中山大学数据科学与计算机学院本科生实验报告

（2016 学年秋季学期）

课程名称：操作系统实验　　任课教师：凌应标　　教学助理（TA）：

年级	15 级	专业（方向）	软件工程（移动信息工程）
学号	15352461	姓名	宗嘉希（组长）
学号	15352448	姓名	周禅城
学号	15352443	姓名	钟凌山
电话	18022724490	Email	zongjx@mail2.sysu.edu.cn
开始日期	2017.05.15	完成日期	2017.05.22

【实验题目】

二状态的进程模型

【实验目的】

1. 实现进程表，实现二状态进程

2. 利用时钟中断完成分时

3. 保证系统调用仍能够正常使用

【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 个。

(2)内核一次性加载 4 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。

【实验方案】

一、 硬件及虚拟机配置

硬件：操作系统为 win10 的笔记本电脑

虚拟机配置：无操作系统，10MB 硬盘，4MB 内存，启动时连接软盘

二、 软件工具及作用

Nasm:用于编译汇编程序，生成.bin 文件

WinHex:用于向软盘写入程序

VMware Workstation 12 Player：用于创建虚拟机，模拟裸机环境

Notepad++: 用于编辑汇编语言文件

TCC：用于编译 C 文件，生成 OBJ 文件

TASM：用于编译 ASM 文件，生成 OBJ 文件

TLINK：用于把两个 OBJ 文件连接，生成 COM 文件

Dosbox：用于提供 16 位运行环境，为 TCC+TASM+TLINK 编译链接提供一个环境

【实验过程】

这一次的实验我们决定把老师给出的原型系统分析一下，把程序看懂、完全分析以后，把我们之前的实验中的功能移植到这个原型系统中，并实现本次的实验要求。

首先，我们先运行了一下老师给出的这一个原型操作系统，经过观察代码和操作总结出了原型系统拥有的功能：

（1）显示帮助菜单

（2）显示当前系统时间

（3）显示当前系统日期

（4）把 ascii 码转换成一个字符

（5）清屏

（6）运行用户程序

（7）调用 21 号中断，其中 21 号中断的功能包括：

1.在屏幕中央显示“OUCH！OUCH！”

2.把小写字符串转换为大写字符串

3.把大写字符串转换为小写字符串

4.把字符串转换为数字串

5.把数字串转换为字符串

6.在屏幕的任意位置打印一个字符串

7.把二进制数转换成十进制数

8.在屏幕的任意位置打印一个颜色变换的字符串

9.判断两个字符串是否相同

10.退出 21 号中断

以上的这些功能将会全部保留，并且将会在基于这些功能的基础上进行改进。

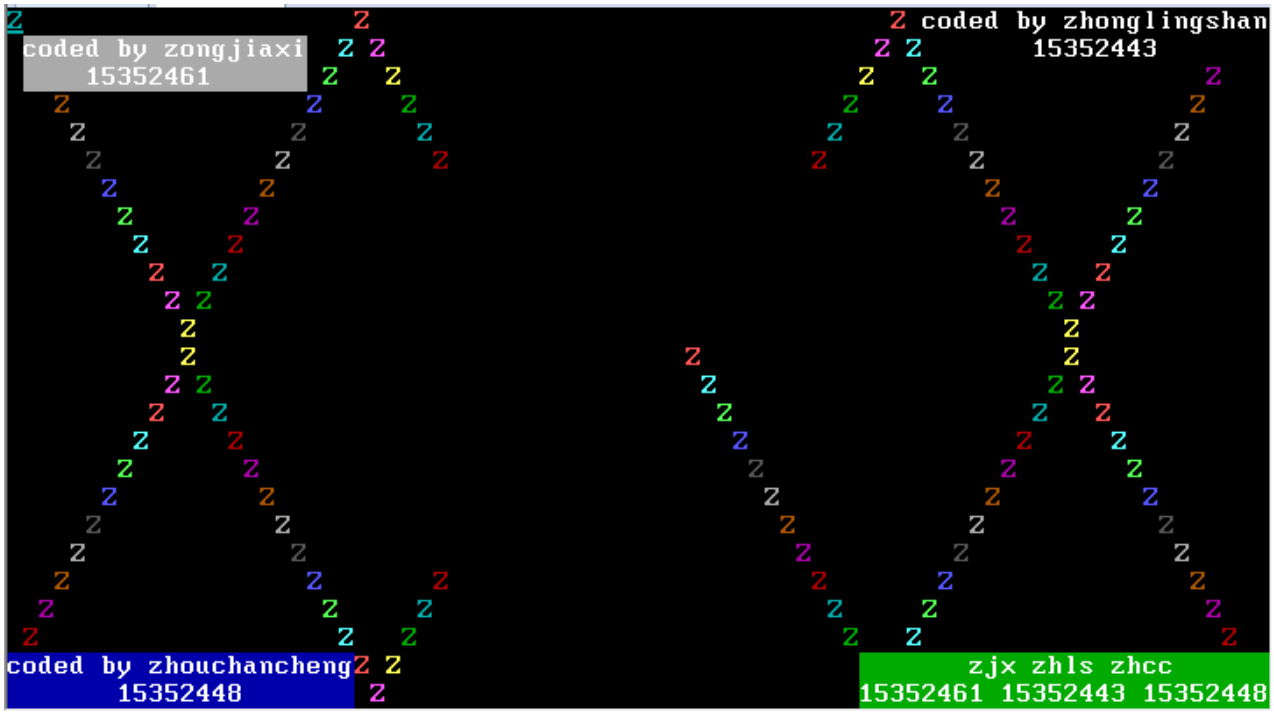
主要的改进包括以下几点：

①原型中的命令输入并不合理。我们在键入时难免会输入非法的命令，符合我们操作习惯的是我们能通过键盘上的 Backspace 按键清除我们的错误输入，再次键入正确的命令。但是在原型操作系统中我们按 Backspace 按键只会向左移动移动光标，就算我们移动到出现错误的地方再次键入正确的命令也是没有用的，只能按回车输入错误的命令后再次输入正确的命令。我们分析原型操作系统发现，例如我们输入的是 ppo 这个错误的命令，然后我们按了两次 Backspace 按键，再次键入 ro，实际上原型操作系统读入的是 pp0\b\bro，也就是说即使经过这样的修改依然不是正确的命令 pro。经过我们修改之后，同样输入错误命令 ppo，然后我们按两次 Backspace 按键，系统就会消除后面两位错误的输入，等于只读入了 p，我们再输入 ro，系统就能得到正确的命令 pro。

②原型操作系统中提供了一个 ascii 功能，但是这个功能经过我们测试是有问题的，比如我们输入 e，他的 ASCII 码是 101，但是输出结果是 1，很明显是错误的。我们分析代码发现原型操作系统中得到字符的 ascii 码后先对其除以 10 得到它的百位，然后再对 10 取模得到它的个位，这样很明显没有考虑到 ASCII 码是三位数的情况。实际上 ASCII 码肯定会出现三位数的情况，于是我们加入了一个 if 判断，当 ASCII 大于 99 时，首先将其除以 100 得到百位，然后将它除以 10 再对 10 取模得到十位，最后将它对 10 取模得到个位，这样我们对于输入的字符的 ASCII 码大于 99 时也能得到正确的输出。

③在原型操作系统中，int21 也就是 21 号中断中有个 0 号功能，就是每按一次 0 键就会在屏幕中间输出一个“Ouch！Ouch！”，但是在原型操作系统中缺乏动态展示，按了一次 0 键之后再按一次没有动态变化，体现不出中断。对此我们进行了一定的修改，按了一次后 0 键输出“Ouch！Ouch！”，再按一次 0 键“Ouch！Ouch！”的颜色会发生变化，这样既美观又体现出了中断。

除了改进以上的功能以外，我们也保留了以前的实验的功能，首先本次实验的四个用户程序是沿用以前的实验的四个用户程序，分别在屏幕的四分之一处动态展示字符的弹射。效果如下图所示：



在以前的实验中，我们都是批处理这四个用户程序,也就是按照顺序来运行这四个程序，一个程序运行完毕以后，下一个程序才开始运行，但是在本次的实验中，我们要利用时钟中断来对程序的运行进行中断、保存和恢复，利用这一种方法可以分时运行四个程序，当时钟中断的频率足够大的时候，就可以实现同时运行四个程序的效果。因此，在程序的开始就要先去设置时钟中断的频率。

时钟中断是有一个叫做 8259A 的芯片控制的。8259A 是专门为了对 8085A 和 8086/8088 进行中断控制而设计的芯片，它是可以用程序控制的中断控制器。8259A 芯片可以处于编程状态和操作状态，编程状态是 CPU 使用 IN 或 OUT 指令对 8259A 芯片进行初始化编程的状态。

```
SetTimer:
    push ax
    mov al,34h   ; 设控制字值
    out 43h,al   ; 写控制字到控制字寄存器
    mov ax,59659 ; 每秒 20 次中断（50ms 一次）
    out 40h,al   ; 写计数器 0 的低字节
    mov al,ah    ; AL=AH
    out 40h,al   ; 写计数器 0 的高字节
    pop ax
    ret
```

以上就是对 8259A 芯片的初始化编程，把时钟中断设置为每秒钟 20 次（1193182/20）。这一个速度足够快使得我们可以看到四个程序同时运行。

接下来就是要想办法做出二状态进程模型了，同一计算机内并发执行多个不同的用户程序，操作系统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。PCB 的主要作用就是模拟 8086 里面的寄存器结构，使用这一个数据结构来存储用户程序运行的状态。由于程序在运行的过程中，所有的数据都是存储在寄存器中，因此 PCB 必须要把所有寄存器的状态记录下来，所以 PCB 的数据成员就是寄存器。在 8086CPU 中，一共有 16 个寄存器，这些寄存器分别为：AX/BX/CX/DX/BP/SP/DI/SI/CS/DS/ES/FS/GS/SS/IP/FLAG。因此以下为 PCB 的数据结构：

```
typedef struct RegisterImage{
    int SS;
    int GS;
    int FS;
    int ES;
    int DS;
    int DI;
    int SI;
    int BP;
    int SP;
    int BX;
    int DX;
    int CX;
    int AX;
    int IP;
    int CS;
    int FLAGS;
}RegisterImage;
```

PCB 创建好之后，下一步我们来分析一下应该如何使用这个 PCB 来保护进程。当程序在运行的过程中，有关的数据都会存到相关的寄存器里面，而在中断发生的时候如果调用了下一个要运行的用户程序的话，由于 8086 里面的寄存器只有这么几个，肯定会把原数据给覆盖掉的，因此我们要使用 PCB 来把当前 CPU 中寄存器的内容保存下来。首先把寄存器依次压栈，但是不要改变栈的基地址，当把所有寄存器都压栈以后，调用 c 程序里保护的函数，把寄存器的值都存到 PCB 里，然后调用进程改变函数，调用下一个用户进程，把该进程的 PCB 模块中保存了的寄存器值调用，重新 pop 出来到相应的寄存器中，就相当于恢复了原来在运行的程序的状态了。以下是保存和恢复的汇编程序：

```
Pro_Timer:
;***************************************
;*                Save                 *
; ***************************************
    cmp word ptr[_Program_Num],0
    jnz Save
    jmp No_Progress
Save:
    inc word ptr[Finite]
    cmp word ptr[Finite],3200
    jnz Lee
    mov word ptr[_CurrentPCBno],0
    mov word ptr[Finite],0
    mov word ptr[_Program_Num],0
    mov word ptr[_Segment],2000h
    jmp Pre
Lee:
    push ss
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086

    mov ax,cs
    mov ds, ax
    mov es, ax

    mov ax,cs
    mov ds, ax
    mov es, ax

    call near ptr _Save_Process
    call near ptr _Schedule

Pre:
    mov ax, cs
    mov ds, ax
    mov es, ax

    call near ptr _Current_Process
    mov bp, ax

    mov ss,word ptr ds:[bp+0]
    mov sp,word ptr ds:[bp+16]

    cmp word ptr ds:[bp+32],0
    jnz No_First_Time
```

```
;*****************************************
;*                Restart                *
; *****************************************
Restart:
    call near ptr _special

    push word ptr ds:[bp+30]
    push word ptr ds:[bp+28]
    push word ptr ds:[bp+26]

    push word ptr ds:[bp+2]
    push word ptr ds:[bp+4]
    push word ptr ds:[bp+6]
    push word ptr ds:[bp+8]
    push word ptr ds:[bp+10]
    push word ptr ds:[bp+12]
    push word ptr ds:[bp+14]
    push word ptr ds:[bp+18]
    push word ptr ds:[bp+20]
    push word ptr ds:[bp+22]
    push word ptr ds:[bp+24]

    pop ax
    pop cx
    pop dx
    pop bx
    pop bp
    pop si
    pop di
    pop ds
    pop es
    .386
    pop fs
    pop gs
    .8086

    push ax
    mov al,20h
    out 20h,al
    out 0A0h,al
    pop ax
    iret

No_First_Time:
    add sp,16
    jmp Restart

No_Progress:
    call another_Timer

    push ax
    mov al,20h
    out 20h,al
    out 0A0h,al
    pop ax
    iret
```

通过以上过程我们可以实现进程的轮转运行！

不过要注意的是，要在数据结构里面声明一个变量去表示该进程的状态，在二状态进程模型中，只有运行和准备两种状态，因此可以很普通地表示出来。

本次实验的重点其实就在这里吧，一定要很好地处理这一个保护还有重启的过程，如果在这个过程中稍微出现差错，就会导致整个程序崩掉，因此这一个过程中，寄存器的存取非常重要。

再仔细地说一下整个使用时钟中断来实现分时运行用户程序的过程吧，首先在主界面中输入想要运行的用户程序，比如说 1、3、4 吧，输入之后，我们读取相对应的扇区，把相应的用户程序读入到指定的段地址中，由于运行用户程序的代码写到了时钟中断中，因此，只要时钟中断发生，1、3、4 号程序就会轮转运行，但是本来时钟中断就在运行了，因此我们要使用一个长时间的延时，是的用户程序正确展示。主要的代码如下图：

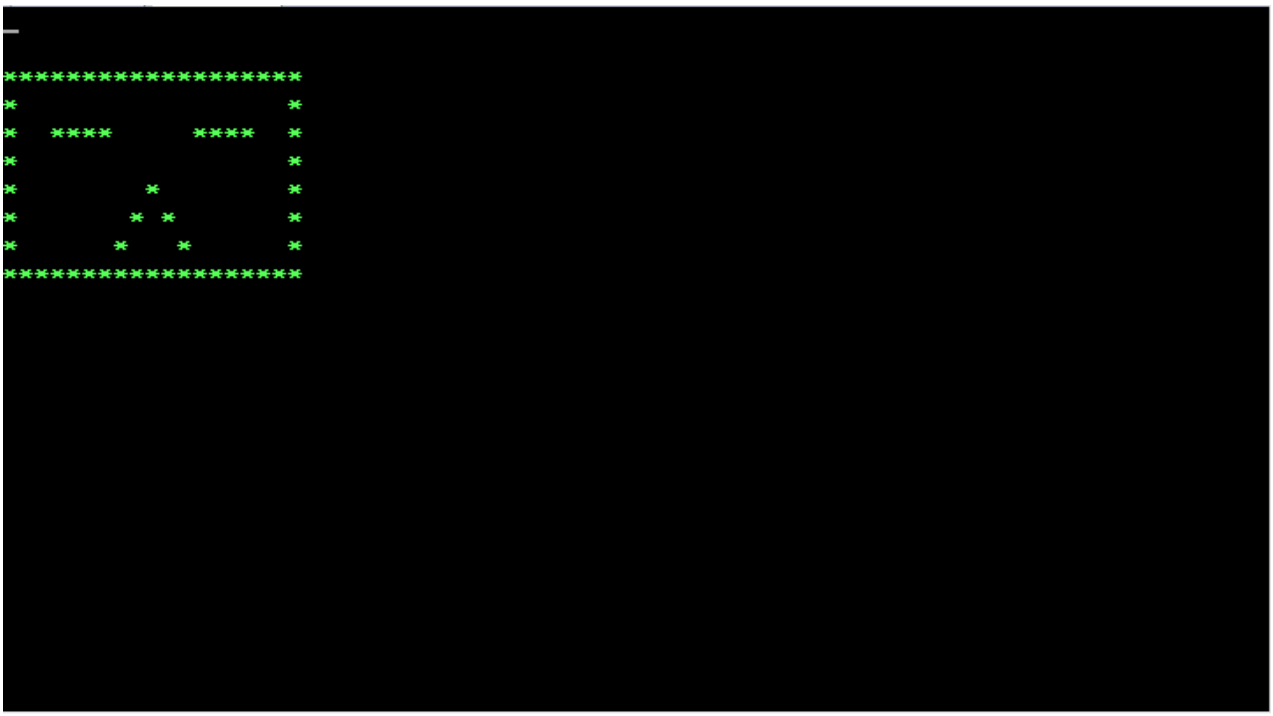
```
void Random_Load()
{
    int i = 0;
    int j = 0;
    for( i=0; i<StringLen;i++ )
        if( Buffer[i]!=' ' && (Buffer[i]<'1' || Buffer[i]>'5') )
        {
            Print("Error Input!");
            Delay2();
            return;
        }
    for( i=0; i<StringLen;i++ )
    {
        if( Buffer[i] ==' ' )
            continue;
        else
        {
            j = Buffer[i] - '0';
            if( Segment > 0x6000 )
            {
                Print(" There have been 5 Processes !");
                break;
            }
            another_load(Segment,2*j);
            Segment += 0x1000;
            Program_Num ++;
        }
    }
}
```

读取扇区并把程序放到指定的代码段中：

```
public _another_load
_another_load proc
    push ax
    push bp

    mov bp,sp

    mov ax,[bp+6]       ;段地址 ; 存放数据的内存基地址
    mov es,ax           ;设置段地址（不能直接mov es,段地址）
    mov bx,100h         ;偏移地址; 存放数据的内存偏移地址
    mov ah,2            ;功能号
    mov al,16           ;扇区数
    mov dl,0            ;驱动器号 ; 软盘为0，硬盘和U盘为80H
    mov dh,1            ;磁头号 ; 起始编号为0
    mov ch,0            ;柱面号 ; 起始编号为0
    mov cl,[bp+8]       ;起始扇区号 ; 起始编号为1
    int 13H             ; 调用中断

    pop bp
    pop ax

    ret
_another_load endp
```

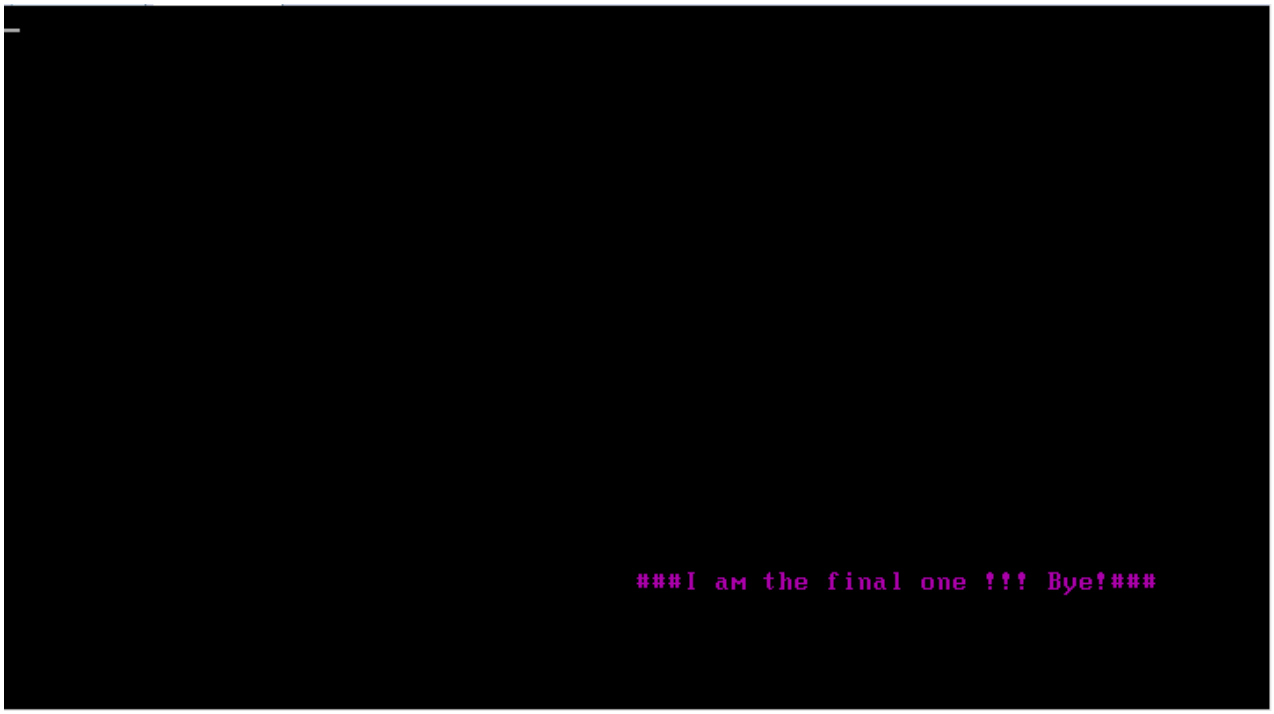
右上角屏幕有一些残留字符，请忽略。

到此，我们已经成功的把本次实验的主要内容给完成了。除了四个用户程序，我们也还多做了一个用户程序，用于测试 33 号、34 号、35 号、36 号中断是否可以继续使用，而这四个中断所显示的内容都分别在屏幕的左上，右上，左下，右下的位置，具体的结果如下面的图的展示：




****Hello! OS World! I Like It****


Ha!Ha! I am the Int 35h !!!


###I am the final one !!! Bye!###

但是由于这些中断显示字符串的方式是使用系统自带的中的，因此每一次调用中断，都要把屏幕清空一次，所以如果通过分时运行来同时运行用户程序和中断显示字符串的话，用户程序会显示不完整（每一次都刷新了屏幕）。

以下是我们本次实验的具体操作过程：

首先是编译的过程，使用 DOSBOX 创建一个 16 位的环境，利用 TCC,TASM,TLINK 去编译 c 文件和汇编文件:

```
D:\>TCC -LINCLUDE CMAIN
Turbo C  Version 2.01  Copyright (c) 1987, 1988 Borland International
cmain.c:
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

        Available memory 415192

D:\>TASM MYOS.ASM -O MYOS.OBJ
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   MYOS.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  454k
```

```
D:\>TASM KLIBC.ASM -O KLIBC.OBJ
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   KLIBC.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  462k


D:\>TLINK /3 /T MYOS.OBJ KLIBC.OBJ CMAIN.OBJ,OS.COM,,
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
```

然后再使用 nasm 把 5 个用户程序和引导程序编译好：

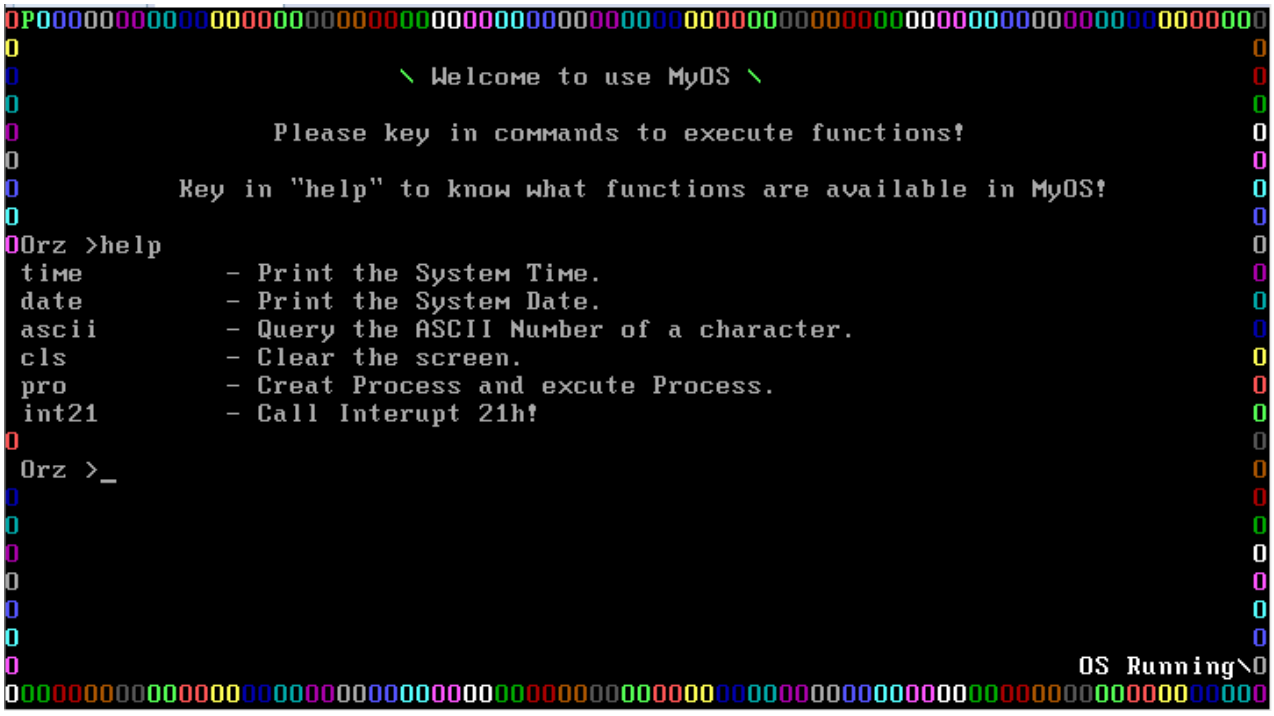
```
D:\exp2>nasm boot.asm -o boot.bin

D:\exp2>nasm prog1.asm -o prog1.bin

D:\exp2>nasm prog2.asm -o prog2.bin

D:\exp2>nasm prog3.asm -o prog3.bin

D:\exp2>nasm prog4.asm -o prog4.bin

D:\exp2>nasm int_caller.asm -o int_caller.bin

D:\exp2>
```

都编译好以后，我们就可以把他们整合成一个 flp 文件了，把相应的文件分配到相应的地方，完成！

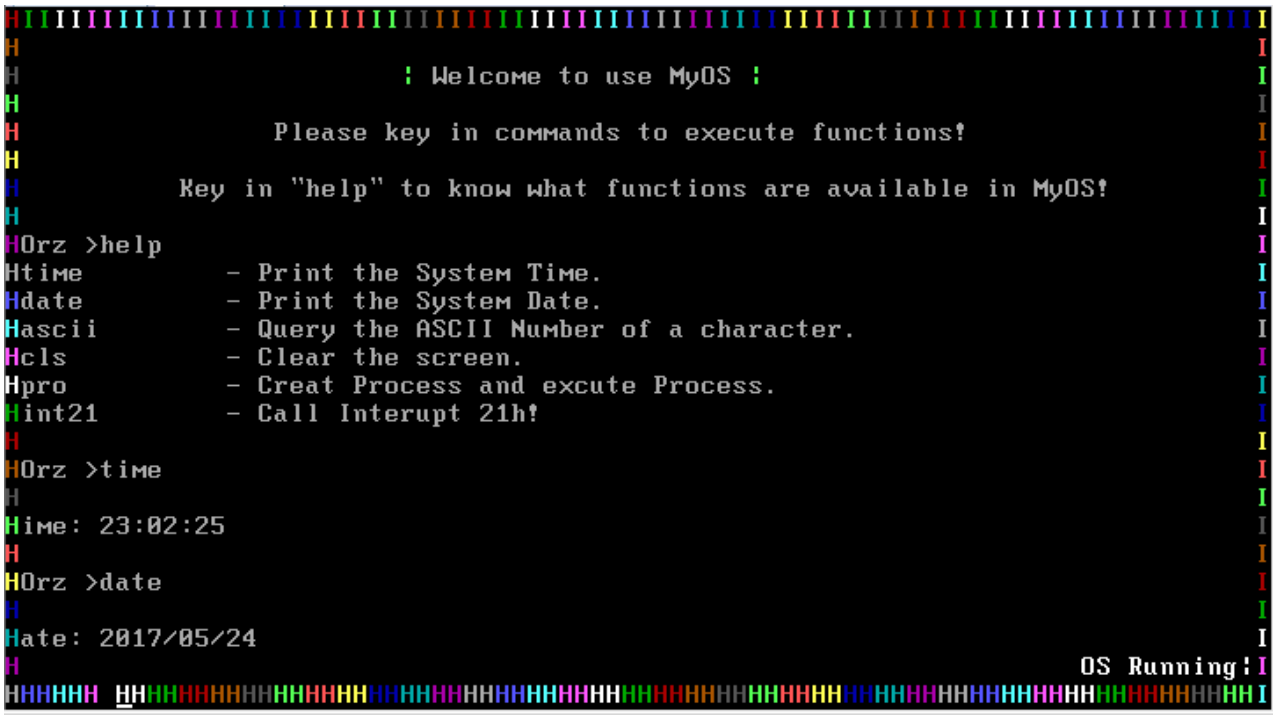
接下来就是使用 wmware 去打开我们的操作系统了：

以上是开始界面，我们可以看到有七彩字符画框，也有明显的横杠旋转标志，证明时钟中断正在运行。

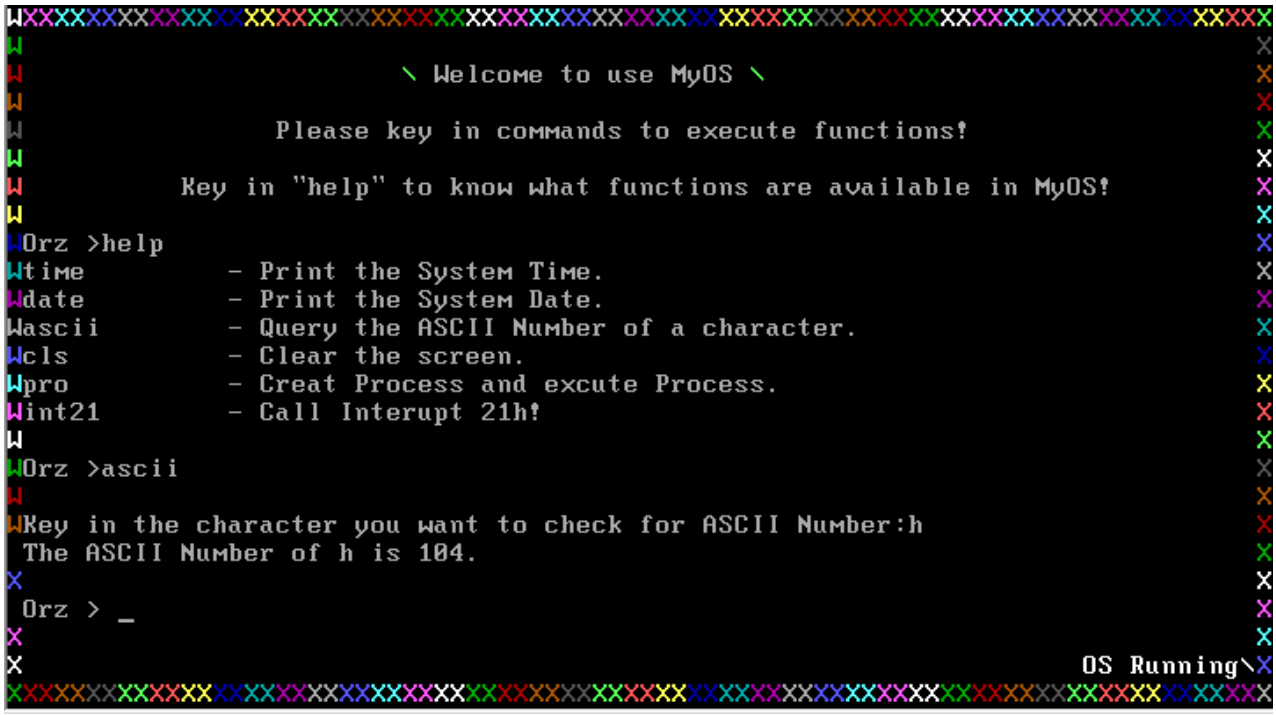
输入 help 指令，得出帮助菜单：


\ Welcome to use MyOS \
Please key in commands to execute functions!
Key in "help" to know what functions are available in MyOS!
Orz >help
time - Print the System Time.
date - Print the System Date.
ascii - Query the ASCII Number of a character.
cls - Clear the screen.
pro - Creat Process and excute Process.
int21 - Call Interupt 21h!
Orz >_
OS Running

分别输出时间和日期：

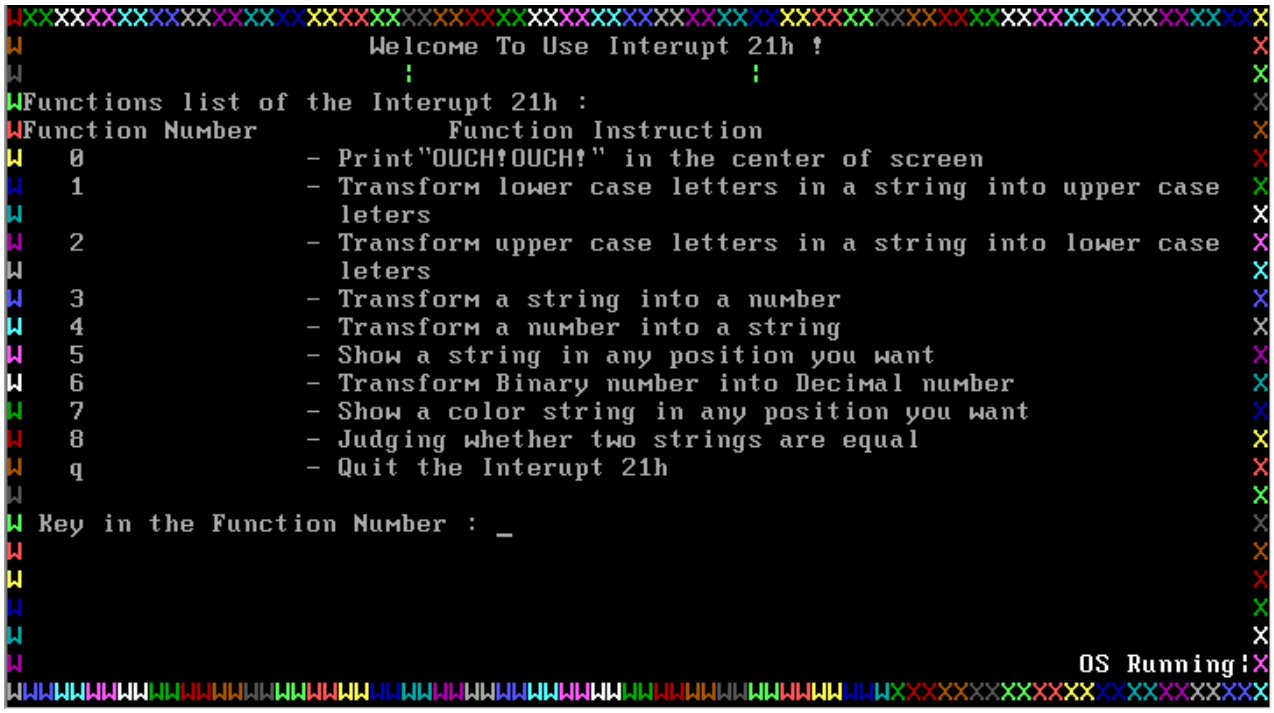

: Welcome to use MyOS :
Please key in commands to execute functions!
Key in "help" to know what functions are available in MyOS!
Orz >help
time - Print the System Time.
date - Print the System Date.
ascii - Query the ASCII Number of a character.
cls - Clear the screen.
pro - Creat Process and excute Process.
int21 - Call Interupt 21h!
Orz >time
Time: 23:02:25
Orz >date
Date: 2017/05/24
OS Running!

转换 ascii 码功能正常：



用户程序的运行和 21 号中断的测试是成功的，但是由于内容太多了，我们将会在附件中传上我们的虚拟软盘，麻烦老师亲自测试了！

以下是 21 号中断功能的界面：



【实验总结】

宗嘉希　15352461： 首先吧，这一次的实验应该是学习了操作系统这门课程这么久以来最难做的实验了。在之前的实验中，我们完成过以批处理的方式去把多个用户程序运行，就是说先运行完一个程序，然后再运行下一个程序，这样子顺序运行。我也以附加功能的形式完成过一种伪分时运行多个用户程序的方案，但是那种方案仅仅是把用户程序放在制定的内存位置中，每一次运行一小段时间都要跳转到其他代码段运行下一段用户程序的代码，但是这样子并不保险，可以说得上是很危险，因为寄存器的值没有得到很好的保护，根本谈不上是分时运行用户程序，因此只一次实验，我们就可以真正的运行分时了。在这一次的实验中，我们学习了什么是二状态进程模型，把程序分为运行中和准备（不运行）状态，然后使用 c 语言来创建一个进程管理模块的数据结构，把每一次一个程序运行过程中，每一个寄存器的值存下来，等到下一次要运行着一个程序的时候再把那些值复原到寄存器中，又可以继续运行本来还没有运行完成的程序了！而且我们利用时钟中断当作触发，每一次出发换一次用户程序，那么就可以把在很快的时间内不停地轮转运行多个用户程序，达到分时的效果。但是不得不说，这一次的实验真的很难，首先要搞清楚 save 和 restart 的过程，太复杂了，关系到栈里面的内容，必须要保证不能出一丝差错，否则就会导致程序完全崩盘，要是没有老师的 ppt 上面的提示和原型代码，真的很难完成本次实验。除了以上这些东西，还有一些别的老问题，比如说用户程序存放的地址问题啊，段地址问题啊什么的，在实验过程中遇到了太多的问题了，但是我们都把困难一一克服了，还修复了原型中一些不合理的地方。所以说，以后要更加努力的学习操作系统啊，因为还有更多更加困难的知识等着我去攻克！

周禅城 15352448：这次的操作系统实验的确如老师所说是个台阶，相比之前的实验，这次实验的难度有了很大的提升，可能是为了以后的实验做铺垫吧。这次的试验主要是关于进程的二状态模型，主要要实现 PCB（进程控制块）与进程表，然后理解进程交替执行的原理，在理解的基础上利用时钟中断实现用户程序轮流执行。这次实验理解就很有困难，在理解时间轮转法时就出现了很大的障碍。具体实现过程中遇到的第一个大问题是连接问题，然后发现连接时 C 文件一定要摆在最后面才会连接成功，不然会出现数据已经定义的错误。第二个问题就是我们的用户程序载入的问题，一开始我们的用户程序只能读入一个扇区，总的来说这次实验收获还是挺大的，希望在这个基础上那个以后的实验会简单吧。

钟凌山 15352443：本次实验的内容是一个非常实用且应用广泛的功能，即在单线程 CPU 中“同时”运行多个程序（可以理解为手机中的“多任务”功能）。此功能依赖于时钟中断，中断每隔一段极小的时间执行，中断每执行一次，则将当前正在运行的程序“挂起”，并把此时寄存器中的数据存储在 PCB 中，然后将下一个将要运行的程序“恢复”，将存储在 PCB 中的数据恢复回寄存器当中，这样就成功地将一个程序挂起并执行下一个程序。由于此过程的间隔很小，在用户看来，多个程序似乎在同时运行。本功能能实现的关键就是各个程序的挂起和恢复，以及进程的管理。PCB 结构用 C 语言非常易于实现。这次的实验能让我们进一步了解一个操作系统内部的运行机制，特别是多任务运行的基本原理。

附录：

源代码：

用户程序 ：

prog1.asm

prog2.asm

prog3.asm

prog4.asm

int_caller.asm

内核代码 ：

klibc.asm

kliba.asm

myos.asm

kdata.h

cmain.c

引导扇区代码：

boot.asm

编译文件：

prog1.bin

prog2.bin

prog3.bin

prog4.bin

int_caller.bin

klibc.obj

myos.obj

cmain.obj

os.com

boot.bin

镜像文件：

144mb.img