

Lab1 系统引导 实验报告

1. 实验环境

- 自制OS的CPU: Intel 80386
- 模拟80386环境的虚拟机: QEMU
- 交叉编译的编译器: GCC
- 调试工具: GDB
- 运行平台: Ubuntu 18.04
- 编程语言: C, X86 Assembly

2. 实验要求

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。

2.1 在实模式下实现一个Hello World程序

在实模式下在终端中打印 `Hello, world!`

2.2 在保护模式下实现一个Hello World程序

从实模式切换至保护模式, 并在保护模式下在终端中打印 `Hello, world!`

2.3. 在保护模式下加载磁盘中的Hello World程序运行

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!`

3. 实验者

- 姓名: 张福翔
- 学号: 171840514
- 联系方式: 499480213@qq.com

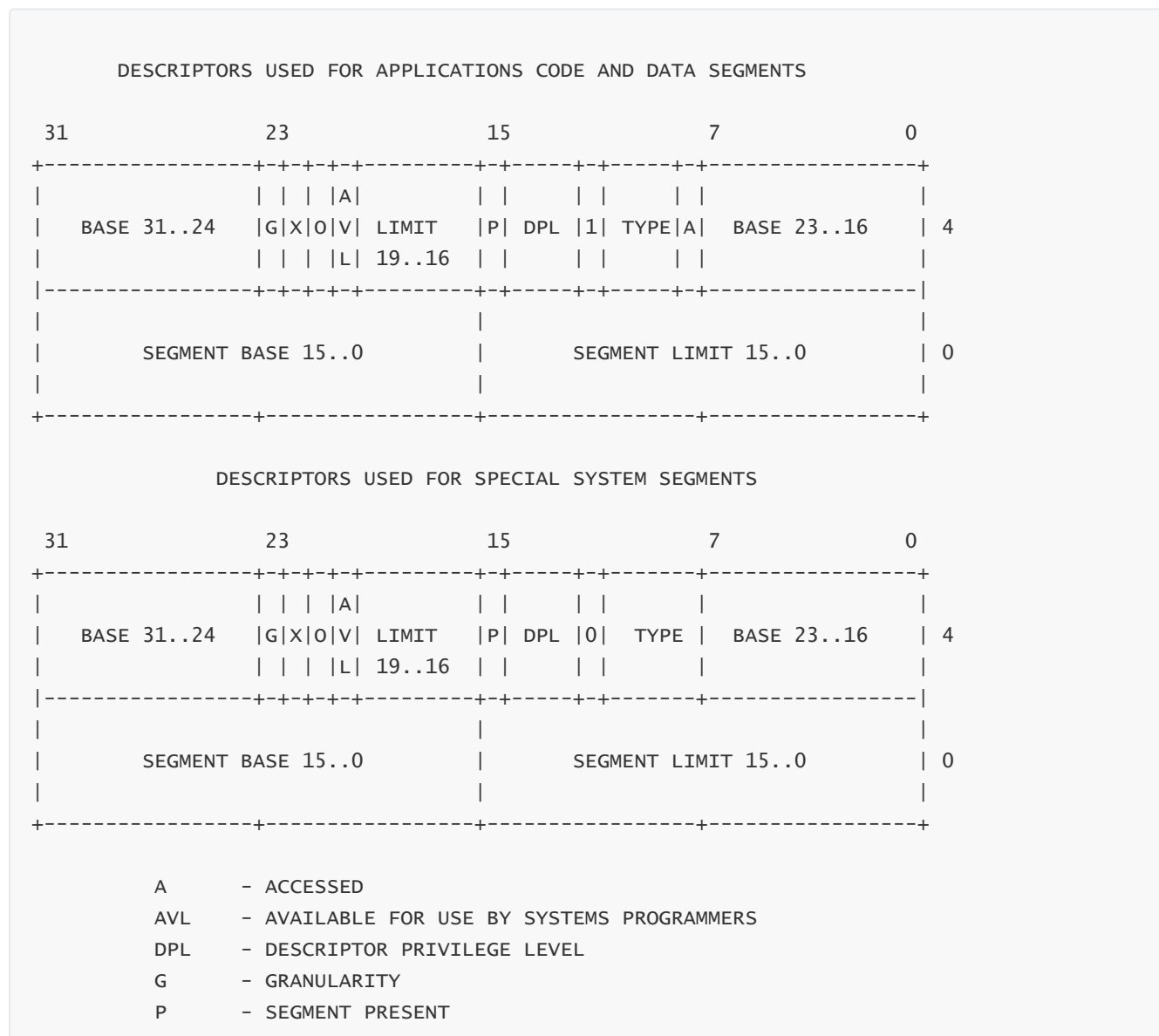
4. 实验原理

4.1 IA-32 的存储管理

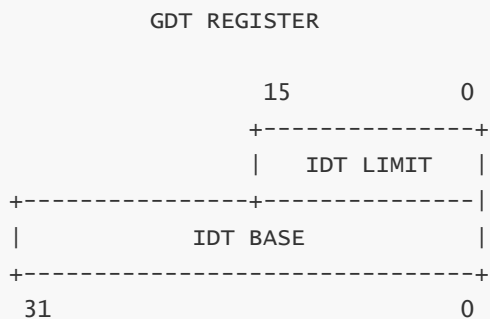
在 IA-32 下, CPU 有两种工作模式: 源于 8086 的实模式与源于 80386 的保护模式。

在实模式下，16 位寄存器通过 Segment:Offset 这种方式实现 1MB 的寻址能力；在保护模式下，虽然寄存器为 32 位，能够进行 4GB 空间的寻址，Segment:Offset 这种寻址方式仍然被保留下来，其中 Segment 仍然由 16 位的段寄存器表示，称为段选择子。

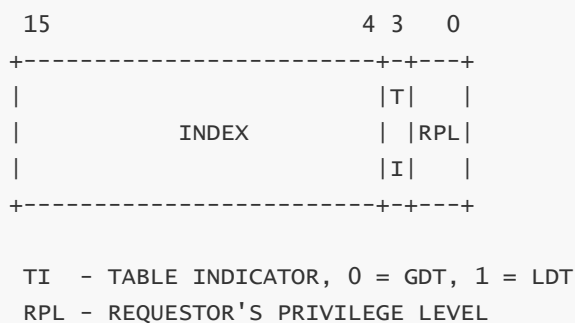
在保护模式下，16 位段寄存器的高 13 位存储着一个索引，该索引指向一个数据结构的表项，表项中定义了段的起始 32 位物理地址，段的界限，属性等内容；而这一数据结构就是 Global Descriptor Table，即 GDT，GDT 中的一个表项称为段描述符；段描述符的结构如下图所示：



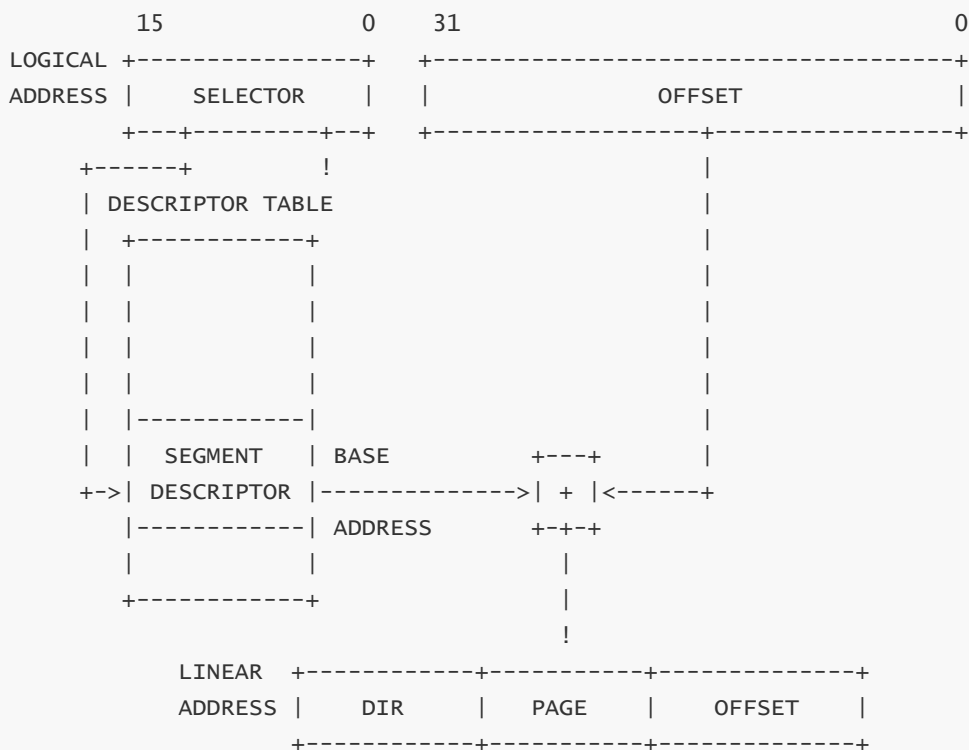
为进入保护模式，需要在内存中开辟一块空间存放 GDT 表；80386 提供了一个寄存器 `GDTR` 用来存放 GDT 的 32 位物理基地址以及表长界限；在将 GDT 设定在内存的某个位置后，可以通过 `LDGT` 指令将 GDT 的入口地址装入此寄存器：



由 GDT 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；段寄存器的数据结构如下：



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 Selector:Offset 表示的逻辑地址可如下图所示转化为线性地址，倘若不采用分页机制，则该线性地址即物理地址。



4.2 系统启动

系统启动时，计算机工作在实模式下，其中 `CS:IP` 指向 BIOS 的第一条指令，即首先取得控制权的是 BIOS，BIOS 将检查各部分硬件是否工作正常，然后按照 CMOS RAM 中设置的启动设备查找顺序来寻找可启动设备。

系统启动时，工作在实模式的 BIOS 程序将主引导扇区（Master Boot Record，0号柱面，0号磁头，0号扇区对应的扇区，512 字节，末尾两字节为魔数 `0x55` 和 `0xaa`）加载至内存 `0x7c00` 处（被加载的程序一般称为 Bootloader），紧接着执行一条跳转指令，将 `CS` 设置为 `0x0000`，`IP` 设置为 `0x7c00`，运行被装入的 Bootloader。

5. 代码框架

```
+lab
|---+bootloader
|   |---boot.h           # 磁盘I/O接口
|   |---boot.c           # 加载磁盘上的用户程序
|   |---start.s          # 引导程序
|   |---Makefile
|---+utils
|   |---genboot.pl       # 生成 MBR
|---+app
|   |---app.s            # 用户程序
|   |---Makefile
|---Makefile
```

我们可以先观察 `Makefile` 文件的命令：

首先是主目录下的 `Makefile` 文件。

```
# ./Makefile
os.img:
    cd bootloader; make bootloader.bin
    cd app; make app.bin
    cat bootloader/bootloader.bin app/app.bin > os.img

clean:
    cd bootloader; make clean
    cd app; make clean
    rm -f os.img

play:
    qemu-system-i386 os.img

debug:
    qemu-system-i386 -s -S os.img
```

默认命令 `make` 用于生成 QEMU 需要的 `os.img` 镜像文件。其生成过程是：先从目录 `bootloader` 生成二进制文件 `bootloader.bin`，然后从目录 `app` 生成 `app.bin`，再通过 `cat` 命令将这两个文件连接为 `os.img`。

我们来分别看看这两个文件的生成过程：

```
# app/Makefile
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
```

```
# bootloader/Makefile
bootloader.bin: start.s boot.c boot.h
    gcc -c -m32 start.s -o start.o
    gcc -c -m32 -O1 -fno-stack-protector boot.c -o boot.o
    ld -m elf_i386 -e start -Ttext 0x7c00 start.o boot.o -o bootloader.elf
    objcopy -S -j .text -O binary bootloader.elf bootloader.bin
    ../utils/genboot.pl bootloader.bin

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

可以看到，我们在链接这两个程序时调用 `ld` 命令的 `-Ttext` 参数手动地为其指定好了程序的起始地址，其中 `app` 为 `0x8c00` 而 `bootloader` 为 `0x7c00`。其中地址 `0x7c00` 即为 BIOS 加载引导程序的默认地址。链接完成后我们使用 `objcopy` 命令将两个程序的 `.text` 节拷贝成为 `.bin` 文件，并调用 `utils/genboot.pl` 对 `bootloader.bin` 文件进行处理，生成主引导记录（**MBR**），检查文件有没有超出 510 个字节的限制。

随后，通过主目录的 `cat` 命令，我们就将两个文件连接了起来。

我们可以注意到，主目录下的 `Makefile` 还有一个 `debug` 命令，这是通过给 `qemu` 赋参数 `-s -S` 来生成一个供 `gdb` 进行调试的服务器，我们可以通过 `gdb` 的断点等一系列功能，对程序运行过程有更好的判断。

6. 解决方法

6.1 在实模式下实现一个Hello World程序

实模式下 BIOS 加载好了基本的中断向量表，我们可以调用系统自陷指令来调用 BIOS 运行相关的处理程序，从而打印出需要的字符串 `Hello, world!`。

```
movw $message, %ax          # 将字符串message的首地址移到AX
movw %ax, %bp
movw $13, %cx               # 打印的字符串长度，不包含'\0'字符
movw $0x1301, %ax          # AH=0x13 打印字符串
movw $0x000c, %bx          # BH=0x00 黑底 BL=0x0c 红字
movw $0x0000, %dx          # 在第0行0列开始打印
int $0x10                  # 陷入0x10号中断

message:
    .string "Hello, world!"
```

也可以通过写显存的方式，将一个个字符显示在屏幕上，例如下面是打印字符 `H` 的代码：

```
movl $((80*5+0)*2), %edi      # 在第5行第0列打印
movb $0x0c, %ah               # 黑底红字
movb $42, %al                 # 42为'H'的ASCII码
movw %ax, %gs:(%edi)          # 写显存
```

实际上由于调用系统自陷指令更为方便，而在实模式下我们还没有关闭中断，因此通过调用自陷指令来实现字符串 `Hello, world!` 的打印显得较为简单。

6.2 在保护模式下实现一个Hello World程序

从实模式下切换到保护模式下需要经过一系列的操作，包括关闭中断，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，初始化 DS，ES，FS，GS，SS。我们设置了三个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。

由于切换到保护模式的过程中需要关闭中断，因此我们没有办法通过调用自陷指令的方式打印出字符串，只能使用写入显存的方式将字符一个个写入显存中，我们设计了函数 `displayStr` 来完成这个工作：

```
start32:
    ...                        # 初始化DS ES FS GS SS 初始化栈顶指针ESP

    # 在保护模式下通过写入显存打印字符串

    pushl $13                  # 字符串长度
    pushl $message             # 字符串地址
    calll displayStr           # 调用显示函数

loop32:
    jmp loop32                 # 完成后通过死循环终止

message:
    .string "Hello, world!\n\0"

displayStr:
    movl 4(%esp), %ebx
    movl 8(%esp), %ecx
    movl $((80*8+0)*2), %edi    # 打印从第五行第一列起
    movb $0x0c, %ah            # 黑底红字

nextChar:
    movb (%ebx), %al
    movw %ax, %gs:(%edi)
    addl $2, %edi
    incl %ebx
    loopnz nextChar
    ret

    # 在6.3中实现
    # jmp bootMain              # 跳转至bootMain函数，加载磁盘中的程序
```

6.3 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。

以下是 `boot.c` 文件中的函数实现，其中 `readSec` 函数与 `waitDisk` 函数使用到了内联函数 `inByte` 和 `outByte`，这两个函数即是对指令 `in`，`out` 的封装。而 `bootMain` 函数则是我们真正执行的函数。其中利用函数指针 `elf` 将我们的用户程序作为函数，在装载过后直接执行。而从磁盘中装载程序则用到了 `readSect` 函数，其先利用 `waitDisk` 指令轮询磁盘的工作状况，在磁盘空闲后通过接口调用 `out` 指令将相应的端口写上需要读取的信息（即磁盘的**1号扇区**）。继续轮询直到磁盘完成任务后，我们再将磁盘里的数据从端口上写到我们的内存中，其地址为 `0x8c00`。

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

void waitDisk(void) { // waiting for disk
    while((inByte(0x1f7) & 0xc0) != 0x40);
}

void readSect(void *dst, int offset) { // reading a 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);
    }
}
```

因此，我们在启动系统并置为保护模式后，只需要通过 `jmp` 指令跳转到我们写好的 `bootMain` 函数即可。

```
start32:
    ...                # 初始化，启动保护模式
    jmp bootMain        # 跳转到 bootMain 函数
```

`bootMain` 函数经过我们上述的那些操作后，会从内存地址为 `0x8c00` 的地方开始执行指令，这就是我们用户程序被从磁盘中读取以后的所在位置。接下来，我们只需要在 `app.s` 中编辑好我们的用户程序，让其能够输出所需的 `Hello, world!` 字符串即可，由于中断的关闭，我们可以采取上述写显存的方式来完成我们的用户程序输出：

```
.code32

.global start
```

```

start:
    # TODO
    pushl $46
    pushl $message
    calll displayStr
    # never return
    hlt

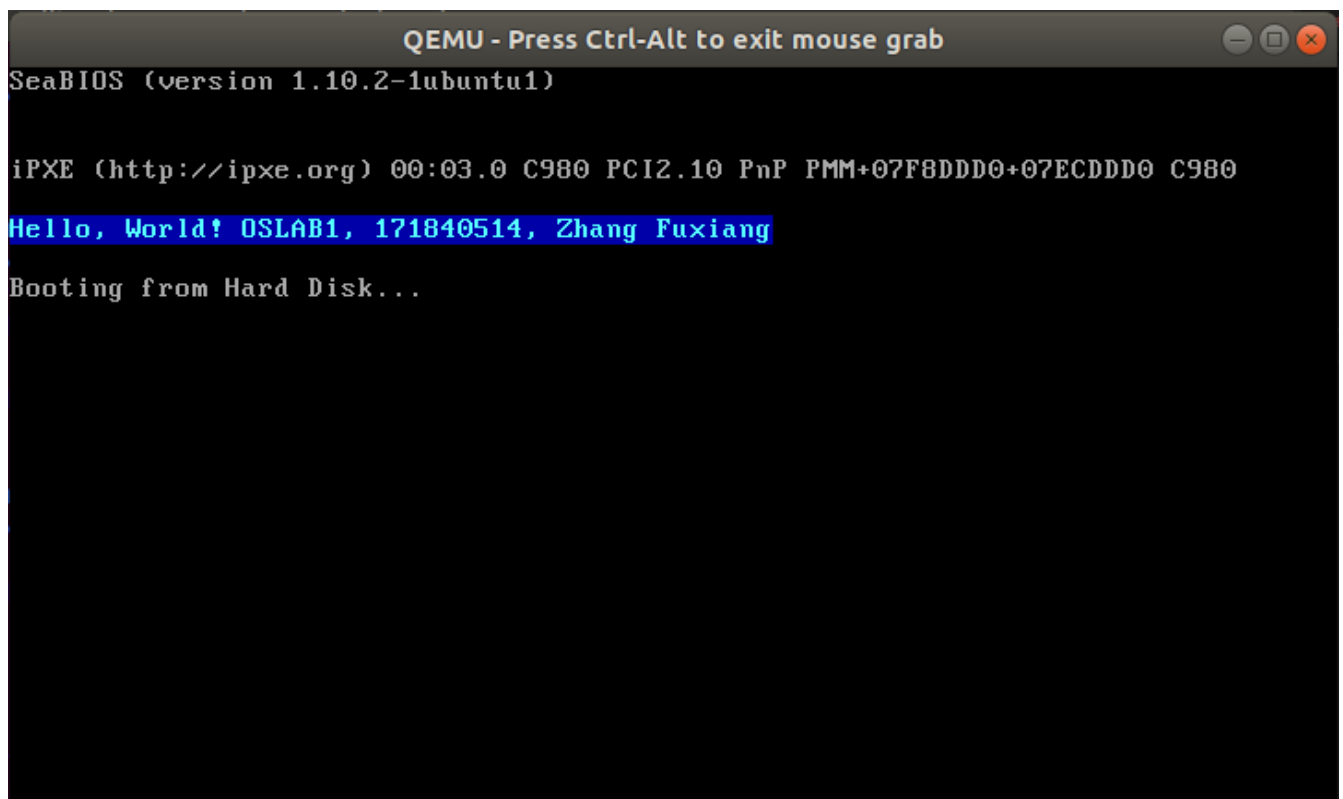
message:
    .string "Hello, world! OSLAB1, 171840514, Zhang Fuxiang\0"

displayStr:
    movl 4(%esp), %ebx
    movl 8(%esp), %ecx
    movl $((80*5+0)*2), %edi
    movb $0x1b, %ah
nextChar:
    movb (%ebx), %al
    movw %ax, %gs:(%edi)
    addl $2, %edi
    incl %ebx
    loopnz nextChar
    ret

```

其中有关输出的部分仍与之前的大体相同，我们使用 `hlt` 指令代替了此前的一个循环来让计算机停止继续工作。此外我们也可以按照规则修改我们输出的字符串和显示的颜色。其中 `movb $0x0b, %ah` 是用来控制输出颜色和背景的，其中的低 4 位是字体颜色而高 4 位是文字背景。

经过上述的修改，我们在 `make play` 命令以后，就可以看到 QEMU 的如下输出：



```

QEMU - Press Ctrl-Alt to exit mouse grab
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980
Hello, World! OSLAB1, 171840514, Zhang Fuxiang
Booting from Hard Disk...

```


这表示我们已经成功地在保护模式下通过装载 `app.s` 中的程序来执行用户程序，达到输出一个字符串的目的。

7.实验心得

通过本次实验我加强了对计算机系统相关知识的了解，并在实际编程中得到了一定的巩固。

实验中唯一不太理解的地方应该是关于 `utils` 目录下的 Perl 语言程序 `genBoot.pl` 的具体机制，因为对该程序语言的陌生导致了不太清楚此处的详细操作，但通过查阅相关资料我还是对该程序的作用有了一定的了解。