

# 实验报告 基于原型操作系统的游戏设计

姓名：王钦 学号：13349112 班级：计科二班

## 实验目的

完成一个汇编语言设计的互动单人游戏程序设计项目

## 实验平台

dd+gcc+ld+nasm+Linux+vim

## 算法流程图

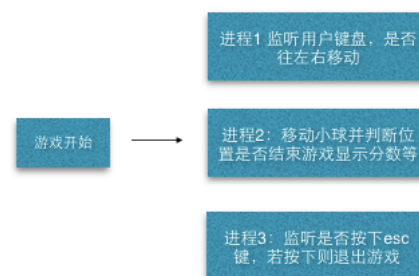


Figure 1: flow

## 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  
`clear`, 清除当前屏幕所有字符, 刷新屏幕  
`help`, 显示系统帮助信息  
`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

### 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:

操作系统所有执行的进程都使用时间片轮转的方法实现进程调度。

#### 5. 游戏:

输入指令 `game`,后可以进入游戏界面,可以看到一个小球弹来弹去,玩家要做的就是移动一个挡板来防止小球掉落下来。(注意: `a`为往左运动, `d`为向右运动,按住不松手,esc为退出), 详细信息可以看实验步骤及效果图的第五步

## 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

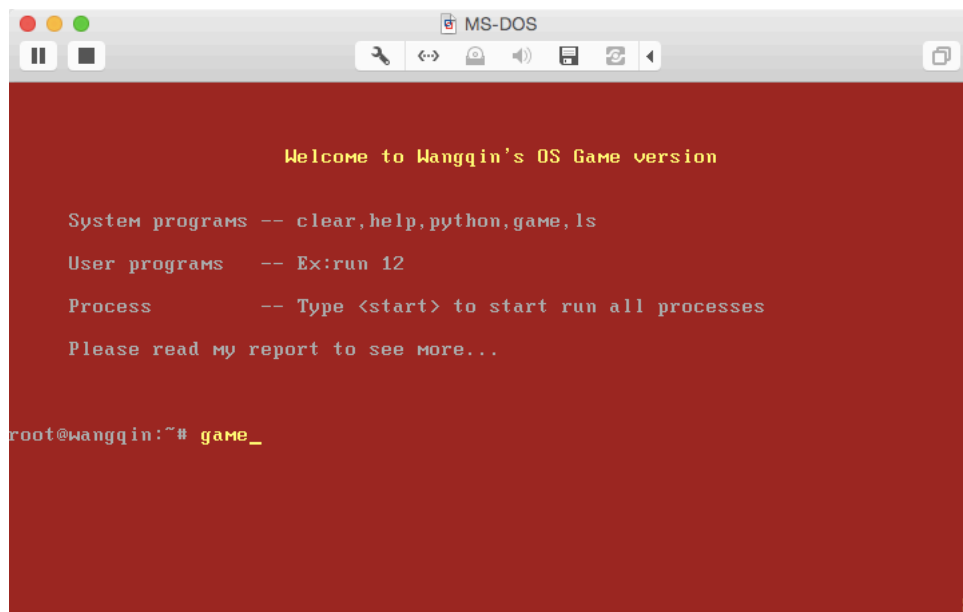
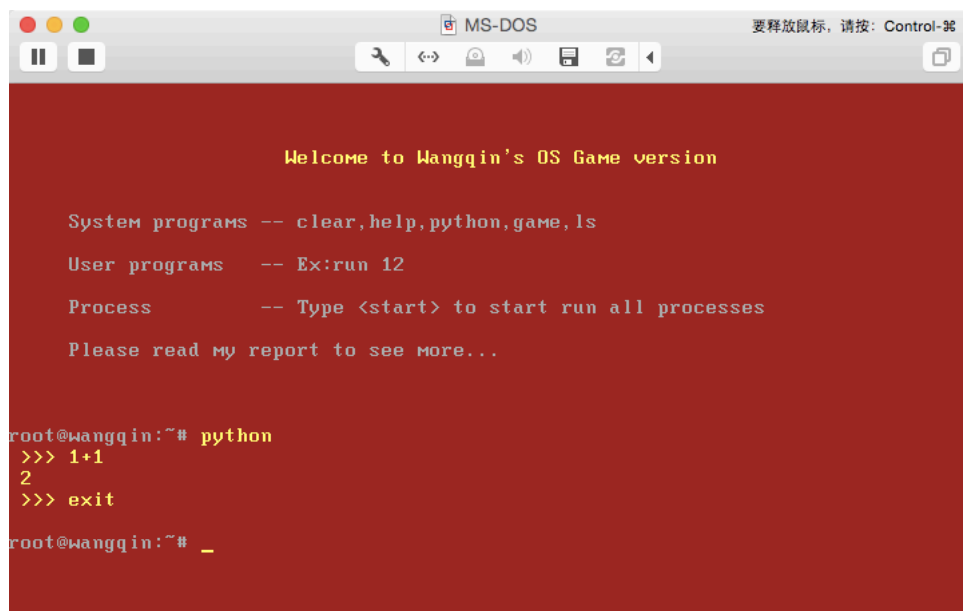


Figure 2: start

#### 4. 我们可以执行 `python`

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。



```

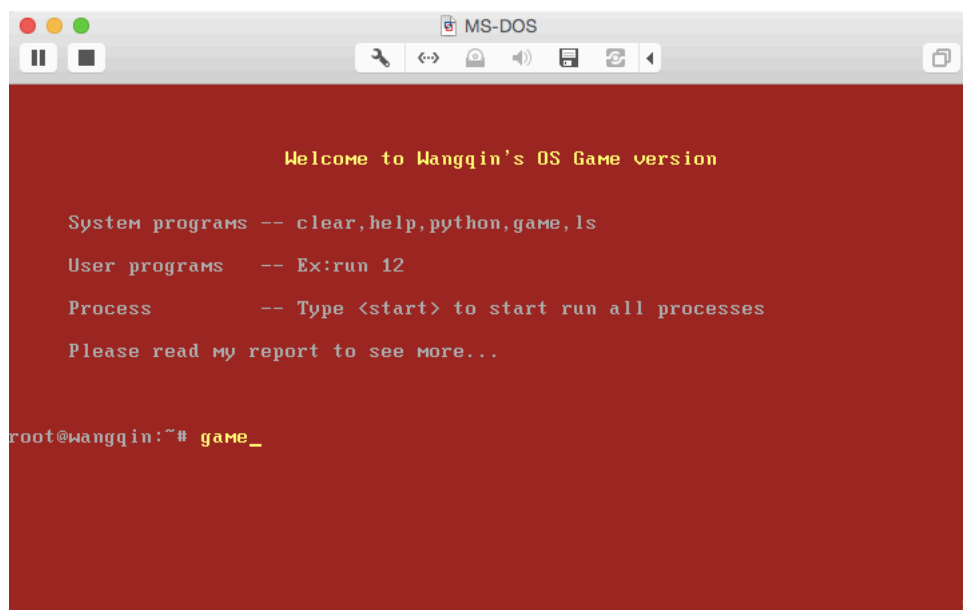
Welcome to Wangqin's OS Game version

System programs -- clear,help,python,game,ls
User programs   -- Ex:run 12
Process         -- Type <start> to start run all processes
Please read my report to see more...

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

Figure 3: python

5. 输入game指令开始装载和运行游戏，可以看到一个小球已经飞入屏幕，你可以使用键盘的a和d键来控制下面的挡板来防止小球跌入下面，每次小球触碰到挡板你的分数都会增加1，显示在右上角，初始化分数是0。并且随着分数的增加小球的运动速度也会增加，你要做的就是合理的控制挡板使分数最高！游戏结束后或过程中都可以按下esc键来退出游戏回到操作系统,回到操作系统后输入game便可再次游戏（注意：a为往左运动，d为向右运动,按住不松手,esc为退出）



```

Welcome to Wangqin's OS Game version

System programs -- clear,help,python,game,ls
User programs   -- Ex:run 12
Process         -- Type <start> to start run all processes
Please read my report to see more...

root@wangqin:~# game_
```

Figure 4: start game

5.1 玩家使用a和d键来控制挡板左右移动来挡住小球(按住不松手)

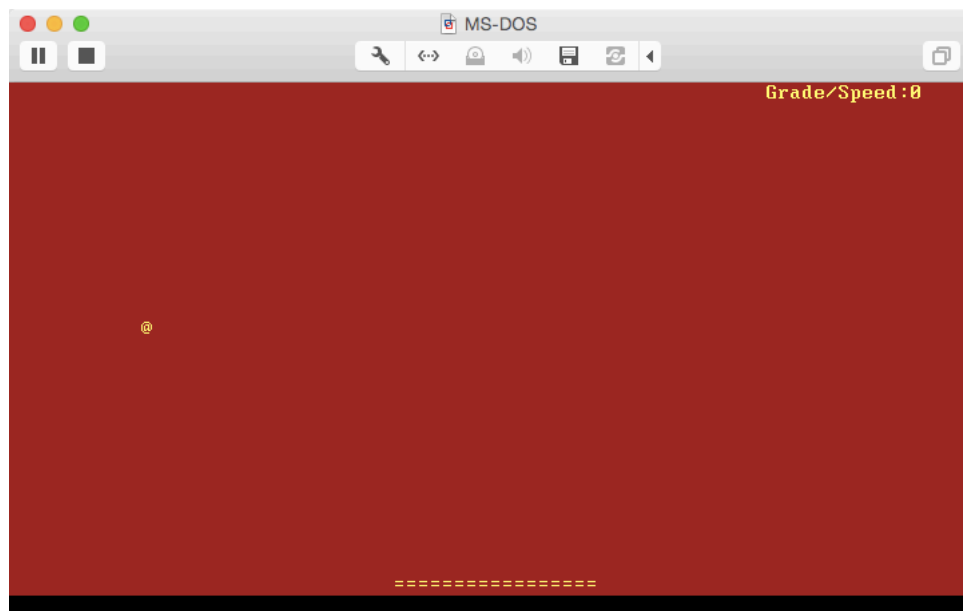


Figure 5: gaming

5.2 右上角会显示你的得分和现在小球的速度

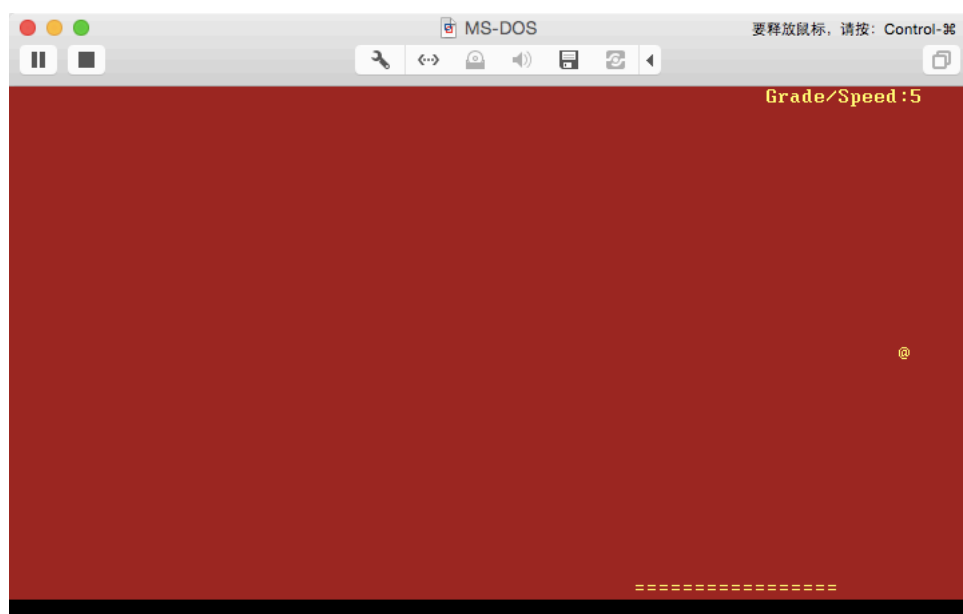


Figure 6: gaming

5.3 当玩家无法挡住小球往下落的时候就会游戏结束，显示 game over!. 可按esc键返回操作系统

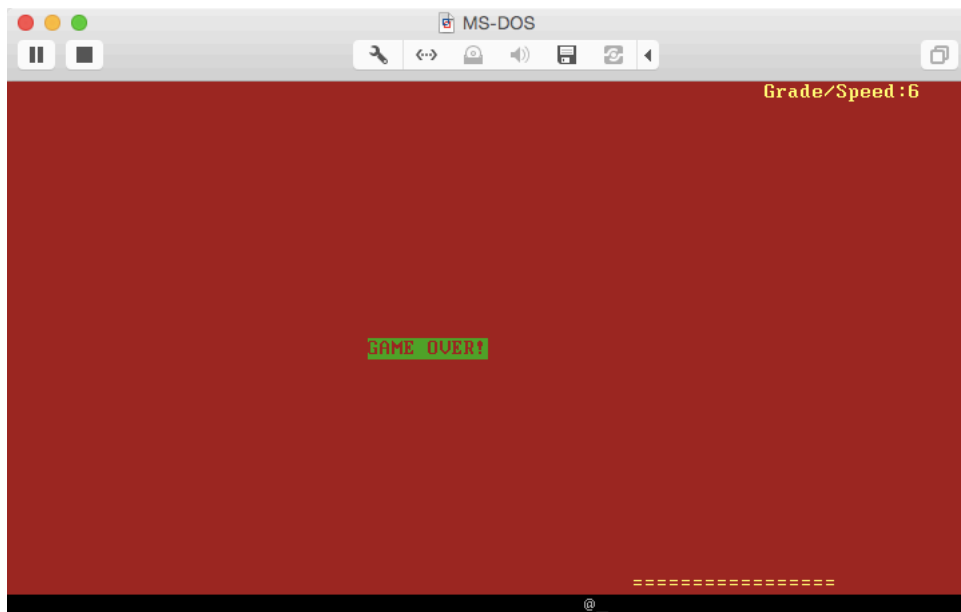


Figure 7: game over

## 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柱面分别存储三个游戏相关的进程
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

Makefile	makefile 文件
README	
bochsrc	
boot.asm	引导程序
disk.img	
kernel	目录中存放内核相关代码
fcb.h	文件控制模块
os.asm	主要为os.c提供函数实现.
os.c	为内核主要控制模块
os.h	主要为os.c提供函数实现.
os_syscall.asm	系统调用相关代码
osclib.c	初始化系统调用和设置系统调用相关模块

- ├── oslib.asm 为osclib.c提供更底层的函数封装
- ├── pcb.h 进程调度创建相关代码
- ├── process.c
- ├── terminal.c 装载shell的工具
- ├── terminal.h
- ├── muti\_process.h
- ├── osclib\_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, 监听ad键移动挡板
- ├── process1.img 第二个进程, 移动小球和判断分数
- ├── process2.asm 第三个进程, 监听esc键是否按下
- ├── process3.asm
- ├── process4.asm
- ├── process\_P2.asm
- ├── python\_extension.c Python扩展
- ├── report 实验报告目录
- ├── ── Illustrations 实验报告插图
  - ├── flow.png
  - ├── gaming.png
  - ├── gaming0.png
  - ├── over.png
  - ├── python.png
  - ├── start.png
- ├── report.pdf 实验报告文件
- ├── report.tex latex文件
- ├── tree
- ├── usr1.asm 用户代码
- ├── usr2.asm
- ├── usrlib.asm

## 主要函数模块(游戏部分从13开始)

```

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=====

```

```
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，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

```

13. 游戏部分:process1.asm,第一个进程为监听用户按下ad键不松手向左或向右移动挡板。`now_index` 代表的是目前挡板的最前端位置相对 `0xb800`的偏移, 每次移动只需移动挡板最前端同时把挡板最后端的字符赋值为空格,挡板长度为`str1`

```

1 loop_k:
2     mov ah,0
3     int 16h
4     mov ah,97D
5     cmp al,ah        判断是否是, a
6     je m_n_l         向左移动
7     mov ah,100D
8     cmp al,ah        判断是否是d

```

```

9  je m_n_r      向右移动
10
11  jmp loop_k
12  m_n_l:
13  call moveleft
14  jmp loop_k
15  m_n_r:
16  call moveright
17  jmp loop_k
18
19  moveleft:
20      mov ax,cs
21      mov ds,ax
22      mov si,[ now_index]
23      dec si
24      dec si
25      mov [ now_index],si
26      mov ax,0xb800
27      mov es,ax
28      mov bx,[now_index]
29      mov byte [es:bx], '='
30      mov cx, strlen
31      add bx, cx
32      add bx, cx
33      mov byte [es:bx], ' '
34  ret
35
36  moveright:
37      mov ax,cs
38      mov ds,ax
39      mov si,[ now_index]
40      inc si
41      inc si
42      mov [ now_index],si
43      mov ax,0xb800
44      mov es,ax
45      mov bx,[now_index]
46      mov byte [es:bx], '='
47      mov cx, strlen
48      sub bx, cx
49      sub bx, cx
50      mov byte [es:bx], ' '
51  ret
52

```

14. 游戏部分: process2.asm,这部分是游戏的主要内容,具体就是控制一个小球相上下左右移动并且 碰到墙壁和挡板就反弹, 如果碰到下面的边缘就显示Game Over游戏结束。 speedi dw 1000D,speedj dw 60000D这两个代表的是游戏的初始化速度, 每当小球碰到一次挡板speedi就会减掉100D, 并且grades要加1分。

```

1  cmp bx,3839D    ;24*160-1 3839      CHANGE
2  jge game_over
3  mov ah, '='
4  mov byte al,[es:bx]
5  cmp al,ah
6  jne bottom
7  mov ax,cs
8  mov ds,ax
9  mov ah,[ grades]
10 inc ah
11 mov byte [ grades],ah    ;change grades
12 mov ax,[ speedi]        ;change speed
13 sub ax,100D
14 mov [speedi],ax
15 call display_grades

```

本文件中有两个很长的模块bottom,top,这两个模块代码非常长我就不贴这里了, 具体 就是控制小球向下和向上的两个函数, 并且判断向下或向上的过程中是否碰到左右的墙壁, 如果碰到做相应处理。  
以下代码是本文件中的延时函数。

```

1  delay:          ;delay time function
2      mov ax,cs
3      mov ds,ax
4      mov dx,00
5      timer2:
6          mov cx,00

```

```

7         timer:
8             inc cx
9             cmp cx,[speedj]
10            jne timer
11            inc dx
12            cmp dx,[ speedi]
13            jne timer2
14            ret

```

15. 游戏模块: process3.asm, 这个文件主要就是监听用户是否按下esc键如果按下就设置 标志在系统进程调度的时候停止调度进程回到操作系统界面。代码也非常简单易懂, 其实这个 功能完全可以放在第一个进程中监听ad键的代码中实现, 但是为了游戏架构更具有扩展性和 层次感还是单独座位一个进程来实现。

```

1  loop_k:
2  mov ah,0
3  int 16h
4
5  mov ah,27D
6  cmp al,ah      判断是否是键esc
7  je m_n_l
8  jmp loop_k     不是继续监听
9  m_n_l:
10 setdi:
11 mov di,0x1234  是则死循环设置标记调度时会识别该标记,
12 jmp setdi
13
14 jmp loop_k
15
16 times 510-($-$$) db 0
17 db 0x55,0xaa

```

## 设计游戏实验心得

1. 由于本次试验完全基于多进程的原型系统上花费去设计游戏的精力并不多, 只是新加了几个汇编的进程而已。本来想写个贪吃蛇, 看到周围很多同学都写贪吃蛇, 于是就决定写个 这种弹小球的游戏, 实验过程中发现必须连续按住ad键, 挡板才会移动, 一下下的按时是不行的, 这个可能和时间片设置的大小有关系具体等待研究。实验过程中也遇到很多bug 比如说小球碰到挡板会把挡板给破坏掉或者撞到显示分数和速度的地方会出一些很细节的问题, 后来经过耐心精心仔细调试发现了这些躲在非常小地方的小bug, 因为有之前做操作系统实验的经验处理这些小bug并不难。
2. 这学期的汇编和实验课总算到此为止了,从中简直是获益匪浅。手动实现各种操作系统底层的结构了解了现代操作系统的运行机制同时通过艰难的调试让我体会到了作为一个 以后还要打几十年代码的人来说写代码的时候细心是最重要的, 基本上每次做实验用来调试的时间总是远远大于写代码的时间。最后查出来只是一句甚至是一个寄存器写错了而已。
3. 最后感谢老师&助教能辛苦的布置和批改检查我们的实验, 带领我们走进操作系统底层的奇妙世界。的确感觉c++语言设计和操作系统是整个大学中最重要两门课, 无论是对 专业能力的提升还是对个人知识面的增加都有非常大的帮助。