本科生实验报告

实验课程：操作系统实验

实验名称：lab2 实验入门

专业名称：计算机科学与技术

学生姓名：杜翊菲

学生学号：23336062

实验地点：实验大楼 B201

实验成绩：

报告时间：2025 年 3 月 21 日

Lab1 实验入门

一、实验要求

1. 学习 x86 汇编、计算机的启动过程、IA-32 处理器架构和字符显存原理

2. 自己编写程序，并且让计算机在启动后加载运行，增进对计算机启动过程的理解，为后面编写操作系统加载程序奠定基础

3. gdb 调试程序的基本方法

二、实验过程（每个部分包括实验操作、关键代码和结果截图）

（一）MBR

1. 根据 Example1 教程，复现 Example 1

现在，我们来正式编写MBR的代码，在MBR被加载到内存地址0x7c00后，向屏幕输出蓝色的Hello World，代码如下所示。

```
org 0x7c00
[bits 16]
xor ax, ax ; eax = 0
; 初始化段寄存器，段地址全部设为0
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

; 初始化栈指针
mov sp, 0x7c00
mov ax, 0xb800
```

```
mov gs, ax


mov ah, 0x01 ;蓝色
mov al, 'H'
mov [gs:2 * 0], ax

mov al, 'e'
mov [gs:2 * 1], ax

mov al, 'l'
mov [gs:2 * 2], ax

mov al, 'l'
mov [gs:2 * 3], ax
```

```
mov al, 'o'
mov [gs:2 * 4], ax

mov al, ' '
mov [gs:2 * 5], ax

mov al, 'W'
mov [gs:2 * 6], ax

mov al, 'o'
mov [gs:2 * 7], ax

mov al, 'r'
mov [gs:2 * 8], ax

mov al, 'l'
mov [gs:2 * 9], ax
```

```
mov al, 'd'
mov [gs:2 * 10], ax

jmp $ ; 死循环

times 510 - ($ - $$) db 0
db 0x55, 0xaa
```

第1、2行的 `org 0x7c00` 和 `[bits 16]` 是汇编伪指令，不是实际的指令。 `org 0x7c00` 是告诉编译器代码中的代码标号和数据标号从 `0x7c00` 开始。也就是说，这些标号的地址会在编译时加上 `0x7c00` 。如果没有这一句，标号的值就默认是标号从代码开始处的偏移地址。此时，如果我们引用标号就会出错。 `[bits 16]` 告诉编译器按16位代码格式编译代码。

指令实际上是从第3行的 `xor ax, ax` 开始执行。我们先将ax置为0，然后借助于ax将段寄存器清0。由于汇编不允许使用立即数直接对段寄存器赋值，所以我们需要借助于ax。

段寄存器初始化后，我们开始对显存地址赋值。由于显存地址是从 `0xB8000` 开始，而16位的段寄存器最大可表示 `0xFFFF` ，因此我们需要借助于段寄存器来寻址到 `0xB8000` 处的地址。于是我们将段寄存器 `gs` 的值赋值为 `0xB800` 。注意我们赋值的是 `0xB800` 而不是 `0xB8000` ，同学们可以自行思考下原因。

然后我们依次对显存地址赋值来实现在显示屏上输出 `Hello World` 。根据显存的显示原理，一个字符使用两个字节表示。因此，我们将ax的高字节部份ah赋值为颜色属性0x01，低字节部份赋值为对应的字符，然后依次放置到显存地址的对应位置。我们在对显存地址赋值时指定了段寄存器gs，因此CPU不会使用默认的段寄存器来计算物理地址。例如，我们想在第0行第1列输出蓝色字符e。

```
mov al, 'e'
mov [gs:2 * 1], ax
```

此时的物理地址的计算过程如下。

$$物理地址 = gs << 4 + 2 * 1 = 0xB800 << 4 + 2 * 1 = 0xB8002$$

恰好是对应的显存地址。依次输出字符后，我们还没有实现下一步的工作，即bootloader加载内核。因此这里就在做死循环。代码的最后的 `times` 指令是汇编伪指令，表示重复执行指令若干次。 `$` 表示当前汇编地址， `$$` 表示代码开始的汇编地址。 `times 510 - ($ - $$) db 0` 表示填充字符0直到第510个字节。最后我们填充0x55，0xaa表示MBR是可启动的。

写完代码后我们使用nasm汇编器来将代码编译成二进制文件。

```
nasm -f bin mbr.asm -o mbr.bin
```

其中， `-f` 参数指定的是输出的文件格式， `-o` 指定的是输出的文件名。 `mbr.bin` 中保存的是机器可以识别的机器指令。同学们可以使用命令 `xxd` 查看其中的内容。

生成了MBR后，我们将其写入到硬盘的首扇区。我们首先创建一个“硬盘”，这个“硬盘”并不是一个真实的硬盘，实际上是一个预先指定大小的文件而已，又被称为“虚拟磁盘”。硬盘的创建使用的是 `qemu-img` ，如下所示。

```
qemu-img create filename [size]
```

`[]` 表示这一项是可选项，也就是说可以不写； `filename` 是生成的硬盘的文件名。我们创建一个10m的磁盘 `hd.img` 。

```
qemu-img create hd.img 10m
```

然后将MBR写入 `hd.img` 的首扇区，写入的命令使用的是linux下的 `dd` 命令。

```
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

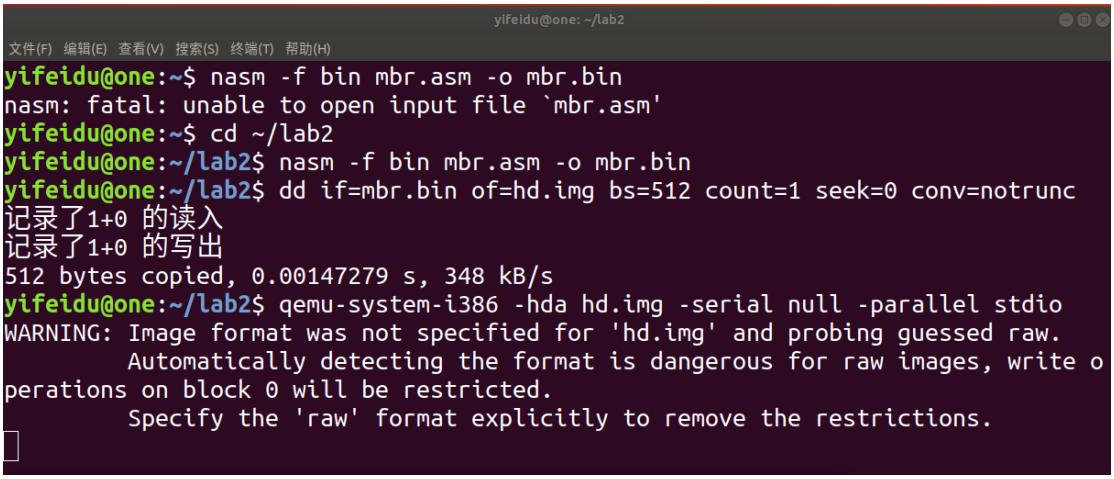
参数的解释如下。

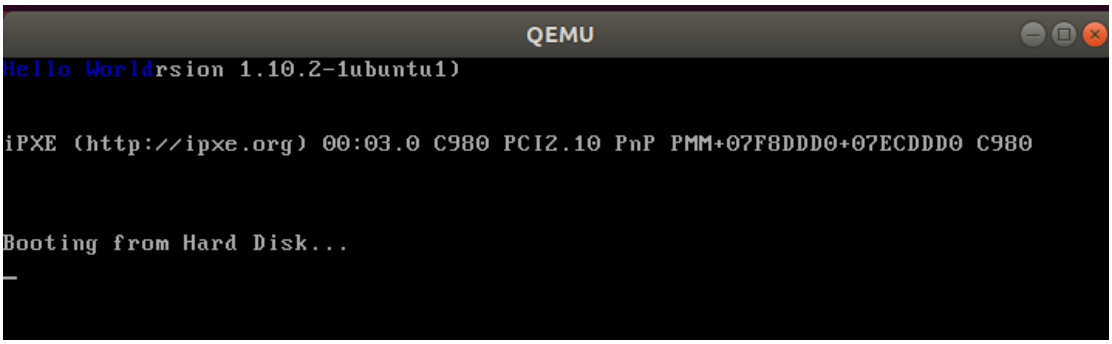
- `if` 表示输入文件。
- `of` 表示输出文件。
- `bs` 表示块大小，以字节表示。
- `count` 表示写入的块数目。
- `seek` 表示越过输出文件中多少块之后再写入。
- `conv=notrunc` 表示不截断输出文件，如果不加上这个参数，那么硬盘在写入后多余部份会被截断。

写入MBR后我们就可以启动qemu来模拟计算机启动了，命令如下。

```
qemu-system-i386 -hda hd.img -serial null -parallel stdio
```

按如上教程操作后，运行截图如下：





在(0,0)成功输出了蓝色的HelloWorld

2. 修改Example 1的代码，使得MBR被加载到0x7C00后在(12,12)处开始输出你的学号。注意，你的学号显示的前景色和背景色必须和教程中不同。

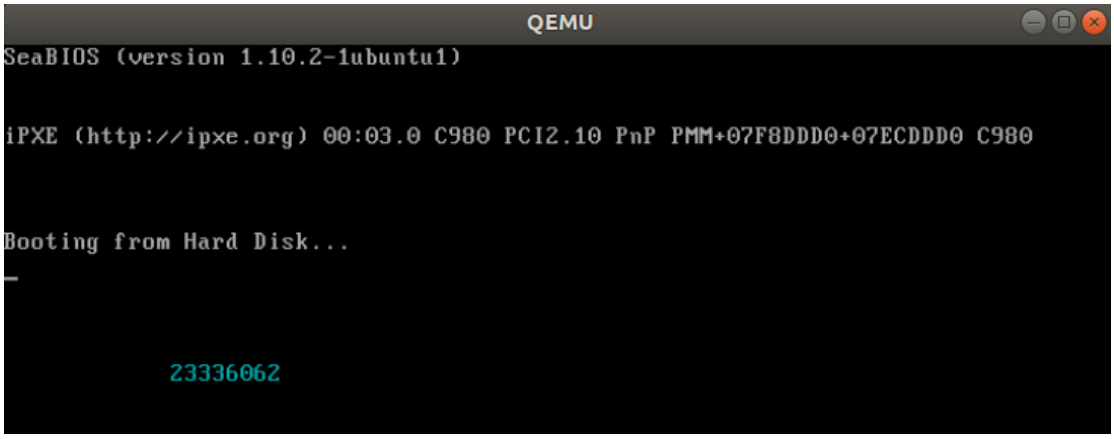
这里我们直接在1.1的基础上修改代码。容易发现1.2和1.1有三个不同需要修改的地方：①修改输出位置至(12,12)；②从输出“HelloWorld”转为输出我自己的学号“23336062”；③修改显示的前景色和背景色

（1）关键代码：

```
; 初始化栈指针
mov sp, 0x7c00
mov ax, 0xb879
mov gs, ax


mov ah, 0x03 ;青色
mov al, '2'
mov [gs:2 * 4], ax

mov al, '3'
mov [gs:2 * 5], ax
```

计算得到(12,12)的显存地址：0xb8000+(12×80+12)×2=0xb8798，所以修改了段地址；并将颜色从蓝色修改为青色（0x03）；最后修改了字符内容，改成了我的学号。

（2）代码成功修改后的运行截图如下：



成功在(12,12)输出了**青色**的学号 23336062

（二）实模式中断

1. 请探索实模式下的光标中断 int 10h，实现将光标移动至(8,8)，获取并输出光标的位置

2.1和2.2使用的都是实模式中断 `int 10h`，由于功能号不同，执行的结果也就不同。在 `int 10h` 中断的资料 中，其只给出10h中断下各个功能号的用途，并未给出实际的用法。因此，同学们可能一开始会感觉不知所云，教程下面给出同学们完成本次实验需要用到的功能号。

功能	功能号	参数	返回值
设置光标位置	AH=02H	BH=页码，DH=行，DL=列	无
获取光标位置和形状	AH=03H	BX=页码	AX=0，CH=行扫描开始，CL=行扫描结束，DH=行，DL=列
在当前光标位置写字符和属性	AH=09H	AL=字符，BH=页码，BL=颜色，CX=输出字符的个数	无

注意，“页码”均设置为0。

一般地，中断的调用方式如下。

```
将参数和功能号写入寄存器
int 中断号
从寄存器中取出返回值
```

（1）关键代码：

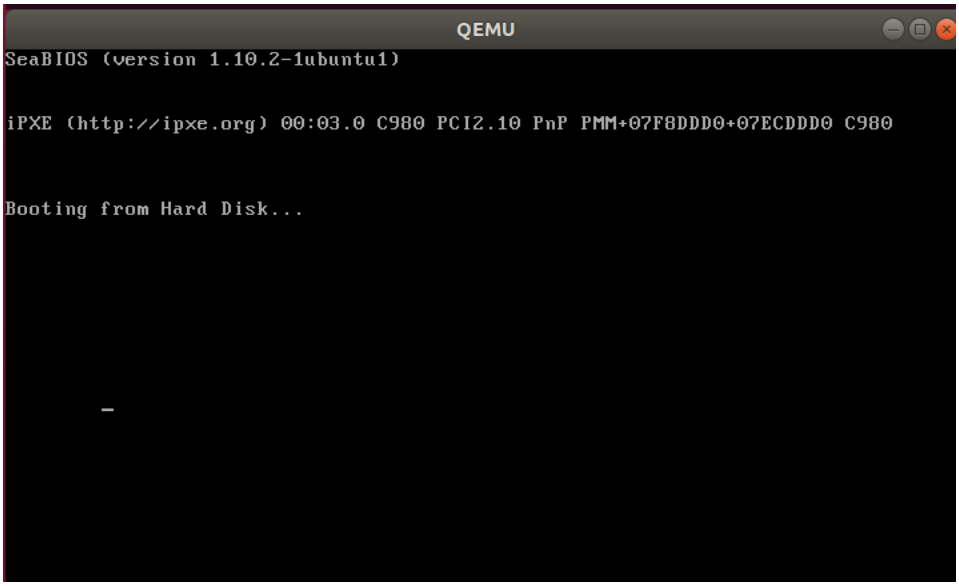
```
mov ah, 2 ; 功能码，光标移动
mov bh, 0 ; 页码，文本状态设为0
mov dh, 8
mov dl, 8
int 10h


mov ah, 3 ; 功能码，位置获取
mov bh, 0 ; 页码，文本状态设为0
int 10h
```

总体框架还是在原代码的大框架基础上修改和增添。题目要求利用中断，获取光标位置，并移动光标。可知移动光标用 0x02 功能码，获取位置用 0x03 功能码。

```
mov ah, 0x04

add dh, 48
add dl, 48

mov al,dh
mov[gs:2*(80*12+8)], ax

mov al,dl
mov[gs:2*(80*12+10)], ax
```

输出光标位置。

（2）运行结果截图：



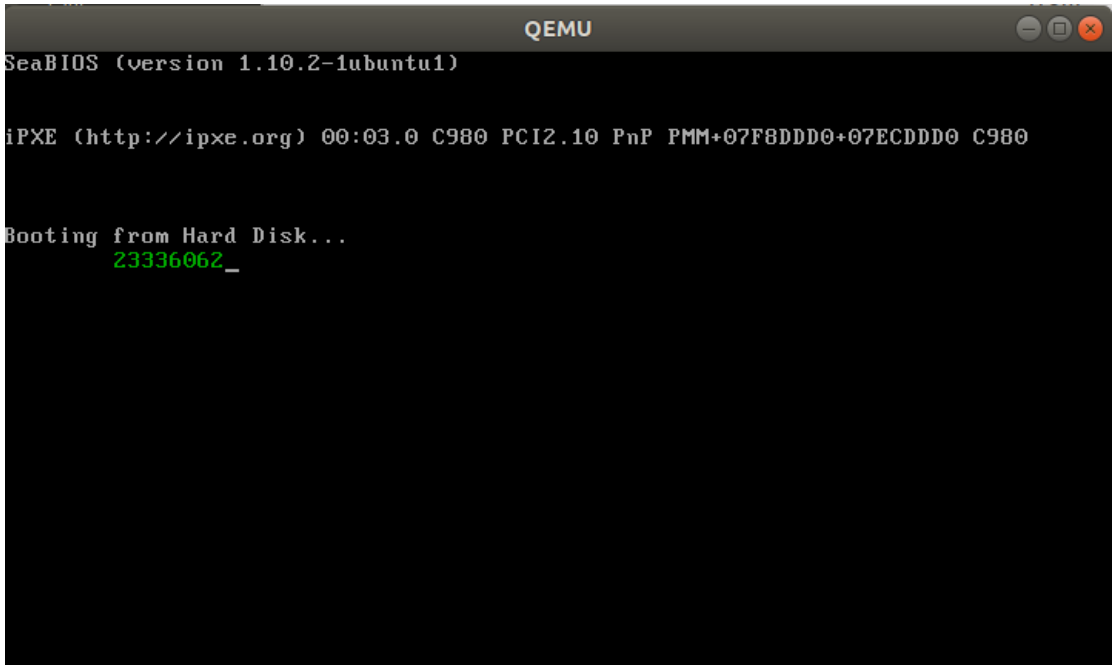
2. 利用实模式下的中断，从(8,8)开始输出你的学号

（1）关键代码：

设置光标位置后移，并且在该位置写字符和属性，用 0x09 功能实现。写字符和前面的代码大差不差。以其中一小段代码为例。

```
mov ah, 0x03 ; 读取光标位置
mov bh, 0x00 ; 文本状态设为0
int 0x10 ; interrupt

mov bl, 0x02 ; green
mov al, '2'
mov ah, 0x09 ; 在当前光标位置写字符和属性的中断功能
mov cx, 0x01 ; 输出字符的个数
int 0x10 ; interrupt

mov ah, 0x02 ; 调用设置光标位置的中断功能
mov bh, 0x00
add dl, 1 ;
int 0x10 ; interrupt
```

（2）运行结果截图：



在(8,8)输出了绿色的学号。

3. 探索实模式下的键盘中断 int 16h，利用键盘中断，实现任意键盘输入并回显的效果

int 16h 能够调用键盘的 I/O 中断，有三个功能，功能号有 0、1、2,，必须放在 ah。从键盘读入字符：00H、10H；设置重复率：03H；读取键盘状态 01H、11H；设置键盘点击：04H；读取键盘标志：02H，12H；字符及其扫描码进栈：05H。具体功能如下：

键盘I/O中断调用有三个功能，功能号为0, 1, 2，且必须把功能号放在AH中。
（1）0号功能调用 格式：MOV AH, 0
INT 16H
功能：从键盘读入字符送AL寄存器。执行时，等待键盘输入，一旦输入，字符的ASCII码放入AL中。若AL = 0，则AH为输入的扩展码。
（esc键的ascii码十进制表示为27，二进制表示为0001 1011，十六进制表示为1B） cmp al，1bh；可以判断刚才输入的字符是否是escape

（2）1号功能调用

格式：MOV AH, 01H

INT 16H

功能：用来查询键盘缓冲区，对键盘扫描但不等待，并设置ZF标志。若有按键操作（即键盘缓冲区不空），则ZF = 0，AL中存放的是输入的ASCII码，AH中存放输入字符的扩展码。若无键按下，则标志位ZF = 1。

（3）2号功能调用

格式： MOV AH, 02H

INT 16H

功能：检查键盘上各特殊功能键的状态。执行后，各种特殊功能键的状态放入AL寄存器中，

这个状态字记录在内存0040H：0017H单元中，若对应位为“1”，表示该键状态为“ON”，处于按下状态；若对应位为“0”，表示该键状态为“OFF”，处于断开状态。

键盘服务(Keyboard Service——INT 16H)

00H、10H —从键盘读入字符03H —设置重复率
01H、11H —读取键盘状态04H —设置键盘点击
02H, 12H —读取键盘标志05H —字符及其扫描码进栈
(1)、功能00H和10H
功能描述：从键盘读入字符
入口参数：
AH = 00H——读键盘
= 10H——读扩展键盘，可根据0000:0496H单元的内容判断：扩展键盘是否有效
出口参数：AH = 键盘的扫描码
AL = 字符的ASCII码
(2)、功能01H和11H
功能描述：读取键盘状态
入口参数：AH = 01H——检查普通键盘
= 11H——检查扩展键盘
出口参数：ZF = 1——无字符输入，否则，AH = 键盘的扫描码，AL = ASCII码。

(3)、功能02H和12H

功能描述：读取键盘标志

入口参数：AH = 02H——普通键盘的移位标志

= 12H——扩展键盘的移位标志

出口参数：AL = 键盘标志(02H和12H都有效)，其各位之值为1时的含义如下： 位7—INS开状态位3—ALT键按下

位6—CAPS LOCK开状态位2—CTRL键按下

位5—NUM LOCK开状态位1—左SHIFT键按下

位4—SCROLL LOCK开状态位0—右SHIFT键按下

AH = 扩展键盘的标志(12H有效)，其各位之值为1时的含义如下：

位7—SysReq键按下位3—右ALT键按下

位6—CAPS LOCK键按下位2—右CTRL键按下

位5—NUM LOCK键按下位1—左ALT键按下

位4—SCROLL键按下位0—左CTRL键按下

(4)、功能03H

功能描述：设置重复率

入口参数：AH = 03H 对于PC/AT和PS/2：AL = 05H

BH = 重复延迟

BL = 重复率

对于PCjr：AL = 00H——装入缺省的速率和延迟

= 01H——增加初始延迟

= 02H——重复频率降低一半

= 03H——增加延迟和降低一半重复频率

= 04H——关闭键盘重复功能

出口参数：无

(5)、功能04H

功能描述：设置键盘点击

入口参数：AH = 04H AL = 00H——关闭键盘点击功能

= 01H——打开键盘点击功能

出口参数：无

（1）关键代码

（2）运行截图

（三）汇编

1. 分支逻辑的实现

请将下列伪代码转换成汇编代码，并放置在标号 `your_if` 之后。

```
if a1 < 12 then
    if_flag = a1 / 2 + 1
else if a1 < 24 then
    if_flag = (24 - a1) * a1
else
    if_flag = a1 << 4
end
```

代码截图：

```
your_if:
; put your implementation here
    cmp eax, 12
    jl lt12 ; if a1 < 12 then
    cmp eax, 24
    jl lt24 ; else if a1 < 24 then
    shl eax, 4  ; a1 = a1 << 4
    mov [if_flag], eax
    jmp your_while
lt12:
    sar eax, 1 ; a1 /= 2
    inc eax ; a1 ++
    mov [if_flag], eax
    jmp your_while
lt24:
    mov ecx, eax
    sub ecx, 24
    neg ecx
    imul ecx, eax
    mov [if_flag], ecx
    jmp your_while
```

2. 循环逻辑的实现

请将下列伪代码转换成汇编代码，并放置在标号 `your_while` 之后。

```
while a2 >= 12 then
    call my_random        // my_random将产生一个随机数放到eax中返回
    while_flag[a2 - 12] = eax
    --a2
end
```

代码截图：

```
your_while:
; put your implementation here
    cmp byte[a2], 12
    jl loopend

    call my_random
    mov ebx, [a2]
    mov ecx, [while_flag]
    mov byte[ecx + ebx - 12] , al
    dec byte[a2]
    jmp your_while
```

3. 函数的实现

请编写函数 `your_function` 并调用之，函数的内容是遍历字符数组 `string`。

```
your_function:
    for i = 0; string[i] != '\0'; ++i then
        pushad
        push string[i] to stack
        call print_a_char
        pop stack
        popad
    end
    return
end
```

代码截图：

```
your_function:
; put your implementation here
    pushad
    mov eax, 0
loop:
    mov ecx, [your_string]
    cmp  byte[ecx + eax], 0
    je funcend
    pushad
    mov  ebx, dword[ecx + eax]
    push ebx
    call print_a_char
    pop ebx
    popad
    add eax, 1
    jmp loop
```

4. 运行结果示意

```
yifeidu@one:~$ cd ~/lab2/assignment
yifeidu@one:~/lab2/assignment$ make run
>>> begin test
>>> if test pass!
>>> while test pass!
Mr.Chen, students and TAs are the best!
```

三、实验总结

lab2 主要是在 lab1 基础上拓展的操作，在 lab1 中我们熟悉了汇编语言的相关语法，并将其在 lab2 的代码编写中使用。在实验过程中，由于写程序的时候一般不会一次便运行成功，所以应该善用 gdb 方法进行 debug，逐步将代码优化至满足要求为止。