592芯片的RT0引脚发出终端号`int 08h`来触发的用户时钟软中断 `int 1ch`,实现在terminal的右下角 一个横杠在转动.

另外有自定义中断`int 33h`,`int 34h`,`int 35h`,`int 36h`分别在屏幕四分之一的位置打印个性化信息

## 4. 进程调度:

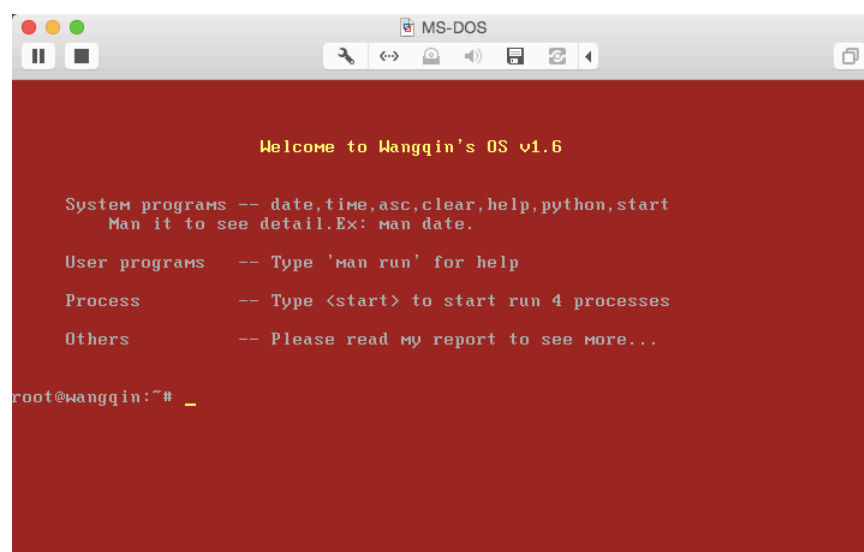
软盘中共存放了用于展示进程调度的五个应用程序，其中一个应用程序为监听用户键盘事件然后退出多进程调度状态回到 `terminal`每个应用程序分别代表一个进程。开启装载进程并进行进程调度由1中系统内置功能的 `start`指令激活。

# 实验步骤及效果图

1. 编辑修改ASM 文件,和C文件

2. 使用make命令配合makefile文件进行编译

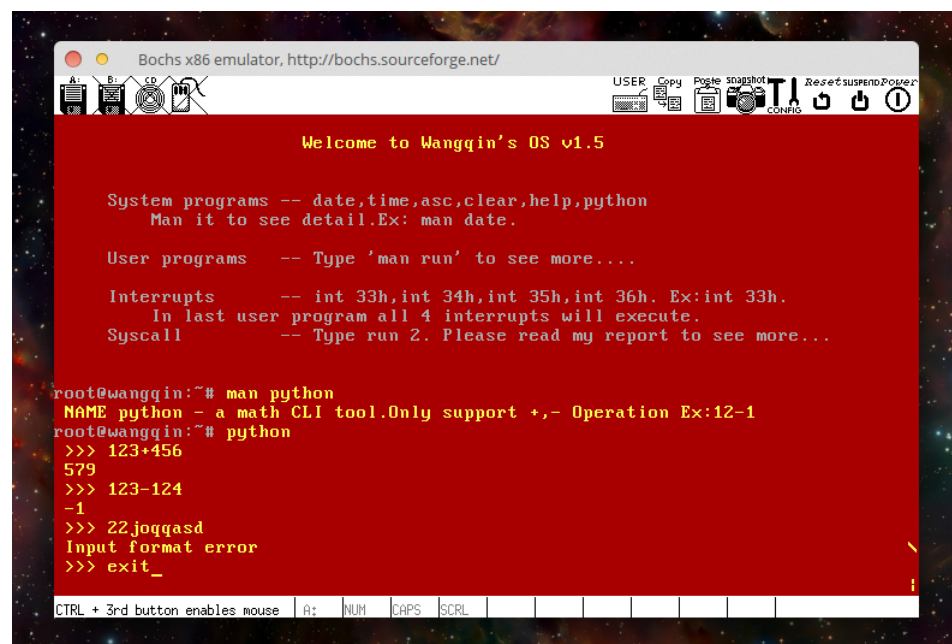
3. 运行bochs or vmware虚拟机进行测试,进入后所看到的欢迎界面,同时进入terminal界面等待输入指令



4. 我们可以执行`man python`

查看python的作用，主要功能是输入一个数学表达式然后返回表达式的结果，类似python命令行的作用。其他类似的系统内置功能也

可以使用man命令来查看帮助信息。但目前python只支持加法和减法且只能有两个操作数（其实主要为了展示ah=3,4将字符串转为数值和将数值转为字符串的系统调用效果），我这里仿照linux系统的系统调用，设置 int 80h 为所有系统调用的入口。可以看到图中分别输入加法减法，返回计算结果。如果输入的不符合格式就会返回错误提示。最后输入exit退出python命令行工具。



```
Bochs x86 emulator, http://bochs.sourceforge.net/

Welcome to Wangqin's OS v1.5

System programs -- date,time,asc,clear,help,python
Man it to see detail.Ex: man date.

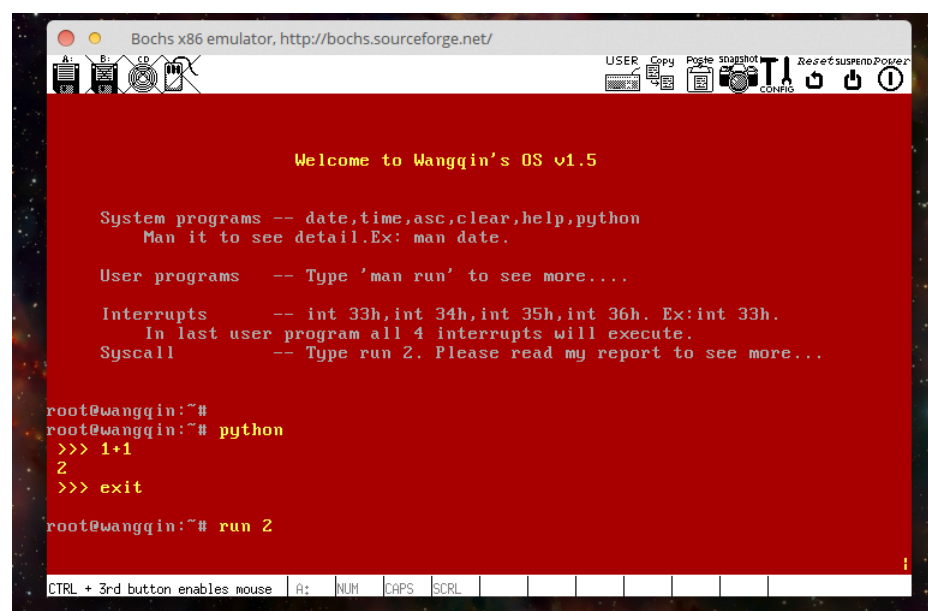
User programs -- Type 'man run' to see more....

Interrupts -- int 33h,int 34h,int 35h,int 36h. Ex:int 33h.
In last user program all 4 interrupts will execute.
Syscall -- Type run 2. Please read my report to see more...

root@wangqin:~# man python
NAME python - a math CLI tool.Only support +,- Operation Ex:12-1
root@wangqin:~# python
>>> 123+456
579
>>> 123-124
-1
>>> 22joqqasd
Input format error
>>> exit_

CTRL + 3rd button enables mouse | A: | NUM | CAPS | SCRL | | | | | | | | | |
```

5. 接下来测试ah等于0,1,2,5的系统调用(显示OUCH, 字母大小写变化等), 输入 run 2 ,运行第二个用户程序,



```
Bochs x86 emulator, http://bochs.sourceforge.net/

Welcome to Wangqin's OS v1.5

System programs -- date,time,asc,clear,help,python
Man it to see detail.Ex: man date.

User programs -- Type 'man run' to see more....

Interrupts -- int 33h,int 34h,int 35h,int 36h. Ex:int 33h.
In last user program all 4 interrupts will execute.
Syscall -- Type run 2. Please read my report to see more...

root@wangqin:~#
root@wangqin:~# python
>>> 1+1
2
>>> exit
root@wangqin:~# run 2

CTRL + 3rd button enables mouse | A: | NUM | CAPS | SCRL | | | | | | | | | |
```

下面是第二个用户程序的代码:

```
1 org 0x1000
2 ;org 0x100
```

```

3
4 ;#0
5 mov ah,0
6 int 80h
7
8 ;LISTEN_EXIT——
9 listen0:
10     mov ah,0
11     int 16h
12
13
14 ;#2
15 mov ax,0xb800
16 mov es,ax
17 mov dx,1994D
18 mov ah,2
19 int 80h
20
21 ;LISTEN_EXIT——
22     mov ah,0
23     int 16h
24
25 ;#1
26 mov ax,0xb800
27 mov es,ax
28 mov dx,1994D
29 mov ah,1
30 int 80h
31
32 ;LISTEN_EXIT——
33     mov ah,0
34     int 16h
35
36
37
38 ;#5
39 mov ax,cs
40 mov ds,ax
41 mov es,ax
42
43 mov cx,0317h ; position
44 mov ah,5
45 mov dx,msg
46 int 80h
47
48
49 ;LISTEN_EXIT——
50 listen:
51     mov ah,0
52     int 16h
53
54 ret
55
56
57
58 msg:
59     db "hello world!"
60
61 times 512-($-$$) db 0    填充剩余扇区;0

```

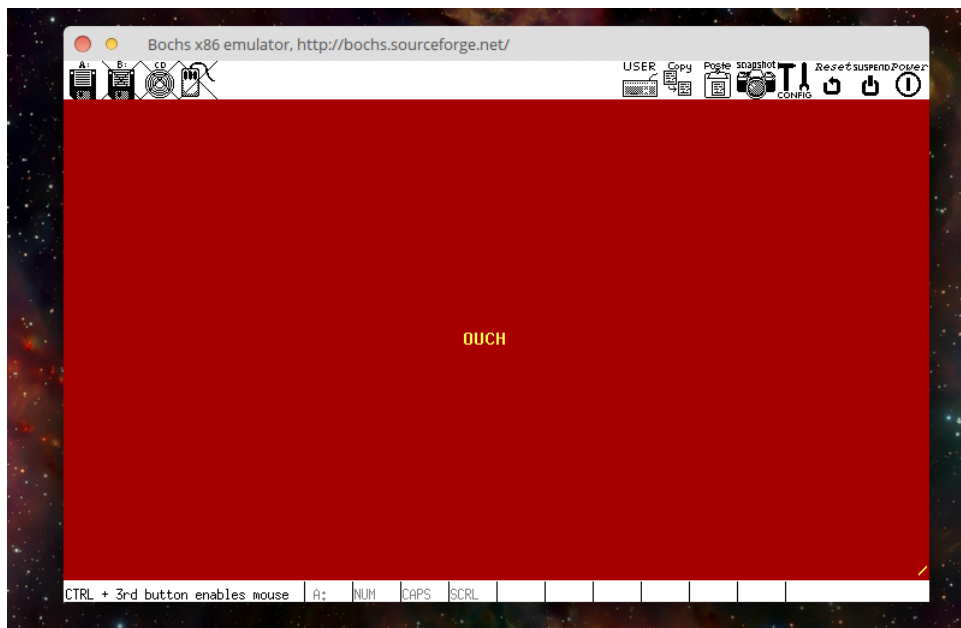
下面我们将上面代码分解一下，详细介绍。

### 5.1 首先执行执行ah=0的系统调用将 OUCH 打印在屏幕中间

```

1 org 0x1000
2 ;org 0x100
3
4 ;#0
5 mov ah,0
6 int 80h
7 ;LISTEN_EXIT——
8 listen0:
9     mov ah,0
10    int 16h

```



5.2 现在按下任意键，将执行ah=2的系统调用把 OUCH的第一个字母大写O变成小写o

```

1 ;#2
2 mov ax,0xb800
3 mov es,ax
4 mov dx,1994D
5 mov ah,2
6 int 80h
7
8 ;LISTEN_EXIT——
9     mov ah,0
10    int 16h

```



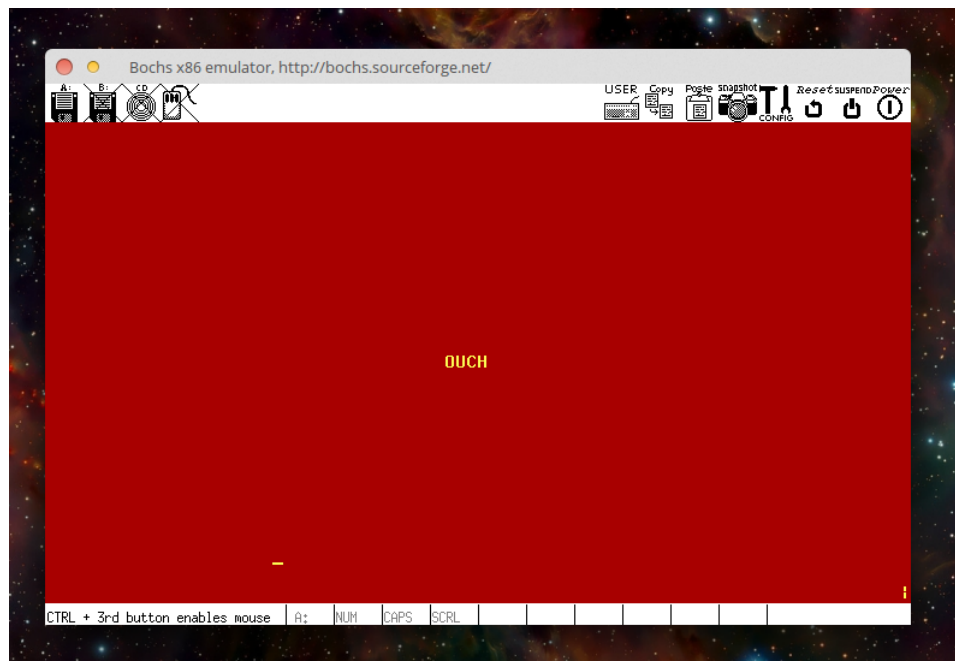
5.3 按下任意键，执行ah=1的系统调用把 oUCH的第一个字母小写o变回大写O

---

```

1 ;#1
2 mov ax,0xb800
3 mov es,ax
4 mov dx,1994D
5 mov ah,1
6 int 80h
7
8 ;LISTEN_EXIT——
9     mov ah,0
10    int 16h

```

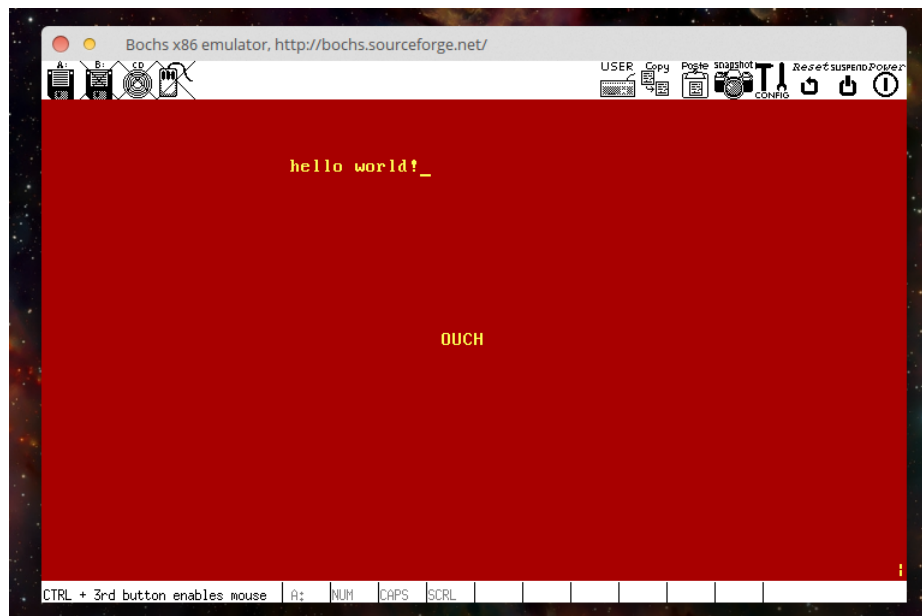


5.4 按下任意键，执行ah=5的系统调用在屏幕3行17列的位置打印一个helloworld

```

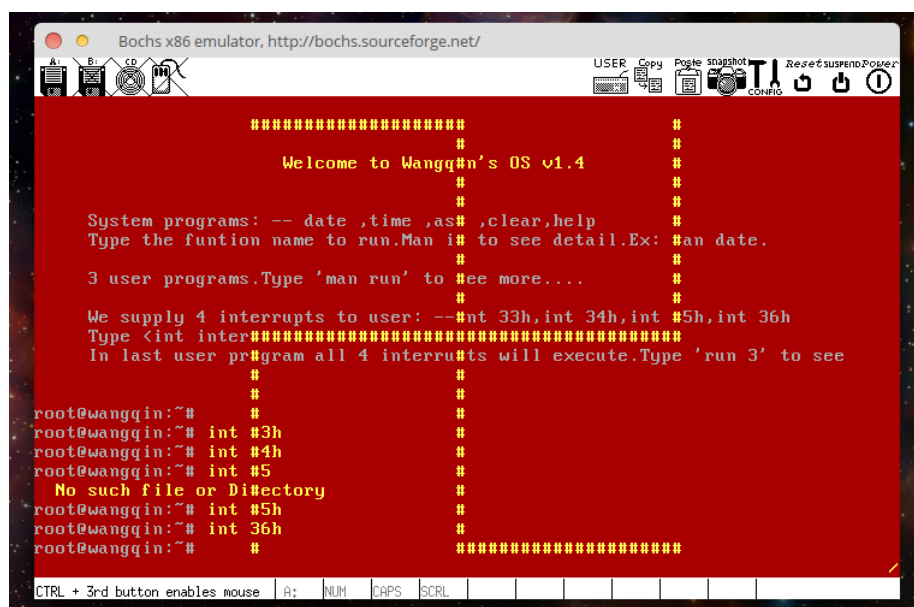
1 ;#5
2 mov ax,cs
3 mov ds,ax
4 mov es,ax
5
6 mov cx,0317h ;position
7 mov ah,5
8 mov dx,msg
9 int 80h
10
11
12 ;LISTEN_EXIT——
13 listen:
14     mov ah,0
15     int 16h
16
17 ret
18
19 msg:
20     db "hello world!"

```

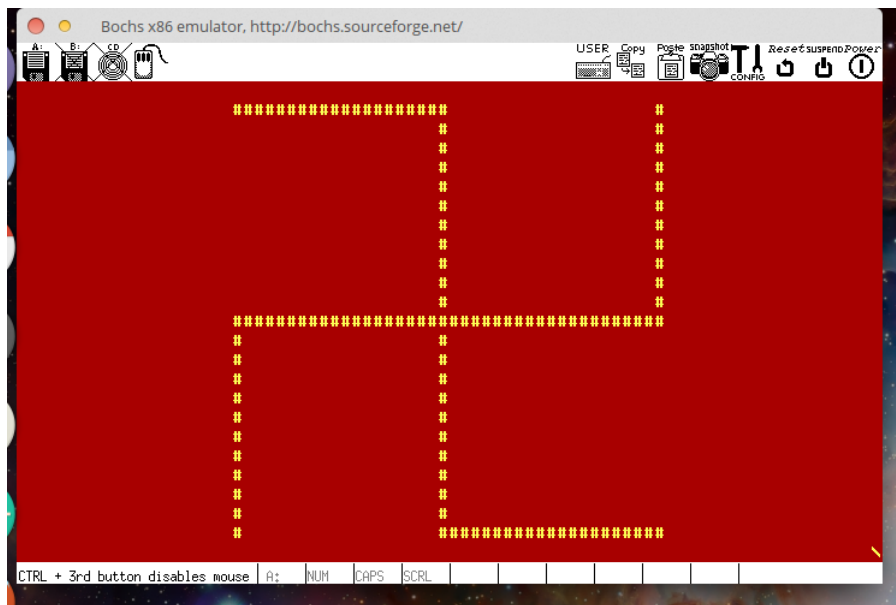


## 5.5 返回操作系统

6. 内核也有设置自定义中断,在 terminal 的右下角一直有个横杠在转动,转动频率由 `0x1c` 时钟中断触发。在第一个用户程序中 `run 1` 可以看到自定义的 `33h~36h` 中断分别向屏幕 `1/4` 位置打印,组合成一个万国旗

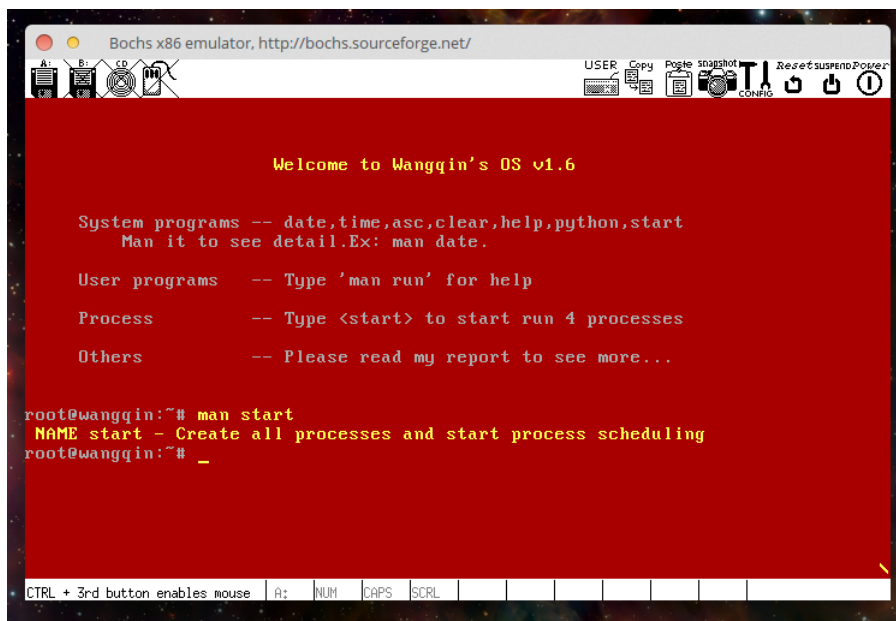






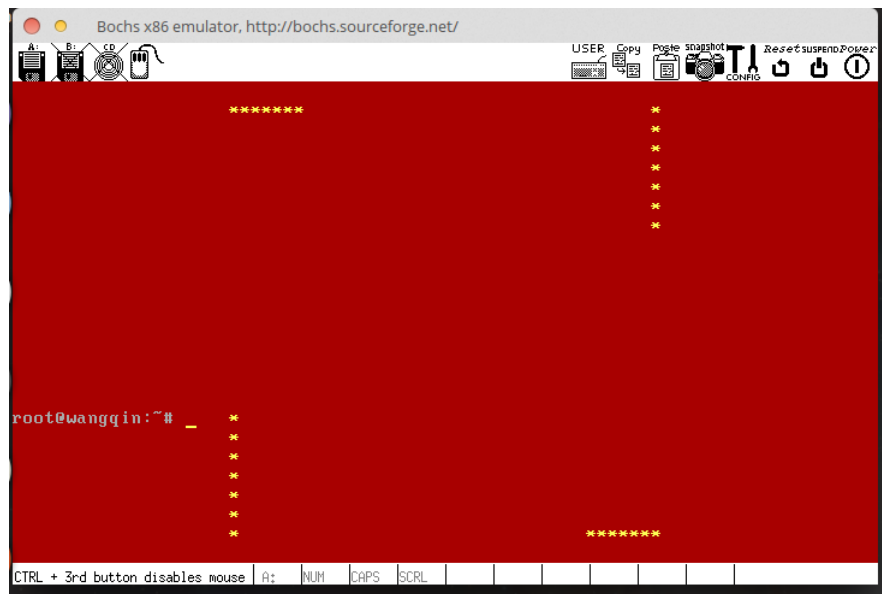
7. 接下来展示这次实验要求实现的多进程调度。

7.1 我这里创建并执行进程由命令 `start` 触发。

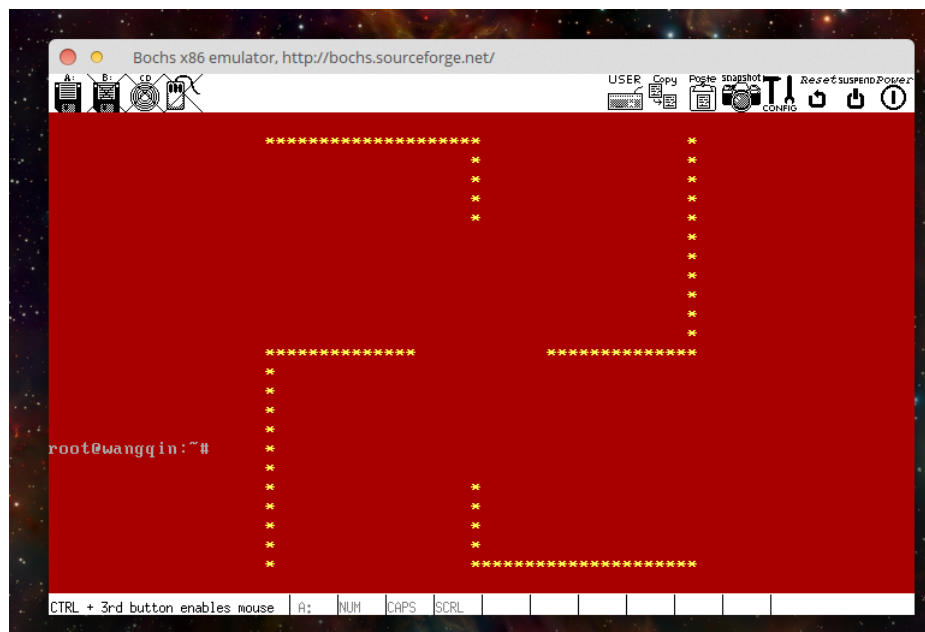


7.2 我们在 `terminal` 中输入 `start` 回车。开始创建四个进程并且进行每 18.2 次的调度，四个进程均分时间片  $T$

$$T = \frac{1}{18.2} S = 0.05494S = 54.94ms$$



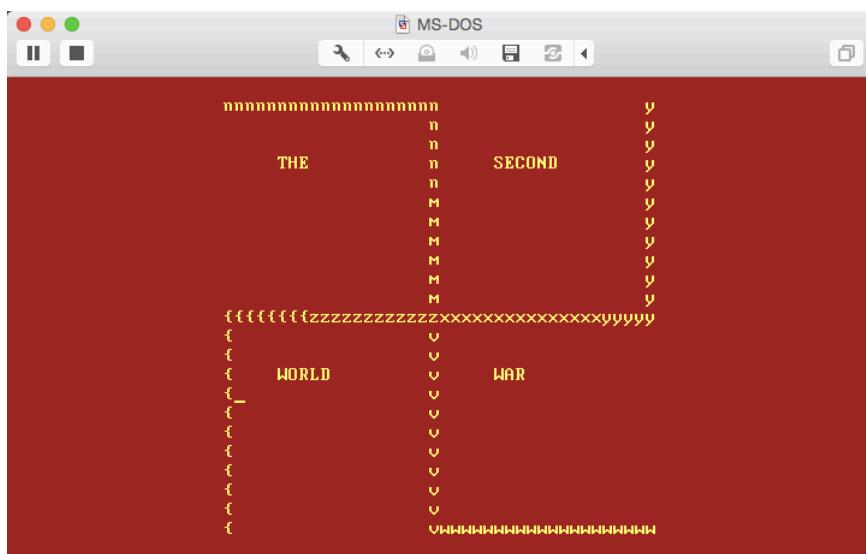
7.3 可以看到每个进程向屏幕四分之一区域打印个性化信息，由于使用的延时函数是用暴力循环的方法，所以存在在不同配置的虚拟机上动画的速度不一样的问题。



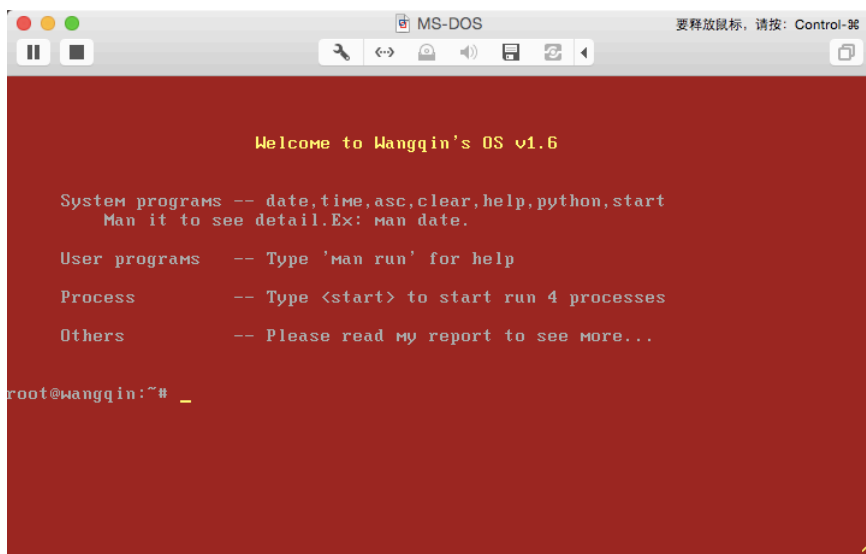
7.4 在屏幕形成一个万字旗，并且打印出 THE SECOND WORLD WAR







7.6 实际上执行 `start` 命令后，总共创建了5个进程，其中四个在上面已经说到，另外一个是为了退出进程调度动画回到 `terminal`的监听键盘事件的后台进程。与其他四个进程拥有同样长的时间片，也就是说在进程调度动画执行的任何时候都可以随便按一个键返回 `terminal`。



## 内存和软盘存储管理

1. 引导程序加载到内存0x7c00处运行
2. 引导程序将操作系统加载到0x7e00处运行
3. 操作系统讲用户程序加载到0x1000处运行
4. 软盘第1个柱面的第一个扇区存储操作系统引导程序
5. 软盘第1个柱面剩下所有扇区2~36扇区存储操作系统内核
6. 软盘第2,3,4,5,6柱面分别存储五个进程代码
7. 软盘第7,8柱面分别存储两个用户程序的程序代码

## 栈结构

内核栈:从内存的 0xffff开始向下扩展

用户栈:从内存的 0x1000开始向下扩展

进程栈:第i个进程对应的进程栈从内存的 i\*0x1000开始向下扩展

更多细节信息请阅读我的Makefile文件

## 系统架构

```
.
├── bochsrc
├── boot.asm 引导程序
├── disk.img
├── kernel    目录中存放内核相关代码
│   ├── os.asm    主要为os.c提供函数实现.
│   ├── os.c      为内核主要控制模块
│   ├── oslib.c   主要为os.c提供函数实现.
│   ├── os.h
│   ├── oslib.asm 为oslib.c提供更底层的函数封装
│   ├── os_syscall.asm 初始化系统调用和设置系统调用相关模块
│   ├── pcb.h
│   ├── process.c 进程调度创建相关代码
│   ├── terminal.c 装载shell的工具
│   └── terminal.h
├── Makefile  makefile 文件
├── oslib_share.c  内核和用户的共享库(使用用户程序体积减少)。
├── oslib_share.asm  内核和用户的共享库(使用用户程序体积减少)。
├── process1.asm    第一个进程
├── process2.asm    第二个进程
├── process3.asm
├── process4.asm
├── process_wait_key.asm 监听退出进程
├── python_extension.c  Python扩展
├── README
├── snapshot.txt
├── stack.png
├── usr1.asm        用户代码
└── usr2.asm
```

1 directory, 28 files

## 主要函数模块

1. os.c: main 函数模块, 由于每次调用汇编函数c都会向栈里面压两个字, 但ret返回的时候却只从栈里面弹出一个字, 为了保持栈中无冗余数据故每次调用汇编函数都要手动再pop出来一个字。

```
1 #include "os.h" //include some extern function declare
2
3 //=====+MAIN=====
4 void main(){
5     //-----init
6     screen_init();
7     asm__("pop %si");
8     interrupt_init();
9     asm__("pop %si");
10    syscall_init();
11    asm__("pop %si");
12    print_welcome_msg();
13    asm__("pop %si");
14    print_message();
15    asm__("pop %si");
16    print_flag(); //root@wangqin4377@: position
17    asm__("pop %si");
18    //-----init_end
19    while(1){
20        char length = listen_key(); // load shell and enter terminal
21
22        if_screen_scroll(); //bottom of screen
23        flag_scroll(); //move flag to next line
```

---

```

24         __asm__("pop %si");
25         if( Print_flag_mark)
26             print_flag();
27         __asm__("pop %si");
28     }
29 }
30
31 //=====MAIN END=====

```

2. 为了实现系统调用的工作，故在 os.asm中实现了下列函数供设置系统调用使用，系统调用表在 0xfe00的内存位置， ecx 为实现系统调用功能的函数内存地址。 ah为系统调用号码

```

1  setting_up_syscall:
2      mov bx,0
3      mov es,bx
4      mov al,ah
5      mov ah,0
6      shl al,2
7      mov bx,0xfe00
8      add bx,ax
9      mov [es:bx],ecx
10     ret

```

3. 为了方便自定义中断，在os.asm中实现以下函数。 interrupt\_num是中断号码。eax中存放着中断处理程序

```

1  insert_interrupt_vector:
2      mov ax,0
3      mov es,ax
4      mov bx,[ interrupt_num]
5      shl bx,2 ;interrupt num * 4 = entry
6      mov ax,cs
7      shl eax,8 ;shl 8 bit *16
8      mov ax,[ interrupt_vector_offset]
9      mov [es:bx], eax
10     ret

```

#### 4. 开机设置系统调用

```

1  syscall_init:
2
3  ;----#0 syscall
4  mov ah,0
5  mov ecx,0
6  mov cx,display_center_ouch
7  call setting_up_syscall
8
9  ;----#1 syscall
10 mov ah,1
11 mov ecx,0
12 mov cx,letter_upper
13 call setting_up_syscall
14
15 ;----#2 syscall
16 mov ah,2
17 mov ecx,0
18 mov cx,letter_lower
19 call setting_up_syscall
20
21 ;----#3 syscall
22 mov ah,3
23 mov ecx,0
24 mov cx,atoi_syscall
25 call setting_up_syscall
26
27 ;----#4 syscall
28 mov ah,4
29 mov ecx,0
30 mov cx,itoa_syscall
31 call setting_up_syscall
32
33 ;----#5 syscall
34 mov ah,5
35 mov ecx,0
36 mov cx,display_str
37 call setting_up_syscall
38 ret

```

5. 设置自定义中断处理程序,第一个设置的就是在terminal右下角不断旋转的小球,原理是使用IRQ0 ( 0x08号中断 ) 自动触发的 0x1c号中断实现.

```

1
2      ;#1 setting up time interrupt
3      mov ax,0x1c
4      mov [ interrupt_num], ax
5      mov ax, timer_interrupt_process
6      mov [ interrupt_vector_offset],ax
7      call insert_interrupt_vector
8
9
10
11     ;#2 int 33
12     mov ax,0x33
13     mov [ interrupt_num], ax
14     mov ax, process_int33
15     mov [ interrupt_vector_offset],ax
16     call insert_interrupt_vector
17
18     ;#3 int 34
19     mov ax,0x34
20     mov [ interrupt_num], ax
21     mov ax, process_int34
22     mov [ interrupt_vector_offset],ax
23     call insert_interrupt_vector
24
25     ;#4 int 35
26     mov ax,0x35
27     mov [ interrupt_num], ax
28     mov ax, process_int35
29     mov [ interrupt_vector_offset],ax
30     call insert_interrupt_vector
31
32     ;#5 int 36
33     mov ax,0x36
34     mov [ interrupt_num], ax
35     mov ax, process_int36
36     mov [ interrupt_vector_offset],ax
37     call insert_interrupt_vector
38
39     ;#5 int 36
40     mov ax,0x80
41     mov [ interrupt_num], ax
42     mov ax, process_int80
43     mov [ interrupt_vector_offset],ax
44     call insert_interrupt_vector

```

6. 执行用户程序,通过执行run函数自动把terminal中输入指定的用户程序加载到 0x1000,并跳转到这里开始执行.

```

1 inline void run( char *str){
2     str += 4;
3
4     while( *str != '\0'){
5         if( '0'< *str && *str< Usr_num){
6
7             load_user( 5 + *str-'0', 0x1000);           //in oslib.asm usri in i sector
8             __asm__(" pop %ax");
9             run_user();
10            __asm__(" pop %ax");
11        }else{
12            run_error();
13            return;
14        }
15        str++;
16    }
17    init_flag_position();
18    screen_init();
19    print_welcome_msg();
20    print_message();
21    print_flag(); //root@wangqin4377@: position
22 }

```

7. 创建进程,首先初始化5个进程(其中有一个是后台监听退出进程),然后分别加载5个进程到相应的内存单元,设置isProcesRun=1 用来表示当前正在运行多进程调度,让时钟中断的处理程序选择调度的代码执行而不是原来的打印横杠转动的代码。

```

1 void Process(){

```



```

2     int current_process_SEG = process_SEG;
3     int i;
4     for( i = 1; i <= process_num_MAX; i++){
5         current_process_SEG += 0x1000;
6         init_pcb( i, current_process_SEG);
7     }
8     load_user(1, 0x1000);
9     load_user(2, 0x2000);
10    load_user(3, 0x3000);
11    load_user(4, 0x4000);
12    load_user(5, 0x5000);
13    w_is_r=0;
14    isProcessRun=1; // enter user process mode
15 }

```

8. 进程调度, 这个是本次试验最核心的要求, `w_is_r`:which process is running, `nw_is_r`:which next process will run 换老进程下来的时候首先保存的是通用寄存器到这个进程的上下文 `TSS`, 具体实现就是先把 `ax, bx...` 等通用寄存器保存到c语言中定义的全局变量中然后再由全局变量保存到进程控制模块队列的pcb结构体中。最后保存的是 `IP, CS, Flags, SP`, 前三个直接从当前栈中直接pop出来即可, 因为cpu在发生中断的时候已经自动把这三个push到栈中了, 最后再保存`sp`, 栈指针寄存器。接下来根据 `di`寄存器的值来判断是否该结束调度程序回到terminal。如果不是则继续换上新的进程, 首先还原的是 `sp, ip, cs, flags`, 后三个直接push到栈里面就可以了, `iret`的时候会自动取出来并跳转到 `cs:ip`位置执行。最后跳转到 `schedule_end`中执行`iret`结束本次调度。

```

1 void schedule(){
2     saveall_reg(); //hurry not inclue sp
3     __asm__("pop %cx");
4     __asm__("pop %eax"); //junk
5
6     nw_is_r = w_is_r + 1;
7     if( nw_is_r > process_num_MAX){
8         nw_is_r = 1;
9     }
10
11
12     saveToQueue(); //code order don't change
13
14
15     //-----set ip cs flag-----
16     __asm__("pop %ax");
17     __asm__("pop %bx");
18     __asm__("pop %cx");
19
20     saveall_reg_seg(); //include sp
21     __asm__("pop %cx");
22
23     if( _di == 0x1234){
24         isProcessRun = 0; //shut down process
25         nw_is_r = 0;
26         backto_os();
27     }else{
28         isProcessRun = 1;
29     }
30
31
32     PCB_queue[ w_is_r ].tss.SP = _sp;
33     PCB_queue[ w_is_r ].tss.IP = _ip;
34     PCB_queue[ w_is_r ].tss.CS = _cs;
35     PCB_queue[ w_is_r ].tss.Flags = _flags;
36
37     //-----end-----
38     _ip = PCB_queue[ nw_is_r ].tss.IP;
39     _cs = PCB_queue[ nw_is_r ].tss.CS;
40     _flags = PCB_queue[ nw_is_r ].tss.Flags;
41     _sp = PCB_queue[ nw_is_r ].tss.SP;
42
43     restore_reg_seg();
44     __asm__("pop %cx");
45
46
47     queueTodata(); // ax bx cx...
48
49     w_is_r++;
50     if( w_is_r > process_num_MAX){

```

---

```

51         w_is_r = 1;
52     }
53
54     restore_reg();
55     __asm__("pop %di");           //don't use di in any process is dangerous
56
57     __asm__("jmp schedule_end");
58     while(1);
59 }

```

9. 保存的时候将全局变量的数据保存到pcb中，还原的时候倒过来赋值就可以了。全局变量实际上就相当于寄存器和pcb之间的一个缓存。

```

1 inline void saveToQueue(){
2     PCB_queue[ w_is_r ].tss.ES = _es;
3     PCB_queue[ w_is_r ].tss.DS = _ds;
4     PCB_queue[ w_is_r ].tss.GS = _gs;
5     PCB_queue[ w_is_r ].tss.FS = _fs;
6     PCB_queue[ w_is_r ].tss.SS = _ss;
7
8     PCB_queue[ w_is_r ].tss.AX = _ax;
9     PCB_queue[ w_is_r ].tss.BX = _bx;
10    PCB_queue[ w_is_r ].tss.CX = _cx;
11    PCB_queue[ w_is_r ].tss.DX = _dx;
12    PCB_queue[ w_is_r ].tss.SI = _si;
13    PCB_queue[ w_is_r ].tss.DI = _di;
14    PCB_queue[ w_is_r ].tss.BP = _bp;
15 }

```

10. 时钟中断处理程序，判断 `isProcessRun` 是否为0来决定该执行进程调度程序还是在terminal的右下角显示转动的横杠。

```

1 timer_interrupt_process:
2     push ax
3     mov ax,0
4     mov ds,ax
5     mov byte al,[ isProcessRun]
6     mov ah,0
7     cmp al,ah
8     je print_corner
9     pop ax
10    jmp schedule

```

11. 保存寄存器数据到全局变量

```

1 extern _ax,_bx,_cx,_dx,_es,_ds,_sp,_bp,_si,_di,_fs,_gs,_ss
2 saveall_reg:
3     mov [_es],es
4     mov [_ds],ds
5     mov [_gs],gs
6     mov [_fs],fs
7     mov [_ss],ss
8
9     mov [_ax],ax
10    mov [_bx],bx
11    mov [_cx],cx
12    mov [_dx],dx
13    mov [_di],di
14    mov [_si],si
15    mov [_bp],bp
16    ret

```

12. 还原 `ip,cs,flags` 寄存器数据和切换进程栈。切换进程栈的时候一定要非常小心不要因为函数调用的问题而在切换的时候出现漏洞，否则调度将会失败。

```

1 restore_reg_seg:
2     mov ax,[ _ip]
3     mov bx,[ _cs]
4     mov cx,[ _flags]
5
6     pop si           ;ret
7     pop di
8
9     mov sp,[ _sp]
10
11    push cx           ;flags
12    push bx           ;cs
13    push ax           ;ip

```

```
14
15         push di
16         push si
17     ret
```

## 实验心得及仍需改进之处

### 实验心得:

本次实验过程中遇到了很多挫折，最大的一个问题就是编写进程调度的算法的时候由于自己疏忽在创建进程启动进程调度的时候没有把时钟中断清零导致，无法发生下一次时钟中断,原本以为 0x1c用户时钟中断可以不需要像0x08那样每次对8259芯片进行重新设置。时钟中断正常工作后又发现调度出现一些问题，比如第一个进程的时间片用完之后不能换到第二个进程执行，当然这个问题还是经过调试很容易解决的。

后来发现第二个进程执行用完时间片后切换到第一个进程时第一个进程无法按照原来停止执行的cs:ip处进行执行，这个问题经过调试也是比较容易解决的。最难调试的问题是栈的切换，我这里为每个进程都设置了一个应用程序栈，进程时间片用完之后切换进程的时候进程的上下文数据RSS中 也会包含这个进程所对应的用户程序栈的栈指针。也就是切换进程的时候也要对应用程序栈 ss:sp进行切换,但是c语言调用汇编的时候是向栈里压两个字，而汇编返回c调用处的时候是从栈里弹出一个字到 IP指令指针寄存器。

这种不对称的出栈入栈很容易造成使用栈的过程中导致栈数据混乱。而且当时钟中断的处理程序即进程调度程序执行完毕后要从栈里弹出三个字的数据到 IP,CS,Flags这些寄存器，调度的时候就要手动的先把原来的 IP,cs,flags弹出来保存然后压入新进程的 ip,cs,flags,所以保持栈清晰，有规律是非常重要的，否则进程调度将会失败。

栈切换这里这里调了很长时间，得到的教训就是一定要注意细节，c调用汇编或汇编调用c都要特别注意有没有冗余的数据被压栈，注意一定要先保存 ip,cs,flags,sp和一些段寄存器的数据,合适的时机再切换栈指针寄存器。当换一个新进程开始执行的时候一定要保证这个进程上次被换下来的时候,栈指针寄存器 sp一定是上次的位置减去三个字，因为要压入 ip,cs,flags。总之注意写代码的时候注意细节总会可以大大降低debug的时间从而减少写代码的总时间，提高编程效率。同时也是 作为一个专业学生好的习惯。这次实验中我优化了操作系统和内核的结构，将 shell和内核分开，使主内核文件 os.c代码量大大减少，向Linux提倡的微内核走进了一步。

### 实验仍需改进之处:

仍需完善细节，比如说python的命令行工具加入乘法，除法完全是几行代码的问题

可以考虑将用户程序做成elf格式，动态链接系统的代码库。目前的情况是用户程序自己带着一份和操作系统一样的代码库，分别联合编译

考虑精简操作系统代码，减少冗余代码

增加文件系统功能

调度算法和进程控制模块的存储有待优化

---

继续优化调整操作系统内核架构和内存磁盘管理