|  |  |  |
| --- | --- | --- |
| **验收成绩** | **报告成绩** | **总评成绩** |
|  |  |  |

**武汉大学计算机学院**

**本科生实验报告**

**操作系统设计与实现**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：操作系统课程设计

指 导 教 师 ：蒋晶珏 副教授

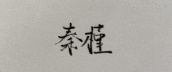
学 生 学 号 ：2020302111394

学 生 姓 名 ：秦槿

二○二三年八月

**郑 重 声 明**

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名：  日期： 2023.08.08\_\_\_

摘要

操作系统课程设计旨在通过模拟实践，引导学生一步步设计一个操作系统。逐步构建操作系统加深了对引导盘、保护模式、操作系统内核、进程、键盘IO以及处理器调度等概念的理解，同时实现了对操作系统设备管理的掌握。此实验帮助学生更深刻地领会课程理论，并直观认识实际操作系统内核。

实验设计遵循操作系统关键理论，包括保护模式、进程、内核、IO处理。涵盖NASM汇编语言和C语言等内容。实验步骤包括搭建Linux虚拟机和Bochs仿真环境，在自己搭建的环境中调试运行《ORANGE'S：一个操作系统的实现》 的前七章代码，并在此基础上自主实现附加实验——实验9的要求。

通过此次实验，我深刻认识到操作系统作为计算机核心，管理软硬件资源的重要性。尽管此次实验仅涵盖操作系统的基础功能部分，实际使用的操作系统远比实验中的复杂，仍需要结合学过的操作系统基础知识对每一步操作都仔细考虑。从另一方面来看，此次实验也充分强调了理论与实践结合以致相互促进的重要性。

**关键词：**操作系统；NASM汇编语言；虚拟机；Linux；bochs

**目录**

**[1&2实验环境及环境搭建，动手写一个最小的“操作系统”](#_1&2_实验环境及环境搭建，动手写一个最小的\“操作系统\”（实验一和实验)** [6](#_1&2_实验环境及环境搭建，动手写一个最小的\“操作系统\”（实验一和实验)

[1.1&2.1 实验内容及上机实验所用平台 6](#_1.1&2.1_实验内容及上机实验所用平台)

[1.1.1&2.1.1 实验内容 6](#_1.1.1&2.1.1__)

[1.1.2&2.1.2 实验所用平台 6](#_1.1.2&2.1.2__实验所用平台)

[1.2&2.2 关键代码分析 8](#_1.2&2.2__结合关键代码分析)

[1.3&2.3 调试过程 10](#_1.3&2.3调试过程)

[1.4&2.4 实验总结 11](#_1.4&2.4_实验总结)

**[3 实现保护模式](#_3.1_实验内容及上机实验所用平台)** [11](#_3.1_实验内容及上机实验所用平台)

[3.1 实验内容及上机实验所用平台 13](#_Toc68187047)

[3.1.1 实验内容 13](#_Toc68187048)

[3.1.2 实验所用平台 13](#_Toc68187049)

[3.2 关键代码分析 15](#_Toc68187050)

[3.3 调试过程 33](#_Toc68187051)

[3.4 实验总结 36](#_Toc68187052)

**[4 切换到保护模式](#_4.1_实验内容及上机实验所用平台)** [37](#_4.1_实验内容及上机实验所用平台)

[4.1 实验内容及上机实验所用平台 37](#_Toc68187054)

[4.1.1 实验内容 37](#_Toc68187055)

[4.1.2 实验所用平台 37](#_Toc68187056)

[4.2 关键代码分析 38](#_Toc68187057)

[4.3 调试过程 53](#_Toc68187058)

[4.4 实验总结 57](#_Toc68187059)

[**5 操作系统内核的雏形** 57](#_Toc68187060)

[5.1 实验内容及上机实验所用平台 57](#_Toc68187061)

[5.1.1 实验内容 57](#_Toc68187062)

[5.1.2 实验所用平台 58](#_Toc68187063)

[5.2 关键代码分析 59](#_Toc68187064)

[5.3 调试过程 64](#_Toc68187065)

[5.4 实验总结 69](#_Toc68187066)

[**6 进程与进程调度** 69](#_Toc68187067)

[6.1 实验内容及上机实验所用平台 70](#_Toc68187068)

[6.1.1 实验内容 70](#_Toc68187069)

[6.1.2 实验所用平台 70](#_Toc68187070)

[6.2 关键代码分析 71](#_Toc68187071)

[6.3 调试过程 86](#_Toc68187072)

[6.4 实验总结 93](#_Toc68187073)

[**7 操作系统的输入/输出系统** 94](#_Toc68187074)

[7.1 实验内容及上机实验所用平台 94](#_Toc68187075)

[7.1.1 实验内容 94](#_Toc68187076)

[7.1.2 实验所用平台 94](#_Toc68187077)

[7.2 关键代码分析 95](#_Toc68187078)

[7.3 调试过程 99](#_Toc68187079)

[7.4 实验总结 104](#_Toc68187080)

[**8 操作系统的输入/输出系统** 94](#_Toc68187074)

[7.1 实验内容及上机实验所用平台 94](#_Toc68187075)

[7.1.1 实验内容 94](#_Toc68187076)

[7.1.2 实验所用平台 94](#_Toc68187077)

[7.2 关键代码分析 95](#_Toc68187078)

[7.3 调试过程 99](#_Toc68187079)

[7.4 实验总结 104](#_Toc68187080)

[心得体会与感悟 105](#_心得体会与感悟)

参考文献 [106](#_Toc68187084)

[教师评语评分 107](#_教师评语评分)

# 1&2 实验环境搭建，**动手写一个最小的“操作系统”（实验一和实验二）**

## 1.1&2.1 实验内容及上机实验所用平台

### 1.1.1&2.1.1 实验内容

环境搭建：在Windows操作系统下，首先安装虚拟机软件VMware，然后在虚拟机中安装Ubuntu操作系统和安装Bochs。Bochs是一款开源的X86硬件平台模拟器，使用纯软件模拟硬件设备，包括键盘、鼠标、VGA卡、磁盘、网卡等，适合操作系统开发。

简化步骤如下：

1.卸载已有虚拟机，安装最新版本的VMware；

2.在虚拟机中安装Ubuntu操作系统；

3.下载并编译安装Bochs源码；

4.编写基本的汇编代码，通过编译、使用dd命令写入磁盘，并在

Bochs模拟器中测试和运行。

环境搭建完成后，通过编译一段最基本的asm代码来初次体验操作系统的设计以及了解NASM编译的使用方法、dd命令写入磁盘的方法以及Bochs的使用方法。

### 1.1.2&2.1.2 实验所用平台

（1）VMware

VMware是虚拟机软件，能在一台机器上同时运行多个不同操作系统，与多启动系统不同，系统切换时不需要重启。它支持隔离的操作环境、互动操作和复原功能。

（2）Ubuntu

Ubuntu是基于Debian的桌面Linux操作系统，提供稳定和新颖的自由软件，有庞大的社区力量。

（3）Bochs

Bochs是一款开源的x86硬件平台模拟器，模拟整个PC平台，包括I/O设备、内存和BIOS。它能在多种平台上运行，并支持模拟多台PC。

（4）NASM

Netwide Assembler（NASM）是x86架构的汇编与反汇编工具，支持多种二进制格式输出，适用于编写16位、32位和64位程序，以及创建引导加载程序等。遵循Intel风格汇编。

## 1.2&2.2 关键代码分析

### 文件夹a:

文件：boot.asm

#### 主要数据结构：

常量BootMessage: db "Hello, OS world!"

#### 关键代码分析：

主要代码结构：

代码主体框架：

org 07c00h ; 告诉编译器程序加载到7c00处

mov ax, cs

mov ds, ax

mov es, ax ; 使ds和es两个段寄存器指向与cs相同的段

call DispStr ; 调用DispStr子程序显示字符串例程

jmp $ ; 无限循环

DispStr: DispStr子程序：

mov ax, BootMessage

mov bp, ax ; 设置ES:BP = 串地址

mov cx, 16 ; 设置CX = 串长度

mov ax, 01301h ; 设置AH = 13, AL = 01h

mov bx, 000ch ; 设置页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)

mov dl, 0

int 10h ; 10h 号中断

ret

BootMessage: db "Hello, OS world!"

times 510-($-$$) db 0 ; 填充剩下的空间，使生成的二进制代码恰好为512字节

dw 0xaa55 ; 结束标志

以上汇编代码中，未被方括号括起来的变量名被看作是地址，如果不加方括号时出现数字，则视作 offset；$表示当前行被汇编后的地址，jmp $因此能够实现无限循环，$$表示一个节的开始处被汇编之后的地址。

### 文件夹b:

文件：boot.asm

#### 关键代码分析：

在文件夹a中boot.asm的基础上添加了预编译宏

%ifdef \_BOOT\_DEBUG\_ 预编译宏

org 0100h ; 调试状态, 做成 .COM 文件, 可调试

%else

org 07c00h ; BIOS 将把 Boot Sector 加载到 0:7C00 处

%endif

## 1.3&2.3调试过程

在Windows环境中安装VMware软件，再下载Linux的Ubuntu镜像。

在VMware中利用下载好的Ubuntu镜像新建虚拟机并安装Ubuntu系统。

* 安装编译环境:

1. sudo apt-get install build-essential
2. sudo apt-get install xorg-dev
3. sudo apt-get install libgtk2.0-dev
4. sudo apt-get install bison
5. sudo apt-get install g++

* 安装具有调试功能的bochs:

1. tar vxzf bochs-2.6.11.tar.gz
2. cd bochs-2.6.11
3. ./configure -enable debugger --enable -disasm（./configure后的参数便是打开调试功能的开关）
4. make
5. sudo make install

* 安装nasm

sudo apt-get install build-essential nasm

* 在bochs中创建软盘映像，如图1-1所示

1. bximage（进入软盘映像创建界面）
2. fd
3. [1.44]或者回车键
4. [a.img]或者回车键

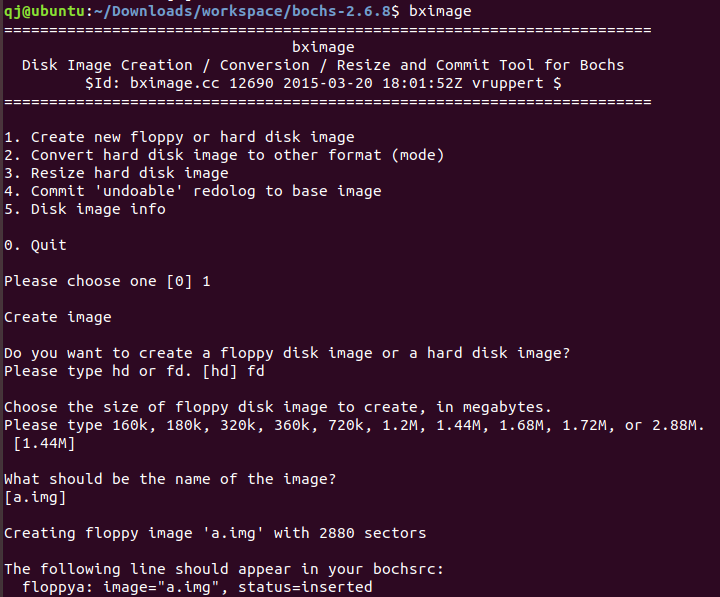


图 1-1

* 用nasm编译boot.asm文件的代码

nasm boot.asm -o boot.bin

* 修改bochsrc使其与本地bochs组件路径对应

romimage: file=/usr/share/bochs/BIOS-bochs-latest

vgaromimage: /usr/share/vgabios/vgabios.bin

keyboard\_mapping:enabled=1,map=/usr/share/bochs/keymaps/x11-pc-us.map

修改为：

romimage:file=/home/qj/Downloads/workspace/bochs-2.6.8/bios/BIOS-bochs-latest

vgaromimage:file=/home/qj/Downloads/workspace/bochs-2.6.8/bios/VGABIOS-lgpl-latest

keyboard:keymap=/home/qj/Downloads/workspace/bochs-2.6.8/gui/keymaps/x11-pc-us.map

* dd命令，用软盘绝对扇区读写工具写到第一个扇区

dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc（notrunc表示文件不会被截断）

* Bochs

bochs -f bochsrc

实验成果如图1-2所示：

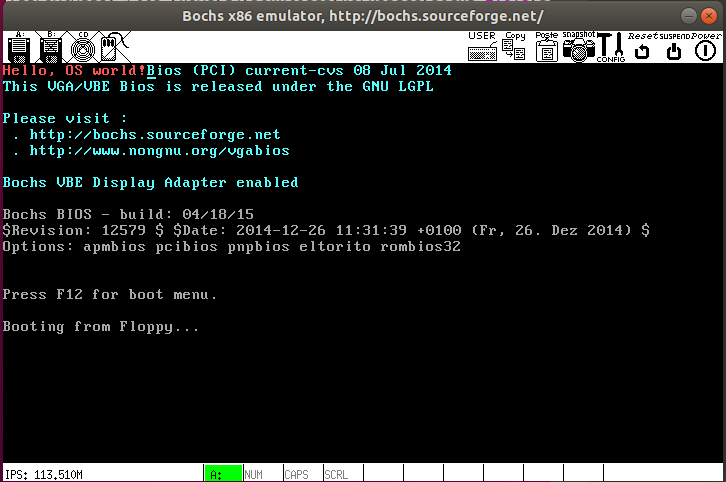


图 1-2

## 1.4&2.4 实验总结

在本实验中，我们编写了一个最简单的“操作系统”，使用Bochs软件进行仿真。事实上，我们编写的程序只是一个最简单的引导扇区，之所以可以称之为最简单的操作系统，是因为它可以脱离我们运行的操作系统，在裸机上运行。

计算机电源打开时，首先会加电自检，然后寻找启动盘，如果是选择从软盘启动，计算机就会检查软盘的0面0磁道1扇区，如果发现它以0xAA55结束，那么BIOS就会认为它是一个引导扇区。一旦BIOS发现了引导扇区，就会将这512字节的内容装载到内存地址0000:7c00处，然后跳转到0000:7c00处，将控制权彻底交给这段引导代码。至此，计算机不再由BIOS中固有的程序来控制，而是变为由操作系统来控制。我们编写的代码中第一行就是将程序加载到0x7c00处，并以0xAA55结束，因而可以作为一个独立的引导扇区运行。

具体来讲，首先是nasm的编译，编译.asm文件为.bin文件，制作引导扇区时，编译为.com进行调试。然后是dd命令写入磁盘：dd if=boot.bin of=/dev/fd0 bs=512 count=1 conv=notrunc。软盘读写工具将文件写到空白软盘的第一个扇区，然后让ds和es两个段寄存器指向与cs相同的段，调用子程序显示字符串，然后无限循环。最后是bochs的用法，用bximage命令创建一个空白映像文件，再使用dd命令将.bin文件写入，修改bochsrc文件中的路径，从指定软盘启动。

3 实现保护模式

## 3.1 实验内容

了解保护模式的相关知识，并掌握从实模式跳转到保护模式的方法；在保护模式下通过GDT描述符对大地址进行读写；实现实模式大于1MB内存的寻址能力，并接着上一次实验，从保护模式返回到实模式，重新设置各个段寄存器的值；了解LDT描述符；学会使用挂载指令将文件挂载在临时目录并运行程序。

## 3.2 关键代码分析

### 文件夹a：

了解保护模式的相关知识，并掌握从实模式跳转到保护模式的方法；在保护模式下通过GDT描述符对大地址进行读写。

文件pmtest1.asm：

#### 主要数据结构：

表 3-1[SECTION .gdt] gdt数据段

|  |  |
| --- | --- |
| 数组GDT | LABEL\_GDT 空描述符 |
| LABEL\_DESC\_CODE32 非一致代码段32描述符 |
| LABEL\_DESC\_VIDEO 显存首地址 |
| 变量 | Gdtlen 表示GDT的长度 |
| GdtPtr dw为GDT界限  dd为GDT基址 |
| 选择子 | SelectorCode32 非一致代码段32选择子 |
| SelectorVideo 视频段选择子 |

#### 主要代码结构：

1. 从实模式跳到保护模式

[SECTION .s16]

[BITS 16]

LABEL\_BEGIN:

初始化 32 位代码段描述符:将[SECTION .S32]段的物理地址分三部分赋给描述符DESC\_CODE32：

xor eax, eax

mov ax, cs

shl eax, 4

add eax, LABEL\_SEG\_CODE32

mov word [LABEL\_DESC\_CODE32 + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_CODE32 + 4], al

mov byte [LABEL\_DESC\_CODE32 + 7], ah

为加载GDT做准备:将GDT的物理地址填充到GdtPtr中：

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_GDT ; eax <- gdt 基地址

mov dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt 基地址

加载 GDTR，将GdtPtr加载到寄存器gdtr中：

lgdt [GdtPtr]

关中断：

cli

通过操作端口92h打开A20地址线：

in al, 92h

or al, 00000010b

out 92h, al

准备切换到保护模式，把寄存器cr0的第0位置为1：

mov eax, cr0

or eax, 1

mov cr0, eax

真正进入保护模式：

jmp dword SelectorCode32:0 ;这一句会把SelectorCode32装入cs并跳转到Code32Selector:0处

; END of [SECTION .s16]

1. 32 位代码段. 由实模式跳入

[SECTION .s32]

[BITS 32]

LABEL\_SEG\_CODE32:

mov ax, SelectorVideo

mov gs, ax ; 将视频段选择子SelectorVideo的地址赋给gs

mov edi, (80 \* 11 + 79) \* 2 ; 设置输出位置为屏幕11行79列

mov ah, 0Ch ; 设置输出为 0000: 黑底 1100: 红字

mov al, 'P'

mov [gs:edi], ax 往显存中edi偏移处（[gs:edi]地址处）写入P

jmp $ ; 到此停止

SegCode32Len equ $ - LABEL\_SEG\_CODE32

; END of [SECTION .s32]

### 文件夹b：

实现实模式大于1MB内存的寻址，并接着上一次实验，在DOS模式下从保护模式返回到实模式，重新设置各个段寄存器的值。

文件pmtest2.asm：

#### 主要数据结构：

表 3-2[SECTION .gdt] gdt数据段

|  |  |
| --- | --- |
| 数组GDT | LABEL\_GDT 空描述符 |
| LABEL\_DESC\_NORMAL Normal 描述符 |
| LABEL\_DESC\_CODE32 非一致代码段32描述符 |
| LABEL\_DESC\_CODE16 非一致代码段16描述符 |
| LABEL\_DESC\_DATA 数据段描述符 |
| LABEL\_DESC\_STACK 32位堆栈段描述符 |
| LABEL\_DESC\_TEST 测试段描述符 |
| LABEL\_DESC\_VIDEO 显存描述符 |
| 变量 | Gdtlen 表示GDT的长度 |
| GdtPtr 前两个字节为GDT界限  后四个字节为GDT基址 |
| 选择子 | SelectorNormal Normal选择子 |
| SelectorCode32 非一致代码段32选择子 |
| SelectorCode16 非一致代码段16选择子 |
| SelectorData 数据段选择子 |
| SelectorStack 堆栈段选择子 |
| SelectorTest 测试段选择子 |
| SelectorVideo 视频段选择子 |

表 3-3[SECTION .data1] 数据段

|  |  |
| --- | --- |
| 常量 | SPValueInRealMode 字符串 |
| PMMessage 在保护模式中显示的信息"In Protect Mode now. ^-^" |
| OffsetPMMessage PMMessage段偏移 |
| StrTest “ABCDEFGHIJKLMNOPQRSTUVWXYZ” |
| OffsetStrTest StrTest段偏移 |
| DataLen 表示数据段的长度 |

表 3-4[SECTION .gs] 全局堆栈段

|  |  |
| --- | --- |
| 常量 | TopOfStack 栈顶指针 |

#### 关键代码结构：

在pmtest1.asm的基础上，结合上述数据结构，对gdt段和selector进行扩充

[SECTION .gdt]

LABEL\_GDT: Descriptor 0, 0, 0; 空描述符

LABEL\_DESC\_NORMAL: Descriptor 0, 0ffffh, DA\_DRW; Normal 描述符

LABEL\_DESC\_CODE32: Descriptor 0, SegCode32Len-1, DA\_C+DA\_32; 非一致代码段,

LABEL\_DESC\_CODE16: Descriptor 0, 0ffffh, DA\_C; 非一致代码段,

LABEL\_DESC\_DATA:Descriptor 0, DataLen-1, DA\_DRW; Data

LABEL\_DESC\_STACK: Descriptor 0, TopOfStack, DA\_DRWA+DA\_32; Stack

LABEL\_DESC\_TEST: Descriptor 0500000h, 0ffffh, DA\_DRW

LABEL\_DESC\_VIDEO:Descriptor 0B8000h, 0ffffh, DA\_DRW ; 显存首地址

; GDT 结束

GdtLen      equ $-LABEL\_GDT; GDT长度

GdtPtr      dw  GdtLen - 1; GDT界限

        dd  0       ; GDT基地址

; GDT 选择子

SelectorNormal      equ LABEL\_DESC\_NORMAL   - LABEL\_GDT

SelectorCode32      equ LABEL\_DESC\_CODE32   - LABEL\_GDT

SelectorCode16      equ LABEL\_DESC\_CODE16   - LABEL\_GDT

SelectorData        equ LABEL\_DESC\_DATA     - LABEL\_GDT

SelectorStack       equ LABEL\_DESC\_STACK    - LABEL\_GDT

SelectorTest        equ LABEL\_DESC\_TEST     - LABEL\_GDT

SelectorVideo       equ LABEL\_DESC\_VIDEO    - LABEL\_GDT

; END of [SECTION .gdt]

[SECTION .s16]

[BITS 16]

LABEL\_BEGIN:

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax

mov sp, 0100h

mov [LABEL\_GO\_BACK\_TO\_REAL+3], ax

mov [SPValueInRealMode], sp

; 初始化 16 位代码段描述符

mov ax, cs

movzx eax, ax

shl eax, 4

add eax, LABEL\_SEG\_CODE16

mov word [LABEL\_DESC\_CODE16 + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_CODE16 + 4], al

mov byte [LABEL\_DESC\_CODE16 + 7], ah

; 初始化 32 位代码段描述符：将[SECTION .S32]段的物理地址分三部分赋给描述符DESC\_CODE32

xor eax, eax

mov ax, cs

shl eax, 4

add eax, LABEL\_SEG\_CODE32

mov word [LABEL\_DESC\_CODE32 + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_CODE32 + 4], al

mov byte [LABEL\_DESC\_CODE32 + 7], ah

; 初始化数据段描述符

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_DATA

mov word [LABEL\_DESC\_DATA + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_DATA + 4], al

mov byte [LABEL\_DESC\_DATA + 7], ah

; 初始化堆栈段描述符

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_STACK

mov word [LABEL\_DESC\_STACK + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_STACK + 4], al

mov byte [LABEL\_DESC\_STACK + 7], ah

; 为加载 GDTR 作准备:将GDT的物理地址填充到GdtPtr中

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_GDT ; eax <- gdt 基地址

mov dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt 基地址

; 加载 GDTR：lgdt [GdtPtr] #将GdtPtr加载到寄存器gdtr中

lgdt [GdtPtr]

cli ; 关中断

; 打开地址线A20：通过操作端口92h

in al, 92h

or al, 00000010b

out 92h, al

; 准备切换到保护模式：把寄存器cr0的第0位置为1

mov eax, cr0

or eax, 1

mov cr0, eax

; 真正进入保护模式：Jump

jmp dword SelectorCode32:0 ; 执行这一句会把 SelectorCode32 装入 cs, 并跳转到 Code32Selector:0 处

LABEL\_REAL\_ENTRY: ; 从保护模式跳回到实模式就到了这里

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax ；程序重设各个段寄存器值，使ds、es、ss指向cs

mov sp, [SPValueInRealMode] ；恢复sp的值

in al, 92h ; \

and al, 11111101b ; | 关闭 A20 地址线

out 92h, al ; /

sti ; 开中断

mov ax, 4c00h ; \

int 21h ; / 回到 DOS

; END of [SECTION .s16]

[SECTION .s32]

[BITS 32]

LABEL\_SEG\_CODE32: 初始化ds、es、gs、ss： mov ax, SelectorData

mov ds, ax ; ds-指向数据段

mov ax, SelectorTest

mov es, ax ; es-指向新增的5MB的内存段

mov ax, SelectorVideo

mov gs, ax ; gs-指向显存

mov ax, SelectorStack

mov ss, ax ; ss-指向测试段

mov esp, TopOfStack

; 下面显示一个字符串

mov ah, 0Ch ; 0000: 黑底 1100: 红字

xor esi, esi

xor edi, edi

mov esi, OffsetPMMessage ; 源数据偏移

mov edi, (80 \* 10 + 0) \* 2;目的数据偏移。屏幕第 10 行, 第 0 列。

cld

.1: lodsb

test al, al

jz .2

mov [gs:edi], ax

add edi, 2

jmp .1

.2: ; 显示完毕

call DispReturn

call TestRead

call TestWrite 调用写函数TestWrite

call TestRead 调用读内存函数TestRead

jmp SelectorCode16:0 ; 到此停止

跳转到[SECTION .s16code]代码段

TestRead:xor esi, esi TestRead函数

mov ecx, 8

.loop: mov al, [es:esi]

call DispAL

inc esi

loop .loop

call DispReturn

ret

TestWrite:push esi TestWrite函数

push edi

xor esi, esi

xor edi, edi

mov esi, OffsetStrTest ; 源数据偏移

cld

.1:lodsb

test al, al

jz .2

mov [es:edi], al

inc edi

jmp .1

.2: pop edi

pop esi

ret

DispAL: 将AL中的字节用十六进制数表示出来，字的前景色是红色

push ecx

push edx

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov dl, al

shr al, 4

mov ecx, 2

.begin:

and al, 01111b

cmp al, 9

ja .1

add al, '0'

jmp .2

.1: sub al, 0Ah

add al, 'A'

.2: mov [gs:edi], ax

add edi, 2

mov al, dl

loop .begin

add edi, 2

pop edx

pop ecx

ret

DispReturn: 模拟一个回车显示，让下一个字符显示在下一行的开头

push eax

push ebx

mov eax, edi

mov bl, 160

div bl

and eax, 0FFh

inc eax

mov bl, 160

mul bl

mov edi, eax

pop ebx

pop eax

ret

; 16 位代码段. 由 32 位代码段跳入, 跳出后到实模式

[SECTION .s16code]

ALIGN 32

[BITS 16]

LABEL\_SEG\_CODE16: ; 跳回实模式:

mov ax, SelectorNormal

mov ds, ax 把选择子SelectorNormal加载到ds

mov es, ax 把选择子SelectorNormal加载到es

mov fs, ax

mov gs, ax

mov ss, ax 把选择子SelectorNormal加载到ss

mov eax, cr0

and al, 11111110b 清cr0的PE位，跳回实模式

mov cr0, eax

LABEL\_GO\_BACK\_TO\_REAL: 跳转到LABEL\_REAL\_ENTRY里

jmp 0:LABEL\_REAL\_ENTRY ;段地址会在程序开始处被设置成正确的值

Code16Len equ $ - LABEL\_SEG\_CODE16

; END of [SECTION .s16code]

SegCode32Len equ $ - LABEL\_SEG\_CODE32

; END of [SECTION .s32]

### 文件夹c：

了解LDT描述符；学会使用挂载指令将文件挂载在临时目录和并运行程序。

文件Pmtest3.asm：

#### 主要数据结构：

[SECTION .gdt] gdt数据段

|  |  |
| --- | --- |
| 数组GDT | LABEL\_GDT 空描述符 |
| LABEL\_DESC\_NORMAL Normal 描述符 |
| LABEL\_DESC\_CODE32 非一致代码段32描述符 |
| LABEL\_DESC\_CODE16 非一致代码段16描述符 |
| LABEL\_DESC\_DATA 数据段描述符 |
| LABEL\_DESC\_STACK 32位堆栈段描述符 |
| LABEL\_DESC\_LDT LDT描述符 |
| LABEL\_DESC\_VIDEO 显存描述符 |
| 变量 | Gdtlen 表示GDT的长度 |
| GdtPtr 前两个字节为GDT界限  后四个字节为GDT基址 |
| 选择子 | SelectorNormal Normal选择子 |
| SelectorCode32 非一致代码段32选择子 |
| SelectorCode16 非一致代码段16选择子 |
| SelectorData 数据段选择子 |
| SelectorStack 堆栈段选择子 |
| SelectoLDT LDT选择子 |
| SelectorVideo 视频段选择子 |

[SECTION .data1] 数据段

|  |  |
| --- | --- |
| 常量 | SPValueInRealMode 字符串 |
| PMMessage 在保护模式中显示的信息"In Protect Mode now. ^-^" |
| OffsetPMMessage PMMessage段偏移 |
| StrTest “ABCDEFGHIJKLMNOPQRSTUVWXYZ” |
| OffsetStrTest StrTest段偏移 |
| DataLen 表示数据段的长度 |

[SECTION .gs] 全局堆栈段

|  |  |
| --- | --- |
| 常量 | TopOfStack 栈顶指针 |

[SECTION .ldt] LDT数据段

|  |  |
| --- | --- |
| LDT数组 | LABEL\_LDT\_DESC\_CODEA LDT描述符 |
| LDTLen 表示LDT的长度 |
| SelectorLDTCodeA LDT选择子 |

#### 主要代码结构：

[SECTION .s16]

[BITS 16]

LABEL\_BEGIN:

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax

mov sp, 0100h

mov [LABEL\_GO\_BACK\_TO\_REAL+3], ax

mov [SPValueInRealMode], sp

; 初始化 16 位代码段描述符

mov ax, cs

movzx eax, ax

shl eax, 4

add eax, LABEL\_SEG\_CODE16

mov word [LABEL\_DESC\_CODE16 + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_CODE16 + 4], al

mov byte [LABEL\_DESC\_CODE16 + 7], ah

; 初始化 32 位代码段描述符：将[SECTION .S32]段的物理地址分三部分赋给描述符DESC\_CODE32

xor eax, eax

mov ax, cs

shl eax, 4

add eax, LABEL\_SEG\_CODE32

mov word [LABEL\_DESC\_CODE32 + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_CODE32 + 4], al

mov byte [LABEL\_DESC\_CODE32 + 7], ah

; 初始化数据段描述符

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_DATA

mov word [LABEL\_DESC\_DATA + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_DATA + 4], al

mov byte [LABEL\_DESC\_DATA + 7], ah

; 初始化堆栈段描述符

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_STACK

mov word [LABEL\_DESC\_STACK + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_STACK + 4], al

mov byte [LABEL\_DESC\_STACK + 7], ah

; 初始化 LDT 在 GDT 中的描述符

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_LDT

mov word [LABEL\_DESC\_LDT + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_LDT + 4], al

mov byte [LABEL\_DESC\_LDT + 7], ah

; 初始化 LDT 中的描述符

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_CODE\_A

mov word [LABEL\_LDT\_DESC\_CODEA + 2], ax

shr eax, 16

mov byte [LABEL\_LDT\_DESC\_CODEA + 4], al

mov byte [LABEL\_LDT\_DESC\_CODEA + 7], ah

; 为加载 GDTR 作准备:将GDT的物理地址填充到GdtPtr中

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_GDT ; eax <- gdt 基地址

mov dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt 基地址

lgdt [GdtPtr] ; 加载 GDTR：将GdtPtr加载到寄存器gdtr中

cli ; 关中断

; 打开地址线A20：通过操作端口92h

in al, 92h

or al, 00000010b

out 92h, al

; 准备切换到保护模式：把寄存器cr0的第0位置为1

mov eax, cr0

or eax, 1

mov cr0, eax

; 真正进入保护模式：Jump

jmp dword SelectorCode32:0 ; 执行这一句会把 SelectorCode32 装入 cs, 并跳转到 Code32Selector:0处

LABEL\_REAL\_ENTRY: ; 从保护模式跳回到实模式就到了这里

；程序重设各个段寄存器值，使ds、es、ss指向as

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax

mov sp, [SPValueInRealMode] ；恢复sp的值

in al, 92h ; ┓

and al, 11111101b ; ┣ 关闭 A20 地址线

out 92h, al ; ┛

sti ; 开中断

mov ax, 4c00h ; ┓

int 21h ; ┛回到 DOS

; END of [SECTION .s16]

[SECTION .s32] ; 32 位代码段. 由实模式跳入

[BITS 32]

LABEL\_SEG\_CODE32:

；初始化ds、gs、ss

mov ax, SelectorData

mov ds, ax ; 数据段选择子

mov ax, SelectorVideo

mov gs, ax ; 视频段选择子

mov ax, SelectorStack

mov ss, ax ; 堆栈段选择子

mov esp, TopOfStack

; 下面显示一个字符串

mov ah, 0Ch ; 0000: 黑底 1100: 红字

xor esi, esi

xor edi, edi

mov esi, OffsetPMMessage ; 源数据偏移

mov edi, (80 \* 10 + 0) \* 2 ; 目的数据偏移。屏幕第10行, 第0列

cld

.1: lodsb

test al, al

jz .2

mov [gs:edi], ax

add edi, 2

jmp .1

.2: ; 显示完毕

call DispReturn

; 加载LDT

mov ax, SelectorLDT

lldt ax

jmp SelectorLDTCodeA:0 ; 跳入局部任务

DispReturn:

push eax

push ebx

mov eax, edi

mov bl, 160

div bl

and eax, 0FFh

inc eax

mov bl, 160

mul bl

mov edi, eax

pop ebx

pop eax

ret

DispReturn: ;模拟一个回车显示，让下一个字符显示在下一行的开头

push eax

push ebx

mov eax, edi

mov bl, 160

div bl

and eax, 0FFh

inc eax

mov bl, 160

mul bl

mov edi, eax

pop ebx

pop eax

ret

[SECTION .s16code] ;16位代码段.由32位代码段跳入,跳出后到实模式

ALIGN 32

[BITS 16]

LABEL\_SEG\_CODE16:

;跳回实模式:把选择子SelectorNormal加载到ds、es、ss、gs、fs

mov ax, SelectorNormal

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

mov ss, ax

;清cr0的PE位，跳回实模式

mov eax, cr0

and al, 11111110b

mov cr0, eax

LABEL\_GO\_BACK\_TO\_REAL:

jmp 0:LABEL\_REAL\_ENTRY ;段地址会在程序开始处被设置成正确的值，跳转到LABEL\_REAL\_ENTRY即段[SECTION .s16]里

Code16Len equ $ - LABEL\_SEG\_CODE16

[SECTION .la] ;LDT32位代码段

ALIGN 32

[BITS 32]

LABEL\_CODE\_A:

mov ax, SelectorVideo

mov gs, ax ; 将视频段选择子SelectorVideo的地址赋给gs

mov edi, (80 \* 12 + 0) \* 2 ;设置输出位置为屏幕第10行第0列

mov ah, 0Ch ;设置输出为 0000: 黑底 1100: 红字

mov al, 'L'

mov [gs:edi], ax ;往显存中edi偏移处写入L

; 准备经由16位代码段跳回实模式

jmp SelectorCode16:0

CodeALen equ $ - LABEL\_CODE\_A

## 3.3 调试过程

可能遇到的问题：

1.将当前文件夹下的pm.img软盘映像挂载到/mnt/floppy目录，并将pmtest.com文件复制到软盘映像，卸载映像时报错“mount：挂载点/mnt/floppy不存在”。

解决方法：

在/mnt目录下创建一个floppy，操作如下：

cd /

sudo mkdir /mnt/floppy

2.将B盘格式化只需要一次就够了，后面不需要再格式化了，后面再格式化就把刚刚存放在里面的文件也一起格式化了，后面在freedos界面也无法正常使用操作的文件。

3.在freedos界面里不要使用小键盘，因为未知原因会导致输入的数字不对应并且会使前面输入的内容消失，使得在freedos界面的操作失败。

文件夹a:

* nasm pmtest1.asm -o pmtest1.bin
* dd if=pmtest1.bin of=a.img bs=512 count=1 conv=notrunc
* 修改bochsrc，确保其中有以下三行

floppya: 1\_44=freedos.img, status=inserted

floppyb: 1\_44=pm.img, status=inserted

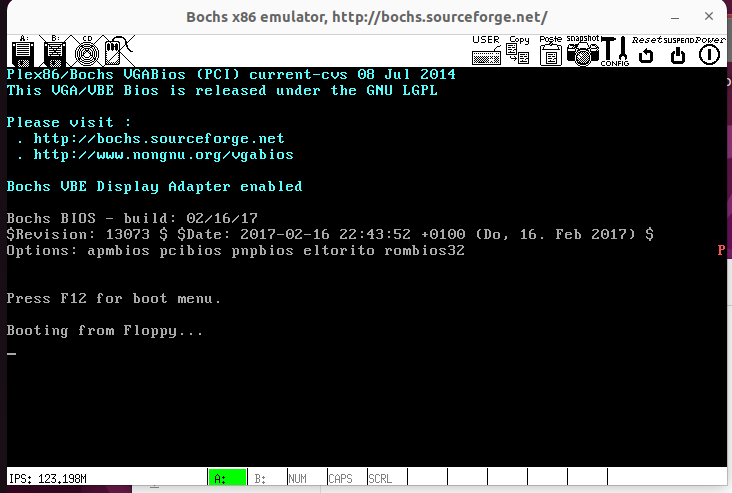
boot: a

* bochs -f bochsrc
* 将B盘格式化，在freedos界面输入:

format b

* nasm pmtest1.asm -o pmtest1.com
* sudo mount -o loop pm.img /mnt/floppy/
* sudo cp pmtest1.com /mnt/floppy/
* sudo umount /mnt/floppy
* bochs -f bochsrc
* freedos 界面输入:
* b:\pmtest1.com

实验成果截图：



文件夹b:

* nasm pmtest2.asm -o pmtest2.bin
* dd if=pmtest2.bin of=a bs=512 count=1 conv=notrunc
* 修改bochsrc，确保其中有以下三行

floppya: 1\_44=freedos.img, status=inserted

floppyb: 1\_44=pm.img, status=inserted

boot: a

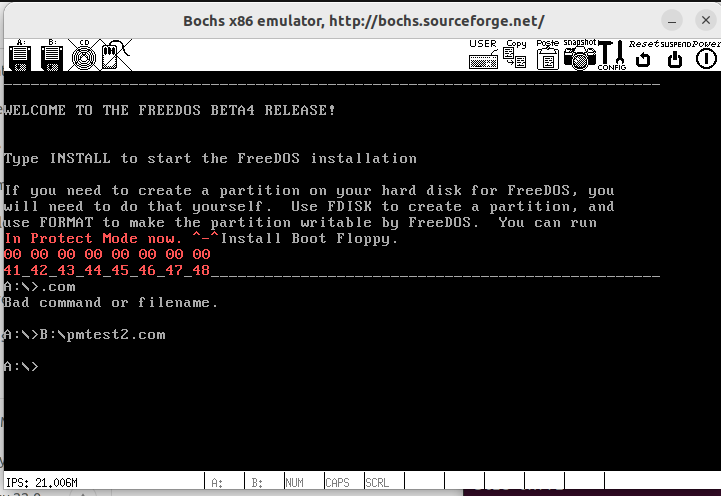
* bochs -f bochsrc
* 将B盘格式化，在freedos界面输入:

format b

* nasm pmtest2.asm -o pmtest2.com
* sudo mount -o loop pm.img /mnt/floppy/
* sudo cp pmtest2.com /mnt/floppy/
* sudo umount /mnt/floppy
* bochs -f bochsrc
* freedos 界面输入:

b:\pmtest2.com

实验成果截图：



文件夹c:

* nasm pmtest3.asm -o pmtest3.bin
* dd if=pmtest3.bin of=a bs=512 count=1 conv=notrunc
* 修改bochsrc，确保其中有以下三行

floppya: 1\_44=freedos.img, status=inserted

floppyb: 1\_44=pm.img, status=inserted

boot: a

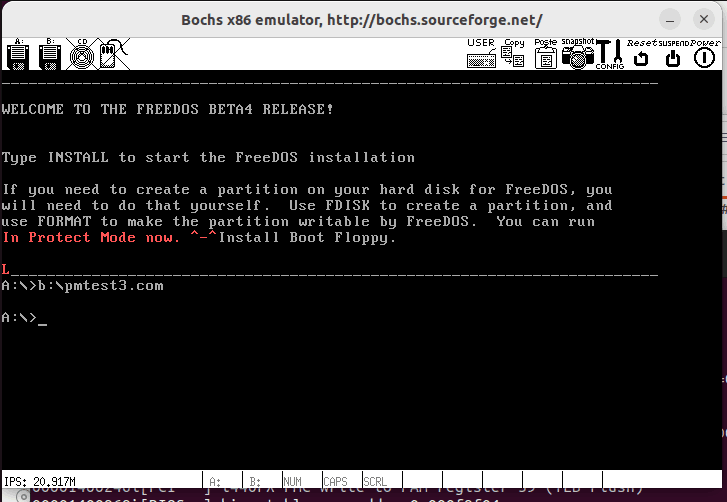
* freedos 界面中输入:

format b

* nasm pmtest3.asm -o pmtest3.com
* sudo mount -o loop pm.img /mnt/floppy/
* sudo cp pmtest3.com /mnt/floppy/
* sudo umount /mnt/floppy
* bochs -f bochsrc
* freedos 界面中输入:

b:\pmtest3.com

实验成果截图：



## 3.4 实验总结

在这七个实验章节之中，困扰我最久的章节反而是第三章。其实第三章的内容不难，相对于其他章节来说第三章其实反而应该算简单的，但是我卡在第三章很久，并不是因为难度高，而是因为会遇到很多奇怪的小问题。这些问题出现的时候很难排查，我重新做了五六遍之后就开始有点暴躁，于是我只能找来同学看着我一步一步做，找出我可能存在的问题。我遇到的第一个问题是修改bochsrc，因为课本上并没有说的很清楚怎么修改，只说明了确保里面有指定的某三行代码。于是我直接添加后尝试进行bochs模拟，发现无法模拟，便将直接添加改成了修改，第一个问题解决。第二个问题是将b盘格式化，由于没仔细看课本的图片我不知道如何格式化，于是上网搜了相关资料后才知道是format，然后往后看的时候发现有图片，懊悔不已。最可恶卡了我最久的一个问题是小键盘输入问题。在freedos界面我每次输入pmtest之后要输入1的时候，我总喜欢用键盘右边的小键盘输入，输入完后那一行前面的内容就自动消失了！一开始我没怀疑这是问题，我以为是正常现象——由于语句过长于是自动折叠了，我还暗自吐槽几个字母能有多长还需要折叠。后面发现是小键盘的问题，在输入123的时候会将前面的内容清除掉，而输入456的时候就变成了123，原因在网上没有搜到，因此我也不清楚如何解决。除此之外，在实验验收过程中，老师问了我一个问题，什么是保护模式。这个问题其实在做实验的时候我已经上网搜过了，但只是因为单纯好奇所以才搜的，并没有记忆，因此我觉得很有必要将这部分内容加到实验总结里。保护模式是一种兼容CPU操作模式，用来增强多工和系统稳定度。CPU在实模式下开机后，由操作系统负责切换到保护模式，现如今大部分x86操作系统都是在保护模式下运行的。

4 切换到保护模式

## 4.1 实验内容及上机实验所用平台

### 4．1．1 实验内容

为了突破512个字节的限制，将新建文件Loader通过引导扇区加载入内存，并把工作分配给它；加载loader进入内存并运行；将控制权交给loader，开始执行内核等由Loader模块去做，这样突破512字节的限制，灵活很多。

### 4．1．2 实验所用平台

（1）VMware

VMWare虚拟机软件是一个“虚拟PC”软件，它使你可以在一台机器上同时运行二个或更多Windows、DOS、LINUX系统。与“多启动”系统相比，VMware采用了完全不同的概念。多启动系统在一个时刻只能运行一个系统，在系统切换时需要重新启动机器。VMWare是真正“同时”运行，多个操作系统在主系统的平台上，就像标准Windows应用程序那样切换。而且每个操作系统你都可以进行虚拟的分区、配置而不影响真实硬盘的数据，可以通过网卡将几台虚拟机用网卡连接为一个局域网。

它主要具有以下功能：

1.不需要分区或重开机就能在同一台PC上使用两种以上的操作系统。

2.完全隔离并且保护不同OS的操作环境以及所有安装在OS上面的应用软件和资料。

3.不同的OS之间还能互动操作，包括网络、周边、文件分享以及复制贴上功能。

4.有复原（Undo）功能。

5.能够设定并且随时修改操作系统的操作环境，如：内存、磁碟空间、周边设备等等。

（2）Ubuntu

Ubuntu是一个以桌面应用为主的[Linux](https://baike.baidu.com/item/Linux/27050" \t "https://baike.baidu.com/item/ubuntu/_blank)操作系统。.Ubuntu基于Debian发行版和[GNOME](https://baike.so.com/doc/5453894-5692279.html" \t "https://baike.so.com/doc/_blank)桌面环境，而从11.04版起，Ubuntu发行版放弃了Gnome桌面环境，改为Unity，与Debian的不同在于它每6个月会发布一个新版本。Ubuntu的目标在于为一般用户提供一个最新的、同时又相当稳定的主要由[自由软件](https://baike.so.com/doc/5925549-6138471.html" \t "https://baike.so.com/doc/_blank)构建而成的操作系统。Ubuntu具有庞大的社区力量，用户可以方便地从社区获得帮助。2013年1月3日，Ubuntu正式发布面向智能手机的移动操作系统。

（3）Bochs

Bochs是一个x86硬件平台的开源模拟器。它可以模拟各种硬件的配置。Bochs模拟的是整个PC平台，包括I/O设备、内存和BIOS。更为有趣的是，甚至可以不使用PC硬件来运行Bochs。事实上，它可以在任何编译运行Bochs的平台上模拟x86硬件。通过改变配置，可以指定使用的CPU(386、486或者586)，以及内存大小等。一句话，Bochs是电脑里的"PC"。根据需要，Bochs还可以模拟多台PC，此外，它甚至还有自己的电源按钮。

## 4.2 数据结构及代码段分析

### 文件夹a：

引导扇区加上BPB等头信息，可以被DOS识别

文件boot.asm

#### 主要数据结构：

|  |  |
| --- | --- |
| FAT12格式 | BS\_OEMName OEM String, 必须 8 个字节 |
| BPB\_BytsPerSec 每扇区字节数：512 |
| BPB\_SecPerClus 每簇扇区数：1 |
| BPB\_RsvdSecCnt Boot记录占用扇区数：1 |
| BPB\_NumFATs FAT 表数码：2 |
| BPB\_RootEntCnt 根目录文件数最大值：224 |
| BPB\_TotSec16 逻辑扇区总数：2880 |
| BPB\_Media 媒体描述符：0xF0 |
| BPB\_FATSz16 每FAT扇区数：9 |
| BPB\_SecPerTrk 每磁道扇区数:18 |
| BPB\_NumHeads 磁头数(面数:2 |
| BPB\_HiddSec 隐藏扇区数：0 |
| BPB\_TotSec32 这个值记录扇区数：0 |
| BS\_DrvNum 中断 13 的驱动器号：0 |
| BS\_Reserved1 未使用：0 |
| BS\_BootSig 扩展引导标记 (29h) |
| BS\_VolID 卷序列号 |
| BS\_VolLab 卷标, 必须 11 个字节：'OrangeS0.02' |
| BS\_FileSysType 文件系统类型, 必须 8个字节：'FAT12 ' |
| 常量 | BootMessage "Hello, OS world!" |

#### 主要代码结构：

LABEL\_START:

;使ds和es两个段寄存器指向与cs相同的段

mov ax, cs

mov ds, ax

mov es, ax

Call DispStr ;调用DispStr子程序显示字符串

jmp $ ; 无限循环

DispStr:

mov ax, BootMessage

mov bp, ax ;设置ES:BP = 串地址

mov cx, 16 ; 设置CX = 串长度

mov ax, 01301h ; 设置AH = 13, AL = 01h

mov bx, 000ch ;设置页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)

mov dl, 0

int 10h ; 10h 号中断

ret

BootMessage: db "Hello, OS world!"

times 510-($-$$) db 0 ;填充剩下的空间，使生成的二进制代码恰好为512字节

dw 0xaa55 ; 结束标志

### 文件夹b：

写代码加载Loader进内存，找到Loader.bin后死循环

#### 主要数据结构：

|  |  |
| --- | --- |
| FAT12格式 | BS\_OEMName OEM String, 必须 8 个字节 |
| BPB\_BytsPerSec 每扇区字节数：512 |
| BPB\_SecPerClus 每簇扇区数：1 |
| BPB\_RsvdSecCnt Boot 记录占用扇区数：1 |
| BPB\_NumFATs FAT 表数码：2 |
| BPB\_RootEntCnt 根目录文件数最大值：224 |
| BPB\_TotSec16 逻辑扇区总数：2880 |
| BPB\_Media 媒体描述符：0xF0 |
| BPB\_FATSz16 每FAT扇区数：9 |
| BPB\_SecPerTrk 每磁道扇区数:18 |
| BPB\_NumHeads 磁头数(面数:2 |
| BPB\_HiddSec 隐藏扇区数：0 |
| BPB\_TotSec32 记录扇区数：0 |
| BS\_DrvNum 中断 13 的驱动器号：0 |
| BS\_Reserved1 未使用：0 |
| BS\_BootSig 扩展引导标记 (29h) |
| BS\_VolID 卷序列号 |
| BS\_VolLab 卷标, 必须 11 个字节：'OrangeS0.02' |
| BS\_FileSysType 文件系统类型, 必须 8个字节 ：'FAT12' |
| 常量 | BootMessage "Hello, OS world!" |
| 变量 | wRootDirSizeForLoop Root Directory 占用的扇区数 |
| wSectorNo 要读取的扇区号 |
| bOdd 奇数还是偶数 |
| 字符串 | LoaderFileName LOADER.BIN 之文件名 |
| BootMessage "Booting " |
| Message1 "Ready." |
| Message2 "No LOADER" |
| 宏 | BaseOfStack 调试状态下堆栈基地址 |
| BaseOfLoader LOADER.BIN被加载到的位置----段地址 |
| OffsetOfLoader LOADER.BIN被加载到的位置----偏移地址 |
| RootDirSectors 根目录占用空间 |
| SectorNoOfRootDirectory Root Directory的第一个扇区号 |

#### 主要代码结构：

文件boot.asm:

LABEL\_START:

;使ds、es、ss三个段寄存器指向与cs相同的段

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax

mov sp, BaseOfStack ;令栈指针sp指向栈底

xor ah, ah ; \

xor dl, dl ; |软驱复位

int 13h ; /

;下面在 A 盘的根目录寻找 LOADER.BIN：遍历根目录取所有的扇区，将每一个扇区加载到内存，然后从中寻找文件名为Loader.bin的条目，直到找到为止。

mov word [wSectorNo], SectorNoOfRootDirectory

LABEL\_SEARCH\_IN\_ROOT\_DIR\_BEGIN:

cmp word [wRootDirSizeForLoop], 0 ;判断根目录区是不是已经读完

jz LABEL\_NO\_LOADERBIN ;如果读完表示没有找到 LOADER.BIN

dec word [wRootDirSizeForLoop]

mov ax, BaseOfLoader

mov es, ax ; es <- BaseOfLoader

mov bx, OffsetOfLoader ; bx <- OffsetOfLoader

mov ax, [wSectorNo] ; ax <- Root Directory 中的某 Sector 号

mov cl, 1

call ReadSector

mov si, LoaderFileName ; ds:si -> "LOADER BIN"

mov di, OffsetOfLoader ; es:di -> BaseOfLoader:0100

cld

mov dx, 10h

LABEL\_SEARCH\_FOR\_LOADERBIN:

cmp dx, 0 ;循环次数控制,

jz LABEL\_GOTO\_NEXT\_SECTOR\_IN\_ROOT\_DIR ;如果已经读完了一个 Sector

dec dx ;就跳到下一个 Sector

mov cx, 11

LABEL\_CMP\_FILENAME:

cmp cx, 0

jz LABEL\_FILENAME\_FOUND ; 如果比较了 11 个字符都相等, 表示找到

dec cx

lodsb ; ds:si -> al

cmp al, byte [es:di]

jz LABEL\_GO\_ON

jmp LABEL\_DIFFERENT ;只要发现不一样的字符就表明本 DirectoryEntry不是我们要找的 LOADER.BIN

LABEL\_GO\_ON:

inc di

jmp LABEL\_CMP\_FILENAME ; 继续循环

LABEL\_DIFFERENT:

and di, 0FFE0h ;为了让它指向本条目开头

add di, 20h

mov si, LoaderFileName ;下一个目录条目

jmp LABEL\_SEARCH\_FOR\_LOADERBIN

LABEL\_GOTO\_NEXT\_SECTOR\_IN\_ROOT\_DIR:

add word [wSectorNo], 1

jmp LABEL\_SEARCH\_IN\_ROOT\_DIR\_BEGIN

LABEL\_NO\_LOADERBIN:

mov dh, 2 ;"No LOADER."

call DispStr ; 显示字符串

%ifdef \_BOOT\_DEBUG\_

mov ax, 4c00h

int 21h ;没有找到 LOADER.BIN, 回到 DOS

%else

jmp $ ; 没有找到 LOADER.BIN, 死循环在这里

%endif

LABEL\_FILENAME\_FOUND: ; 找到 LOADER.BIN 后便来到这里继续

jmp $ ; 代码暂时停在这里

;显示一个字符串, 函数开始时 dh 中应该是字符串序号(0-based)

DispStr:

mov ax, MessageLength

mul dh

add ax, BootMessage

mov bp, ax ; \

mov ax, ds ; | ES:BP = 串地址

mov es, ax ; /

mov cx, MessageLength ; CX = 串长度

mov ax, 01301h ; AH = 13, AL = 01h

mov bx, 0007h ;页号为0(BH = 0) 黑底白字(BL = 07h)

mov dl, 0

int 10h ;int 10h

ret

;从第ax个Sector开始,将cl个Sector读入es:bx中ReadSector:

;怎样由扇区号求扇区在磁盘中的位置 (扇区号 -> 柱面号, 起始扇区, 磁头号)

; 设扇区号为 x

; ┌ 柱面号 = y >> 1

; x ┌ 商 y ┤

; -------------- => ┤ └ 磁头号 = y & 1

; 每磁道扇区数 │

; └ 余 z => 起始扇区号 = z + 1

push bp

mov bp, sp

sub esp, 2 ;辟出两个字节的堆栈区域保存要读的扇区数:byte[bp-2]

mov byte [bp-2], cl

push bx ;保存 bx

mov bl, [BPB\_SecPerTrk] ; bl: 除数

div bl ; y 在 al 中, z 在 ah 中

inc ah ; z ++

mov cl, ah ; cl <- 起始扇区号

mov dh, al ; dh <- y

shr al, 1 ; y >> 1 (y/BPB\_NumHeads)

mov ch, al ; ch <- 柱面号

and dh, 1 ; dh & 1 = 磁头号

pop bx ; 恢复 bx

; 至此, "柱面号, 起始扇区, 磁头号" 全部得到

mov dl, [BS\_DrvNum] ; 驱动器号 (0 表示 A 盘)

.GoOnReading:

mov ah, 2

mov al, byte [bp-2] ; 读 al 个扇区

int 13h

jc .GoOnReading ;如果读取错误CF会被置为1,不停地读,直到正确为止

add esp, 2

pop bp

ret

times 510-($-$$) db 0 ;填充剩下的空间，使二进制代码恰好为512字节

dw 0xaa55 ; 结束标志

文件Loader.asm:

org 0100h

mov ax, 0B800h

mov gs, ax ;令gs指向0B8h00h处

mov ah, 0Fh ;设置 0000:黑底 1111:白字

mov al, 'L' ;设置AL=‘L’

mov [gs:((80 \* 0 + 39) \* 2)], ax ;在(屏幕第0行39列)写入L

jmp $ ; 到此停住

### 文件夹c：

由扇区号求FAT的值，加载Loader，将工作交给它

文件：boot.asm

#### 主要数据结构：

|  |  |
| --- | --- |
| FAT12格式 | BS\_OEMName OEM String, 必须 8 个字节 |
| BPB\_BytsPerSec 每扇区字节数：512 |
| BPB\_SecPerClus 每簇扇区数：1 |
| BPB\_RsvdSecCnt Boot 记录占用扇区数：1 |
| BPB\_NumFATs FAT 表数码：2 |
| BPB\_RootEntCnt 根目录文件数最大值：224 |
| BPB\_TotSec16 逻辑扇区总数：2880 |
| BPB\_Media 媒体描述符：0xF0 |
| BPB\_FATSz16 每FAT扇区数：9 |
| BPB\_SecPerTrk 每磁道扇区数:18 |
| BPB\_NumHeads 磁头数(面数):2 |
| BPB\_HiddSec 隐藏扇区数：0 |
| BPB\_TotSec32 记录扇区数：0 |
| BS\_DrvNum 中断 13 的驱动器号：0 |
| BS\_Reserved1 未使用：0 |
| BS\_BootSig 扩展引导标记 (29h) |
| BS\_VolID 卷序列号 |
| BS\_VolLab 卷标, 必须 11 个字节：'OrangeS0.02' |
| BS\_FileSysType 文件系统类型, 必须 8个字节：'FAT12' |
| 常量 | BootMessage "Hello, OS world!" |
| 变量 | wRootDirSizeForLoop Root Directory 占用的扇区数 |
| wSectorNo 要读取的扇区号 |
| bOdd 奇数还是偶数 |
| 字符串 | LoaderFileName LOADER.BIN 之文件名 |
| BootMessage "Booting " |
| Message1 "Ready. " |
| Message2 "No LOADER" |
| 宏 | BaseOfStack 调试状态下堆栈基地址 |
| BaseOfLoader LOADER.BIN 被加载到的位置(段地址) |
| OffsetOfLoader LOADER.BIN 被加载到的位置(偏移地址) |
| RootDirSectors 根目录占用空间 |
| SectorNoOfRootDirectory Root Directory 的第一个扇区号 |
| SectorNoOfFAT1 FAT1 的第一个扇区号 = BPB\_RsvdSecCnt |
| DeltaSectorNo  DeltaSectorNo = BPB\_RsvdSecCnt + (BPB\_NumFATs \* FATSz) - 2  文件的开始Sector号 = DirEntry中的开始Sector号 + 根目录占用Sector数目 + DeltaSectorNo |

#### 主要代码结构：

LABEL\_START:

;使ds、es、ss三个段寄存器指向与cs相同的段

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax

mov sp, BaseOfStack ;令栈指针sp指向栈底

; 清屏并显示字符串Booting

mov ax, 0600h ; AH = 6, AL = 0h

mov bx, 0700h ; 黑底白字(BL = 07h)

mov cx, 0 ; 左上角: (0, 0)

mov dx, 0184fh ; 右下角: (80, 50)

int 10h

mov dh, 0 ; "Booting "

call DispStr ; 显示字符串

xor ah, ah ; ┓

xor dl, dl ; ┣ 软驱复位

int 13h ; ┛

;在 A 盘的根目录寻找 LOADER.BIN：遍历根目录取所有的扇区，将每一个扇区加载到内存，然后从中寻找文件名为Loader.bin的条目，直到找到为止。

mov word [wSectorNo], SectorNoOfRootDirectory

LABEL\_SEARCH\_IN\_ROOT\_DIR\_BEGIN:

cmp word [wRootDirSizeForLoop], 0 ; ┓

jz LABEL\_NO\_LOADERBIN ; ┣ 判断根目录区是不是已经读完

dec word [wRootDirSizeForLoop] ; ┛ 读完表示没有找到 LOADER.BIN:

mov ax, BaseOfLoader

mov es, ax ; es <- BaseOfLoader

mov bx, OffsetOfLoader ; bx <- OffsetOfLoader , es = BaseOfLoader

mov ax, [wSectorNo] ; ax <- Root Directory 中的某Sector 号

mov cl, 1

call ReadSector

mov si, LoaderFileName ; ds:si -> "LOADER BIN"

mov di, OffsetOfLoader ; es:di -> BaseOfLoader:0100 = BaseOfLoader\*10h+100

cld

mov dx, 10h

LABEL\_SEARCH\_FOR\_LOADERBIN:

; 循环次数控制, 如果已经读完了一个 Sector, 就跳到下一个 Sector

cmp dx, 0

jz LABEL\_GOTO\_NEXT\_SECTOR\_IN\_ROOT\_DIR

dec dx

mov cx, 11

LABEL\_CMP\_FILENAME:

cmp cx, 0

jz LABEL\_FILENAME\_FOUND ; 如果比较 11 个字符都相等, 表示找到

dec cx

lodsb ; ds:si -> al

cmp al, byte [es:di]

jz LABEL\_GO\_ON

jmp LABEL\_DIFFERENT ; 发现不一样的字符表明这个DirectoryEntry 不是我们要找的 LOADER.BIN

LABEL\_GO\_ON:

inc di

jmp LABEL\_CMP\_FILENAME ;继续循环

LABEL\_DIFFERENT:

and di, 0FFE0h ; di &= E0 为了让它指向本条目开头

add di, 20h

mov si, LoaderFileName ;di += 20h 下一个目录条目

jmp LABEL\_SEARCH\_FOR\_LOADERBIN

LABEL\_GOTO\_NEXT\_SECTOR\_IN\_ROOT\_DIR:

add word [wSectorNo], 1

jmp LABEL\_SEARCH\_IN\_ROOT\_DIR\_BEGIN

LABEL\_NO\_LOADERBIN:

mov dh, 2 ; "No LOADER."

call DispStr ; 显示字符串

%ifdef \_BOOT\_DEBUG\_

mov ax, 4c00h

int 21h ; 没有找到 LOADER.BIN, 回到 DOS

%else

jmp $ ; 没有找到 LOADER.BIN, 死循环在这里

%endif

LABEL\_FILENAME\_FOUND: ; 找到 LOADER.BIN 后便来到这里继续

mov ax, RootDirSectors

and di, 0FFE0h ; di -> 当前条目的开始

add di, 01Ah ; di -> 首 Sector

mov cx, word [es:di]

push cx ; 保存此 Sector 在 FAT 中的序号

add cx, ax

add cx, DeltaSectorNo ; cl <- LOADER.BIN的起始扇区号(0-based)

mov ax, BaseOfLoader

mov es, ax ; es <- BaseOfLoader

mov bx, OffsetOfLoader ; bx <- OffsetOfLoader

mov ax, cx ; ax <- Sector 号

LABEL\_GOON\_LOADING\_FILE:

; 每读一个扇区就在 "Booting " 后面打一个点, 形成这样的效果: Booting ......

push ax

push bx

mov ah, 0Eh

mov al, '.'

mov bl, 0Fh

int 10h

pop bx

pop ax

mov cl, 1

call ReadSector

pop ax ; 取出此 Sector 在 FAT 中的序号

call GetFATEntry

cmp ax, 0FFFh

jz LABEL\_FILE\_LOADED

push ax ; 保存 Sector 在 FAT 中的序号

mov dx, RootDirSectors

add ax, dx

add ax, DeltaSectorNo

add bx, [BPB\_BytsPerSec]

jmp LABEL\_GOON\_LOADING\_FILE

LABEL\_FILE\_LOADED:

mov dh, 1 ; "Ready."

call DispStr ; 显示字符串"Ready."

jmp BaseOfLoader:OffsetOfLoader ;正式跳转到加载到内存中的 LOADER.BIN 的开始处,开始执行 LOADER.BIN 的代码,Boot Sector 使命结束。

DispStr:

;显示一个字符串, 函数开始时 dh 中应该是字符串序号(0-based)

mov ax, MessageLength

mul dh

add ax, BootMessage

mov bp, ax ; ┓

mov ax, ds ; ┣ ES:BP = 串地址

mov es, ax ; ┛

mov cx, MessageLength ; CX = 串长度

mov ax, 01301h ; AH = 13, AL = 01h

mov bx, 0007h ; 页号为0(BH = 0) 黑底白字(BL = 07h)

mov dl, 0

int 10h

ret

;从第 ax 个 Sector 开始, 将 cl 个 Sector 读入 es:bx 中

ReadSector:

; 怎样由扇区号求扇区在磁盘中的位置 (扇区号 -> 柱面号, 起始扇区, 磁头号)

; 设扇区号为 x

; ┌ 柱面号 = y >> 1

; x ┌ 商 y ┤

; -------------- => ┤ └ 磁头号 = y & 1

; 每磁道扇区数│

; └ 余 z => 起始扇区号 = z + 1

push bp

mov bp, sp

sub esp, 2 ; 辟出两字节的堆栈区域保存要读的扇区数: byte [bp-2]

mov byte [bp-2], cl

push bx ; 保存 bx

mov bl, [BPB\_SecPerTrk] ; bl: 除数

div bl ; y 在 al 中, z 在 ah 中

inc ah ; z ++

mov cl, ah ; cl <- 起始扇区号

mov dh, al ; dh <- y

shr al, 1 ; y >> 1 (其实是 y/BPB\_NumHeads, 这里BPB\_NumHeads=2)

mov ch, al ; ch <- 柱面号

and dh, 1 ; dh & 1 = 磁头号

pop bx ; 恢复 bx

; 至此, "柱面号, 起始扇区, 磁头号" 全部得到

mov dl, [BS\_DrvNum] ; 驱动器号 (0 表示 A 盘)

.GoOnReading:

mov ah, 2

mov al, byte [bp-2] ; 读 al 个扇区

int 13h

jc .GoOnReading ; 读取错误 CF 会被置为 1, 这时不停读,到正确为止

add esp, 2

pop bp

ret

;找到序号为 ax 的 Sector 在 FAT 中的条目, 结果放在 ax 中

;需要注意, 中间需要读 FAT 的扇区到 es:bx 处, 所以函数一开始保存了 es 和 bx

GetFATEntry:

push es

push bx

push ax

mov ax, BaseOfLoader ; \

sub ax, 0100h ; | 在 BaseOfLoader 后留出 4K 空间用于存放 FAT

mov es, ax ; /

pop ax

mov byte [bOdd], 0

mov bx, 3

mul bx ; dx:ax = ax \* 3

mov bx, 2

div bx ; dx:ax / 2 ==> ax <- 商, dx <- 余数

cmp dx, 0

jz LABEL\_EVEN

mov byte [bOdd], 1

LABEL\_EVEN:;偶数

; ax是 FATEntry 在 FAT 中的偏移量,下面计算在哪个扇区中(FAT占多个扇区)

xor dx, dx

mov bx, [BPB\_BytsPerSec]

div bx ; dx:ax / BPB\_BytsPerSec

; ax <- 商 (FATEntry相对FAT所在扇区号)，dx <- 余数(FATEntry在扇区的偏移)。

push dx

mov bx, 0 ; bx <- 0 es:bx = (BaseOfLoader - 100):00

add ax, SectorNoOfFAT1 ; 此句后ax 就是 FATEntry 所在扇区号

mov cl, 2

call ReadSector ; 一次读两个扇区,因为一个 FATEntry 可能跨两个扇区

pop dx

add bx, dx

mov ax, [es:bx]

cmp byte [bOdd], 1

jnz LABEL\_EVEN\_2

shr ax, 4

LABEL\_EVEN\_2:

and ax, 0FFFh

LABEL\_GET\_FAT\_ENRY\_OK:

pop bx

pop es

ret

Loader.asm

主要代码结构:

org 0100h

mov ax, 0B800h ;令gs指向0B8h00h处

mov gs, ax

mov ah, 0Fh ;设置0000: 黑底 1111: 白字

mov al, 'L' ;设置AL=‘L’

mov [gs:((80 \* 0 + 39) \* 2)], ax ; 在gs偏移处(屏幕第0行39列）写入L

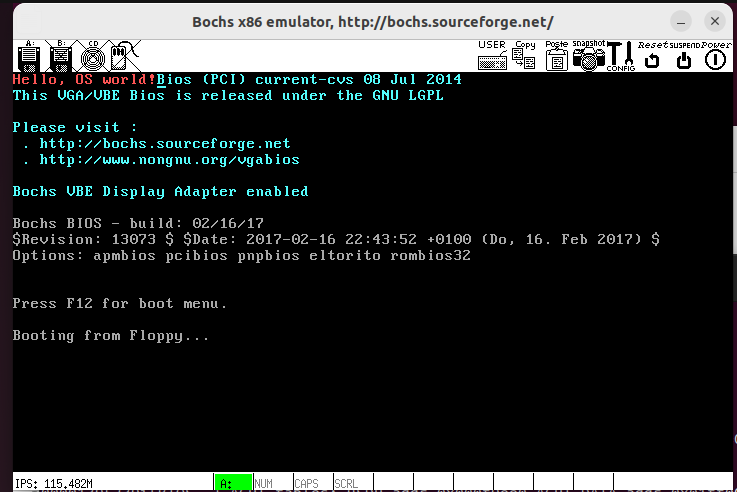
jmp $ ; Start

## 4.3 调试过程

文件夹a:

* nasm boot.asm -o boot.bin
* dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
* 修改bochsrc
* 调试:

bochs -f bochsrc

实验成果截图：

文件夹b:

* nasm loader.asm -o loader.bin
* nasm boot.asm -o boot.bin
* dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
* 修改bochsrc
* 挂载

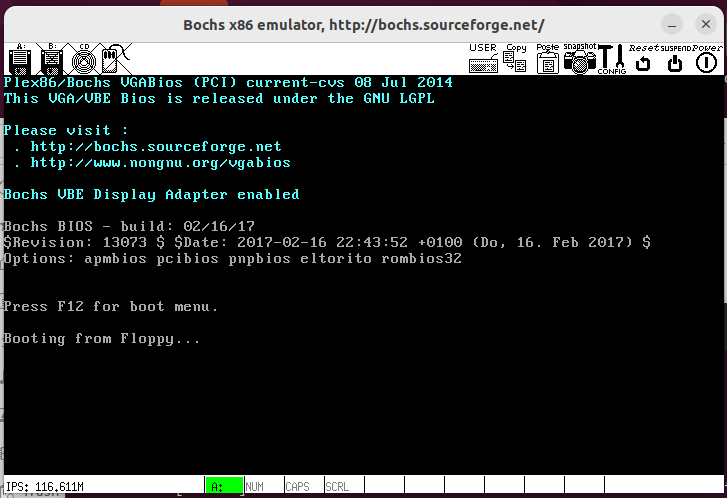
sudo mount -o loop a.img /mnt/floppy/

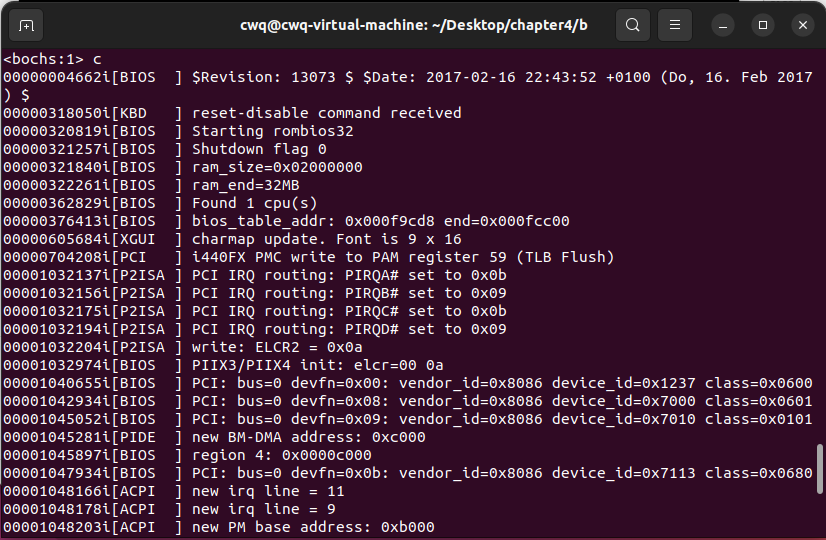
sudo cp loader.bin /mnt/floppy/ -v（cp，copy file，将文件复制到指定工作目录下；-v，详细显示命令执行的操作）

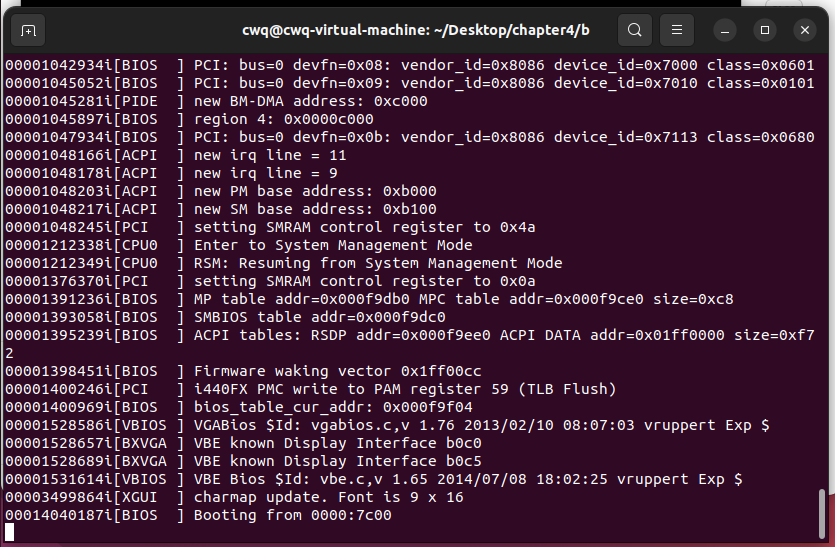
sudo umount /mnt/floppy

* 调试:

bochs -f bochsrc

实验成果截图：





文件夹c:

* nasm loader.asm -o loader.bin
* nasm boot.asm -o boot.bin
* dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
* bochs -f bochsrc
* 挂载

sudo mount -o loop a.img /mnt/floppy/

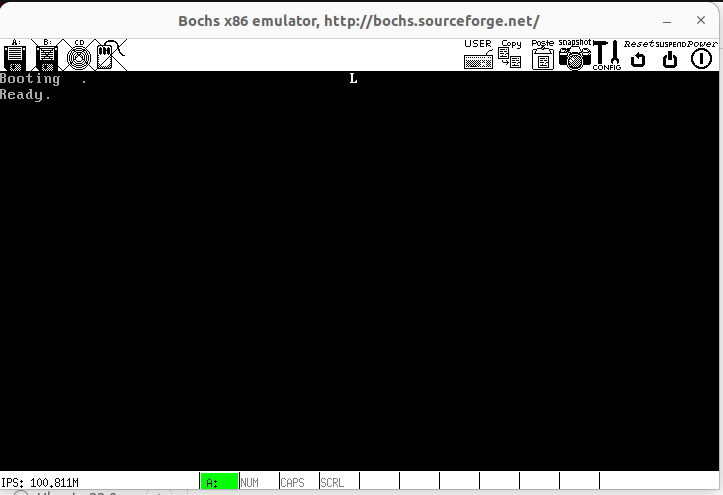
sudo cp loader.bin /mnt/floppy/ -v（cp，copy file，将文件复制到指定工作目录下；-v，详细显示命令执行的操作）

sudo umount /mnt/floppy

* 调试:

bochs -f bochsrc

实验成果截图：



## 4.4 实验总结

在经历了前三章实验后，我对这次的操作系统复现实验有了更大的把握。我学会了挂载，Linux操作系统的特殊操作。由于操作系统的完善以及犯过的很多小白错误我已经知道该如何解决，本次实验我几乎没有遇到什么很大的问题，基本上只要做两三次就能完成，因此没有太多需要记录的东西。

5 操作系统内核的雏形

## 5.1 实验内容及上机实验所用平台

### 5．1．1 实验内容

**在Linux下用汇编写Hello, World！；进一步，汇编和C同时使用；研究ELF，用Loader加载ELF文件，从loader到kernel内核，把kernel内核加载到内存；将控制权交给kernel内核；跳入保护模式,并显示内存的使用情况。**

### 5．1．2 实验所用平台

（1）VMware

VMWare虚拟机软件是一个“虚拟PC”软件，它使你可以在一台机器上同时运行二个或更多Windows、DOS、LINUX系统。与“多启动”系统相比，VMware采用了完全不同的概念。多启动系统在一个时刻只能运行一个系统，在系统切换时需要重新启动机器。VMWare是真正“同时”运行，多个操作系统在主系统的平台上，就像标准Windows应用程序那样切换。而且每个操作系统你都可以进行虚拟的分区、配置而不影响真实硬盘的数据，可以通过网卡将几台虚拟机用网卡连接为一个局域网。

它主要具有以下功能：

1.不需要分区或重开机就能在同一台PC上使用两种以上的操作系统。

2.完全隔离并且保护不同OS的操作环境以及所有安装在OS上面的应用软件和资料。

3.不同的OS之间还能互动操作，包括网络、周边、文件分享以及复制贴上功能。

4.有复原（Undo）功能。

5.能够设定并且随时修改操作系统的操作环境，如：内存、磁碟空间、周边设备等等。

（2）Ubuntu

Ubuntu是一个以桌面应用为主的[Linux](https://baike.baidu.com/item/Linux/27050" \t "https://baike.baidu.com/item/ubuntu/_blank)操作系统。.Ubuntu基于Debian发行版和[GNOME](https://baike.so.com/doc/5453894-5692279.html" \t "https://baike.so.com/doc/_blank)桌面环境，而从11.04版起，Ubuntu发行版放弃了Gnome桌面环境，改为Unity，与Debian的不同在于它每6个月会发布一个新版本。Ubuntu的目标在于为一般用户提供一个最新的、同时又相当稳定的主要由[自由软件](https://baike.so.com/doc/5925549-6138471.html" \t "https://baike.so.com/doc/_blank)构建而成的操作系统。Ubuntu具有庞大的社区力量，用户可以方便地从社区获得帮助。2013年1月3日，Ubuntu正式发布面向智能手机的移动操作系统。

（3）Bochs

Bochs是一个x86硬件平台的开源模拟器。它可以模拟各种硬件的配置。Bochs模拟的是整个PC平台，包括I/O设备、内存和BIOS。更为有趣的是，甚至可以不使用PC硬件来运行Bochs。事实上，它可以在任何编译运行Bochs的平台上模拟x86硬件。通过改变配置，可以指定使用的CPU(386、486或者586)，以及内存大小等。一句话，Bochs是电脑里的"PC"。根据需要，Bochs还可以模拟多台PC，此外，它甚至还有自己的电源按钮。

## 5.2 数据结构及代码段分析

### 文件夹a：

**在Linux下用汇编写Hello, World！**

文件hello.c

#### 主要数据结构：

[section .data]数据段

|  |  |
| --- | --- |
| 字符串 | strHello "Hello, world!", 0Ah |
| 宏 | STRLEN 字符串长度：$ - strHello |

#### 主要代码结构：

[section .text]

global \_start ; 让\_start符号成为可见的标识符，这样链接器就知道跳转到程序中的什么地方并开始执行程序。

\_start:

mov edx, STRLEN ;设置参数三：字符串长度

mov ecx, strHello ;设置参数二：要显示的字符串

mov ebx, 1 ;设置参数一：文件描述符(stdout)

mov eax, 4 ; 系统调用号(sys\_write)

int 0x80 ; 系统调用(调用内核功能)

mov ebx, 0 ;退出程序,参数一：退出代码

mov eax, 1 ;系统调用号(sys\_exit)

int 0x80 ; 系统调用(调用内核功能)

### 文件夹b：

汇编和C同步使用

#### 文件foo.asm

主要数据结构：

[section .data]

|  |  |
| --- | --- |
| 常量 | num1st 3 |
| num2nd 4 |

主要代码结构：

[section .text]

global \_start ; 我们必须导出 \_start 这个入口，以便让链接器识别。

global myprint ; 导出这个函数为了让 bar.c 使用

\_start:

;把num2nd、num1st先后压入栈,调用外部函数choose,栈指针加8

push dword [num2nd] ; \

push dword [num1st] ; |

call choose ; | choose(num1st, num2nd);

add esp, 8 ; /

mov ebx, 0 ;参数一：退出代码

mov eax, 1 ; 系统调用号(sys\_exit)

int 0x80 ; 系统调用(调用内核功能)

; void myprint(char\* msg, int len)

myprint:

mov edx, [esp + 8] ;设置参数2：len

mov ecx, [esp + 4] ; 设置参数1：msg

mov ebx, 1 ;设置参数：文件描述符(stdout)

mov eax, 4 ; 系统调用号(sys\_write)

int 0x80 ; 系统调用(调用内核功能)

ret ;返回

#### 文件bar.c:

主要代码结构：

void myprint(char\* msg, int len); //外部函数myprint()的声明

int choose(int a, int b) //choose（）函数：参数a,b

{

if(a >= b){ //如果a>=b:调用myprint输出"the 1st one"

myprint("the 1st one\n", 13);

}

else{ //否则：调用myprint输出"the 2nd one"

myprint("the 2nd one\n", 13);

}

return 0;

}

### 文件夹c：

**研究ELF，用Loader加载ELF文件，从loader到kernel内核，把kernel内核加载到内存**

#### 文件Fat12hdr.inc

主要数据结构：

|  |  |
| --- | --- |
| FAT12格式 | BS\_OEMName OEM String, 必须 8 个字节 |
| BPB\_BytsPerSec 每扇区字节数：512 |
| BPB\_SecPerClus 每簇扇区数：1 |
| BPB\_RsvdSecCnt Boot 记录占用扇区数：1 |
| BPB\_NumFATs FAT 表数码：2 |
| BPB\_RootEntCnt 根目录文件数最大值：224 |
| BPB\_TotSec16 逻辑扇区总数：2880 |
| BPB\_Media 媒体描述符：0xF0 |
| BPB\_FATSz16 每FAT扇区数：9 |
| BPB\_SecPerTrk 每磁道扇区数:18 |
| BPB\_NumHeads 磁头数(面数:2 |
| BPB\_HiddSec 隐藏扇区数：0 |
| BPB\_TotSec32 wTotalSectorCount为0时记录扇区数：0 |
| BS\_DrvNum 中断 13 的驱动器号：0 |
| BS\_Reserved1 未使用：0 |
| BS\_BootSig 扩展引导标记 (29h) |
| BS\_VolID 卷序列号 |
| BS\_VolLab 卷标, 必须 11 个字节：'OrangeS0.02' |
| BS\_FileSysType 文件系统类型, 必须 8个字节 ：'FAT12 ' |
| 基于 FAT12 头的一些常量定 | RootDirSectors 根目录占用空间 |
| SectorNoOfRootDirectory Root Directory 的第一个扇区号 |
| SectorNoOfFAT1 FAT1 的第一个扇区号 = BPB\_RsvdSecCnt |
| DeltaSectorNo  DeltaSectorNo = BPB\_RsvdSecCnt + (BPB\_NumFATs \* FATSz) - 2  文件的开始Sector号 = DirEntry中的开始Sector号 + 根目录占用Sector数目 + DeltaSectorNo |
| FATSz BPB\_FATSz16：9 |

#### 文件Kernel.asm

主要代码结构：

[section .text]

global \_start ; 导出 \_start

\_start: ; 跳到这里来的时候，我们假设 gs 指向显存

mov ah, 0Fh ; 0000: 黑底 1111: 白字

mov al, 'K' ;设置AL=‘K’

mov [gs:((80 \* 1 + 39) \* 2)], ax ; 在屏幕第1行39列写入L

jmp $ ;无限循环

### 文件夹d：

**将控制权交给kernel内核**

#### 文件load.inc:

主要数据结构：

|  |  |  |
| --- | --- | --- |
| BaseOfLoader | 09000h | LOADER.BIN 被加载到的位置 ---- 段地址 |
| OffsetOfLoader | 0100h | LOADER.BIN 被加载到的位置 ---- 偏移地址 |
| BaseOfLoaderPhyAddr | BaseOfLoader\*10h | LOADER.BIN 被加载到的位置 ---- 物理地址 |
| BaseOfKernelFile | 08000h | KERNEL.BIN 被加载到的位置 ---- 段地址 |
| OffsetOfKernelFile | 0h | KERNEL.BIN 被加载到的位置 ---- 偏移地址 |

#### 文件loader.asm:

jmp SelectorFlatC :KernelEntryPointPhyAddr ;正式进入内核

### 文件夹e：

**跳入保护模式,并显示内存的使用情况**

#### 主要代码结构：

文件loader.asm：

; 将 KERNEL.BIN 的内容经过整理对齐后放到新的位置

; 遍历每一个 Program Header，根据 Program Header 中的信息来确定把什么放进内存，放到什么位置，以及放多少。

InitKernel:

xor esi, esi

mov cx,word [BaseOfKernelFilePhyAddr+2Ch];\ ecx <- pELFHdr ->

movzx ecx, cx ;/ e\_phnum

mov esi,[BaseOfKernelFilePhyAddr+1Ch];esi<- pELFHdr -> e\_phoff

add esi, BaseOfKernelFilePhyAddr ;esi<-OffsetOfKernel+pELFHdr ->e\_phoff

.Begin:

mov eax, [esi + 0]

cmp eax, 0 ; PT\_NULL

jz .NoAction

push dword [esi + 010h];size ;\

mov eax, [esi + 04h] ; |

add eax, BaseOfKernelFilePhyAddr

;|memcpy((void\*)(pPHdr->p\_vaddr)

push eax ;src ;| uchCode + pPHdr->p\_offset,

push dword [esi + 08h] ;dst ;| pPHdr->p\_filesz;

call MemCpy ;|

add esp, 12 ;/

.NoAction:

add esi, 020h ; esi += pELFHdr->e\_phentsize

dec ecx

jnz .Begin

ret

## 调试过程

注意：

1. 安装运行环境:
   1. sudo apt install lzop
   2. sudo apt-get install gcc-multilib g++-multilib module-assistant
   3. sudo apt-get install build-essential libssl-dev libffi-dev python-dev
   4. sudo apt-get update
2. 如何修改:
   1. sudo apt-get install ia32-libs libc6-dev-i386 linux32

dpkg --print-foreign-architectures

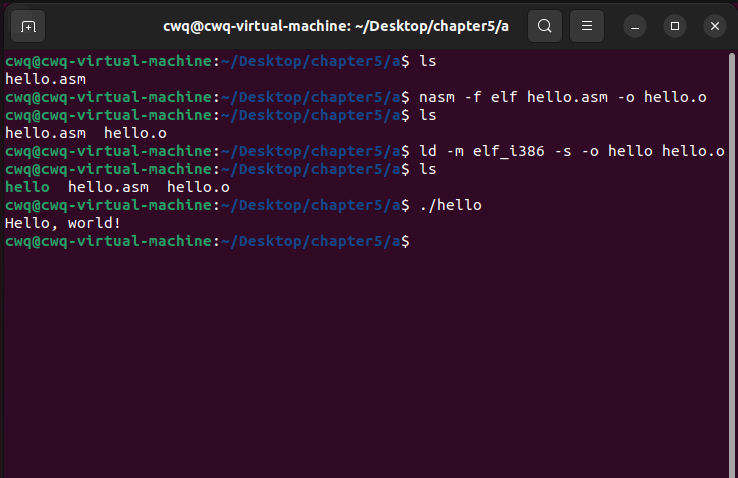
出现结果：i386,则为已经打开

文件夹a:

* ls
* nasm -f elf hello.asm -o hello.o（-f elf指定输出文件为elf格式）
* ls
* ld -m elf\_i386 -s -o hello hello.o（ld将目标文件连接为可执行程序，-s意为strip all）

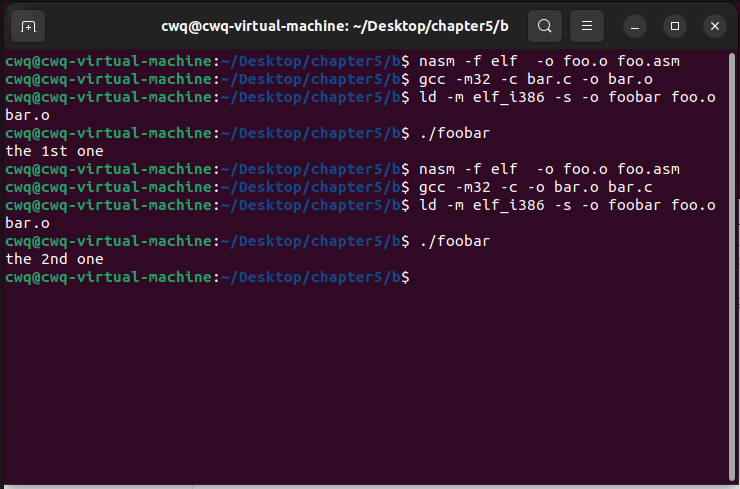
如果报错： ld: i386 architecture of input file `hello.o' is incompatible with i386:x86-64 output说明装的是64位系统.

* ls
* ./hello

实验成果截图：

文件夹b:

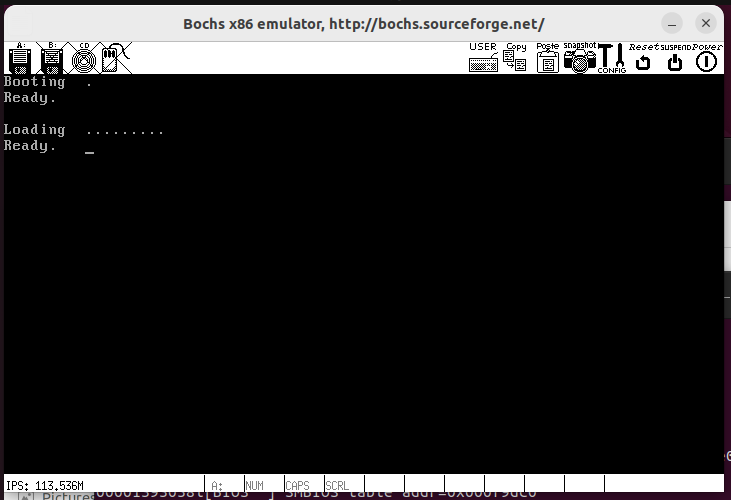
* nasm -f elf -o foo.o foo.asm
* gcc -m32 -c bar.c -o bar.o
* ld -m elf\_i386 -s -o foobar foo.o bar.o
* ./foobar
* 修改foo.asm中的num2nd=2(原来为num2nd=4)
* nasm -f elf -o foo.o foo.asm
* gcc -m32 -c -o bar.o bar.c
* ld -m elf\_i386 -s -o foobar foo.o bar.o
* ./foobar

实验成果截图：

文件夹c:

* nasm loader.asm -o loader.bin
* nasm boot.asm -o boot.bin
* dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
* nasm -f elf -o kernel.o kernel.asm
* ld -m elf\_i386 -s -o kernel.bin kernel.o
* sudo mount -o loop a.img /mnt/floppy/
* sudo cp kernel.bin /mnt/floppy/ -v
* sudo umount /mnt/floppy
* 更改bochsrc
* 调试：

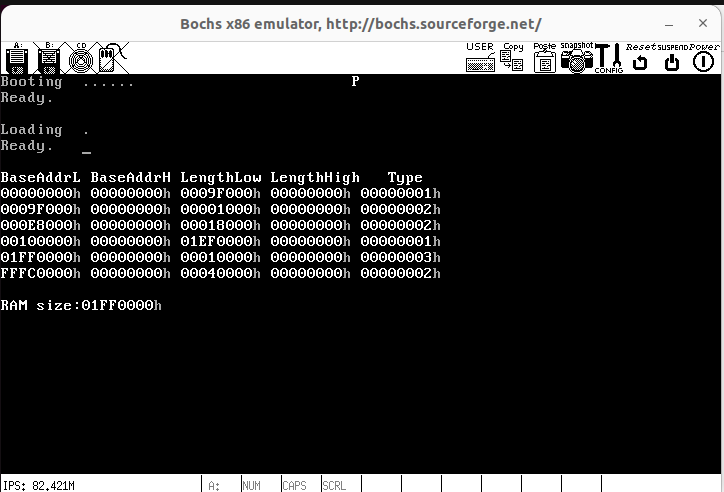
bochs -f bochsrc

实验成果截图：

文件夹d:

* nasm loader.asm -o loader.bin
* nasm boot.asm -o boot.bin
* dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
* nasm -f elf -o kernel.o kernel.asm
* ld -m elf\_i386 -s -o kernel.bin kernel.o
* sudo mount -o loop a.img /mnt/floppy/
* sudo cp kernel.bin /mnt/floppy/ -v
* sudo umount /mnt/floppy
* 更改bochsrc
* 调试：

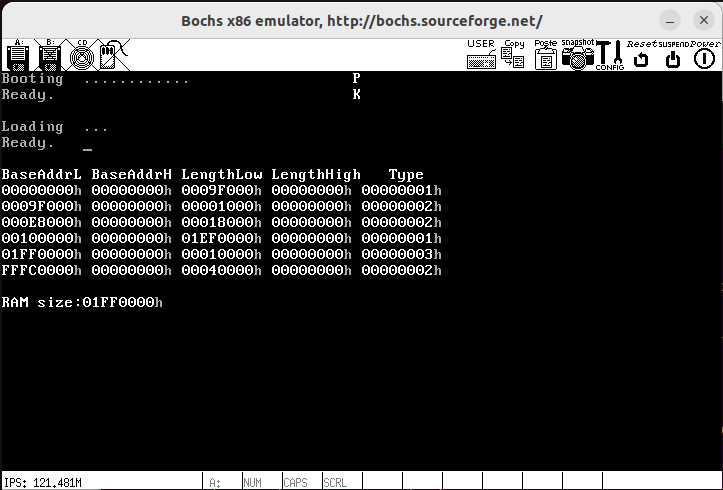
bochs -f bochsrc

实验成果截图：

文件夹e:

* nasm loader.asm -o loader.bin
* nasm boot.asm -o boot.bin
* dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
* nasm -f elf -o kernel.o kernel.asm
* ld -m elf\_i386 -s -o kernel.bin kernel.o
* sudo mount -o loop a.img /mnt/floppy/
* sudo cp kernel.bin /mnt/floppy/ -v
* sudo umount /mnt/floppy
* 更改bochsrc
* 调试：

bochs -f bochsrc

实验成果截图：

## 实验总结

在刚开始做实验的时候，我一开始是直接按照课本中的代码来进行实验，显然我的实验一开始失败了。在网上搜索查询资料过后，我找到了原因——这本书是用32位虚拟机做的，而我使用的是之前下载的64位虚拟机，因此有些地方需要更改。我重新下载了新的匹配的运行环境并让其运行起来。然后发现这个代码也是无法正常执行，因此上网搜查完资料再加上询问完同学过后更改成了可以在64位虚拟机上实现的代码。同时本章节所做的操作系统内核已经达到可以使用高级编程语言的程度了，这使得我们后面增加新功能的时候更加地方便，可以更多地使用我们更习惯的c++而不需要再去写底层的汇编语言。除此之外我们也为后面使用Makefile文件做好了铺垫，以便我们可以免去那些繁杂重复的工作。虽然这个章节操作并不难，但是却是我们意义重大的一步——我们将进入保护模式的复杂操作交给了内核，以后这些重复性工作将由内核替我们完成。

6 进程与进程调度

## 6.1 实验内容及上机实验所用平台

### 6．1．1 实验内容

**进程切换；丰富中断处理程序，进程状态的保存与恢复，进程调度，解决中断重入问题。**

### 6．1．2 实验所用平台

（1）VMware

VMWare虚拟机软件是一个“虚拟PC”软件，它使你可以在一台机器上同时运行二个或更多Windows、DOS、LINUX系统。与“多启动”系统相比，VMware采用了完全不同的概念。多启动系统在一个时刻只能运行一个系统，在系统切换时需要重新启动机器。VMWare是真正“同时”运行，多个操作系统在主系统的平台上，就像标准Windows应用程序那样切换。而且每个操作系统你都可以进行虚拟的分区、配置而不影响真实硬盘的数据，可以通过网卡将几台虚拟机用网卡连接为一个局域网。

它主要具有以下功能：

1.不需要分区或重开机就能在同一台PC上使用两种以上的操作系统。

2.完全隔离并且保护不同OS的操作环境以及所有安装在OS上面的应用软件和资料。

3.不同的OS之间还能互动操作，包括网络、周边、文件分享以及复制贴上功能。

4.有复原（Undo）功能。

5.能够设定并且随时修改操作系统的操作环境，如：内存、磁碟空间、周边设备等等。

（2）Ubuntu

Ubuntu是一个以桌面应用为主的[Linux](https://baike.baidu.com/item/Linux/27050" \t "https://baike.baidu.com/item/ubuntu/_blank)操作系统。.Ubuntu基于Debian发行版和[GNOME](https://baike.so.com/doc/5453894-5692279.html" \t "https://baike.so.com/doc/_blank)桌面环境，而从11.04版起，Ubuntu发行版放弃了Gnome桌面环境，改为Unity，与Debian的不同在于它每6个月会发布一个新版本。Ubuntu的目标在于为一般用户提供一个最新的、同时又相当稳定的主要由[自由软件](https://baike.so.com/doc/5925549-6138471.html" \t "https://baike.so.com/doc/_blank)构建而成的操作系统。Ubuntu具有庞大的社区力量，用户可以方便地从社区获得帮助。2013年1月3日，Ubuntu正式发布面向智能手机的移动操作系统。

（3）Bochs

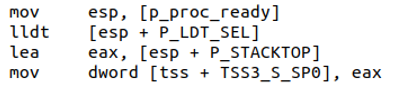
Bochs是一个x86硬件平台的开源模拟器。它可以模拟各种硬件的配置。Bochs模拟的是整个PC平台，包括I/O设备、内存和BIOS。更为有趣的是，甚至可以不使用PC硬件来运行Bochs。事实上，它可以在任何编译运行Bochs的平台上模拟x86硬件。通过改变配置，可以指定使用的CPU(386、486或者586)，以及内存大小等。一句话，Bochs是电脑里的"PC"。根据需要，Bochs还可以模拟多台PC，此外，它甚至还有自己的电源按钮。

## 6.2 数据结构及代码段分析

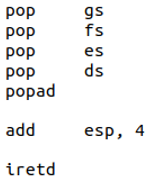
在做完实验报告之后重新阅读发现此章节部分所填较多内容较乱不方便老师查看，故在最前方总结一个大概：

**进程切换，实现从ring0到ring1的切换**

中断发生时，从内核栈切换到进程栈，指向就绪队列中下一个应该被执行的进程，同时用TSS保存ring0的信息：



取出下一个进程所需要的的寄存器值，利用iretd转到ring1特权级，执行该进程。



**丰富中断处理程序，让时钟中断处理可以不停发生，并在进行进程调度的时候做到进程的保存与恢复**

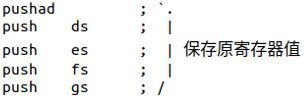
让中断可以不停发生：

设置EOI：

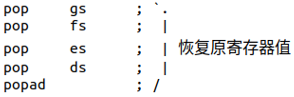


为了让被中断的进程能够顺利恢复，在进程表中，我们给每一个寄存器预留了位置，以便让他们把所有的值都保存下来，这样在进程奇幻的时候就不会对进程产生不良影响。

push保存原寄存器值：



pop恢复原寄存器值：



为了保存内核栈信息，在从ring0到ring1的时候，我们使用TSS保存ring0的信息。在每一次restart的时候，使用iretd返回ring1之前，我们都要进行检查，以确保tee.esp0是正确的。



**解决中断重入**

如果中断处理的时间足够长，以至于在它完成之前就会有下一个中断产生，我们就会发现中断一直在进行，而不会回到我们的程序。之所以会产生这种情况，是因为在一次中断还未处理完时，又一次中断发生了。这时程序又跳到中断处理程序的开头，如此反复，永远也执行不到中断处理程序的结尾—跳回进程继续执行。而且，由于压栈操作多而出栈操作少,随着时间的继续，当堆栈溢出的时候，错误就会发生。

为了解决中断重入问题，我们在main.c中设置一个全局变量k\_reenter。

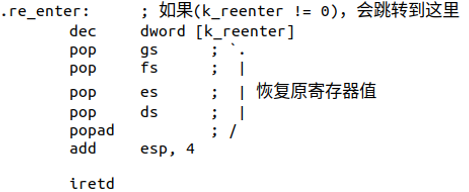


这个全局变量有一个初值-1，当中断处理程序开始执行时它自加，结束时自减。在处理程序开头处这个变量需要被检查一下，如果值不是0(0=-1+1)，则说明在一次中断未处理完之前就又发生了一次中断，这时直接跳到最后，结束中断处理程序的执行。

然后判断是否需要跳转



如果k\_reenter不为0，则直接结束当前中断，恢复上一个进程的执行



**进程调度**

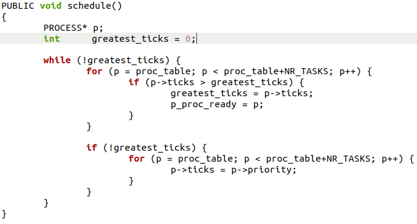
我们在发生时钟中断选择下一个执行的进程时，直接选择进程表中的下一个进程,这种时间片轮转的方式给了每个进程均等的机会。现在我们不再给每个进程以相等的机会了，我们希望能够实现进程的优先级。实现的方法并不困难，稍微修改调度算法就可以了。

给每一个进程都添加一个变量ticks，在一段时间的开头，ticks的值有大有小，等于其初值priority。进程每获得一个运行周期，ticks就减1,当减到0时，此进程就不再获得执行的机会，直到所有进程的ticks都减到0为止。这样，每个进程获得的执行时间就不一样了。

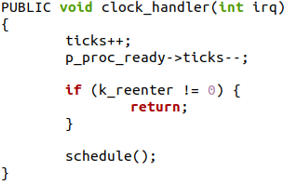
首先变量定义如下



然后对于进程调度，我们可以单独写一个函数，叫做schedule()，放在proc.c中



同时我们修改时钟中断如下



在运行程序之后我们发现，当前的A、B、C三个字母的个数之比是139:71:54,大体相当于2.57:1.31:1，与进程优先级5:1.67:1 （15:5:3)相差比较大。

这是因为，当多个进程ticks相当时，每一次进程调度的时候只有某一个进程的ticks会减1(而不是3个进程ticks同时减1)，所以，总共调度的次数应该是3个进程的ticks之和(150+50+30=230)。这个规律放在中间某个过程中也是适用的，比如到最后阶段，当A和B的ticks都减到30时，3个进程同时运行，总共运行的时间将是30×3=90ticks。所以我们总结出:

进程A执行的循环数=(100+20×2+30×3)/20 = 230/ 20=11.5次

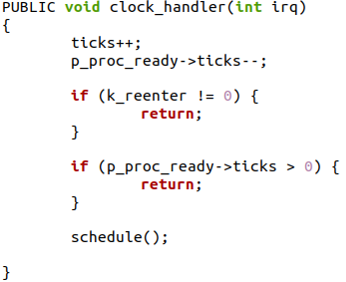
进程B执行的循环数=(0+20×2+30×3)/20= 130/ 20 =6.5次

进程C执行的循环数=(0+0×2+ 30×3)/20= 90 / 20=4.5次

这个结论与试验结果11、7、5是相吻合的。

解决方案如下：

我们让一个进程只有在ticks变为0之后才交出控制权。通过在clock\_handler()中增加一个判断就可以解决：



### 文件kernel.asm

主要代码分析：

; 把 esp 从 LOADER 挪到 KERNEL

mov esp, StackTop ; 堆栈在 bss 段中

mov dword [disp\_pos], 0

sgdt [gdt\_ptr] ; cstart() 中将会用到 gdt\_ptr

call cstart ; 调用cstart改变gdt\_ptr，让它指向新的GDT

lgdt [gdt\_ptr] ; 使用新的GDT

lidt [idt\_ptr]

jmp SELECTOR\_KERNEL\_CS:csinit

csinit: ; “跳转指令强制使用刚刚初始化的结构”——<<OS:D&I 2nd>> P90.

;jmp 0x40:0

;ud2

xor eax, eax

mov ax, SELECTOR\_TSS

ltr ax

;sti

jmp kernel\_main ；跳转到kernel\_main()

;hlt

; 中断和异常 -- 硬件中断

%macro hwint\_master 1

push %1

call spurious\_irq

add esp, 4

hlt

%endmacro

; 中断和异常 -- 异常

divide\_error:

push 0xFFFFFFFF ; no err code

push 0 ; vector\_no = 0

jmp exception

single\_step\_exception:

push 0xFFFFFFFF ; no err code

push 1 ; vector\_no = 1

jmp exception

nmi:

push 0xFFFFFFFF ; no err code

push 2 ; vector\_no = 2

jmp exception

breakpoint\_exception:

push 0xFFFFFFFF ; no err code

push 3 ; vector\_no = 3

jmp exception

overflow:

push 0xFFFFFFFF ; no err code

push 4 ; vector\_no = 4

jmp exception

bounds\_check:

push 0xFFFFFFFF ; no err code

push 5 ; vector\_no = 5

jmp exception

inval\_opcode:

push 0xFFFFFFFF ; no err code

push 6 ; vector\_no = 6

jmp exception

copr\_not\_available:

push 0xFFFFFFFF ; no err code

push 7 ; vector\_no = 7

jmp exception

double\_fault:

push 8 ; vector\_no = 8

jmp exception

copr\_seg\_overrun:

push 0xFFFFFFFF ; no err code

push 9 ; vector\_no = 9

jmp exception

inval\_tss:

push 10 ; vector\_no = A

jmp exception

segment\_not\_present:

push 11 ; vector\_no = B

jmp exception

stack\_exception:

push 12 ; vector\_no = C

jmp exception

general\_protection:

push 13 ; vector\_no = D

jmp exception

page\_fault:

push 14 ; vector\_no = E

jmp exception

copr\_error:

push 0xFFFFFFFF ; no err code

push 16 ; vector\_no = 10h

jmp exception

exception:

call exception\_handler

add esp, 4\*2 ; 让栈顶指向 EIP，堆栈中从顶向下依次是：EIP、CS、EFLAGS

hlt

restart: ;完成ring0到ring1的跳转

; 中断发生时，从内核栈切换到进程栈，指向就绪队列中下一个应该被执行的进程，同时用TSS保存ring0的信息

mov esp, [p\_proc\_ready]

lldt [esp + P\_LDT\_SEL]

lea eax, [esp + P\_STACKTOP]

mov dword [tss + TSS3\_S\_SP0], eax

;取出下个进程所需寄存器值，利用iretd转到ring1特权级执行该进程。

pop gs

pop fs

pop es

pop ds

popad

add esp, 4

iretd

### 文件main.c

主要代码分析：

PUBLIC int kernel\_main()

{

disp\_str("-----\"kernel\_main\" begins-----\n"); //打印字符串"-----\"kernel\_main\" begins-----\n"

//初始化进程表：初始化寄存器、LDTSelector和LDT，LDT selector被赋值为SELECTOR\_LDT\_FIRST，LDT中共有两个描述符，为简化起见，分别被初始化成内核代码段和内核数据段，只是改变了一下DPL以让其运行在低的特权级下。

PROCESS\* p\_proc = proc\_table;

p\_proc->ldt\_sel = SELECTOR\_LDT\_FIRST;

memcpy(&p\_proc->ldts[0], &gdt[SELECTOR\_KERNEL\_CS >> 3], sizeof(DESCRIPTOR));

p\_proc->ldts[0].attr1 = DA\_C | PRIVILEGE\_TASK << 5; // change the DPL

memcpy(&p\_proc->ldts[1], &gdt[SELECTOR\_KERNEL\_DS >> 3], sizeof(DESCRIPTOR));

p\_proc->ldts[1].attr1 = DA\_DRW | PRIVILEGE\_TASK << 5; // change the DPL

//要初始化的寄存器比较多，cs指向LDT中的第一个描述符，ds,es,fs,ss都设置为指向LDT中的第二个描述符，gs仍然指向显存，只是其RPL发生改变。接下来eip指向TestA，这表明进程将从TestA的入口地址开始运行，另外，esp指向单独的栈

p\_proc->regs.cs = ((8 \* 0) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.ds = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.es = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.fs = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.ss = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.gs = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK) | RPL\_TASK;

p\_proc->regs.eip = (u32)TestA;

p\_proc->regs.esp = (u32) task\_stack + STACK\_SIZE\_TOTAL;

p\_proc->regs.eflags = 0x1202; // IF=1, IOPL=1, bit 2 is always 1.

k\_reenter = -1;

/\* 我们在main.c中设置一个全局变量k\_reenter。这个全局变量有一个初值-1，当中断处理程序开始执行时它自加，结束时自减。在处理程序开头处这个变量需要被检查一下，如果值不是0(0=-1+1)，则说明在一次中断未处理完之前就又发生了一次中断，这时直接跳到最后，结束中断处理程序的执行。\*/

p\_proc\_ready = proc\_table; //扳机

restart(); //无限循环

while(1){}

}

void TestA()

{

int i = 0;

while(1){

disp\_str("A"); //调用函数disp\_str输出字符”A”

disp\_int(i++); //调用函数disp\_int输出数字

disp\_str("."); //调用函数disp\_str输出字符”.”

delay(1); //调用函数delay()

}

}

### 文件klib.c

主要代码分析：

PUBLIC char \* itoa(char \* str, int num)

// 数字前面的 0 不被显示出来, 比如 0000B800 被显示成 B800

{

char \* p = str;

char ch;

int i;

int flag = FALSE;

\*p++ = '0';

\*p++ = 'x';

if(num == 0){ \*p++ = '0'; }

else{

for(i=28;i>=0;i-=4){

ch = (num >> i) & 0xF;

if(flag || (ch > 0)){

flag = TRUE;

ch += '0';

if(ch > '9'){ ch += 7; }

\*p++ = ch;

}

}

}

\*p = 0;

return str;

}

PUBLIC void disp\_int(int input) //输出数字

{

char output[16];

itoa(output, input);

disp\_str(output);

}

PUBLIC void delay(int time) //延迟

{

int i, j, k;

for(k=0;k<time;k++){

/\*for(i=0;i<10000;i++){ for Virtual PC \*/

for(i=0;i<10;i++){ // for Bochs

for(j=0;j<10000;j++){}

}

}

}

### 文件proc.h

主要数据结构

|  |  |
| --- | --- |
| 结构体 | STACK\_FRAME 进程表结构体的定义 |
| PROCESS 进程结构体的定义 |
| 宏 | NR\_TASKS 最大允许进程：1 |
| STACK\_SIZE\_TESTA 栈的大小 |
| STACK\_SIZE\_TOTAL STACK\_SIZE\_TESTA |

### 文件protect.c

主要代码结构：

PUBLIC void init\_prot() //初始化 IDT

{

init\_8259A(); //调用init\_8259A()函数

// 全部初始化成中断门(没有陷阱门)

init\_idt\_desc(INT\_VECTOR\_DIVIDE, DA\_386IGate, divide\_error, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_DEBUG, DA\_386IGate, single\_step\_exception, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_NMI, DA\_386IGate, nmi, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_BREAKPOINT, DA\_386IGate, breakpoint\_exception, PRIVILEGE\_USER);

init\_idt\_desc(INT\_VECTOR\_OVERFLOW, DA\_386IGate, overflow, PRIVILEGE\_USER);

init\_idt\_desc(INT\_VECTOR\_BOUNDS, DA\_386IGate, bounds\_check, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_INVAL\_OP, DA\_386IGate, inval\_opcode, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_COPROC\_NOT, DA\_386IGate, copr\_not\_available, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_DOUBLE\_FAULT, DA\_386IGate, double\_fault, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_COPROC\_SEG, DA\_386IGate, copr\_seg\_overrun, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_INVAL\_TSS, DA\_386IGate, inval\_tss, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_SEG\_NOT, DA\_386IGate, segment\_not\_present, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_STACK\_FAULT, DA\_386IGate, stack\_exception, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_PROTECTION, DA\_386IGate, general\_protection, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_PAGE\_FAULT, DA\_386IGate, page\_fault, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_COPROC\_ERR, DA\_386IGate, copr\_error, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 0, DA\_386IGate, hwint00, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 1, DA\_386IGate, hwint01, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 2, DA\_386IGate, hwint02, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 3, DA\_386IGate, hwint03, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 4, DA\_386IGate, hwint04, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 5, DA\_386IGate, hwint05, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 6, DA\_386IGate, hwint06, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ0 + 7, DA\_386IGate, hwint07, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 0, DA\_386IGate, hwint08, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 1, DA\_386IGate, hwint09, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 2, DA\_386IGate, hwint10, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 3, DA\_386IGate, hwint11, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 4, DA\_386IGate, hwint12, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 5, DA\_386IGate, hwint13, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 6, DA\_386IGate, hwint14, PRIVILEGE\_KRNL);

init\_idt\_desc(INT\_VECTOR\_IRQ8 + 7, DA\_386IGate, hwint15, PRIVILEGE\_KRNL);

//填充 GDT 中 TSS 这个描述符

memset(&tss, 0, sizeof(tss));

tss.ss0 = SELECTOR\_KERNEL\_DS;

init\_descriptor(&gdt[INDEX\_TSS],

vir2phys(seg2phys(SELECTOR\_KERNEL\_DS), &tss),

sizeof(tss) - 1,

DA\_386TSS);

tss.iobase = sizeof(tss); //没有I/O许可位图

// 填充 GDT 中进程的 LDT 的描述符

init\_descriptor(&gdt[INDEX\_LDT\_FIRST],

vir2phys(seg2phys(SELECTOR\_KERNEL\_DS), proc\_table[0].ldts),

LDT\_SIZE \* sizeof(DESCRIPTOR) - 1,

DA\_LDT);

}

PUBLIC void init\_idt\_desc(unsigned char vector, u8 desc\_type, int\_handler handler, unsigned char privilege) //初始化 386 中断门

{

GATE \* p\_gate = &idt[vector];

u32 base = (u32)handler;

p\_gate->offset\_low = base & 0xFFFF;

p\_gate->selector = SELECTOR\_KERNEL\_CS;

p\_gate->dcount = 0;

p\_gate->attr = desc\_type | (privilege << 5);

p\_gate->offset\_high = (base >> 16) & 0xFFFF;

}

## 6.3 调试过程

### 修改Makefile(只有64位系统下才需要)

CFLAGS = -I include/ -c -fno-builtin

LDFLAGS = -s -Ttext $(ENTRYPOINT)

DASMFLAGS = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET)

改为：

CFLAGS = -m32 -I include/ -c -fno-builtin -fno-stack-protector

LDFLAGS = -m elf\_i386 -s -Ttext $(ENTRYPOINT)

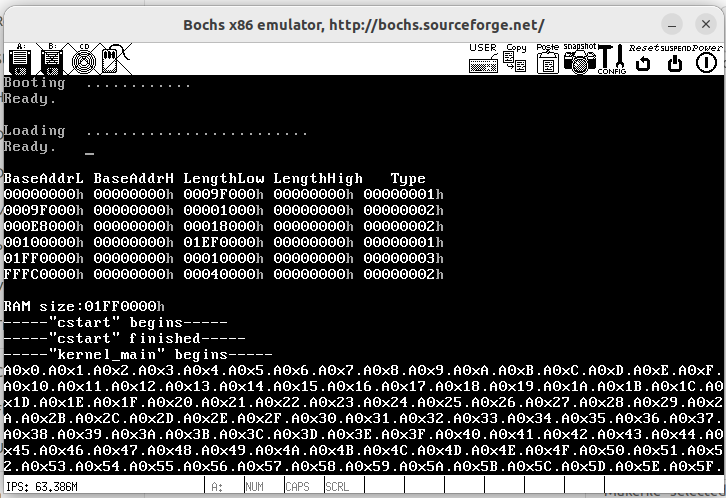
DASMFLAGS = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET) -m32

* （可以不输入）dd if=boot/boot.bin of=a.img bs=512 count=1 conv=notrunc
* 使用Make进行编译与写入a.img，然后启动bochs。具体对应文件夹对应操作如下：

### 文件夹a：

#### 进程切换：1.make 2.bochs

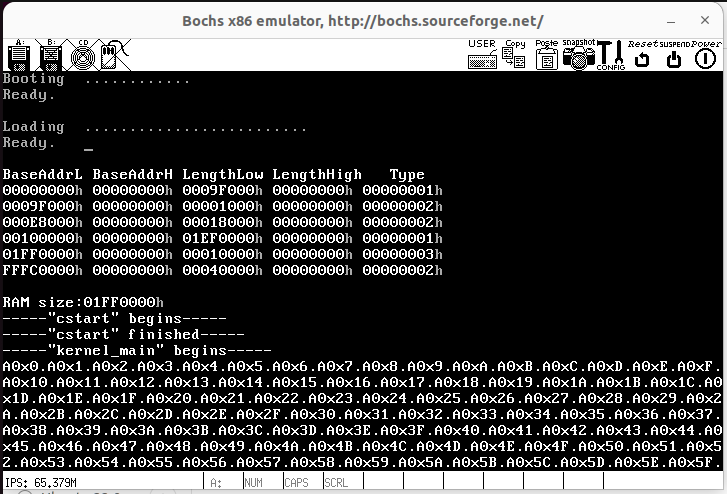
实验成果截图：



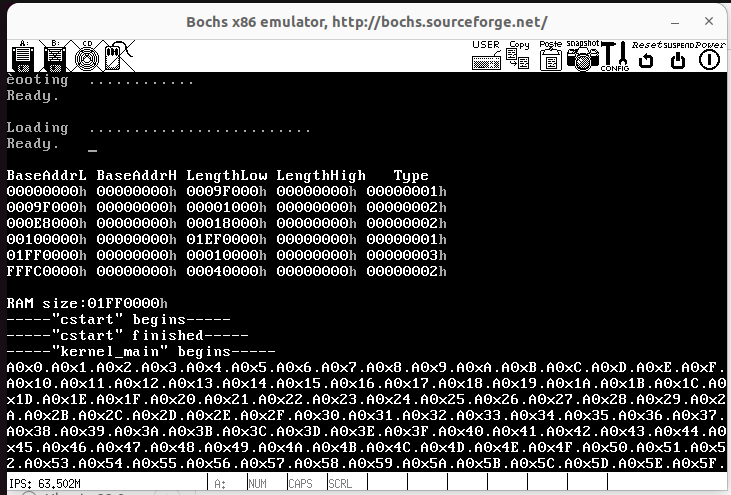
### 文件夹b：

#### 丰富中断处理程序，让时钟中断处理可以不停地发生而不是只发生一次：

Stage1: 1. make stage1 2. bochs

实验成果截图：

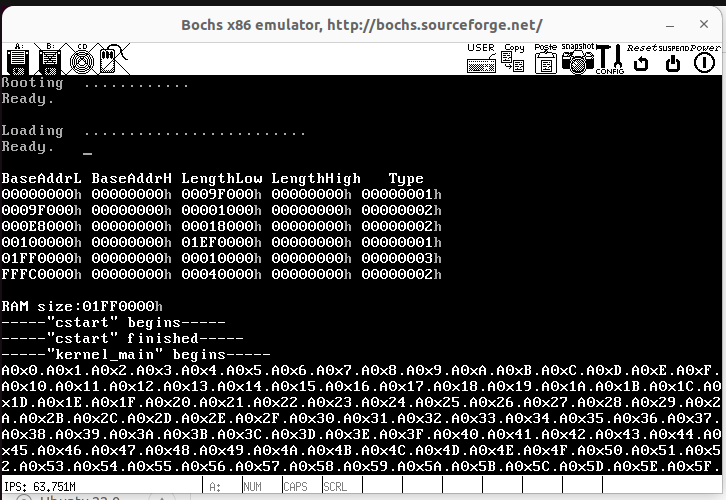
Stage2: 1. make stage2 2. bochs

实验成果截图：

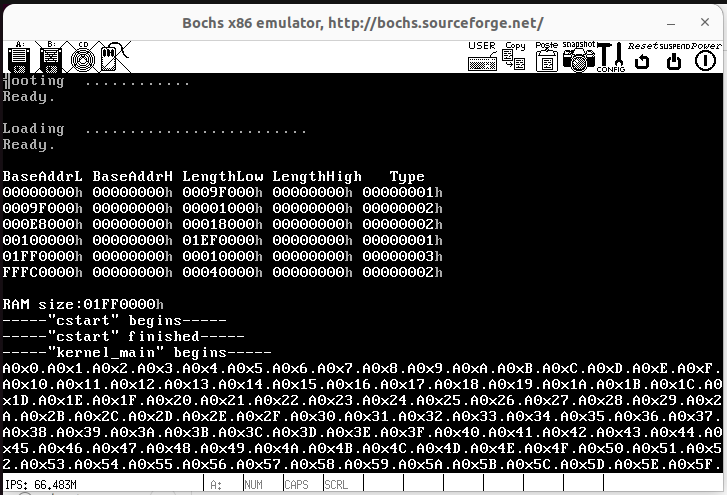
#### 进程状态的保存与恢复：

Stage3: 1. make stage3 2. bochs

实验成果截图：

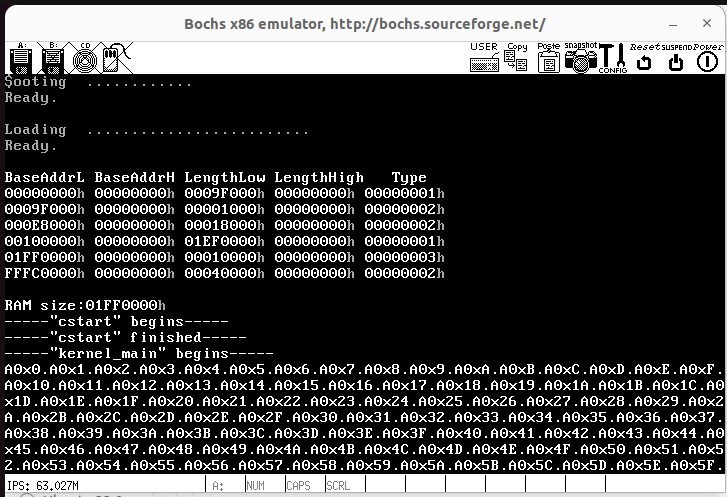


Stage4: 1. make stage4 2. bochs

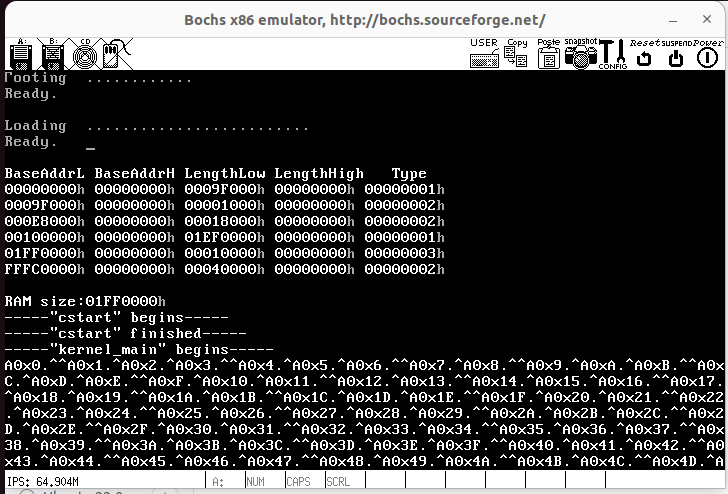
实验成果截图：

#### 进程调度：

Stage5: 1. make stage5 2. bochs

实验成果截图：

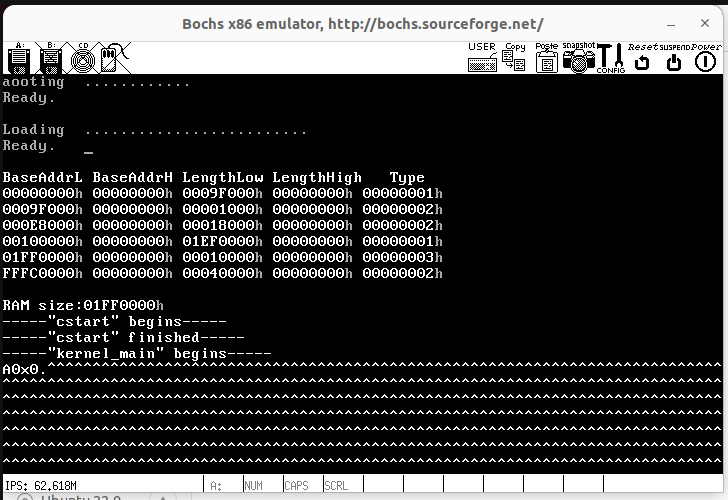
final: 1. make final 2. bochs

实验成果截图：

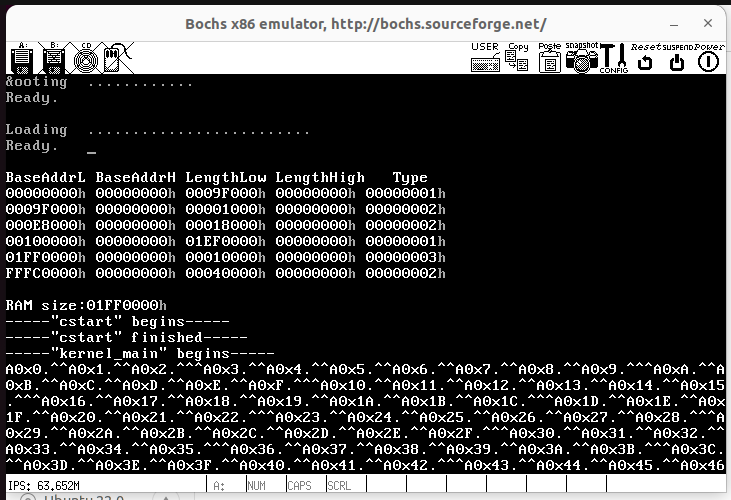
### 文件夹c：

解决中断重入问题：

Stage1: 1. make stage1 2. bochs

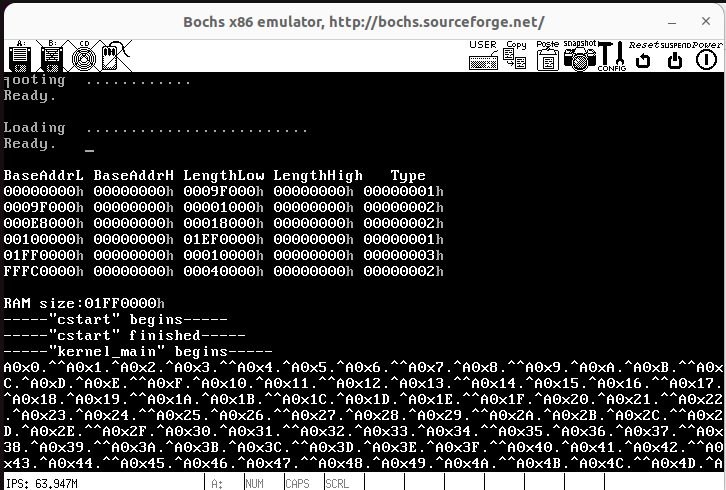
实验成果截图：

Stage2: 1. make stage2 2. bochs

实验成果截图：

Stage3: 1. make stage3 2. bochs

实验成果截图：

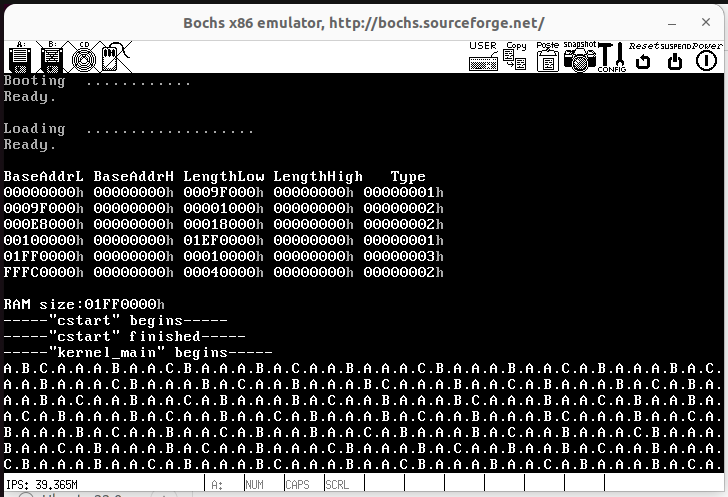


### 文件夹o：

进程调度：

1. make image 2. bochs

实验成果截图：



## 6.4 实验总结

本章节将重点放到了进程上。从进程的切换，中断处理程序的丰富，再到进程切换时进程状态的保存与恢复，最后到进程调度，同时也考虑并解决了中断重入问题。对于我们做的小型的简陋操作系统来说，进程的出现以及进程相关操作的完善是一个重大的突破！虽然还没办法进行输入输出，但我们可以说这个操作系统已经是一个基本完善的操作系统了。本次实验进行了make的概念理解并学会了使用Makefile来化简我们的操作步骤以及在同一个终端里面展现不一样的内核效果。操作途中遇到耗费较多时间的问题就是Makefile里面有些内容需要更改，因为里面的内容是以32位虚拟机为操作对象来做的，而我们使用的是64位虚拟机，将对应代码更改完后直接进行对应操作即完成实验。但是在做完实验之后，我也没有直接跳过，而是一个函数一个函数去看了他们各自的功能，大部分在里面都注释了，即使没有的我也在课本里找到了相对应的解释，这使得我做实验报告时再一次看代码的时候没有之前做的时候那么懵懂了，大概也能想到哪些应该是主要代码结构，提高了我做实验报告的速度。

7 操作系统的输入/输出系统

## 7.1 实验内容及上机实验所用平台

### 7．1．1 实验内容

**实现简单的I/O，从键盘输入字符的中断开始；敲击键盘时获取并打印扫描码；创建对应打印扫描码解析数组，在敲击键盘时可以打印对应字符。**

### 7．1．2 实验所用平台

（1）VMware

VMWare虚拟机软件是一个“虚拟PC”软件，它使你可以在一台机器上同时运行二个或更多Windows、DOS、LINUX系统。与“多启动”系统相比，VMware采用了完全不同的概念。多启动系统在一个时刻只能运行一个系统，在系统切换时需要重新启动机器。VMWare是真正“同时”运行，多个操作系统在主系统的平台上，就像标准Windows应用程序那样切换。而且每个操作系统你都可以进行虚拟的分区、配置而不影响真实硬盘的数据，可以通过网卡将几台虚拟机用网卡连接为一个局域网。

它主要具有以下功能：

1.不需要分区或重开机就能在同一台PC上使用两种以上的操作系统。

2.完全隔离并且保护不同OS的操作环境以及所有安装在OS上面的应用软件和资料。

3.不同的OS之间还能互动操作，包括网络、周边、文件分享以及复制贴上功能。

4.有复原（Undo）功能。

5.能够设定并且随时修改操作系统的操作环境，如：内存、磁碟空间、周边设备等等。

（2）Ubuntu

Ubuntu是一个以桌面应用为主的[Linux](https://baike.baidu.com/item/Linux/27050" \t "https://baike.baidu.com/item/ubuntu/_blank)操作系统。.Ubuntu基于Debian发行版和[GNOME](https://baike.so.com/doc/5453894-5692279.html" \t "https://baike.so.com/doc/_blank)桌面环境，而从11.04版起，Ubuntu发行版放弃了Gnome桌面环境，改为Unity，与Debian的不同在于它每6个月会发布一个新版本。Ubuntu的目标在于为一般用户提供一个最新的、同时又相当稳定的主要由[自由软件](https://baike.so.com/doc/5925549-6138471.html" \t "https://baike.so.com/doc/_blank)构建而成的操作系统。Ubuntu具有庞大的社区力量，用户可以方便地从社区获得帮助。2013年1月3日，Ubuntu正式发布面向智能手机的移动操作系统。

（3）Bochs

Bochs是一个x86硬件平台的开源模拟器。它可以模拟各种硬件的配置。Bochs模拟的是整个PC平台，包括I/O设备、内存和BIOS。更为有趣的是，甚至可以不使用PC硬件来运行Bochs。事实上，它可以在任何编译运行Bochs的平台上模拟x86硬件。通过改变配置，可以指定使用的CPU(386、486或者586)，以及内存大小等。一句话，Bochs是电脑里的"PC"。根据需要，Bochs还可以模拟多台PC，此外，它甚至还有自己的电源按钮。

## 7.2 数据结构及代码段分析

### 文件夹a:

实现简单的I/O，从键盘输入字符的中断开始

文件keyboard.c

主要代码结构：

PUBLIC void keyboard\_handler(int irq)// 键盘中断处理程序，每按一次键，键盘输出字符"\*"

{

disp\_str("\*");

}

PUBLIC void init\_keyboard()

{

put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler);/\*设定键盘中断处理程序\*/

enable\_irq(KEYBOARD\_IRQ); /\*打开键盘中断\*/

}

### 文件夹b:

敲击键盘时获取并打印扫描码

文件keyboard.c

主要代码结构：

PUBLIC void keyboard\_handler(int irq) //键盘中断处理程序，将读取的扫描码打印出来

{

u8 scan\_code = in\_byte(0x60);

disp\_int(scan\_code);

}

PUBLIC void init\_keyboard()

{

put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler);/\*设定键盘中断处理程序\*/

enable\_irq(KEYBOARD\_IRQ); /\*开键盘中断\*/

}

### 文件夹d:

创建对应打印扫描码解析数组，用数组表示扫描码和键盘输入缓冲区，在敲击键盘时可以打印对应字符。

在代码keyboard.c中，总体的思想就是0xE0和oxE1单独处理,因为从扫描符表中知道,除去以这两个数字开头的扫描码，其余的都是单字节的。

我们暂时对oxEo和oxE1不加理会。如果遇到不是以它们开头的,则判断是Make Code还是Break Code,如果是后者同样不加理会，如果是前者就打印出来。Break Code是Make Code与ox80进行“或(OR)”操作的结果，代码中的FLAG BREAK就是被定义成了0x80。

从keymap[]中取出字符的时候进行了一个“与”操作（scan code&0x7F)。一方面，如果当前扫描码是Break Code,“与”操作之后就变成Make Code了;另一方面，这样做也是为了避免越界的发生，因为数组keymap[]的大小是0x80。

文件keymap.h

建立一个数组，以扫描码为下标，对应的元素就是相应的字符

文件keyboard.h

主要数据结构——缓冲区KB\_INPUT

|  |  |
| --- | --- |
| 缓冲区KB\_INPUT | p\_head 指向缓冲区中下一个空闲位置 |
| p\_tail 指向键盘任务应处理的字节 |
| Count 缓冲区中字节数 |
| buf[KB\_IN\_BYTES] 缓冲区 |

文件keyboard.c

主要代码结构

PUBLIC void keyboard\_handler(int irq) //对缓冲区进行添加操作，如果缓冲区已满，则直接把收到的字节丢弃

{

u8 scan\_code = in\_byte(KB\_DATA);

if (kb\_in.count < KB\_IN\_BYTES) {

\*(kb\_in.p\_head) = scan\_code;

kb\_in.p\_head++;

if (kb\_in.p\_head == kb\_in.buf + KB\_IN\_BYTES) {

kb\_in.p\_head = kb\_in.buf;

}

kb\_in.count++;

}

}

PUBLIC void init\_keyboard()

{

//初始化kb\_in的成员

kb\_in.count = 0;

kb\_in.p\_head = kb\_in.p\_tail = kb\_in.buf;

put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler);/\*设定键盘中断处理程序\*/

enable\_irq(KEYBOARD\_IRQ); /\*开键盘中断\*/

}

PUBLIC void keyboard\_read()

{

u8 scan\_code;

char output[2];

int make; // TRUE: make; FALSE: break.

memset(output, 0, 2);

/\*判断kb\_in.count是否为0。如果不为0，表示缓冲区有扫描码，则开始读。读缓冲区开始时关闭中断，到结束时打开。\*/

if(kb\_in.count > 0){

disable\_int();

scan\_code = \*(kb\_in.p\_tail);

kb\_in.p\_tail++;

if (kb\_in.p\_tail == kb\_in.buf + KB\_IN\_BYTES) {

kb\_in.p\_tail = kb\_in.buf;

}

kb\_in.count--;

enable\_int();

//下面开始解析扫描码

if (scan\_code == 0xE1) {

/\* 暂时不做任何操作 \*/

}

else if (scan\_code == 0xE0) {

/\* 暂时不做任何操作 \*/

}

else { //下面处理可打印字符

//首先判断Make Code 还是 Break Code

make = (scan\_code & FLAG\_BREAK ? FALSE : TRUE);

//如果是Make Code 就打印，是 Break Code 则不做处理

if(make) {

output[0] = keymap[(scan\_code&0x7F)\*MAP\_COLS];

disp\_str(output);

}

}

}

}

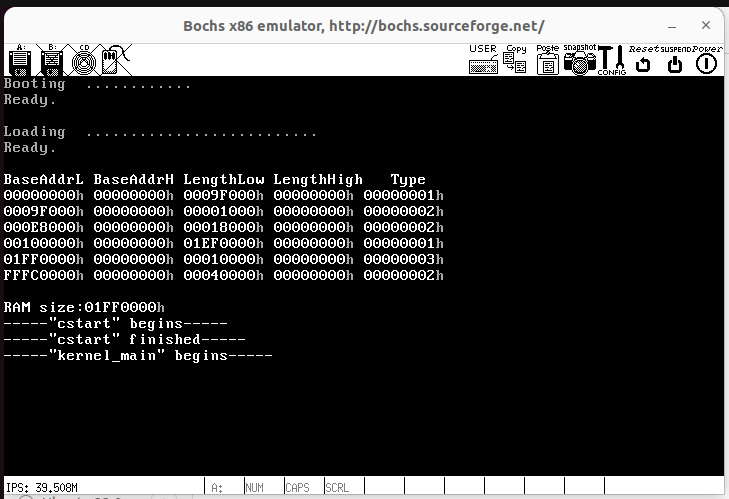
## 7.3 调试过程

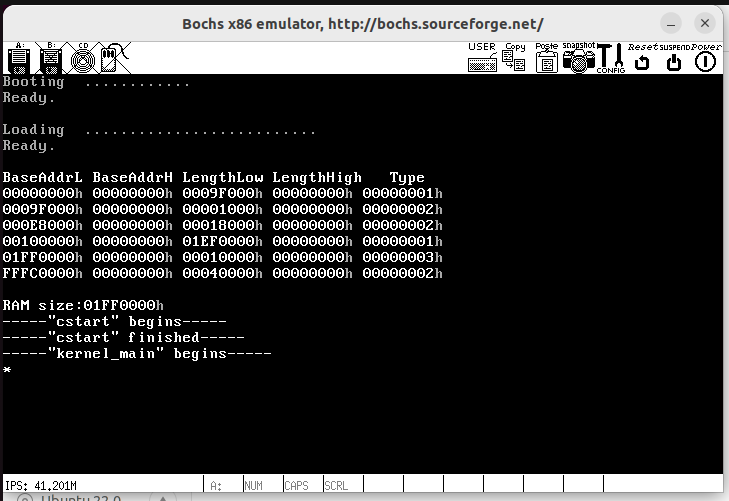
abd三个文件夹的实验操作都是：1. make image 2. Bochs

实验成果截图如下：

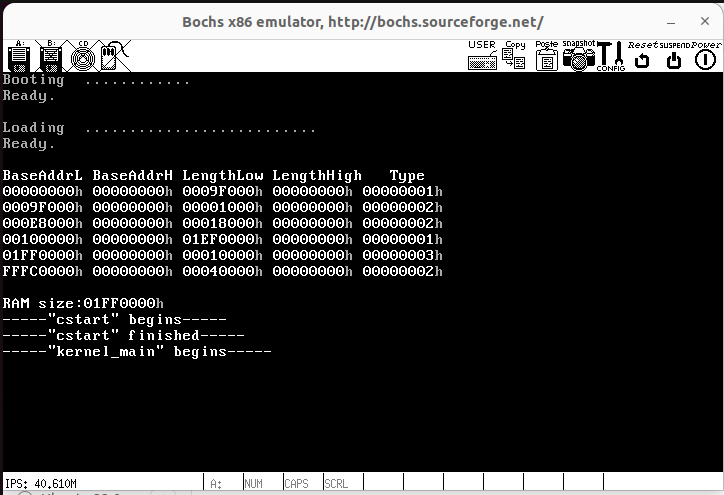
### 文件夹a:

实现简单的I/O，从键盘输入字符的中断开始

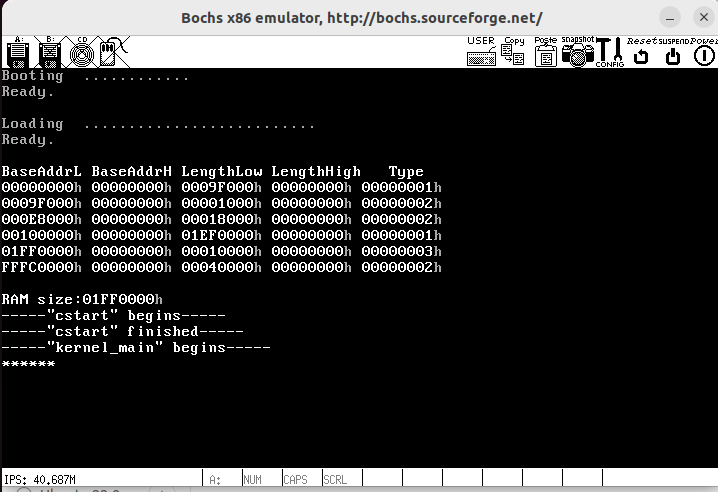
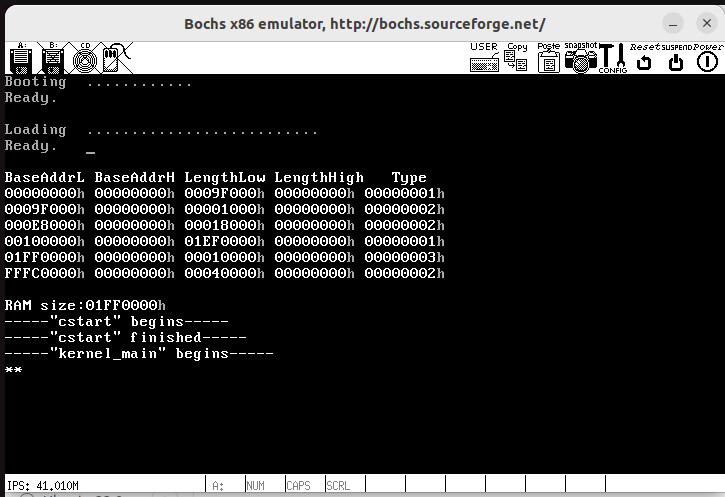
刚开始模拟未进行任何操作时：

当敲击任意键时会出现一个\*号并卡住

在课本上看到，如果在kernel中的keyboard.c中把第21行的注释符号删去，那么每次敲击键盘后都会出现两颗星星，因此我们删去注释符号后重新进行实验。

此时未操作

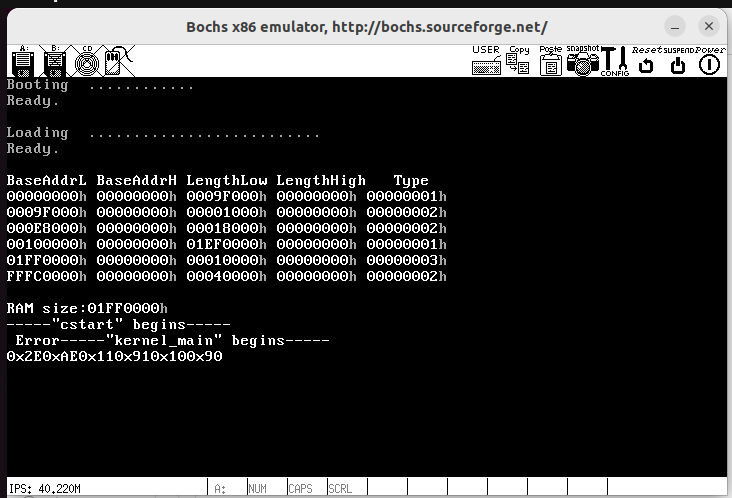
敲击一次键盘后

敲击三次键盘后

### 文件夹b:

敲击键盘时获取并打印扫描码

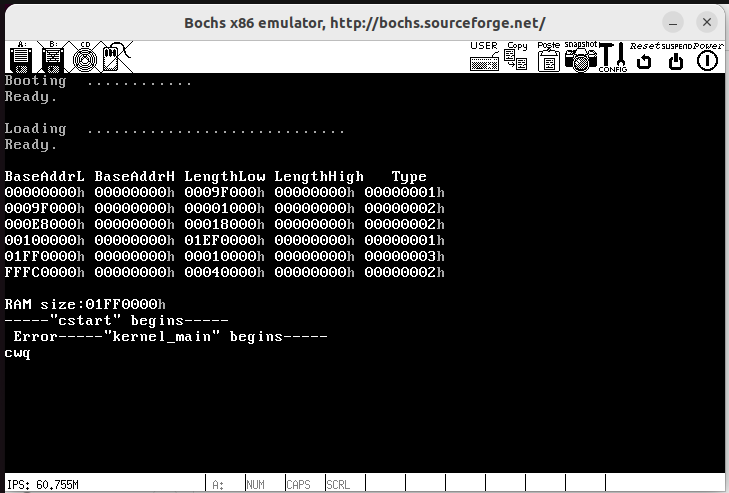
输入名字简写cwq得到以下扫描码



### 文件夹d:

创建对应打印扫描码解析数组，用数组表示扫描码和键盘输入缓冲区，在敲击键盘时可以打印对应字符。

输入名字简写cwq得到以下截图



## 7.4 实验总结

本次实验研究的是输入与输出。经过这次实验，我对电脑如何将外界输入处理成人类对应的语言有了更深的了解。在此之前我从未思考过这些东西，因为这些东西都已经成为习惯了，我没有想着去探究其深层次的原理。当我得知我们敲击键盘时按下和拿起分别会有两个扫描码时，我一下子就理解了快捷键是怎么运行的，这极大地勾起了我的兴趣，因此这个章节我也往后多看了几章，也清楚了文件夹d里面keyboard.c的keyboard\_read()函数剩下的空白部分是在为完善输入输出功能而留空，以便可以进行一些特殊符号的输入输出。

8

## 7.1 实验内容及上机实验所用平台

### 7．1．1 实验内容

**实现简单的I/O，从键盘输入字符的中断开始；敲击键盘时获取并打印扫描码；创建对应打印扫描码解析数组，在敲击键盘时可以打印对应字符。**

### 7．1．2 实验所用平台

（1）VMware

VMWare虚拟机软件是一个“虚拟PC”软件，它使你可以在一台机器上同时运行二个或更多Windows、DOS、LINUX系统。与“多启动”系统相比，VMware采用了完全不同的概念。多启动系统在一个时刻只能运行一个系统，在系统切换时需要重新启动机器。VMWare是真正“同时”运行，多个操作系统在主系统的平台上，就像标准Windows应用程序那样切换。而且每个操作系统你都可以进行虚拟的分区、配置而不影响真实硬盘的数据，可以通过网卡将几台虚拟机用网卡连接为一个局域网。

它主要具有以下功能：

1.不需要分区或重开机就能在同一台PC上使用两种以上的操作系统。

2.完全隔离并且保护不同OS的操作环境以及所有安装在OS上面的应用软件和资料。

3.不同的OS之间还能互动操作，包括网络、周边、文件分享以及复制贴上功能。

4.有复原（Undo）功能。

5.能够设定并且随时修改操作系统的操作环境，如：内存、磁碟空间、周边设备等等。

（2）Ubuntu

Ubuntu是一个以桌面应用为主的[Linux](https://baike.baidu.com/item/Linux/27050" \t "https://baike.baidu.com/item/ubuntu/_blank)操作系统。.Ubuntu基于Debian发行版和[GNOME](https://baike.so.com/doc/5453894-5692279.html" \t "https://baike.so.com/doc/_blank)桌面环境，而从11.04版起，Ubuntu发行版放弃了Gnome桌面环境，改为Unity，与Debian的不同在于它每6个月会发布一个新版本。Ubuntu的目标在于为一般用户提供一个最新的、同时又相当稳定的主要由[自由软件](https://baike.so.com/doc/5925549-6138471.html" \t "https://baike.so.com/doc/_blank)构建而成的操作系统。Ubuntu具有庞大的社区力量，用户可以方便地从社区获得帮助。2013年1月3日，Ubuntu正式发布面向智能手机的移动操作系统。

（3）Bochs

Bochs是一个x86硬件平台的开源模拟器。它可以模拟各种硬件的配置。Bochs模拟的是整个PC平台，包括I/O设备、内存和BIOS。更为有趣的是，甚至可以不使用PC硬件来运行Bochs。事实上，它可以在任何编译运行Bochs的平台上模拟x86硬件。通过改变配置，可以指定使用的CPU(386、486或者586)，以及内存大小等。一句话，Bochs是电脑里的"PC"。根据需要，Bochs还可以模拟多台PC，此外，它甚至还有自己的电源按钮。

## 7.2 数据结构及代码段分析

### 文件夹a:

实现简单的I/O，从键盘输入字符的中断开始

文件keyboard.c

主要代码结构：

PUBLIC void keyboard\_handler(int irq)// 键盘中断处理程序，每按一次键，键盘输出字符"\*"

{

disp\_str("\*");

}

PUBLIC void init\_keyboard()

{

put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler);/\*设定键盘中断处理程序\*/

enable\_irq(KEYBOARD\_IRQ); /\*打开键盘中断\*/

}

### 文件夹b:

敲击键盘时获取并打印扫描码

文件keyboard.c

主要代码结构：

PUBLIC void keyboard\_handler(int irq) //键盘中断处理程序，将读取的扫描码打印出来

{

u8 scan\_code = in\_byte(0x60);

disp\_int(scan\_code);

}

PUBLIC void init\_keyboard()

{

put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler);/\*设定键盘中断处理程序\*/

enable\_irq(KEYBOARD\_IRQ); /\*开键盘中断\*/

}

### 文件夹d:

创建对应打印扫描码解析数组，用数组表示扫描码和键盘输入缓冲区，在敲击键盘时可以打印对应字符。

在代码keyboard.c中，总体的思想就是0xE0和oxE1单独处理,因为从扫描符表中知道,除去以这两个数字开头的扫描码，其余的都是单字节的。

我们暂时对oxEo和oxE1不加理会。如果遇到不是以它们开头的,则判断是Make Code还是Break Code,如果是后者同样不加理会，如果是前者就打印出来。Break Code是Make Code与ox80进行“或(OR)”操作的结果，代码中的FLAG BREAK就是被定义成了0x80。

从keymap[]中取出字符的时候进行了一个“与”操作（scan code&0x7F)。一方面，如果当前扫描码是Break Code,“与”操作之后就变成Make Code了;另一方面，这样做也是为了避免越界的发生，因为数组keymap[]的大小是0x80。

文件keymap.h

建立一个数组，以扫描码为下标，对应的元素就是相应的字符

文件keyboard.h

主要数据结构——缓冲区KB\_INPUT

|  |  |
| --- | --- |
| 缓冲区KB\_INPUT | p\_head 指向缓冲区中下一个空闲位置 |
| p\_tail 指向键盘任务应处理的字节 |
| Count 缓冲区中字节数 |
| buf[KB\_IN\_BYTES] 缓冲区 |

文件keyboard.c

主要代码结构

PUBLIC void keyboard\_handler(int irq) //对缓冲区进行添加操作，如果缓冲区已满，则直接把收到的字节丢弃

{

u8 scan\_code = in\_byte(KB\_DATA);

if (kb\_in.count < KB\_IN\_BYTES) {

\*(kb\_in.p\_head) = scan\_code;

kb\_in.p\_head++;

if (kb\_in.p\_head == kb\_in.buf + KB\_IN\_BYTES) {

kb\_in.p\_head = kb\_in.buf;

}

kb\_in.count++;

}

}

PUBLIC void init\_keyboard()

{

//初始化kb\_in的成员

kb\_in.count = 0;

kb\_in.p\_head = kb\_in.p\_tail = kb\_in.buf;

put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler);/\*设定键盘中断处理程序\*/

enable\_irq(KEYBOARD\_IRQ); /\*开键盘中断\*/

}

PUBLIC void keyboard\_read()

{

u8 scan\_code;

char output[2];

int make; // TRUE: make; FALSE: break.

memset(output, 0, 2);

/\*判断kb\_in.count是否为0。如果不为0，表示缓冲区有扫描码，则开始读。读缓冲区开始时关闭中断，到结束时打开。\*/

if(kb\_in.count > 0){

disable\_int();

scan\_code = \*(kb\_in.p\_tail);

kb\_in.p\_tail++;

if (kb\_in.p\_tail == kb\_in.buf + KB\_IN\_BYTES) {

kb\_in.p\_tail = kb\_in.buf;

}

kb\_in.count--;

enable\_int();

//下面开始解析扫描码

if (scan\_code == 0xE1) {

/\* 暂时不做任何操作 \*/

}

else if (scan\_code == 0xE0) {

/\* 暂时不做任何操作 \*/

}

else { //下面处理可打印字符

//首先判断Make Code 还是 Break Code

make = (scan\_code & FLAG\_BREAK ? FALSE : TRUE);

//如果是Make Code 就打印，是 Break Code 则不做处理

if(make) {

output[0] = keymap[(scan\_code&0x7F)\*MAP\_COLS];

disp\_str(output);

}

}

}

}

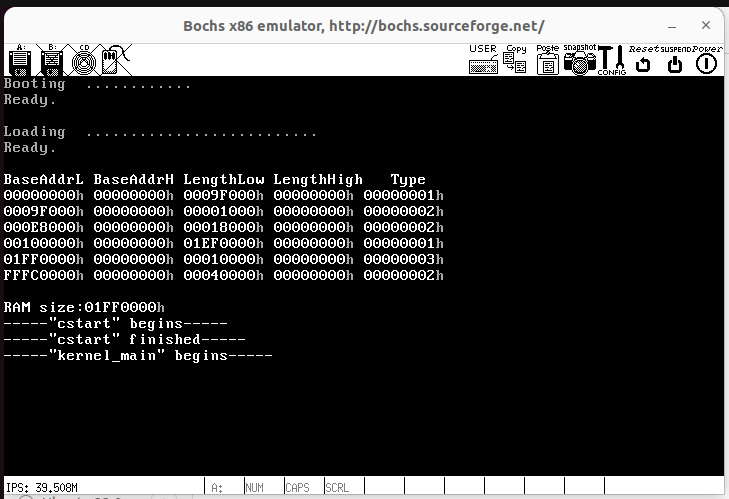
## 7.3 调试过程

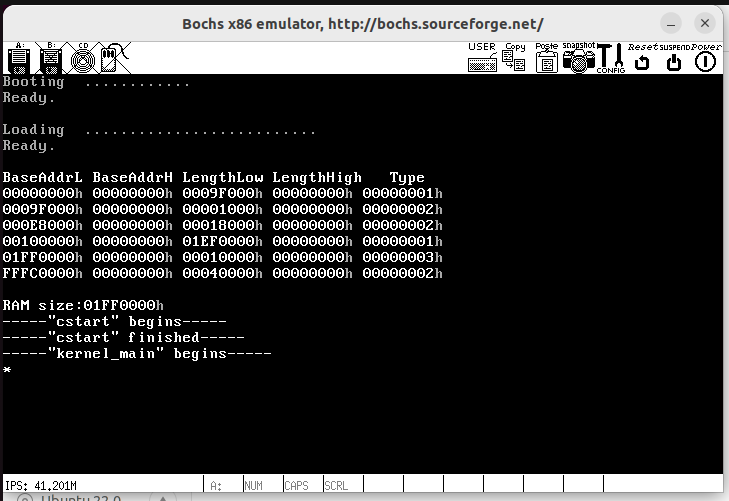
abd三个文件夹的实验操作都是：1. make image 2. Bochs

实验成果截图如下：

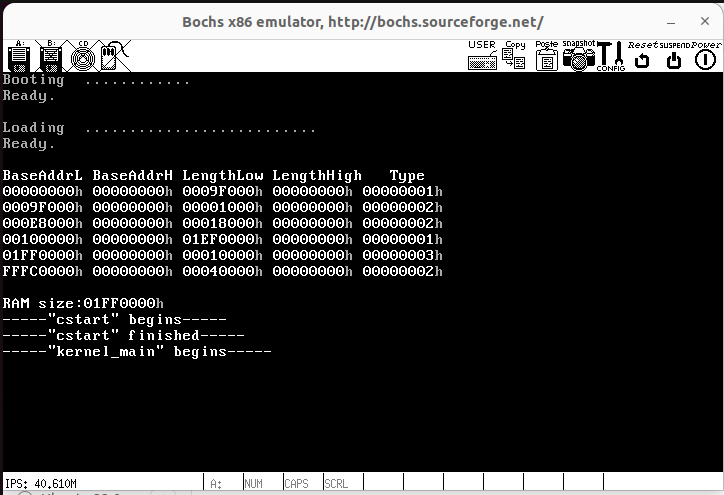
### 文件夹a:

实现简单的I/O，从键盘输入字符的中断开始

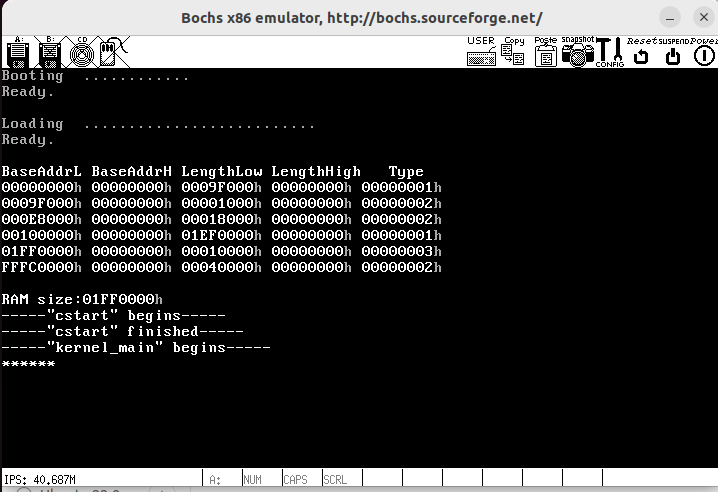
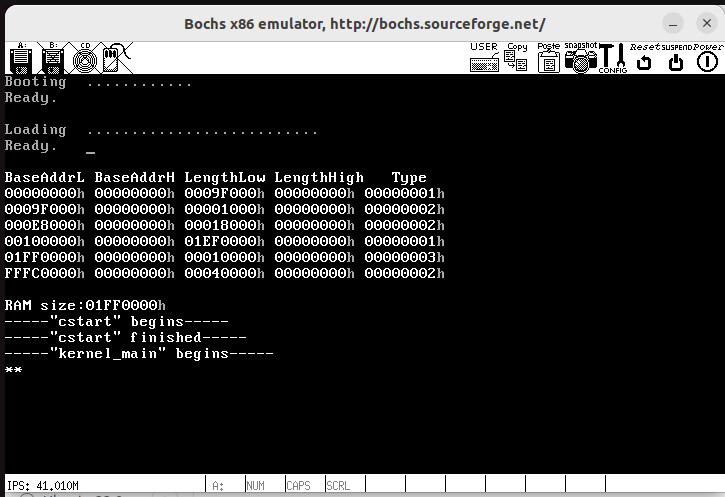
刚开始模拟未进行任何操作时：

当敲击任意键时会出现一个\*号并卡住

在课本上看到，如果在kernel中的keyboard.c中把第21行的注释符号删去，那么每次敲击键盘后都会出现两颗星星，因此我们删去注释符号后重新进行实验。

此时未操作

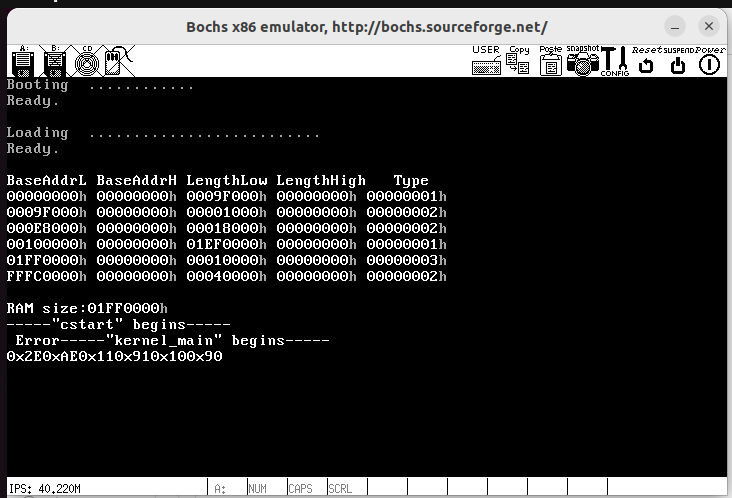
敲击一次键盘后

敲击三次键盘后

### 文件夹b:

敲击键盘时获取并打印扫描码

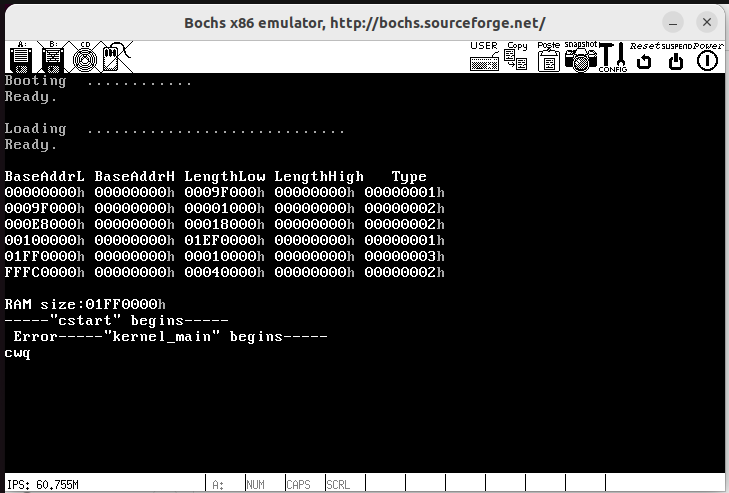
输入名字简写cwq得到以下扫描码



### 文件夹d:

创建对应打印扫描码解析数组，用数组表示扫描码和键盘输入缓冲区，在敲击键盘时可以打印对应字符。

输入名字简写cwq得到以下截图



## 7.4 实验总结

本次实验研究的是输入与输出。经过这次实验，我对电脑如何将外界输入处理成人类对应的语言有了更深的了解。在此之前我从未思考过这些东西，因为这些东西都已经成为习惯了，我没有想着去探究其深层次的原理。当我得知我们敲击键盘时按下和拿起分别会有两个扫描码时，我一下子就理解了快捷键是怎么运行的，这极大地勾起了我的兴趣，因此这个章节我也往后多看了几章，也清楚了文件夹d里面keyboard.c的keyboard\_read()函数剩下的空白部分是在为完善输入输出功能而留空，以便可以进行一些特殊符号的输入输出。

# 心得体会与感悟

在本次操作系统复现实验中，我参考老师发的电子课本《一个操作系统的实现》，借助上学期学习的操作系统原理的知识，将理论知识与实践相结合做出了一个简陋的操作系统，这个操作系统可以粗略地做到一个操作系统应该做到的功能。从搭建虚拟机到搭建工作环境，然后一步步实现保护模式和实模式的转换，切换到保护模式之后实加载Loader进入内存并将控制权交给loader，随后将loader替换成kernel内核，实现进程切换、保存与恢复、调度，同时解决可能会出现的中断重入问题，最后实现I/O系统，一个由我们自己设计的简单的操作系统就此完成。

当我复现这个简陋的操作系统的时候，我通过每一次实验之间代码的不同以及最后呈现的效果的不同，对不同代码的功能有了更清晰明确的理解，也对汇编语言有了初步的认识。不得不说这本书在引领初学者入门方面是很优秀的，但是还是有一些小瑕疵，比如说里面的代码都是对应的32位的虚拟机但是我下载使用的是64位的，有些Linux的指令由于版本的原因可能需要更改，以及作者考虑到了读者可能没有操作系统方面的理论知识但是没有考虑到读者也可能没有汇编语言方面的基础，不过整体来说这本书瑕不掩瑜，还是值得我们作为初学者去入门的。当然，除了这本书，网上很多前辈的回答以及学长的经验也给予了我很大的帮助，我也因此清楚地感受到了善于利用身边的资源快速查阅符合版本的相关资料的重要性。

在本次操作系统复现实验中，我参考老师发的电子课本《一个操作系统的实现》，借助上学期学习的操作系统原理的知识，将理论知识与实践相结合做出了一个简陋的操作系统。从搭建虚拟机到搭建工作环境，然后一步步实现保护模式和实模式的转换，切换到保护模式之后实加载Loader进入内存并将控制权交给loader，随后将loader替换成kernel内核，实现进程切换、保存与恢复、调度，同时解决可能会出现的中断重入问题，最后实现I/O系统，一个由我们自己设计的简单的操作系统就此完成。

当我复现这个简陋的操作系统的时候，我通过每一次实验之间代码的不同以及最后呈现的效果的不同，对不同代码的功能有了更清晰明确的理解，也对汇编语言有了初步的认识。经过两周的学习，我认为《ORANGE’S：一个操作系统的实现》这本书在引领初学者入门方面是很优秀的，但是还是有一些小瑕疵，比如说里面的代码都是对应的32位的虚拟机但是我下载使用的是64位的，有些Linux的指令由于版本的原因可能需要更改，以及作者考虑到了读者可能没有操作系统

方面的理论知识但是没有考虑到读者也可能没有汇编语言方面的基础，不过整体来说这本书瑕不掩瑜，还是值得我们作为初学者去入门的。

通过参与这次操作系统复现实验，我获得了对CPU硬件的初步了解。实现一个操作系统不可避免地需要与硬件进行交互，而这个实验让我对一些关键芯片有了初步的了解。其中，我对中断控制芯片8259A、时钟控制芯片8253以及键盘控制芯片8042有了更深入的认识。了解这些芯片的功能和工作原理，让我意识到操作系统是如何与硬件进行通信和控制的。除了对硬件的认识，这个实验还深化了我对操作系统的理解。我对操作系统的多进程管理、输入输出处理、多控制台支持以及分页机制等方面有了更加深刻和直观的认识。通过亲自实践，我体会到了操作系统是如何协调和管理多个进程的执行，如何处理输入输出请求以及如何实现多个控制台的切换和管理。此外，我也了解到了分页机制在操作系统中的重要性，它能够有效地管理内存资源，提高系统的性能和安全性。这个实验还让我初步接触了x86汇编语言。通过阅读源码并亲自实践，我逐渐掌握了x86汇编语言的基本使用。我学会了如何编写汇编指令来实现特定的功能，如何与其他编程语言进行交互，并且开始理解汇编语言与底层硬件之间的密切关系。这为我进一步深入学习和应用汇编语言打下了坚实的基础。此外，这个实验还让我对虚拟机和Linux操作系统有了初步的认识。在实验过程中，我尝试了使用Linux操作系统，并学习了一些基本的常用指令。通过亲身体验，我逐渐熟悉了Linux操作系统的特点和使用方法，了解了虚拟机在操作系统开发和测试中的重要作用。这为我今后深入学习虚拟化技术和探索更多操作系统的可能性奠定了基础。

同时，我非常感谢老师对我的指导和帮助，没有他的支持和鼓励，我无法完成这个实验。通过这次实验，我不仅掌握了操作系统的基本原理和实践技能，还培养了解决问题的能力和动手实践的精神。这对我的职业发展和学术研究都具有重要意义。我将继续深入学习操作系统的知识，并将其应用到未来的项目和研究中。

**参考文献**

**1.《Orange'S：一个操作系统的实现》。**

**2.CSDN网址：**[**http://blog.csdn.net/zgh1988**](http://blog.csdn.net/zgh1988)

**3. 知乎网址：如何从零开始写一个简单操作系统**

[**https://www.zhihu.com/question/25628124**](https://www.zhihu.com/question/25628124)

**4.《操作系统真相还原》，人民邮电出版社，郑钢，2016**

**5. MIT的操作系统设计课程，网址：**

[**https://pdos.csail.mit.edu/6.828/2017/xv6.html**](https://pdos.csail.mit.edu/6.828/2017/xv6.html)

**6. 《Linux内核完全注释》，赵炯，机械工业出版社， 2007。**

**7. 《30天自制操作系统》，川合秀实，人民邮电出版社，2012。**

**8. 《一个64位操作系统的设计与实现》，田宇，人民邮电出版社，2018年。**

**9. 清华大学操作系统设计课程网址：**

[**https://chyyuu.gitbooks.io/ucore\_os\_docs/content/**](https://chyyuu.gitbooks.io/ucore_os_docs/content/)

# 教师评语评分

评语：

评分：

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分制评分。）