

# 实验3：C与汇编开发独立批处理的内核

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、加深理解操作系统内核概念
- 2、了解操作系统开发方法
- 3、掌握汇编语言与高级语言混合编程的方法
- 4、掌握独立内核的设计与加载方法
- 5、加强磁盘空间管理工作

## 二、实验要求：

- 1、知道独立内核设计的需求
- 2、掌握一种 x86 汇编语言与一种C高级语言混合编程的规定和要求
- 3、设计一个程序，以汇编程序为主入口模块，调用一个C语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据，将程序生成COM格式程序，在DOS或虚拟环境运行。
- 4、汇编语言与高级语言混合编程的方法，重写和扩展实验二的的监控程序，从引导程序分离独立，生成一个COM格式程序的独立内核。
- 5、再设计新的引导程序，实现独立内核的加载引导，确保内核功能不比实验二的监控程序弱，展示原有功能或加强功能可以工作。
- 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下运行。

有两种方案可供选择，一种是 TASM+tcc+tlink，一种是 NASM+gcc+ld，由于第一种方案较为古老，代码语法等与现在差别较大，故选择了第二种方案来完成本次实验。

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.熟悉环境

- (1) 写一个混编程序，掌握编译链接的方法
- (2) 对一个C程序进行分析

### 3.完成本次实验：开发独立批处理内核

- (1) 编写引导程序
- (2) 编写独立内核
- (3) 根据实验一和实验二编写用户程序
- (4) 编译、链接
- (5) 制作软盘
- (6) 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。

## 3.程序分析：

### 1.简单程序：

- (1) 该程序是用汇编和C混编实现字符串大小写转换，大小写转换的函数用C实现，显示则由汇编实现。

C程序 `hello.c`：传入字符串，upper函数实现大写转小写。

```
char Message[10]="AaBbCcDdEe";
/*变量_Message,初值为AaBbCcDdEe*/
void upper(){
    for(int i=0;i<10;i++){
        if (Message[i]>='A'&&Message[i]<='Z')
            Message[i]=Message[i]-'A'+'a';
    }
}
```

汇编程序 `hello.asm`：在屏幕显示转换后的结果。

```
BITS 16                ;按照16位进行编译的，代码地址都是16位
extern upper           ;引用upper
extern Message        ;引用Message
global _start         ;程序开始的标志
_start:
    mov ax,cs
    mov ds,ax
    mov es,ax
    mov ss,ax
```

```

push cs
call upper      ;调用函数
mov bp,Message
mov ax,ds
mov es,ax
mov cx,10
mov ax,1301h
mov bx,0007h
mov dh,10
mov dl,10
int 10h
_end:
jmp $

```

编译链接则通过如下指令实现：

`nasm -f elf32 hello.asm -o show.o` 将汇编文件编译成中间文件 `show.o`；

`gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -masm=intel -c hello.c -o hello.o` 将C文件编译成中间文件 `hello.o`。其中 `-march=i386` 表示使用 i386 指令集，`-m16` 表示使用16位代码，`-mpreferred-stack-boundary=2` 表示栈指针按照 $2*2=4$ 字节对齐，`-ffreestanding` 使输出程序能够独立运行，`-fno-PIE` 使程序不受位置限制，能在主存任何位置装载，故 `-fno-PIE` 则表示关闭 PIE 这个功能，`-masm=intel` 表示使用 nasm 汇编语法。

`ld -m elf_i386 -N --oformat binary -Ttext 0x7c00 show.o hello.o -o boot.bin` 用 `ld` 将两个文件链接得到二进制文件 `boot.bin`。

接下来生成镜像文件导入虚拟机即可，此处不再赘述。

(2) 下面来看一个很简单的C程序 `try.c`，该程序代码如下：

```

#include<stdio.h>
int main(){
    int a=3;
    int b=5;
    int c;
    c=a+b;
    printf("%d",c);
}

```

先用 `gcc -c -o try.o try.c` 得到 `.obj` 文件，然后通过 `objdump` 命令可以查到更多关于该文件的信息：

`-d`表示进行反汇编，可以看到如下信息：

```
chench@LAPTOP-T0EITVUA: $ objdump -d try.o
```

```
try.o: 文件格式 elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <main>:
 0: 55          push    %rbp
 1: 48 89 e5    mov     %rsp,%rbp
 4: 48 83 ec 10  sub     $0x10,%rsp
 8: c7 45 f4 03 00 00 00  movl    $0x3,-0xc(%rbp)
 f: c7 45 f8 05 00 00 00  movl    $0x5,-0x8(%rbp)
16: 8b 55 f4    mov     -0xc(%rbp),%edx
19: 8b 45 f8    mov     -0x8(%rbp),%eax
1c: 01 d0      add     %edx,%eax
1e: 89 45 fc    mov     %eax,-0x4(%rbp)
21: 8b 45 fc    mov     -0x4(%rbp),%eax
24: 89 c6      mov     %eax,%esi
26: 48 8d 3d 00 00 00 00  lea     0x0(%rip),%rdi    # 2d <main+0x2d>
2d: b8 00 00 00 00  mov     $0x0,%eax
32: e8 00 00 00 00  callq   37 <main+0x37>
37: b8 00 00 00 00  mov     $0x0,%eax
3c: c9        leaveq  %eax
3d: c3        retq
```

-t可以看到该程序的符号列表文档:

```
chench@LAPTOP-T0EITVUA: $ objdump -t try.o
```

```
try.o: 文件格式 elf64-x86-64
```

```
SYMBOL TABLE:
0000000000000000 1 df *ABS* 0000000000000000 try.c
0000000000000000 1 d .text 0000000000000000 .text
0000000000000000 1 d .data 0000000000000000 .data
0000000000000000 1 d .bss 0000000000000000 .bss
0000000000000000 1 d .rodata 0000000000000000 .rodata
0000000000000000 1 d .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 1 d .eh_frame 0000000000000000 .eh_frame
0000000000000000 1 d .comment 0000000000000000 .comment
0000000000000000 g F .text 000000000000003e main
0000000000000000 *UND* 0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000 *UND* 0000000000000000 printf
```

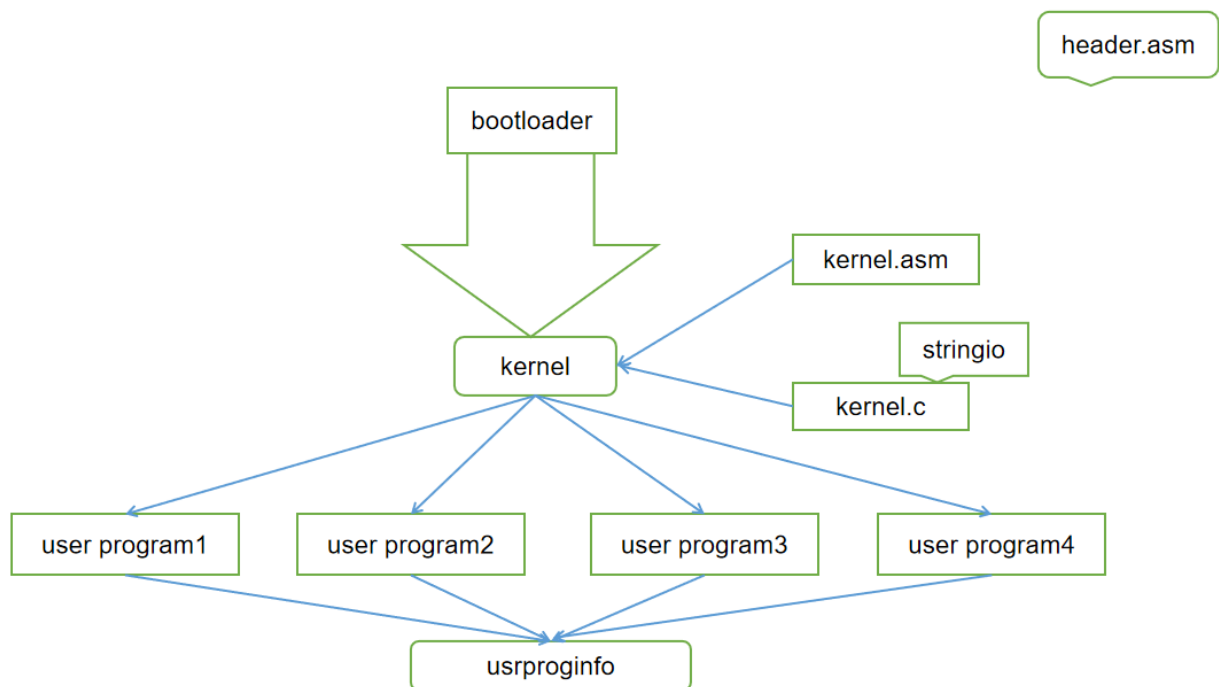
其中 `.text` 为已编译程序的机器代码; `.rodata` 为只读数据; `.data` 为已初始化的全局C变量; `.bss` 为未初始化的全局C变量; 局部C变量在运行时被保存在栈中, 因此既不出现在 `.data` 中, 也不出现在 `.bss` 中。

## 2.本次实验:

程序安排如下:

磁头	扇区	大小	内容
0	1	512B	引导程序
0	2	512B	存放用户程序信息
0	3-18	8KB	内核
1	1-2	1KB	用户程序1
1	3-4	1KB	用户程序2
1	5-6	1KB	用户程序3
1	7-8	1KB	用户程序4

程序结构如下：



接下来对每个模块的重点进行分析：

- (1) `header.asm`：该文件是用来实现在指定位置显示相关信息，可以被其他汇编程序和C程序调用。

```

; 用于在指定位置显示字符串，参数：(字符串首地址，字符串字节数，行数，列数)
%macro PRINT_IN_POS 4
    pusha                ; 压栈
    mov ax, cs           ; 置其他段寄存器值与CS相同
    mov ds, ax           ; DS=CS
    mov bp, %1           ; BP=当前串的偏移地址
    mov ax, ds            ; ES:BP = 串地址
    mov es, ax           ; 置ES=DS
    mov cx, %2           ; CX = 串长 (=9)
    mov ax, 1301h        ; AH = 13h (功能号)、AL = 01h (光标置于串尾)
    mov bx, 0007h        ; 页号为0(BH = 0) 黑底白字(BL = 07h)

```

```

    mov dh, %3      ; 行号=0
    mov dl, %4      ; 列号=0
    int 10h         ; BIOS的10h功能: 显示一行字符
    popa           ; 出栈
%endmacro

```

(2) 4个用户程序可根据前两次实验代码做简单改动即可:

在 `start` 后将寄存器压栈:

```

start:
    pusha

```

这样可以保证从用户程序返回操作系统内核后, 内核能够以与之前相同的状态继续执行。

在 `QuitUsrProg` 处添加出栈和返回指令:

```

QuitUsrProg:
    popa
    retf

```

压栈就要出栈, 因此要 `popa`。`retf` 是为了返回操作系统内核, 该指令从栈中取得IP和CS寄存器的值, 从而实现跳转。

包含文件 `header.asm`, 通过调用函数来在指定位置显示内容:

```

#include "header.asm"
PRINT_IN_POS hint1, hint1len, 7, 29
;hint1为'This is usrprog xxx. Press e to exit.'

```

(3) `usrproginfo.asm`: 显示用户程序信息。

```

#include "header.asm"      ; 包含header.asm
%macro UsrProgInfoBlock 7  ; 参数: PID,程序名,字节数,柱面,磁头,扇区,内存地址
    pid%1 db %1           ; 程序编号PID
    name%1 db %2          ; 程序名
    times 16-($-name%1) db 0
    size%1 dw %3           ; 程序大小
    cylinder%1 db %4       ; 柱面
    head%1 db %5           ; 磁头
    sector%1 db %6         ; 扇区
    addr%1 dw %7           ; 内存地址
%endmacro

UsrProgNumber:
    db 4                  ; 用户程序数

UserProgInfo:             ; 详细信息
    UsrProgInfoBlock 1, 'stone1u', 1024, 0, 1, 1, offset_usrprog1
    UsrProgInfoBlock 2, 'stone1u', 1024, 0, 1, 3, offset_usrprog2
    UsrProgInfoBlock 3, 'stone1d', 1024, 0, 1, 5, offset_usrprog3

```

```
UsrProgInfoBlock 4, 'stonerd', 1024, 0, 1, 7, offset_usrprog4
```

(4) 引导程序 `bootloader.asm`：实现了4个函数：

- ① `_start`：通过使用BIOS的 `int 10h` 功能调用显示交互信息。
- ② `LoadUsrProgInfo`：加载用户程序信息表到内存中。
- ③ `LoadOsKernel`：加载操作系统内核。使用 `int 13h` 读软盘。
- ④ `EnterOs`：用 `jmp` 跳转至操作系统内核。

注：需要加BITS 16伪指令，表示这个段是按照16位进行编译的，代码地址都是16位。

(5) 内核程序：

- ① `oskernel.asm`：操作系统入口。
- ② `ka.asm`：里面是用汇编语言编写的函数，可供汇编程序或C程序调用。  
如：`printInPos`、`loadAndRun`、`powerOff`、`putchar`、`getchar`，得到用户程序的扇区、磁头的函数。
- ③ `kc.c`：里面是用C语言编写的函数，可供汇编程序或C程序调用。  
`void startUp()`、`void promptString()`、`void showHelp()` 显示交互界面的相关信息；  
`void listUsrProg()` 显示用户程序的相关信息；  
`void shell()` 执行程序的函数。
- ④ `stringio.h`：自己写的关于字符串输入输出的函数，包含在 `kc.c` 内。  
如：未使用C库自己实现的 `strlen`、`print`、`strcmp` 等等。

## 四、实验过程：

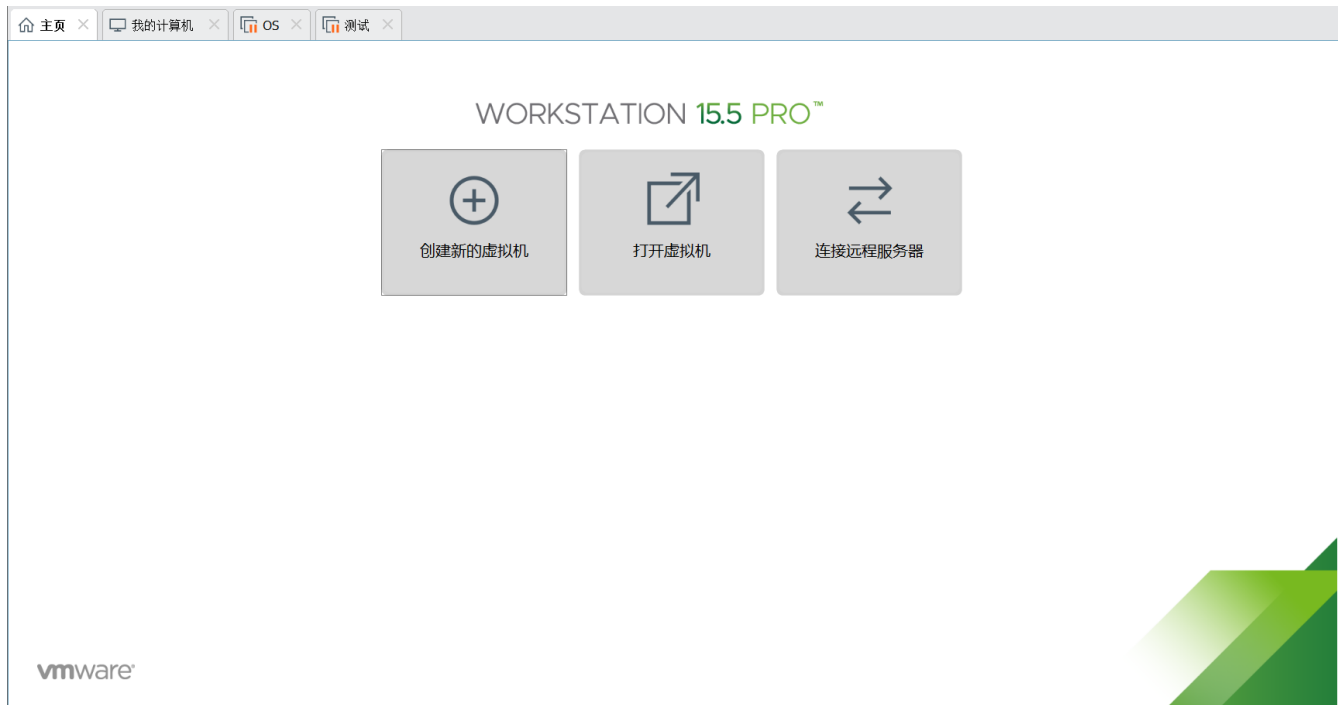
注：该部分介绍的是设计内核的整个流程，用来熟悉环境的简单程序将C和汇编代码链接后，后续步骤和前两次实验一样，就不在此详细介绍了。

### 1.软件的下载与安装：

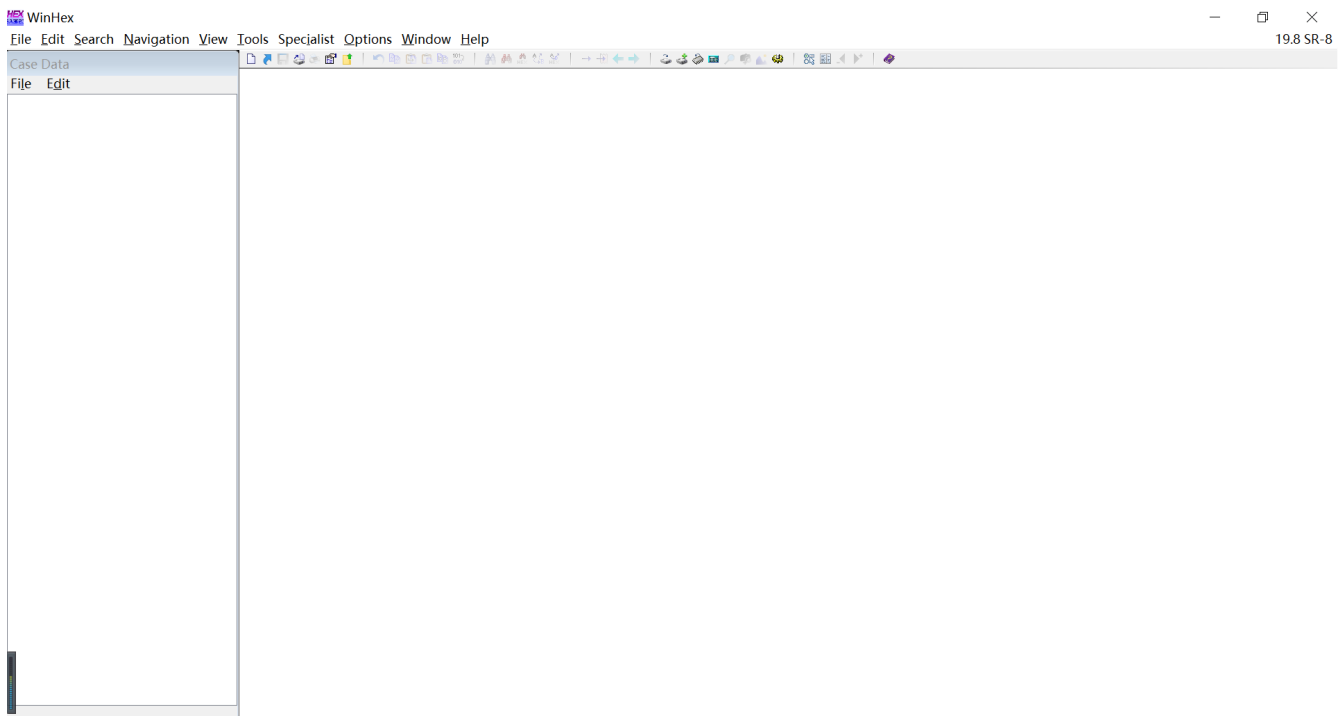
- (1) 在官网下载 `VMware Workstation 15.5 Pro`：

<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，下载 windows Subsystem for Linux (Ubuntu 18.04 LTS)：

首先以管理员身份运行 Powershell，输入如下指令：

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-windows-Subsystem-Linux
```

重启后在 Microsoft Store 下载 Ubuntu 18.04 LTS，安装完成后设置用户名和密码。由于国外源下载会比较慢，所以更换为国内源（如阿里云）。至此一个简单的 Linux 环境就搭好了：



```
chench@LAPTOP-T0EITVUA: ~
chench@LAPTOP-T0EITVUA: $
```

(4) 安装 gcc、NASM、GNU ld：

输入 `sudo apt-get install build-essential` 安装 gcc；输入 `sudo apt install nasm` 安装 NASM；  
输入 `sudo apt install binutils` 安装 GNU ld。

## 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：

**Step1:**由于4个用户程序、用户程序信息表和引导程序单独编译即可，故在 Windows 下用 NASM 直接编译：

```
nasm
Microsoft Windows [版本 10.0.18362.836]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\czh\Desktop\OS>nasm -f bin stonelu.asm -o stonelu.bin
C:\Users\czh\Desktop\OS>nasm -f bin stoneld.asm -o stoneld.bin
C:\Users\czh\Desktop\OS>nasm -f bin stonerd.asm -o stonerd.bin
C:\Users\czh\Desktop\OS>nasm -f bin stoneru.asm -o stoneru.bin
C:\Users\czh\Desktop\OS>nasm -f bin bootloader.asm -o bootloader.bin
C:\Users\czh\Desktop\OS>nasm -f bin usrproginfo.asm -o usrproginfo.bin
C:\Users\czh\Desktop\OS>
```

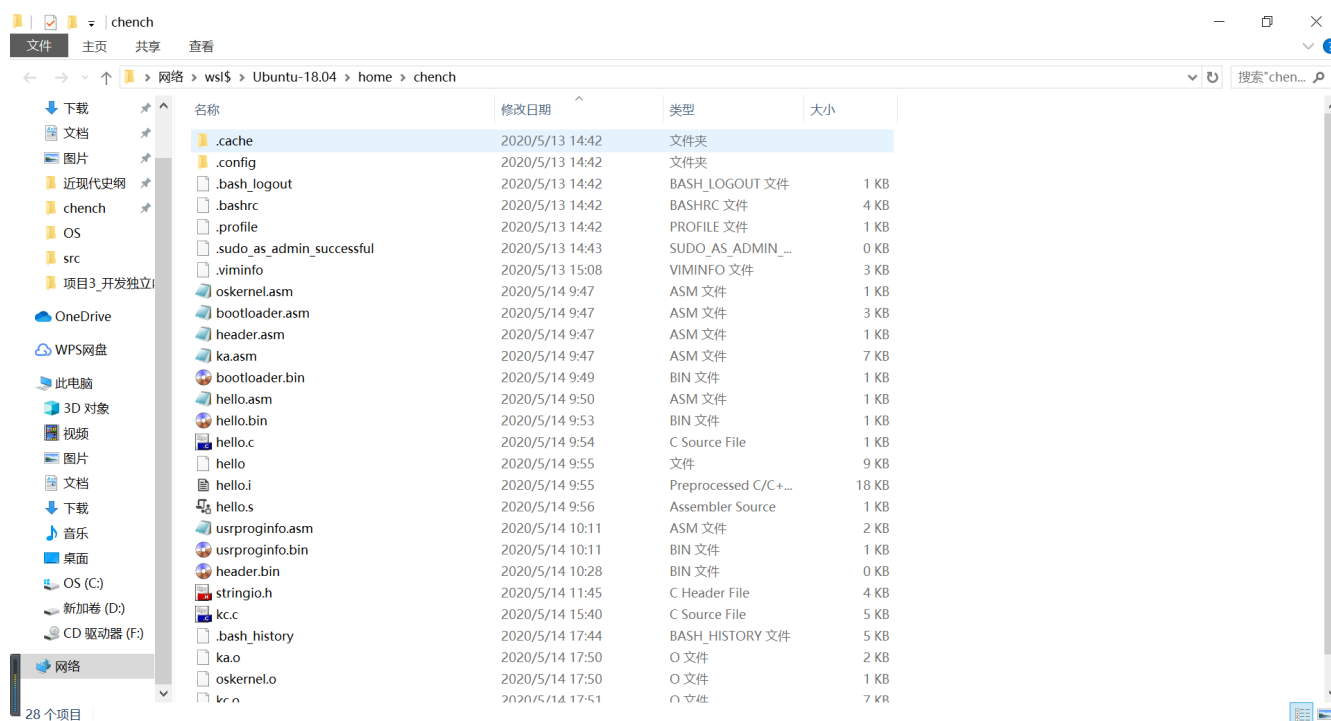
可以在和 NASM 相同的目录下找到上述6个 .bin 文件:

	usrproginfo.bin	2020/5/14 17:19	BIN 文件	1 KB
	bootloader.bin	2020/5/14 17:19	BIN 文件	1 KB
	stoneru.bin	2020/5/14 17:04	BIN 文件	1 KB
	stonerd.bin	2020/5/14 17:04	BIN 文件	1 KB
	stoneld.bin	2020/5/14 17:04	BIN 文件	1 KB
	stonelu.bin	2020/5/14 17:03	BIN 文件	1 KB
	stoneld.asm	2020/5/14 17:03	ASM 文件	5 KB
	stoneru.asm	2020/5/14 17:02	ASM 文件	5 KB
	stonerd.asm	2020/5/14 16:59	ASM 文件	5 KB
	stonelu.asm	2020/5/14 16:58	ASM 文件	5 KB
	usrproginfo.asm	2020/5/14 10:11	ASM 文件	2 KB
	bootloader.asm	2020/5/14 9:47	ASM 文件	3 KB

Step2:本实验最关键的一步：**得到独立内核**。步骤如下：

①将编写独立内核用到的代码文件放入WSL系统下，以保证可以使用 NASM+gcc+ld 进行编译链接，方法如下：

在 wsl 中输入如下指令：`explorer.exe .`，让 windows 访问 wsl 系统下的文件，可以看到弹出这样的页面：

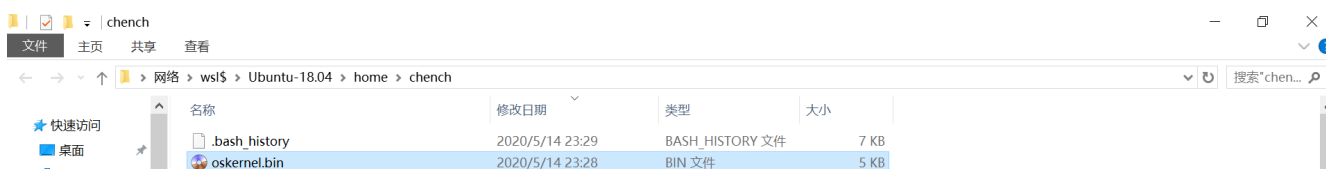


将 oskernel.asm、ka.asm、kc.c、stringio.h 文件复制到该文件夹内即可。

②输入如下指令，将C程序与汇编程序联合编译：

```
chench@LAPTOP-TOEITVUA: ~
chench@LAPTOP-TOEITVUA: $ nasm -f elf32 ka.asm -o ka.o
chench@LAPTOP-TOEITVUA: $ nasm -f elf32 oskernel.asm -o oskernel.o
chench@LAPTOP-TOEITVUA: $ gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -masm=intel -c kc.c -o kc.o
chench@LAPTOP-TOEITVUA: $ ld -m elf_i386 -N --oformat binary -Ttext 0x8000 oskernel.o ka.o kc.o -o oskernel.bin
chench@LAPTOP-TOEITVUA: $
```

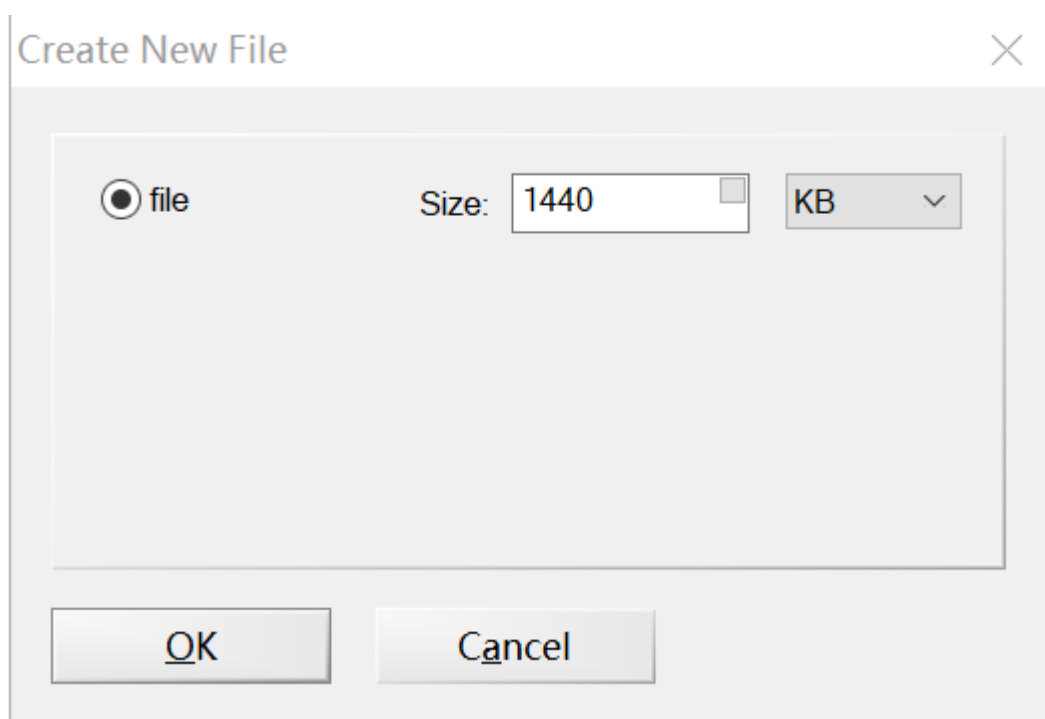
③在WSL目录下可以看到 `oskernel.bin` :



至此，所有需要的 `.bin` 文件已准备完毕，接下来就是要将这些文件导入一个镜像文件中。

### Step3: 创建镜像文件:

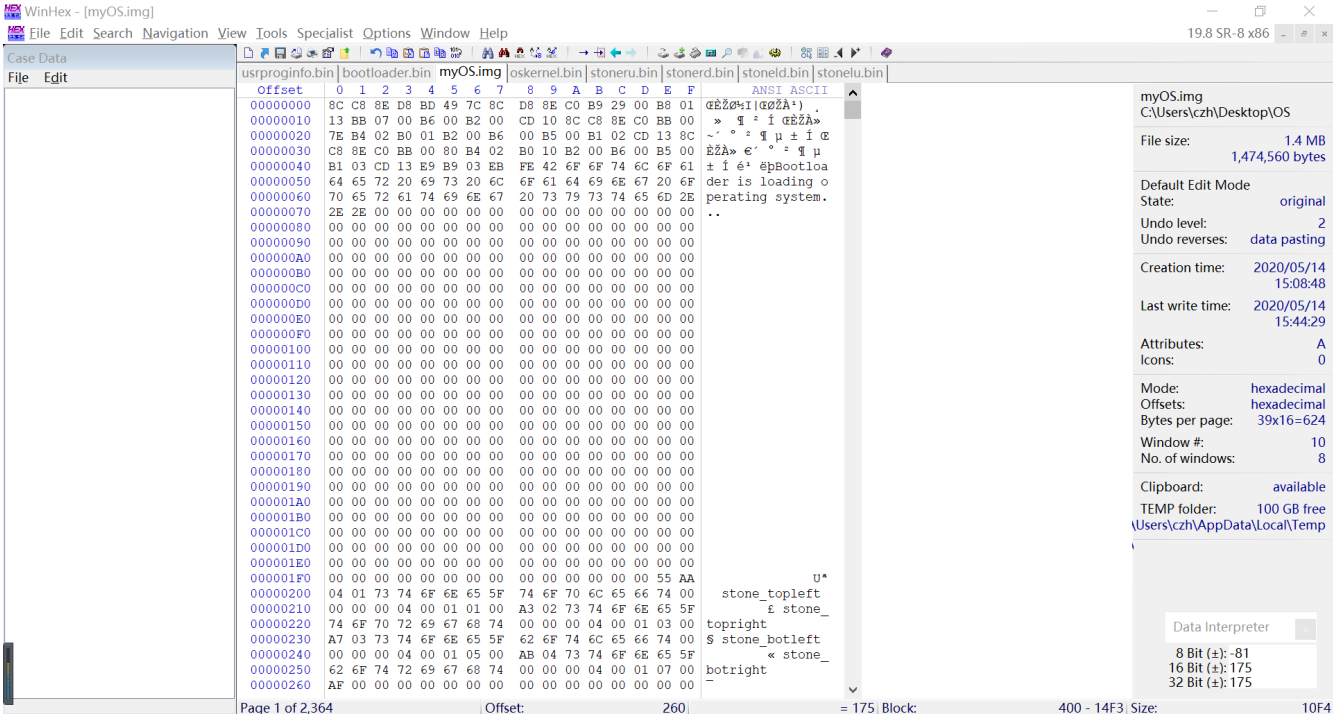
打开 `WinHex`，首先新创建一个文件: `File -> New`，然后选择 `1440KB`，该文件就是最终要的镜像文件。



然后打开制作软盘所需的所有文件（共7个），然后将这7个文件按照如下顺序复制粘贴至新建的文件中：

bootloader.bin->usrproginfo.bin->oskernel.bin->stonelu.bin->stoneru.bin->stoneid.bin->stonerd.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，oskernel.bin 的偏移为 0x0400，第一个用户程序的偏移为 0x2400 .....



至此，便得到了命名为 myos.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

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)

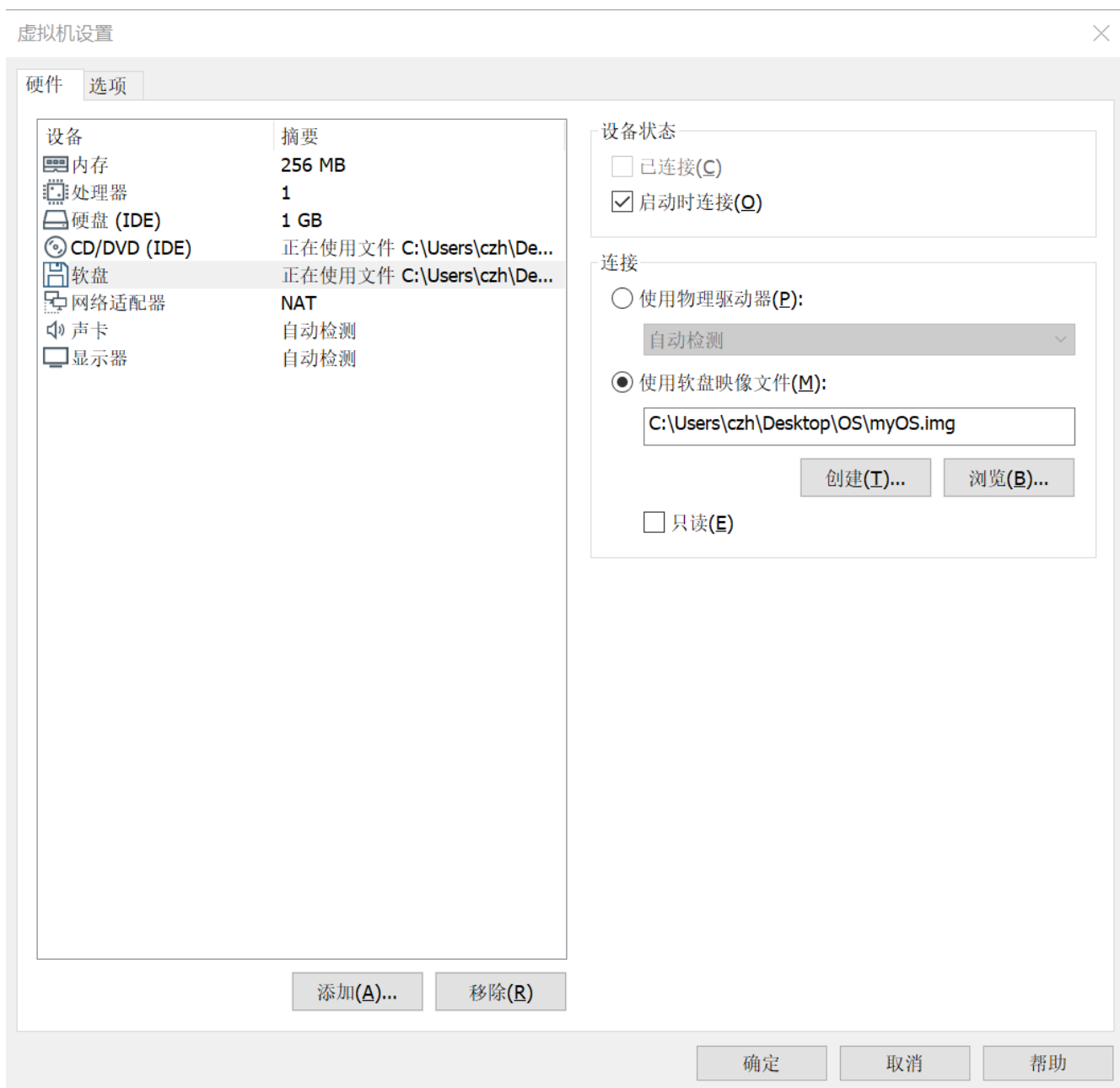
下一步(N) >

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 myOS.img 镜像文件，接下来就是将 myOS.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 myOS.img 导入， 确定 即可。



注：简单程序显示结果的镜像文件为 `boot.img`

## 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：

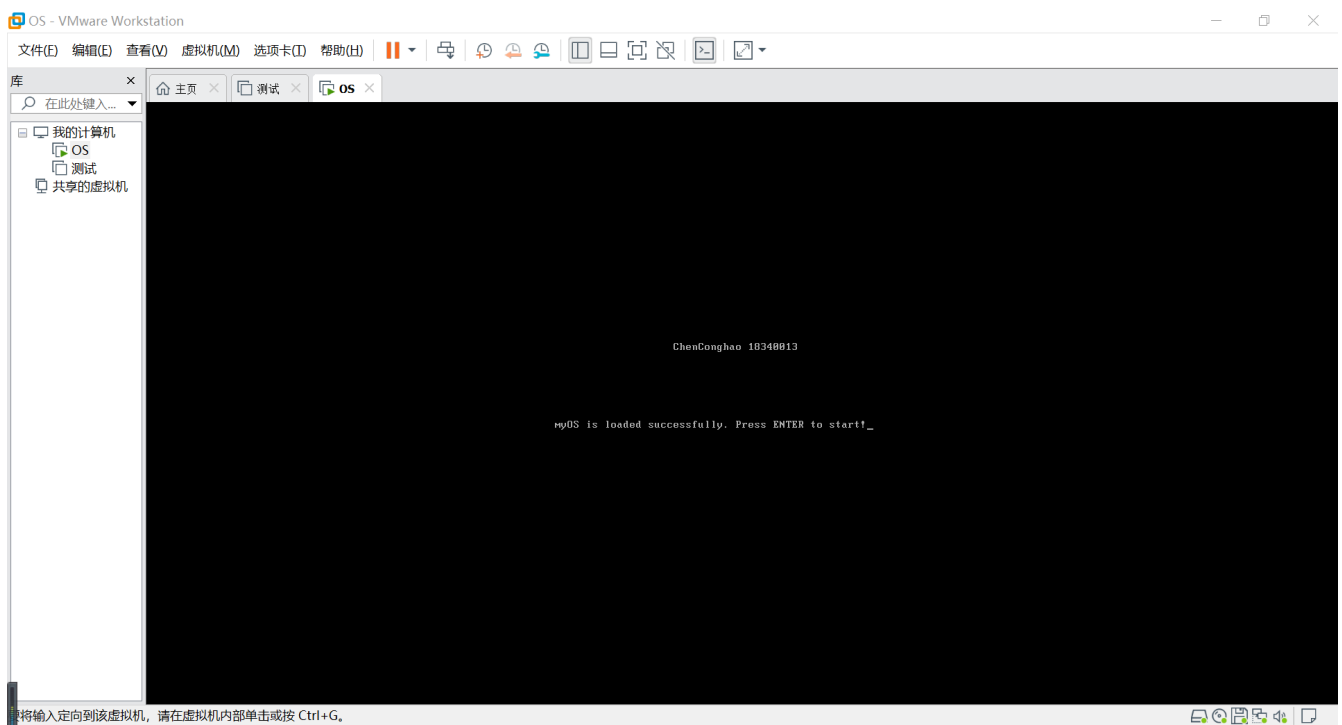
### 1.简单程序：

aabbccddee

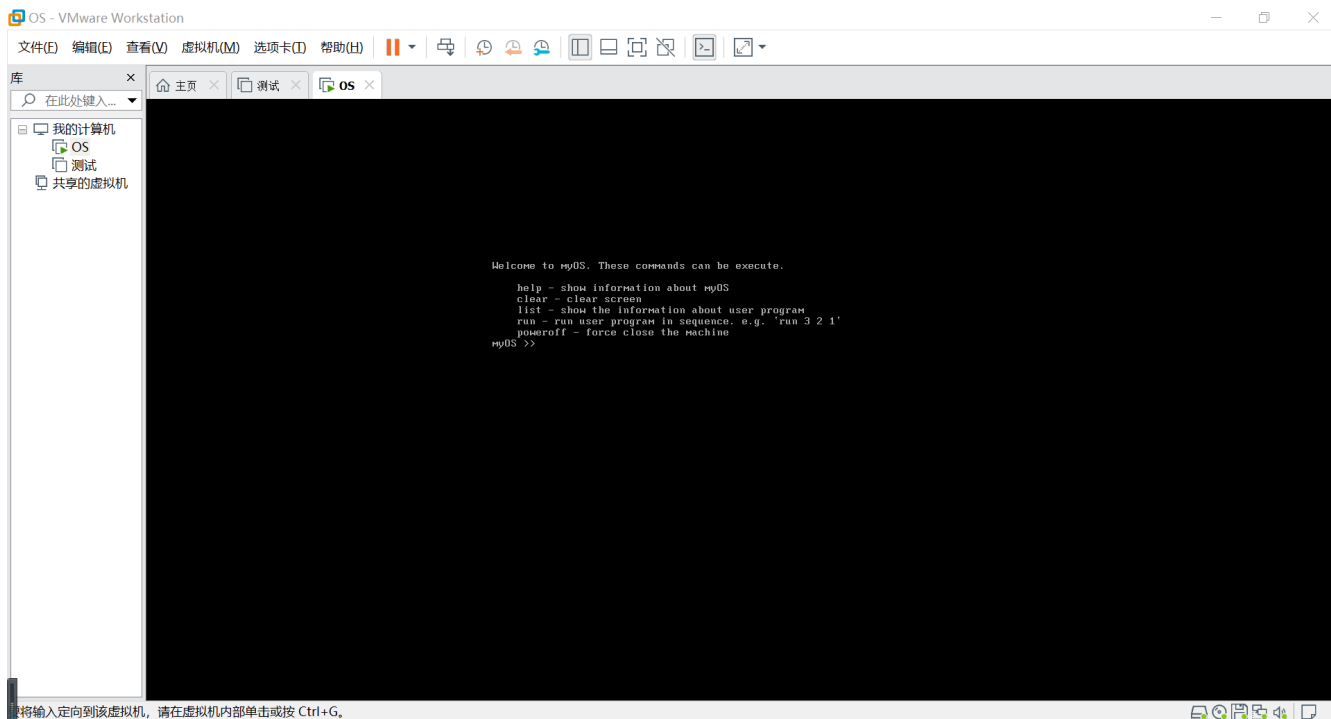
可以看到输入的字符串全部转为小写字母输出了。

## 2.本次实验：

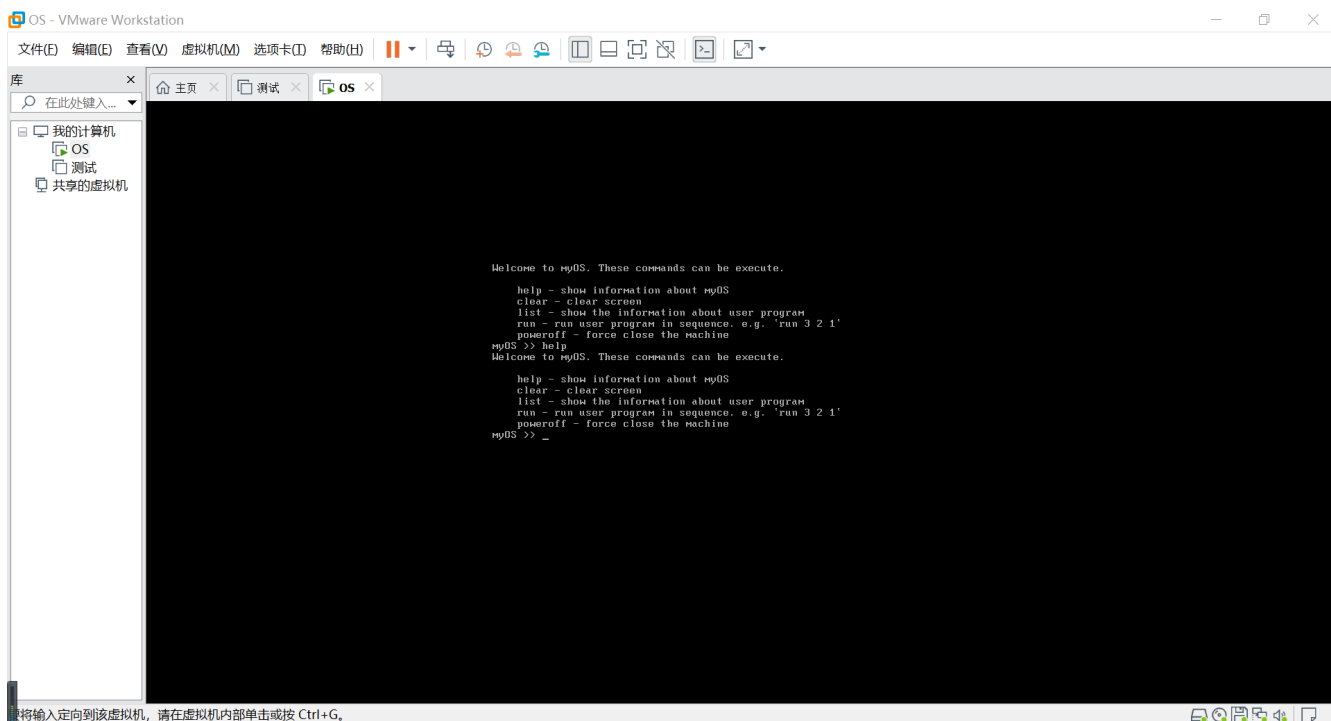
(1) 初始界面：



(2) 按 ENTER 进入后：



(3) 执行 `help` 操作：把有关信息显示出来。



(4) 执行 `list` 操作：显示4个用户程序的信息。



```
Welcome to myOS. These commands can be execute.
```

```
help - show information about myOS
```

```
clear - clear screen
```

```
list - show the information about user program
```

```
run - run user program in sequence. e.g. 'run 3 2 1'
```

```
poweroff - force close the machine
```

```
myOS >> list
```

```
You can use 'run <PID>' to run a user program.
```

```
PID - Name - Size - Addr - Cylinder - Head - Sector
```

```
1 - stonelu - 1024 - A300 - 0 - 1 - 1
```

```
2 - stoneru - 1024 - A700 - 0 - 1 - 3
```

```
3 - stoneld - 1024 - AB00 - 0 - 1 - 5
```

```
4 - stonerd - 1024 - AF00 - 0 - 1 - 7
```

```
myOS >>
```

(5) run 操作：在 run 操作中，以下几种命令合法：run 1（只执行第一个用户程序）、run 4 3 2 1（按一定顺序执行4个用户程序）、run 1 1 1（连续执行相同用户程序），且后面的数字只能是1-4。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最简单的界面。

如：键入run 4 3 2 1：

```
Program have been executed successfully!
```

```
myOS >> run 4 3 2 1
```

```

This is usrprog 4. Press e to exit.

```

[illegible]

紧接着执行完 `usrprog2`、`usrprog1` 后，返回如下界面：

```
Program have been executed successfully!  
myOS >> _
```

(6) 执行 `clear` 操作：清屏，只留下命令行。

```
myOS >>
```

(7) 执行 `poweroff` 操作：直接关闭虚拟机。



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

```
Welcome to myOS. These commands can be execute.

help - show information about myOS
clear - clear screen
list - show the information about user program
run - run user program in sequence. e.g. 'run 3 2 1'
poweroff - force close the machine
myOS >> aaaaa
aaaaa: command not found
myOS >>
```

## 五、实验总结:

本次实验在我看来真的很难很难! 我花了整整4天时间才搞定。老师在实验课上讲解相关的知识点的时候觉得并不难, 感觉无非就是通过操作一些指令来将C和汇编程序进行链接, 听上去很轻松。可自己做实验的时候就发现各种bug。首先是方案的选择, 要么 `TASM+tcc+tlink`, 要么 `NASM+gcc+ld`, 看群里的讨论感觉两种方案都有不少需要解决的问题, 于是就选择了自己熟悉的 `gcc`。环境配置倒没有什么问题, 无非就是下载安装, 很容易。这一步解决以后, 各种问题就接二连三的开始出现。在C与汇编的交互的时候有许多问题要注意, 比如变量如何相互调用, 函数如何互相调用, 涉及到入栈、出栈、`extern`、`global` 等等。参考了一些资料以后才搞明白是怎么回事。还有就是不能引用C库函数的头文件, 否则就是“用操作系统设计操作系统”, 所以一些功能需要自己实现。好不容易搞完代码了, 在编译链接得到内核的过程中, 发现链接不上, 于是就开始检查代码有无问题, 各种改动发现还是不行; 最后发

现代码改完没有问题而链接就是不成功的时候，群里有人问编译链接的相关指令，这才提醒我有可能和指令有一定的关系，于是按照群里发的几条指令逐个试了一遍，一直试了5次才成功。后来我把弹出的error上百度搜了一下，说有可能是没加 `-fno-PIE`，于是就明白了为什么会选择这条指令，最后编译链接终于成功了！这个过程可以说是让我崩溃的，在这一步我卡了整整一个上午。编译链接完成之后写软盘就简单多了，跟前两次实验一样常规操作，就可以得到镜像文件。最终看到自己的程序可以运行时，紧绷的神经这才放松下来！讨论群整整一周都是相当火爆，在讨论群里可以看到同学们在实验过程中碰到的各种bug，当然还有大佬们提供的各种解决方案，可以说给了我不少帮助，让我对这部分的内容有了更加清晰的认识，也在我做实验的过程中起到了一定的作用。

## 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.

凌应标. 02实验课.pptx

<https://blog.csdn.net/trochiluses/article/details/19301663>

[http://blog.sina.com.cn/s/blog\\_55c555f10102vtz1.html](http://blog.sina.com.cn/s/blog_55c555f10102vtz1.html)

<https://www.jb51.net/LINUXjishu/310967.html>

<https://www.cnblogs.com/tlm1992/p/3790195.html>

<https://www.cnblogs.com/tongongV/p/10745040.html>

<https://blog.csdn.net/wwchao2012/article/details/79980514>

<https://blog.csdn.net/cloudblaze/article/details/78526456>

[https://blog.csdn.net/lmz\\_lmz/article/details/88341807](https://blog.csdn.net/lmz_lmz/article/details/88341807)