实验6：实现时间片轮转的二态进程模型

院系	专业	年级	姓名
数据科学与计算机学院	人工智能与大数据	2018级	陈琮昊

一、实验目的：

1、学习多道程序与CPU分时技术

2、掌握操作系统内核的二态进程模型设计与实现方法

3、掌握进程表示方法

4、掌握时间片轮转调度的实现

二、实验要求：

1、了解操作系统内核的二态进程模型。

2、扩展实验五的的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。

3、修改时钟中断处理程序，调用时间片轮转调度算法。

4、设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行。

5、修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的现场。

6、编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性。

三、实验方案：

1.运行环境：

该实验在`Windows+Linux`下，选用`gcc+NASM+ld`的组合运行。

所需软件：`VMware Workstation 15.5 Pro (Windows)`、`WinHex 19.8 (Windows)`、`WSL Ubuntu 18.04 LTS`、`GCC 7.5.0 (Linux)`、`NASM 2.13.02 (Linux)`、`GNU ld 2.30 (Linux)`

虚拟机`VMware`用来运行程序，`WSL Ubuntu 18.04 LTS`是以软件形式在`Windows`下运行的`Linux`子系统，`NASM`为汇编语言编译器，`WinHex`用来将文件用16进制显示，`GCC`为C语言编译器，`GNU ld`用来链接代码。

2.实验流程：

1.配备好相关的软硬件

2.编写代码以实现相关功能

3.得到所需镜像文件

4.导入虚拟机运行并观察结果

其中编写代码在`windows`的文本编辑器下进行；编译、链接则在`WSL`下进行；最终在虚拟机观察运行结果则是在`windows`下的`VMware`。

3.程序分析：

1.程序结构如下：

柱面号	磁头号	扇区号	扇区数（大小）	内容
0	0	1	1（512B）	引导程序
0	0	2	1（512B）	用户程序信息表
0	0~1	0:3~1:18	34（17KB）	操作系统内核
1	0	1~2	2（1KB）	用户程序1
1	0	3~4	2（1KB）	用户程序2
1	0	5~6	2（1KB）	用户程序3
1	0	7~8	2（1KB）	用户程序4
1	0	10	2（1KB）	系统调用

2.文件目录如下：

主题	所包含文件
内核	`osstarter.asm`、`liba.asm`、`libc.c`（包含`stringio.h`）
用户程序	`stonelu.asm`、`stoneld.asm`、`stoneru.asm`、`stonerd.asm`
引导程序	`bootloader.asm`
相关功能	`usrproginfo.asm`、`hotwheel.asm`、`multiprocess.asm`
宏定义	`macro.asm`
系统调用	`test.asm`、`systema.asm`、`systemc.c`

3.重点分析：

注：与之前实验相同的模块不再进行分析

（1）每个用户程序独占一个段

前几次实验不涉及多进程的问题，所以设计时就从简了，没考虑那么多，用户程序和内核处于同一个段中，但这不利于进程切换。本次实验中，设定内核加载在第一个`64K`中（段值为`0x0000`）；用户程序从第二个`64K`内存开始分配，每个进程`64K`，即四个用户程序的段值分别为`0x1000`、`0x2000`、`0x3000`、`0x4000`。这些地址以常量的形式定义在`macro.asm`中：

```
addr_oskernel equ 08000h     ; 操作系统内核被装入的位置
addr_usrprog1 equ 10000h     ; 四个普通用户程序的物理地址
addr_usrprog2 equ 20000h     ; 四个普通用户程序的物理地址
addr_usrprog3 equ 30000h     ; 四个普通用户程序的物理地址
addr_usrprog4 equ 40000h     ; 四个普通用户程序的物理地址
```

四个反弹的用户程序不做任何修改也没有问题，不过考虑到对后面的多进程执行有可能带来影响，因此做了一点简单的修改：

1. 删除了用户程序显示交互信息的字符串，因为这样显示起来字符会叠加，屏幕因此会很乱；
2. 修改了每个用户程序显示字符的起始位置，让每个程序显示各自字符时稍微错开一点。

（2）定义进程控制块（PCB）和进程表

在汇编中定义8个进程控制块，编号从0到7。每个PCB的结构都相同，因此考虑使用宏：

```
%macro ProcessControlBlock 1  ; 参数：段值
    dw 0                      ; ax，偏移量=+0
    dw 0                      ; cx，偏移量=+2
    dw 0                      ; dx，偏移量=+4
    dw 0                      ; bx，偏移量=+6
    dw 0FE00h                 ; sp，偏移量=+8
    dw 0                      ; bp，偏移量=+10
    dw 0                      ; si，偏移量=+12
    dw 0                      ; di，偏移量=+14
    dw %1                     ; ds，偏移量=+16
    dw %1                     ; es，偏移量=+18
    dw %1                     ; fs，偏移量=+20
    dw 0B800h                 ; gs，偏移量=+22
    dw %1                     ; ss，偏移量=+24
    dw 0                      ; ip，偏移量=+26
    dw %1                     ; cs，偏移量=+28
    dw 512                    ; flags，偏移量=+30
    db 0                      ; id，进程ID，偏移量=+32
    db 0                      ; state，{0:新建态；1:就绪态；2:运行态}，偏移量=+33
%endmacro

pcb_table:                    ; 定义PCB表
pcb_0: ProcessControlBlock 0
pcb_1: ProcessControlBlock 1000h
pcb_2: ProcessControlBlock 2000h
pcb_3: ProcessControlBlock 3000h
pcb_4: ProcessControlBlock 4000h
pcb_5: ProcessControlBlock 0
pcb_6: ProcessControlBlock 0
pcb_7: ProcessControlBlock 0
```

每个PCB占34个字节，注释中的**偏移量**的意思是相对于当前PCB首地址偏移的字节数。

（3）现场保护：save 过程

基本思路是，将此过程写成一个汇编函数`pcbSave`，在进入时钟中断后立即将所有寄存器的值压栈，然后调用该函数。在`pcbSave`函数中，从栈上取得寄存器的值，并将它们保存在当前进程的PCB中。

找到当前PCB的地址并不难：前面定义了8个PCB，每个PCB占34个字节，它们是连续存放的，首地址在标签`pcb_table`处。变量`current_process_id`保存着当前进程的序号，故当前PCB的首地址可由`pcb_table + current_process_id*34`算出。

`pcbSave`函数的代码如下：

```
pcbSave:                        ; 函数：现场保护
    pusha
    mov bp, sp
    add bp, 16+2                ; 参数首地址
    mov di, pcb_table

    mov ax, 34
    mul word[cs:current_process_id]
    add di, ax                  ; di指向当前PCB的首地址

    mov ax, [bp]
    mov [cs:di], ax
    mov ax, [bp+2]
    mov [cs:di+2], ax
    mov ax, [bp+4]
    mov [cs:di+4], ax
    mov ax, [bp+6]
    mov [cs:di+6], ax
    mov ax, [bp+8]
    mov [cs:di+8], ax
    mov ax, [bp+10]
    mov [cs:di+10], ax
    mov ax, [bp+12]
    mov [cs:di+12], ax
    mov ax, [bp+14]
    mov [cs:di+14], ax
    mov ax, [bp+16]
    mov [cs:di+16], ax
    mov ax, [bp+18]
    mov [cs:di+18], ax
    mov ax, [bp+20]
    mov [cs:di+20], ax
    mov ax, [bp+22]
    mov [cs:di+22], ax
    mov ax, [bp+24]
    mov [cs:di+24], ax
    mov ax, [bp+26]
    mov [cs:di+26], ax
    mov ax, [bp+28]
    mov [cs:di+28], ax
```

```
    mov ax, [bp+30]
    mov [cs:di+30], ax

    popa
    ret
```

（4）进程调度：schedule 过程

在二态进程模型中，进程调度的目的是把`current_process_id`的值修改为下一个处于就绪态进程的序号。

可以将其写成一个汇编函数`pcbSchedule`，实现思路如下：首先将当前进程的状态由运行态改为就绪态，然后递增`current_process_id`使其指向下一个进程，并检查下一进程的状态是否是就绪态，若是就绪态，则调度完毕；若不是就绪态，则循环递增`current_process_id`直到找到一个就绪进程。由于`current_process_id`的值不应该超过7，因此需要在其超过7时把它重置为1。基于以上思路编写代码：

```
pcbSchedule:                    ; 函数：进程调度
    pusha
    mov si, pcb_table
    mov ax, 34
    mul word[cs:current_process_id]
    add si, ax                  ; si指向当前PCB的首地址
    mov byte[cs:si+33], 1       ; 将当前进程设置为就绪态

    try_next_pcb:               ; 循环地寻找下一个处于就绪态的进程
        inc word[cs:current_process_id]
        add si, 34              ; si指向下一PCB的首地址
        cmp word[cs:current_process_id], 7
        jna pcb_not_exceed      ; 若id递增到8，则将其恢复为1
        mov word[cs:current_process_id], 1
        mov si, pcb_table+34    ; si指向1号进程的PCB的首地址

    pcb_not_exceed:
        cmp byte[cs:si+33], 1 ; 判断下一进程是否处于就绪态
        jne try_next_pcb        ; 不是就绪态，则尝试下一个进程
        mov byte[cs:si+33], 2 ; 是就绪态，则设置为运行态。调度完毕
    popa
    ret
```

注：后面还会提到这一过程。

（5）现场恢复：restart 过程

该过程直接写在时钟中断里面。具体来说就是从调度后的新PCB中取出各个寄存器的值并将它们恢复到相应寄存器中去。我使用`si`寄存器来间址，在恢复了所有其他寄存器后，最后恢复`si`寄存器。

在这个过程中，需要格外注意的是`sp`寄存器。注意到在为`pcbSave`函数传参的过程中，传递的`sp`的值并不是发生时钟中断前的那个用户进程的`sp`的值。在进入时钟中断后，首先栈中会增加`psw`、`cs`、`ip`3 个字，然后又压入了`ss`、`gs`、`fs`、`es`、`ds`、`di`、`si`、`bp`8个字。这样，最后恢复的`sp`的值实际上比正确的`sp`的值偏小了22 个字节，不要忘记将其加上。

在恢复完所有寄存器的值后，就可以准备返回进程了。只需依次将新进程的`psw`、`cs`、`ip`压栈，然后中断返回即可。这部分代码如下：

```
pcbRestart:
    mov si, pcb_table
    mov ax, 34
    mul word[cs:current_process_id]
    add si, ax                  ; si指向调度后的PCB的首地址

    mov ax, [cs:si+0]
    mov cx, [cs:si+2]
    mov dx, [cs:si+4]
    mov bx, [cs:si+6]
    mov sp, [cs:si+8]
    mov bp, [cs:si+10]
    mov di, [cs:si+14]
    mov ds, [cs:si+16]
    mov es, [cs:si+18]
    mov fs, [cs:si+20]
    mov gs, [cs:si+22]
    mov ss, [cs:si+24]
    add sp, 11*2                ; 恢复正确的sp

    push word[cs:si+30]         ; 新进程flags
    push word[cs:si+28]         ; 新进程cs
    push word[cs:si+26]         ; 新进程ip

    push word[cs:si+12]
    pop si                      ; 恢复si

QuitTimer:
    push ax
    mov al, 20h
    out 20h, al
    out 0A0h, al
    pop ax
    sti                         ; 开中断
    iret
```

（6）bat指令进入多进程

在输入`bat`指令后，开始并行运行相应编号的用户程序。比如，当输入`bat 1 2 3 4`时，同时运行四个用户程序；当输入`bat 1 3`时，同时执行这两个用户程序。

首先，需要把将要运行的用户程序写入内存中的规定地址去，并把对应的PCB中的`state`字段设置为就绪态，还要初始化PCB中的段寄存器。上面这几个功能写成一个函数，并放在`multiprocess.asm`文件中。该函数是由32位的C代码调用的，因此需要用`retf`来返回。

```
loadProcessMem:                 ; 函数：将某个用户程序加载入内存并初始化其PCB
    pusha
    mov bp, sp
    add bp, 16+4                ; 参数地址
    LOAD_TO_MEM [bp+12], [bp], [bp+4], [bp+8], [bp+16], [bp+20]

    mov si, pcb_table
```

```
    mov ax, 34
    mul word[bp+24]                 ; progid_to_run
    add si, ax                      ; si指向新进程的PCB

    mov ax, [bp+24]                 ; ax=progid_to_run
    mov byte[cs:si+32], al          ; id
    mov ax, [bp+16]                 ; ax=用户程序的段值
    mov word[cs:si+16], ax          ; ds
    mov word[cs:si+18], ax          ; es
    mov word[cs:si+20], ax          ; fs
    mov word[cs:si+24], ax          ; ss
    mov word[cs:si+28], ax          ; cs
    mov byte[cs:si+33], 1           ; state设其状态为就绪态
    popa
    retf
```

同时，在`libc.c`中要编写`bat`命令所调用的函数`multiProcessing`：先是参数检查（写起来比较容易）；然后循环地为每一个用户进程调用上面的`loadProcessMem`函数，将用户程序从软盘装入内存，然后将对应的PCB中的状态设置为就绪态。最后，设置标志量`timer_flag`为1（后面会提到），从而时钟中断开始在多个进程中轮转。

`multiProcessing`函数的代码比较容易理解，此处就不再解释了。

（7）杀死所有进程

这里提供了一种从多进程运行状态返回到内核的方法，当键入`e`时退出用户程序并回到内核。

首次从内核进入多进程的运行模式时，内核的寄存器状态信息被保存在了0号PCB中，因此在返回内核前只需使调度器选择0号PCB即可。除此之外，返回内核前还要将其余PCB全部重置，以便下次重新进入多进程。修改后的`pcbSchedule`如下：（前文已提到）

```
pcbSchedule:                        ; 函数：进程调度
    pusha
    mov si, pcb_table
    mov ax, 34
    mul word[cs:current_process_id]
    add si, ax                      ; si指向当前PCB的首地址
    mov byte[cs:si+33], 1           ; 将当前进程设置为就绪态

    mov ah, 01h                     ; 功能号：查询键盘缓冲区但不等待
    int 16h
    jz try_next_pcb                 ; 无键盘按下，继续
    mov ah, 0                       ; 功能号：查询键盘输入
    int 16h
    cmp al, 101                      ; 是否按下e
    jne try_next_pcb                ; 若按下e，回到内核；否则调度用户程序

    mov word[cs:current_process_id], 0
    mov word[cs:timer_flag], 0      ; 禁止时钟中断处理多进程

    call resetAllPcbExceptZero

    jmp QuitSchedule                ; 跳转回内核
    try_next_pcb:                   ; 循环地寻找下一个处于就绪态的进程
```

```
        inc word[cs:current_process_id]
        add si, 34                  ; si指向下一PCB的首地址
        cmp word[cs:current_process_id], 7
        jna pcb_not_exceed          ; 若id递增到8，则将其恢复为1
        mov word[cs:current_process_id], 1
        mov si, pcb_table+34        ; si指向1号进程的PCB的首地址
    pcb_not_exceed:
        cmp byte[cs:si+33], 1       ; 判断下一进程是否处于就绪态
        jne try_next_pcb            ; 不是就绪态，则尝试下一个进程
        mov byte[cs:si+33], 2       ; 是就绪态，则设置为运行态。调度完毕
    QuitSchedule:
    popa
    ret
```

为了保证下次还能正常地使用`bat`命令进入多进程，需要及时地将PCB表中的PCB重置为初始状态。这就是`resetAllPcbExceptZero`函数的功能，该函数把1到7号PCB中的所有寄存器映像全部重置为默认值。该函数在`pcbSchedule`中被调用。

```
resetAllPcbExceptZero:
    push cx
    push si
    mov cx, 7                       ; 共8个PCB
    mov si, pcb_table+34

    loop1:
        mov word[cs:si+0], 0        ; ax
        mov word[cs:si+2], 0        ; cx
        mov word[cs:si+4], 0        ; dx
        mov word[cs:si+6], 0        ; bx
        mov word[cs:si+8], 0FE00h   ; sp
        mov word[cs:si+10], 0       ; bp
        mov word[cs:si+12], 0       ; si
        mov word[cs:si+14], 0       ; di
        mov word[cs:si+16], 0       ; ds
        mov word[cs:si+18], 0       ; es
        mov word[cs:si+20], 0       ; fs
        mov word[cs:si+22], 0B800h  ; gs
        mov word[cs:si+24], 0       ; ss
        mov word[cs:si+26], 0       ; ip
        mov word[cs:si+28], 0       ; cs
        mov word[cs:si+30], 512     ; flags
        mov byte[cs:si+32], 0       ; id
        mov byte[cs:si+33], 0       ; state=新建态
        add si, 34                  ; si指向下一个PCB
        loop loop1

    pop si
    pop cx
    ret
```

(8) 设立时钟中断标志 timer_flag

只有在多进程的时候才需要时钟中断进行`save-schedule-restart`操作，其他时候直接返回即可。实现方法是设置一个标志量`timer_flag`，初始化为0，表示无需操作；当用户执行了`bat`命令后、将要运行多进程时，将`timer_flag`标志置为1，表示开始进行多进程。

在`multiprocess.asm`中定义标志：

```
timer_flag dw 0
```

在`libc.c`中引用该变量：

```
extern uint16_t timer_flag;
```

时钟中断处理程序的关键代码：

```
Timerm:                          ; 08h号时钟中断处理程序
    cli                         ; 关中断，不允许时钟中断嵌套
    cmp word[cs:timer_flag], 0
    je QuitTimer

    ...  ; save过程
    ...  ; schedule过程
    ...  ; restart过程

QuitTimer:
    push ax
    mov al, 20h
    out 20h, al
    out 0A0h, al
    pop ax
    sti                         ; 开中断
    iret
```

（9）测试系统调用：

1.功能号为`ah=00h`的系统调用：实现的功能是**在屏幕中间显示“OUCH”。**

利用BIOS的`int 10h`实现在屏幕某一位置打印彩色的“OUCH”字符串。实现见`systema.asm`中封装的`sys_showOuch`函数。

2.功能号为`ah=01h`的系统调用：实现的功能是**将`es:dx`的一个字符串中的小写字母变为大写字母。**

显然用C语言实现此功能比较方便。在`systemc.c`中实现`toupper`函数：

```
void toupper(char* str)
```

在`systema.asm`中封装：

```
[extern toupper]
sys_toUpper:
    push es             ; 传递参数
    push dx             ; 传递参数
    call dword toupper
    pop dx              ; 丢弃参数
    pop es              ; 丢弃参数
    ret
```

3.功能号为`ah=02h`的系统调用：实现的功能是**将`es:dx`的一个字符串中的大写字母变为小写字母。**

与上一条基本类似，不再赘述。

4.功能号为`ah=03h`的系统调用：实现的功能是**将`es:dx`的一个数字字符串转变对应的数值。**

使用C语言实现此功能。在`systemc.c`中实现`atoi`函数，然后在`systema.asm`内封装即可。

5.功能号为`ah=04h`的系统调用：实现的功能是**将`bx`的数值转变对应的`es:dx`的一个数字字符串。**

使用C语言实现此功能。在`systemc.c`中实现`itoa_buf`函数，该函数接受三个参数：`num`为待转换的数字，`base`为进制，`str`为存放字符串的缓冲区地址。

```
char* itoa_buf(int num, int base, char* str)
```

6.功能号为`ah=05h`的系统调用：实现的功能是**将`es:dx`的一个字符串显示在屏幕指定位置。**

在`systema.asm`中实现`sys_printInPos`函数：

```
[extern strlen]       ;该函数由C实现
sys_printInPos:
    pusha
    mov bp, dx          ; es:bp=串地址
    push es             ; 传递参数
    push bp             ; 传递参数
    call dword strlen ; 返回值ax=串长
    pop bp              ; 丢弃参数
    pop es              ; 丢弃参数
    mov bl, 07h         ; 颜色
    mov dh, ch          ; 行号
    mov dl, cl          ; 列号
    mov cx, ax          ; 串长
    mov bh, 0           ; 页码
    mov al, 0           ; 光标不动
    mov ah, 13h         ; BIOS功能号
    int 10h
    popa
    ret
```

测试系统调用的用户程序见`test.asm`。

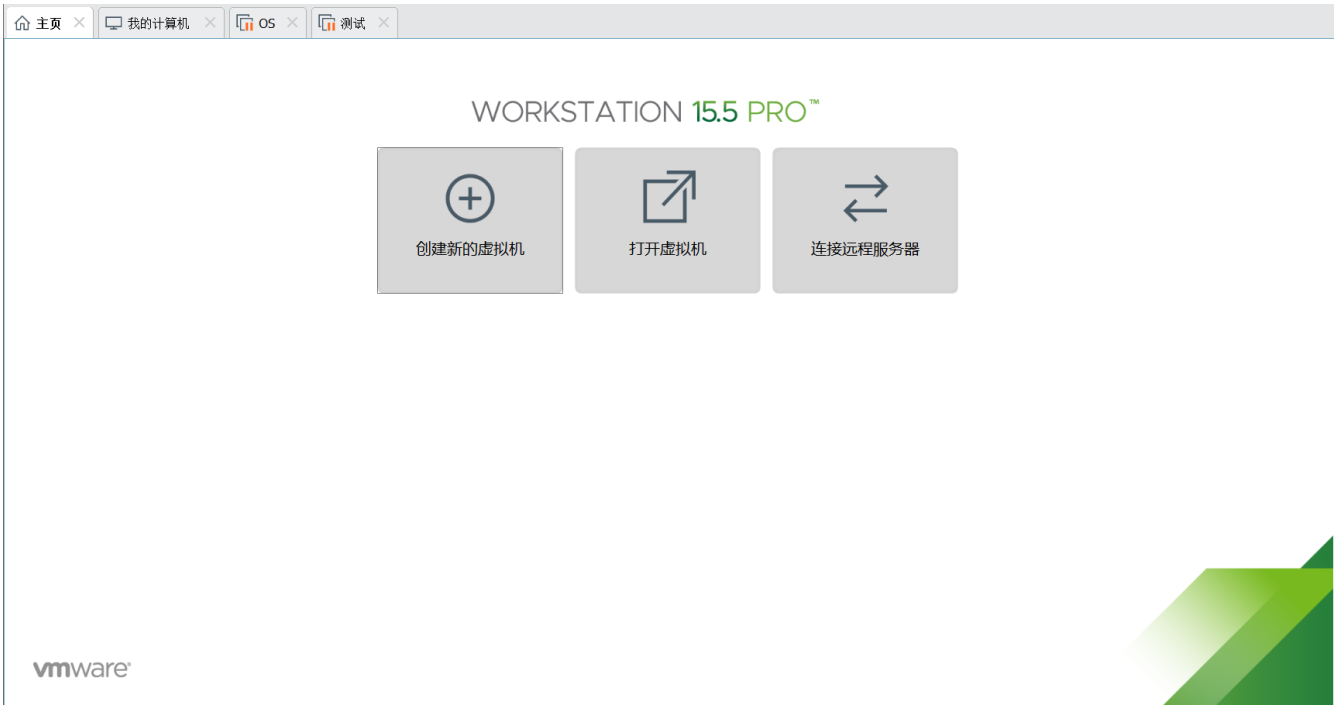
四、实验过程：

1.软件的下载与安装：

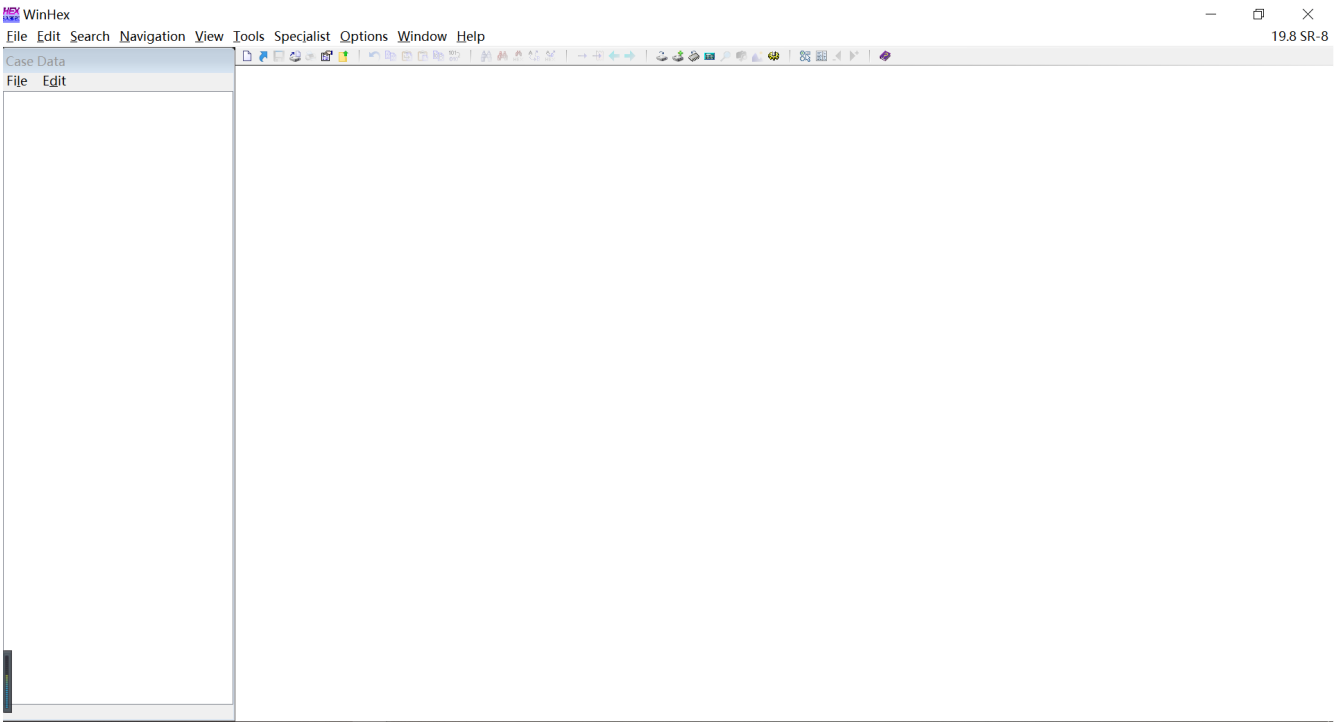
（1）在官网下载`VMware Workstation 15.5 Pro`：

https://www.vmware.com/cn/products/workstation-pro/workstation-pro-evaluation.html

安装该软件后初始界面如下（已经输入密钥并初步设置后）：



（2）下载`WinHex 19.8`压缩包，解压后找到`winhex.exe`，打开并解锁，界面如下：



（3）配置`Linux`环境，使用`Windows Subsystem for Linux (Ubuntu 18.04 LTS)`：

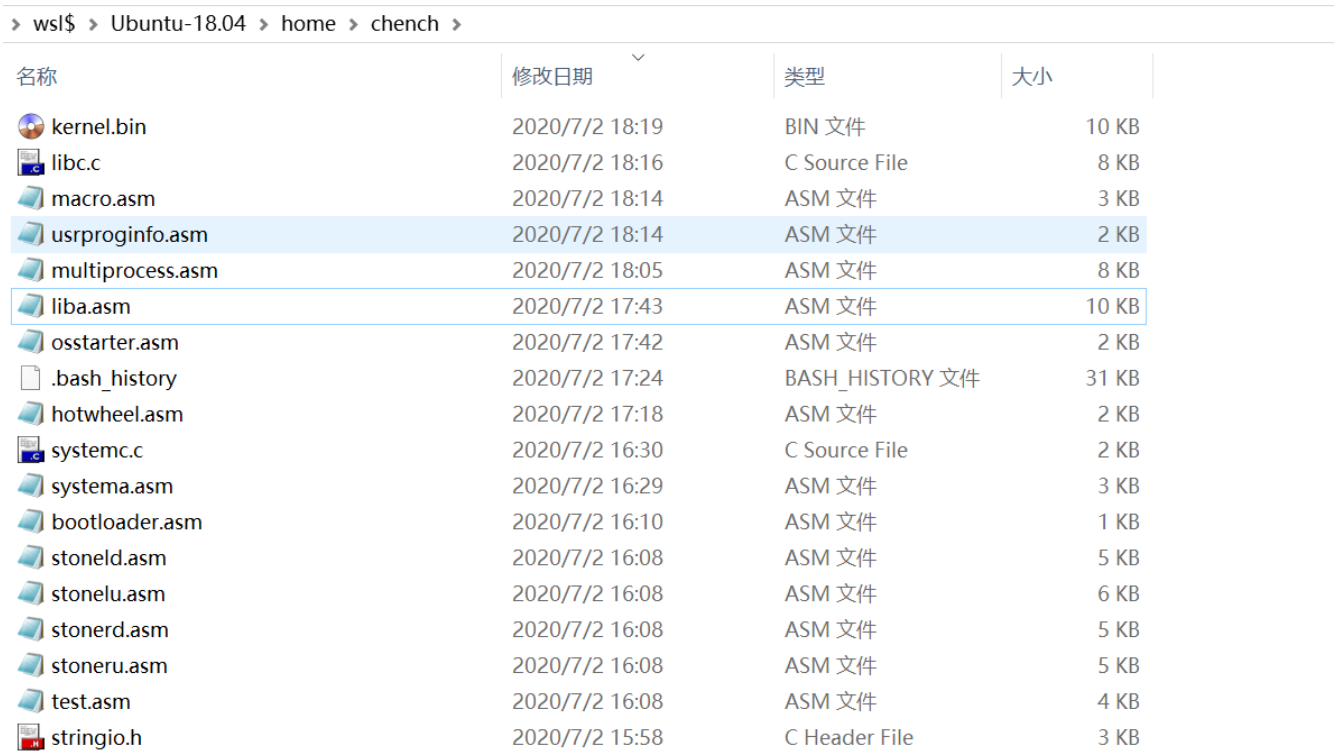
然后输入相关指令安装`gcc`、`NASM`、`GNU ld`，至此本次实验所需软件已配备完成。

2.Windows下编写代码：

代码关键部分已在`程序分析`一栏中介绍，具体代码内容见`src`文件夹。

3.配置相关文件：

首先，进入`WSL`的目录，将需要在`Linux`下编译的代码放在`WSL`目录下：



其次，进行编译链接：（借用之前实验且无需改动的程序就未再次进行编译）

```
chench@LAPTOP-TOEITVUA:~$ nasm -f bin bootloader.asm -o bootloader.bin
chench@LAPTOP-TOEITVUA:~$ nasm -f bin usrproginfo.asm -o usrproginfo.bin
chench@LAPTOP-TOEITVUA:~$ nasm -f bin test.asm -o test.bin
chench@LAPTOP-TOEITVUA:~$ nasm -f elf32 systema.asm -o systema.o
chench@LAPTOP-TOEITVUA:~$ nasm -f elf32 liba.asm -o liba.o
chench@LAPTOP-TOEITVUA:~$ nasm -f elf32 osstarter.asm -o osstarter.o
chench@LAPTOP-TOEITVUA:~$ nasm -f elf32 multiprocess.asm -o multiprocess.o
chench@LAPTOP-TOEITVUA:~$ gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -masm=intel -c libc.
c -o libc.o
chench@LAPTOP-TOEITVUA:~$ gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -masm=intel -c syste
mc.c -o systemc.o
chench@LAPTOP-TOEITVUA:~$ ld -m elf_i386 -N --oformat binary -Ttext 0x8000 osstarter.o liba.o libc.o hotwheel.o systema.
o systemc.o multiprocess.o -o kernel.bin
```

编译完成后可以在该目录下找到上述所有`.bin`文件。

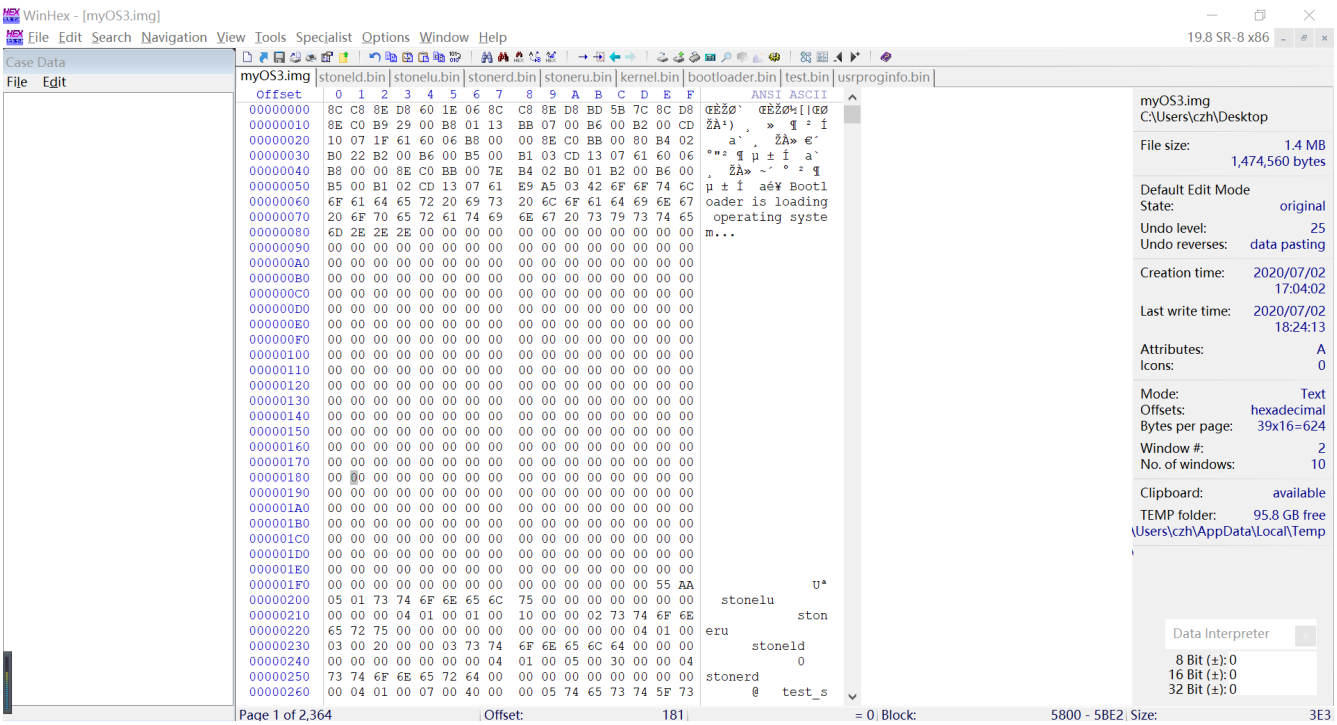
最后，便是得到本次实验所需的镜像文件：

将实验5中生成的镜像文件`myOS__.img`复制一份并在`WinHex`内打开，需要改动的地方将新生成的`bin`文件进行复制粘贴，将原文件的内容进行覆盖，命名为`myOS3.img`。

打开制作软盘所需的所有文件（共8个），然后将这8个文件按照如下顺序复制粘贴至新建的文件中：

`bootloader.bin`->`usrproginfo.bin`->`kernel.bin`->`stonelu.bin`->`stoneru.bin`->`stoneld.bin`->`stonerd.bin`->`test.bin`

`程序分析`部分已经列出了每个文件的扇区、大小等相关信息。因此，复制粘贴时需要注意`Offset`，如`usrproginfo.bin`的偏移为`0x0200`，`kernel.bin`的偏移为`0x0400`，第一个用户程序的偏移为`0x4800`.....`test.bin`的偏移为`0x5800`。



至此，便得到了命名为`myOS3.img`的镜像文件。

4.导入虚拟机：

打开虚拟机，选择`创建新的虚拟机`一项，然后进行如下配置：

安装客户机操作系统

虚拟机如同物理机，需要操作系统。您将如何安装客户机操作系统?

◉ 稍后安装操作系统(S)。

创建的虚拟机将包含一个空白硬盘。

帮助 | < 上一步(B) | 下一步(N) > | 取消

新建虚拟机向导

选择客户机操作系统

此虚拟机中将安装哪种操作系统?

客户机操作系统

○ Microsoft Windows(W)
○ Linux(L)
○ VMware ESX(X)
◉ 其他(O)

版本(V)

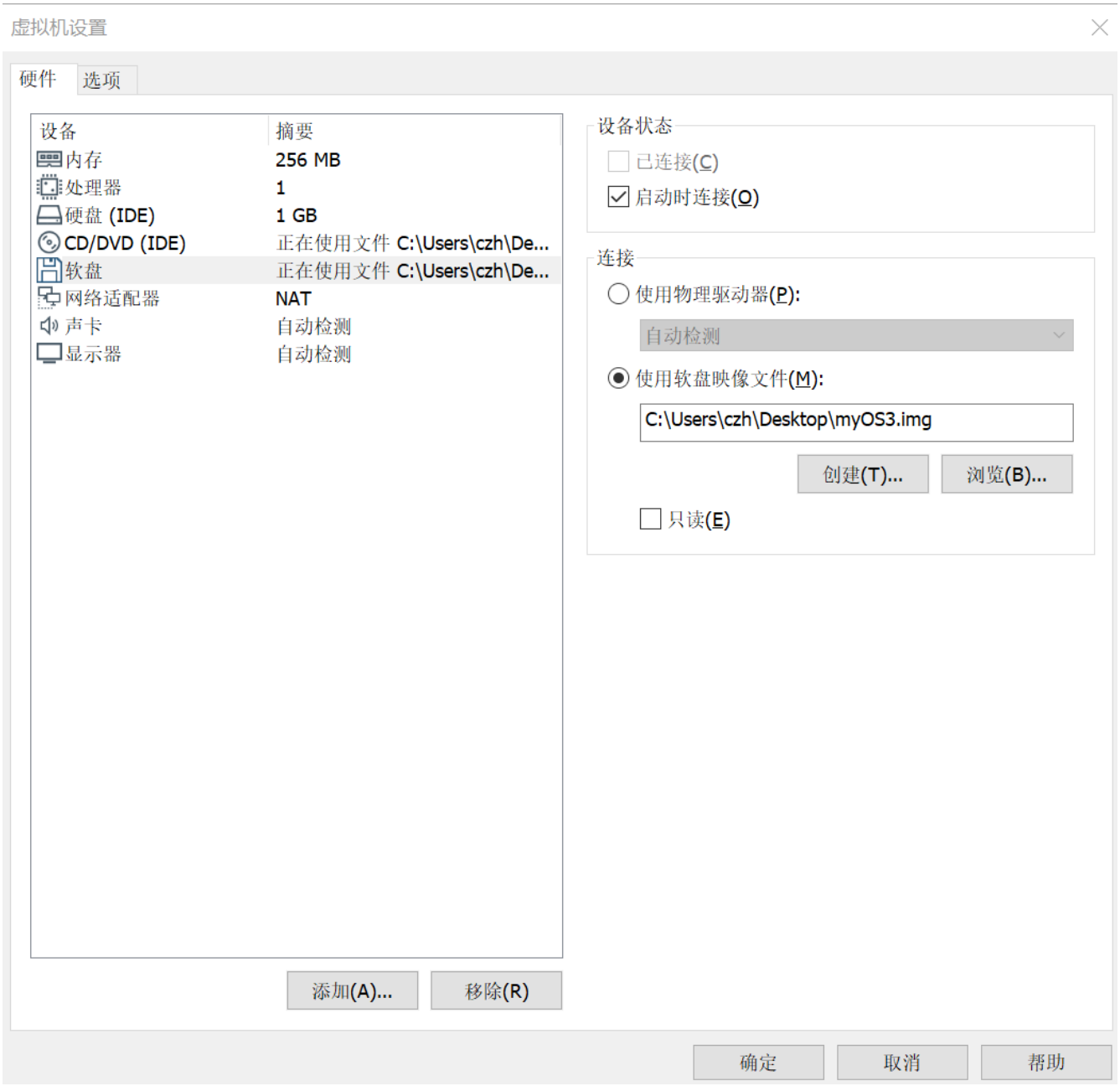
其他

帮助 | < 上一步(B) | 下一步(N) > | 取消

于是所使用的虚拟机也已配置完成。（后续步骤中选择磁盘容量可视情况而定，其他步骤则不赘述了。）

在上一步中，已经通过 `WinHex` 得到了最终的 `myOS3.img` 镜像文件，接下来就是将 `myOS3.img` 文件导入虚拟机，步骤如下：

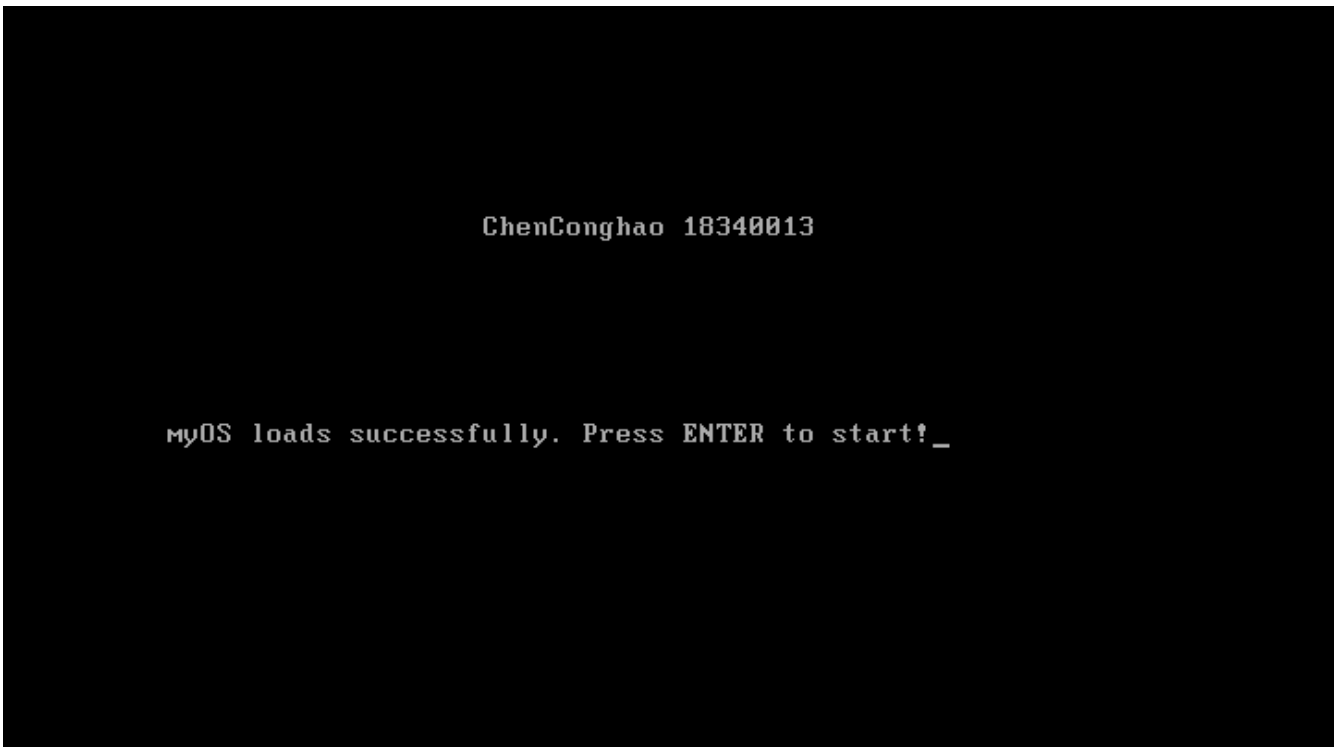
打开已创建好的虚拟机，然后选择 `编辑虚拟机设置`，然后添加 `软盘`，在 `连接` 一栏选择 `使用软盘映像文件`，将 `myOS3.img` 导入，`确定` 即可。



5.运行显示：

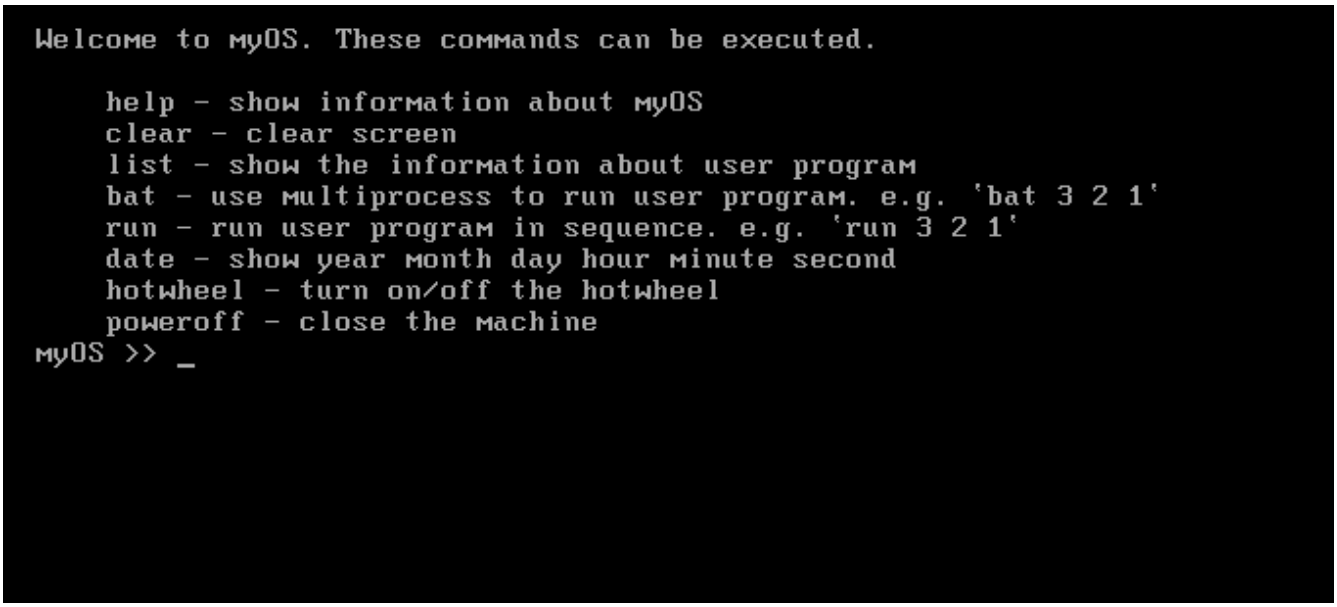
添加软盘完成后，打开虚拟机，从虚拟软盘启动后即可看到如下结果：

（1）初始界面：



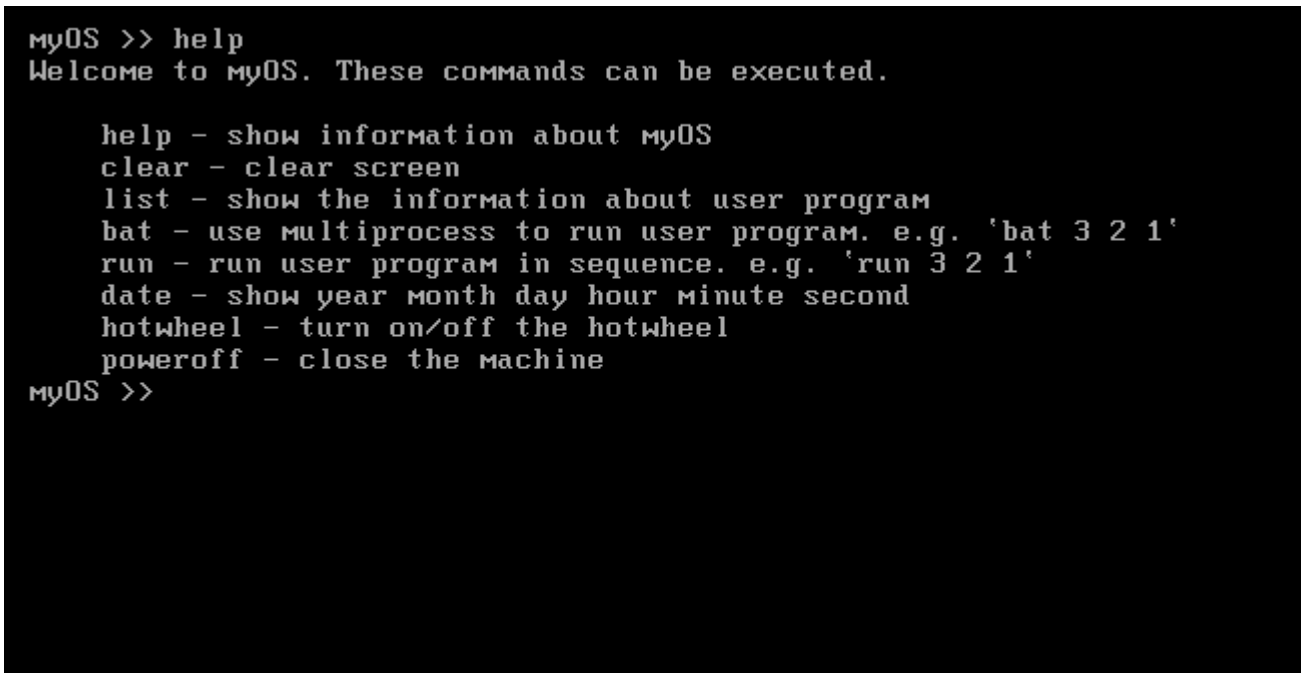
注：在一开始时风火轮是不转的。

（2）按`ENTER`进入后：



可以看到显示了可以执行的指令。

（3）执行`help`操作：



显示了可以执行的指令。

（4）执行`clear`操作：

可以看到清屏后只留下了命令行。

（5）执行`list`操作：

```
myOS >> help
Welcome to myOS. These commands can be executed.

    help - show information about myOS
    clear - clear screen
    list - show the information about user program
    bat - use multiprocess to run user program. e.g. 'bat 3 2 1'
    run - run user program in sequence. e.g. 'run 3 2 1'
    date - show year month day hour minute second
    hotwheel - turn on/off the hotwheel
    poweroff - close the machine
myOS >> list
You can use 'run <ProgID>' to run a user program.
ProgID  - Program Name  -  Size  -  Addr - Cylinder - Head - Sector
1  -  stonelu          -  1024  -  0x1000  -  1  -  0  -  1
2  -  stoneru          -  1024  -  0x2000  -  1  -  0  -  3
3  -  stoneld          -  1024  -  0x3000  -  1  -  0  -  5
4  -  stonerd          -  1024  -  0x4000  -  1  -  0  -  7
5  -  test_syscall     -  1024  -  0x0  -  1  -  0  -  9
myOS >> _
```

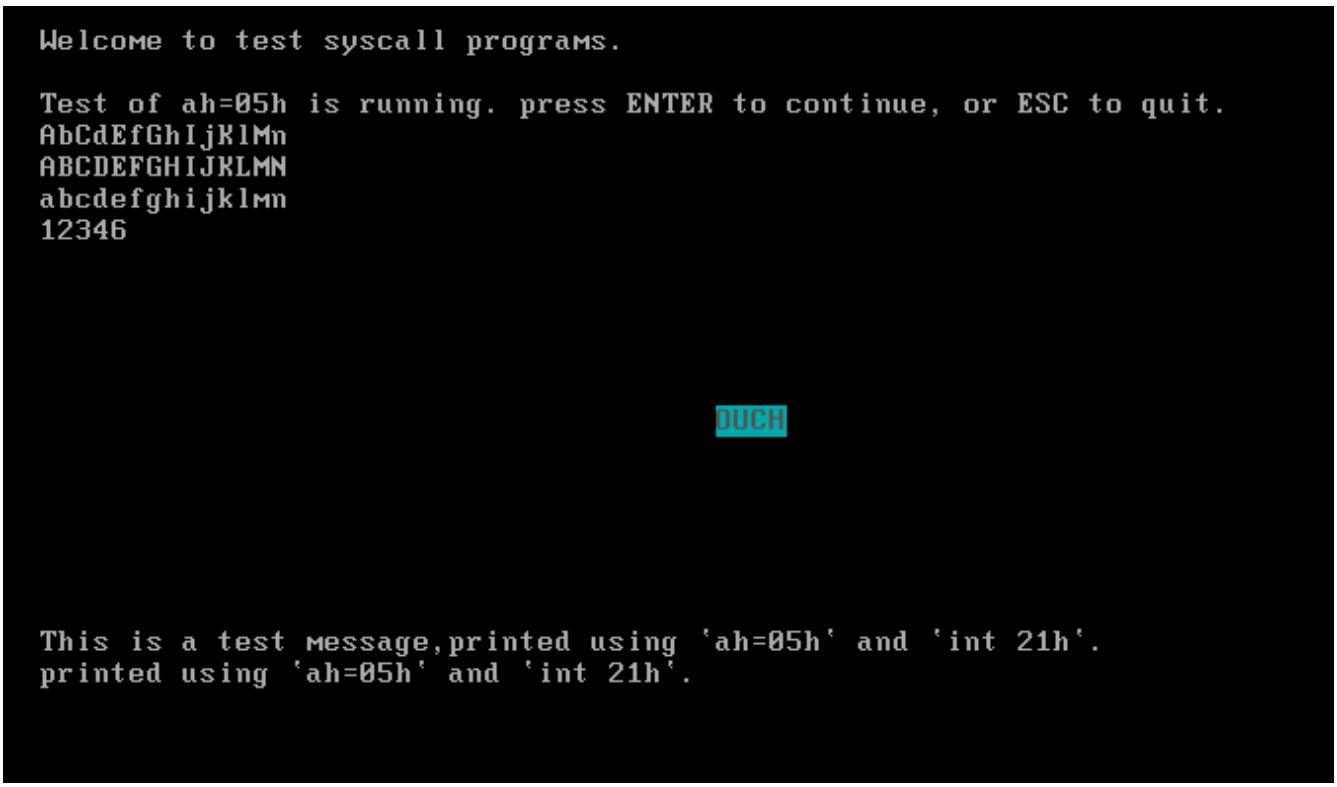
显示了4个用户程序以及测试系统调用程序的相关信息。

（6）`run`操作：连续的执行多个程序，且run后面的数字只能是1-5（见后面可知run 5是执行系统调用）。在执行多个程序过程中，中途可按`e`退出进入下一程序。所有程序执行完毕后会返回到该界面：

```
myOS >>
```

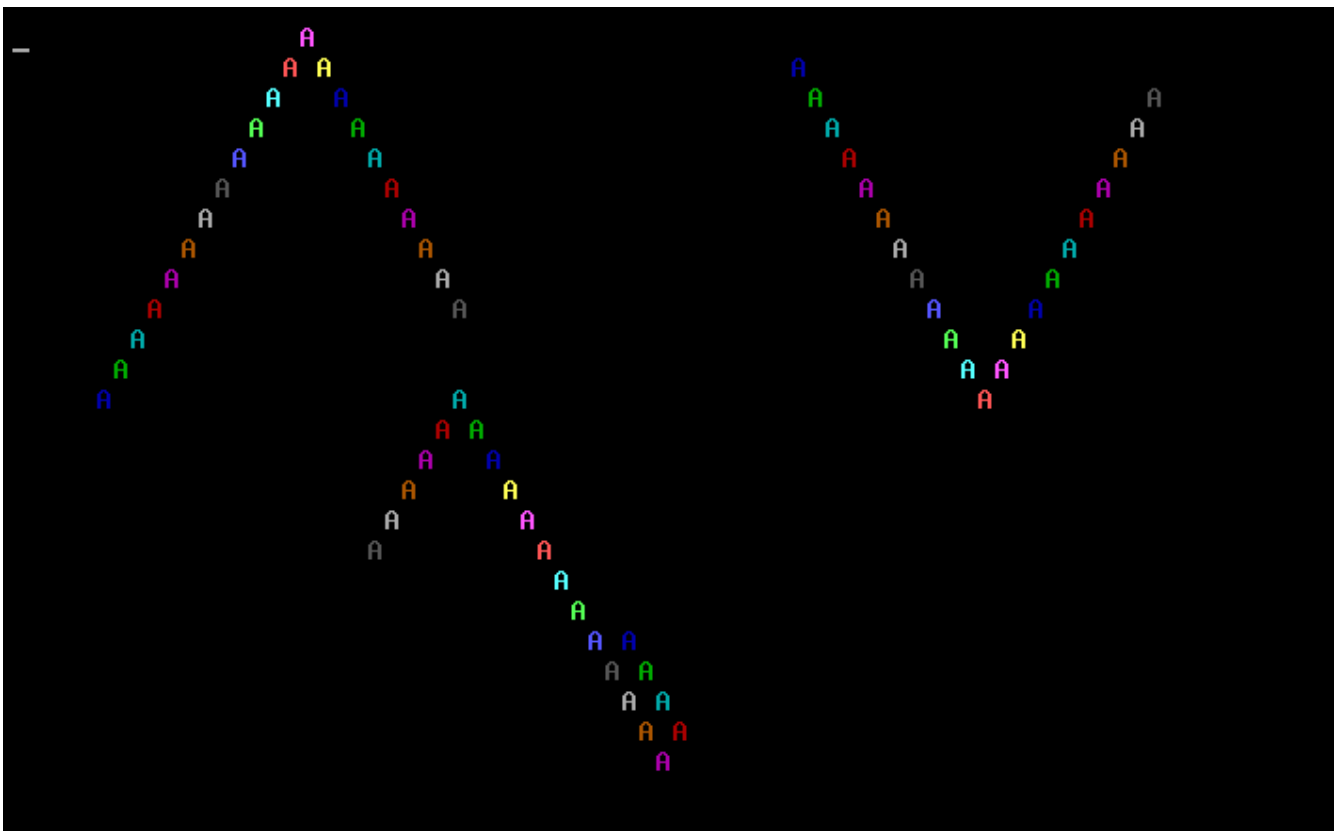
注：该功能见视频。

（7）执行系统调用：键入`run 5`即可进入系统调用界面，进入后通过按`ENTER`来执行不同的调用：当然，在中间过程按`e`也可退出系统调用。



注：该功能见视频。

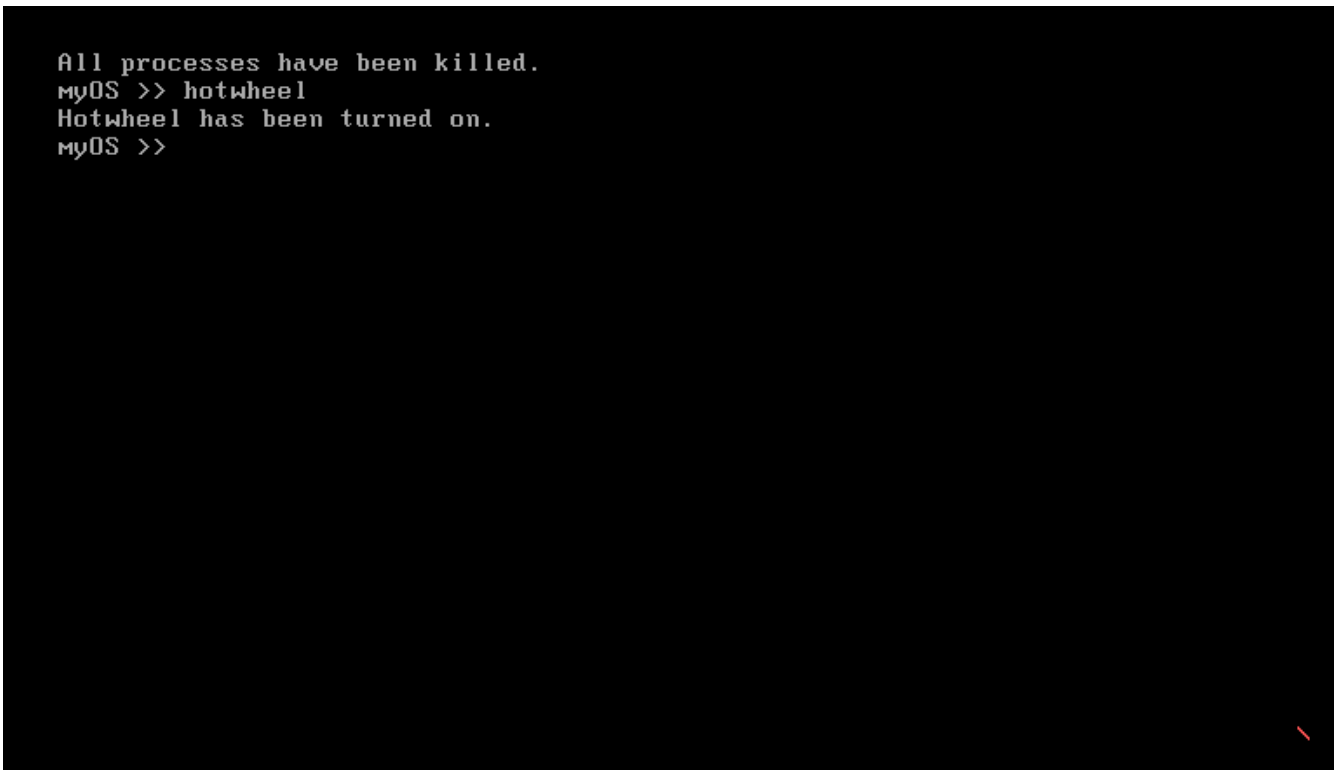
（8）执行`bat`操作：输入格式为`bat 3 2 1`，且`bat`后面的数字只能是`1-4`（不能是`5`！）该操作可执行多线程，同时显示多个用户程序：

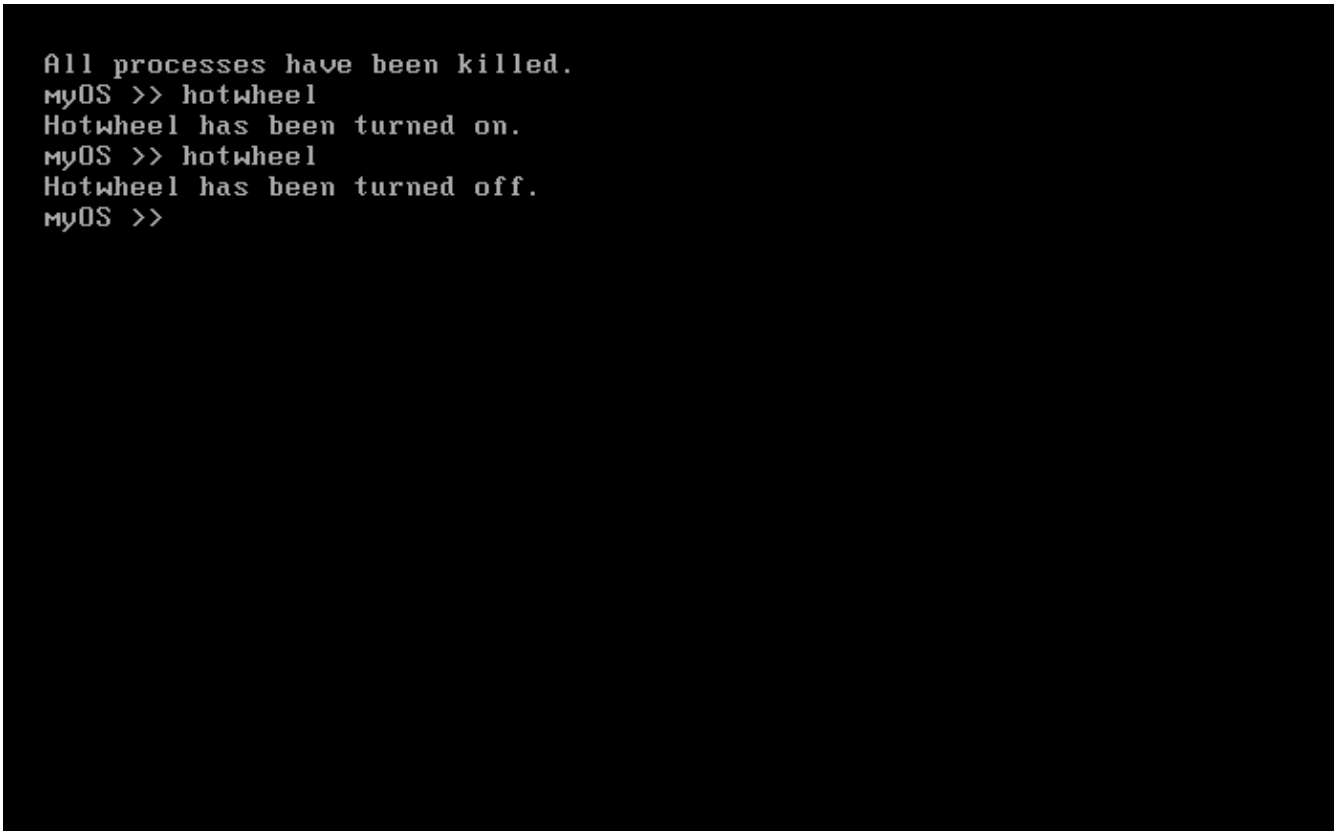


在中途键入`e`则可退出多线程的执行模式，并返回到如下界面：

注：该功能见视频。

（9）执行 `hotwheel` 操作：默认情况下不会在右下角显示"风火轮"，执行一次 `hotwheel` 则显示"风火轮"，再执行一次 `hotwheel` 则又关闭。

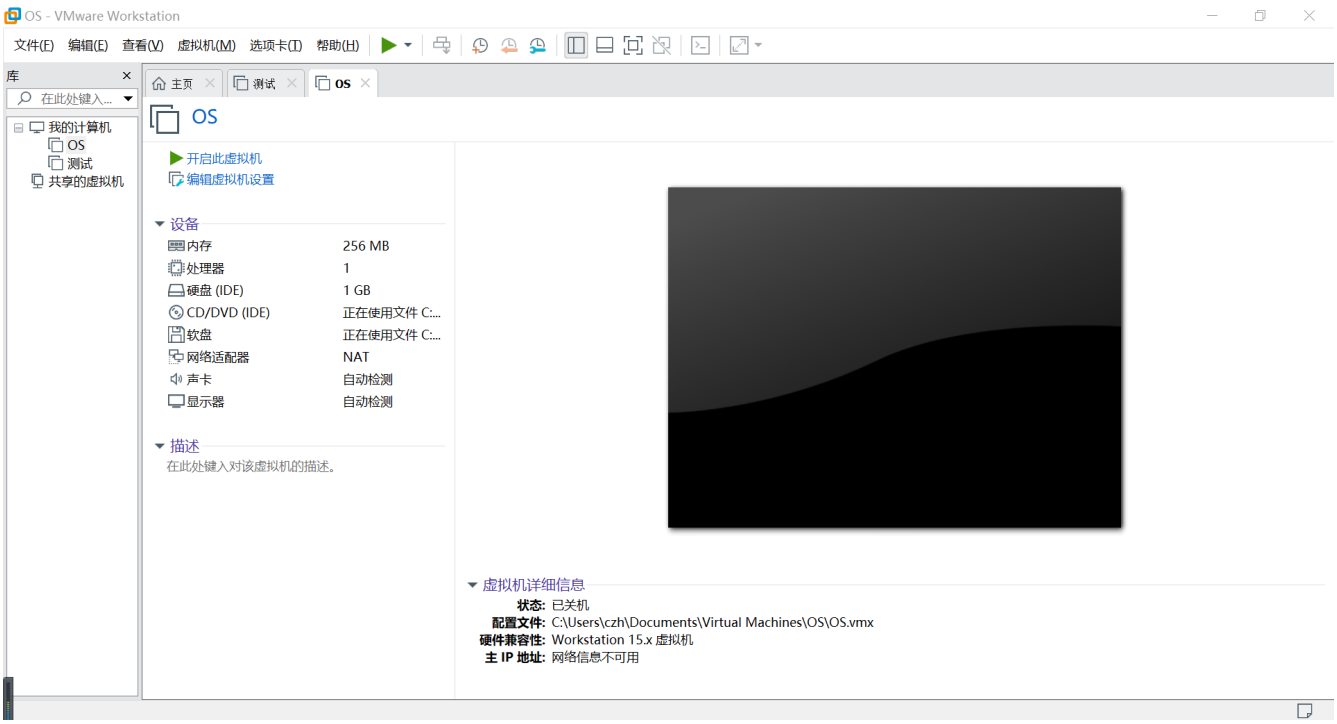




(10) 执行`date`指令：显示当前时间。

(11) 如果输入不正确的指令（如`aaa`），则会显示`command not found`：

（12）执行`poweroff`操作：直接关闭虚拟机。



五、实验总结：

就像在`程序分析`中提到的，在没有开启分页机制的情况下，线性地址就是物理地址，所以较为简单。在8086中逻辑地址由16位的段值和16位的偏移量共同构成20位的物理地址。计算方法是：将段值左移 4 位再加上偏移量即的到物理地址。一个物理地址可能由多个不同的逻辑地址表示，因此将物理地址拆分成逻辑地址的结果不是唯一的。不过我的设定简单粗暴：认为段值的低12位全为0。这样规定后，将物理地址拆分成逻辑地址的方法就是`phys_addr>>4&0F000h`。实现多进程的思路是十分清晰的：save过程负责将所有寄存器的值保存到存储器中的PCB内，schedule过程负责选择下一个就绪进程，restart过程负责把被调度的下一个进程PCB中的值恢复到对应的寄存器中，最后返回到被调度进程即可。但是思路清晰做起来可不容易！在`程序分析`一栏里可以看到如何处理**寄存器**就是个

非常麻烦的问题。可以看到由于寄存器的保护、恢复非常严谨，不能有任何错误。我就是在处理寄存器这一步上耗费了大量时间，在做的时候需要理清楚好多东西，要不感觉脑袋里一团乱。有好几次调试完编译链接没有问题了，结果导入虚拟机的时候崩了，真的是有点“心态爆炸”，不过好在最终坚持了下来，大约经过了6次这样的尝试，终于成功了！不过有两个小问题：第一就是四个用户程序的显示区域有交集。因为用户程序前几次涉及到的边界的值等参数都没动，所以在并行执行时会在边界附近有交集。第二就是由于加入了多线程，所以风火轮的默认状态是关闭而不是打开。

六、参考文献：

`李忠，王晓波，余洁.《x86汇编语言：从实模式到保护模式》.电子工业出版社，2012.`

`王爽.《汇编语言（第3版）》.`

`凌应标. 05实验课.pptx`

https://blog.csdn.net/jinixin/article/details/90345661?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase