Lab1

一、实验目的

本实验通过实现一个简单的引导程序，介绍系统启动的基本过程

二、实验要求

从实模式切换至保护模式，在保护模式下读取磁盘 1 号扇区中的 Hello World 程序至内存中的相应位置，跳转执行该 Hello World 程序，并在终端中打印 Hello, World!

三、实验过程

具体步骤如下：

1、 由实模式切换保护模式

关闭中断，打开 A20 数据总线，加载 GDTR，设置 CR0 的 PE 位（第 0 位）为 1b，通过长跳转设置 CS 进入保护模式；此部分代码在实模式（.code16）下实现，在教程中已给出。如图所示。

```
.code16

.global start
start:
        cli                                     #关闭中断
        inb $0x92, %al                          #启动A20总线
        orb $0x02, %al
        outb %al, $0x92
        data32 addr32 lgdt gdtDesc              #加载GDTR
        movl %cr0, %eax                         #启动保护模式
        orb $0x01, %al
        movl %eax, %cr0
        data32 ljmp $0x08, $start32             #长跳转切换至保护模式
```

初始化 DS，ES，FS，GS，SS，初始化栈顶指针 ESP

设置了三个 GDT 表项，其中代码段与数据段的基地址都为 0x0，视频段的基地址为 0xb8000。

由 GDTR 访问 GDT 是由段选择子来完成的；为访问一个段，需将段选择子存储入段寄存器，比如数据段选择子存储入 DS，代码段选择子存储入 CS；其数据结构如下

```
15                    4 3   0
+-------------------------+-+---+
|                         |T|   |
|          INDEX          | |RPL|
|                         |I|   |
+-------------------------+-+---+


 TI  - TABLE INDICATOR, 0 = GDT, 1 = LDT

        RPL - REQUESTOR'S PRIVILEGE LEVEL
```

TI 位表示该段选择子为全局段还是局部段，PRL 表示该段选择子的特权等级，13 位 Index 表示描述符表中的编号。

那么 ds、gs 的 Index 分别为 0x2、0x3，TI = 0；PRL = 0；则段选择子分别为 0x10、0x18。而 es、fs、ss 默认则与 ds 相同。同时在给 ds 等段寄存器赋值时，不能通过 mov 指令将立即数赋给段寄存器，通过赋值给 ax 再将其赋给段寄存器。而 esp 赋值则令其为（128 << 20），当小于等于此数时，均可。然后通过 jmp bootmain 指令跳转到 boot.c 中完成切换状态，从而开始通过 bootloader 装载应用程序。

```
.code32
start32:
        movw $0x10, %ax
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %fs
        movw %ax, %ss
        movw $0x18, %ax
        movw %ax, %gs
        movl $0x0, %ebp
        movl $(128 << 20), %esp
        subl $0x16, %ebp                    #初始化DS ES FS GS SS 初始化栈顶指针ESP
        jmp bootMain                      #跳转至bootMain函数 定义于boot.c
```

```
.p2align 2
gdt:
        .word 0,0                           #GDT第一个表项必须为空
        .byte 0,0,0,0

        .word 0xffff,0                      #代码段描述符
        .byte 0,0x9a,0xcf,0

        .word 0xffff,0                      #数据段描述符
        .byte 0,0x92,0xcf,0

        .word 0xffff,0x8000                 #视频段描述符
        .byte 0x0b,0x92,0xcf,0
```

2、 加载磁盘中的程序并运行

由于中断关闭，无法通过陷入磁盘中断调用 BIOS 进行磁盘读取，本次实验提供的代码框架中实现了 readSec(void *dst,int offset)这一接口（定义于 bootloader/boot.c 中），其通过读写（in，out 指令）磁盘的相应端口（Port）来实现磁盘特定扇区的读取

通过上述接口读取磁盘 MBR 之后扇区中的程序至内存的特定位置并跳转执行（注意代码框架 app/Makefile 中设置的该 Hello World 程序入口地址）

而我们通过查找 app/Makefile 中的地址，确定了程序读取到内存的特定位置为：0x8c00。如图所示：

```
zyh@ubuntu: ~/oslab/lab1/app
app.bin: app.s
        gcc -c -m32 app.s -o app.o
        ld -m elf_i386 -e start -Ttext 0x8c00 app.o -o app.elf
        objcopy -S -j .text -O binary app.elf app.bin

clean:
        rm -rf *.o *.elf *.bin
~
```

而同时通过阅读理解 boot.c 中的代码，我们可以知晓，通过调用 readSect 函数来将 1 号扇区中的 app 应用程序读取到内存中的特定位置（即 0x8c00），此处 dst 即为地址 0x8c00，offset 为扇区号 1）。而 elf 作为函数，又是一个指针，只要指向 0x8c00 即可通过调用 elf 函数来运行已经读取到主存的 app 应用程序。即为图中代码所示。

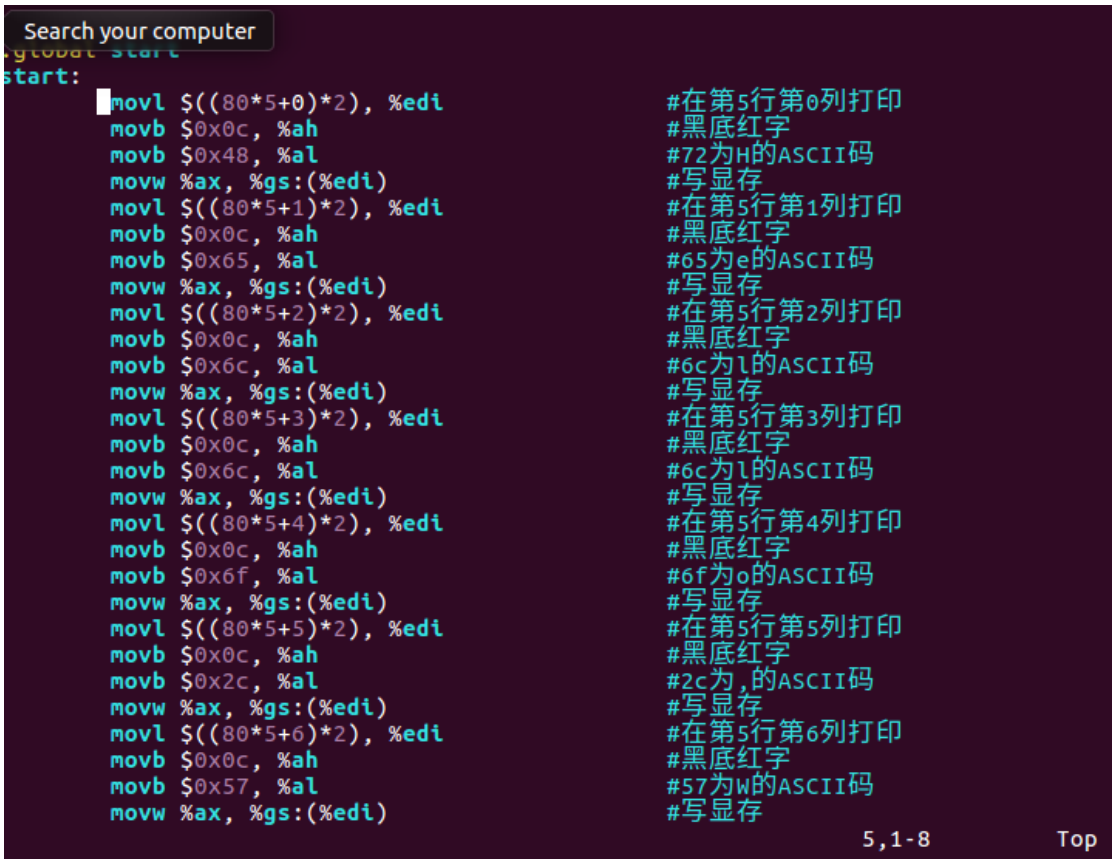
```
#define SECTSIZE 512

void bootMain(void) {
        void (*elf)(void);
        // loading sector 1 to memory
        readSect((void*)0x8c00, 1);
        elf = 0x8c00;
        elf();
}
```

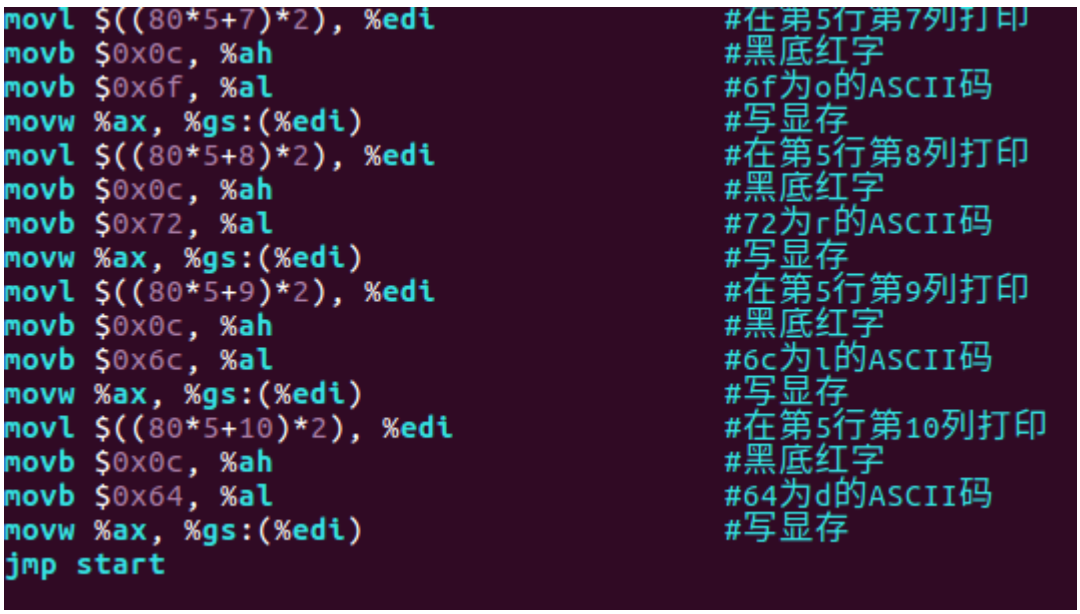
```
void readSect(void *dst, int offset) { // reading one sector of disk
        int i;
        waitDisk();
        outByte(0x1F2, 1);
        outByte(0x1F3, offset);
        outByte(0x1F4, offset >> 8);
        outByte(0x1F5, offset >> 16);
        outByte(0x1F6, (offset >> 24) | 0xE0);
        outByte(0x1F7, 0x20);

        waitDisk();
        for (i = 0; i < SECTSIZE / 4; i ++) {
                ((int *)dst)[i] = inLong(0x1F0);
        }
}
```

最后补充 app 应用程序的内容，实验要求通过运行 app 应用程序，在屏幕上输出“Hello，World!”，因此我们将 app.s 中的内容填充为输出“Hello，World!”，通过写汇编代码实现。同时注意由于中断关闭，因此不能通过陷入屏幕中断调用 BIOS 打印字符串 Hello, World!，只能通过写显存打印字符串。如下所示：

```
movl $((80*5+0)*2), %edi                    #在第 5 行第 0 列打印
movb $0x0c, %ah                             #黑底红字
movb $72, %al                               #72 为 H 的 ASCII 码
movw %ax, %gs:(%edi)                        #写显存
```

所以我们在 app.s 中如此写：





即可实现我们的要求。当然我们注意到在最末尾处我们加了一行 jmp start 的代码，此处是希望通过不断重复执行此段代码来保持操作系统运行，以避免 core dumped。

以下为不添加此句代码与添加后的对比：

不添加：

```
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0018 000b8000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00007c44 0000001f
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
CCS=00000b03 CCD=00000b3d CCO=ADDB
EFER=0000000000000000
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
Aborted (core dumped)
zyh@ubuntu:~/oslab/lab1$
```

添加：

```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980

Hello,World!

Booting from Hard Disk...
```

```
zyh@ubuntu:~/oslab/lab1$ qemu-system-i386 os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
```

至此：实验已经完成。

四、实验启示

在此次实验中虽然理解操作系统通过 bootloader 引导启动的大致流程，但对于实验具体的实现并不熟悉导致我们一开始的手足无措，代码并不复杂，但是由于对堆栈、段寄存器等性质的性质不了解，所以对于初始化等问题不清楚。

在实验过程中，我们基于教程代码和计算机系统基础实验 PA 系统代码及其他小型操作系统代码的阅读和理解比较，对我们的代码进行操作，最终将 start.s 补充完整，也对实模式切换到保护模式的具体实现有了更加清晰的认知。