

《操作系统实验》实验报告

实验名称：系统引导

姓名：胡育玮

学号：171860574

邮箱：yeevee@qq.com

班级：17 级计算机科学与技术系 2 班

时间：2019 年 3 月 17 日

一、实验目的

- 1、学习在 Linux 环境下编写、调试程序,初步掌握 Shell、Vim、GCC、Binutils、Make、QEMU、GDB 的使用
- 2、学习 AT&T 汇编程序的特点
- 3、理解系统引导程序的含义,理解系统引导的启动过程

二、实验内容

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. 在保护模式下实现一个 Hello World 程序
-----在保护模式下在终端中打印 Hello, World!
3. 在保护模式下加载磁盘中的 Hello World 程序并运行
-----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!

三、实验过程

(1) 分析：

根据课程网站的说法，本实验仅需提交保护模式下加载磁盘中的 Hello World 程序并运行的相关源码与报告，因此工作的核心就是实现在保护模式下启动 Hello World 程序并让屏幕显示 “Hello, World!”。

下载好课程网站提供的框架代码后，打开其根目录下的 Makefile 文件，发现有如下命令：

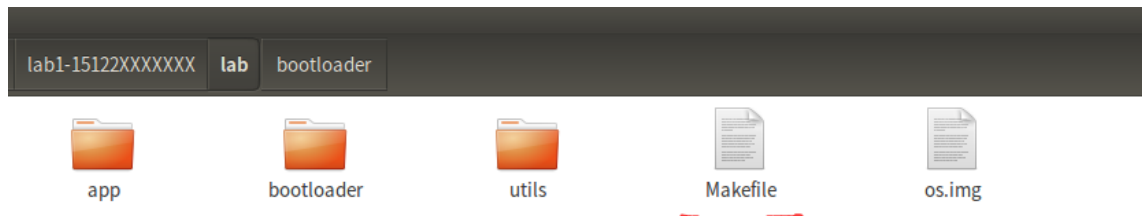


图 1：框架代码文件根目录下的 Makefile 文件

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

play:
    qemu-system-i386 os.img
```

图 2：Makefile 里的命令

于是在终端进入根目录，直接进行编译并执行，执行结果如下：（依次使用 `make clean`、`make`、`make play` 三个命令进行编译和执行）

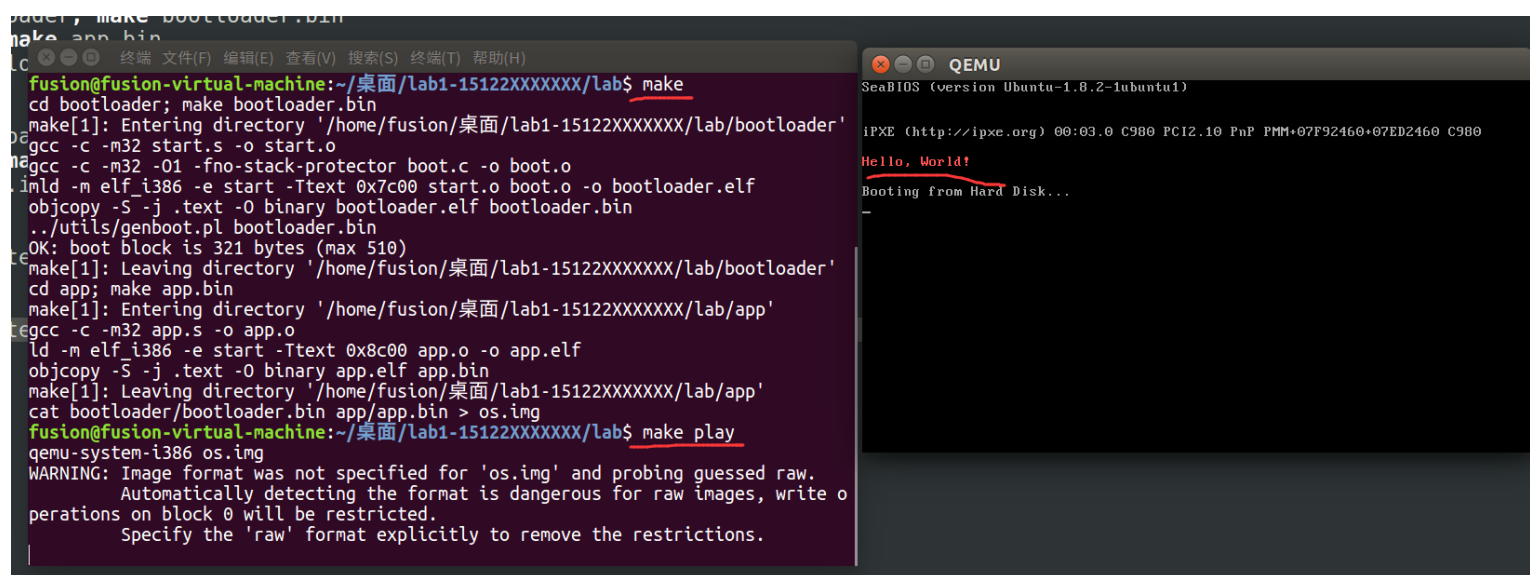


图 3：执行结果

由图 3 可知，**直接执行框架代码**，便可以在 QEMU 的界面内输出 **Hello, world!**，说明框架代码内已有与输出字符串相关的指令。

根据课程网站，框架代码文件夹的结构如下：



图 4：框架代码文件夹的结构

根据讲义：在 PC 启动时，首先会在实模式下运行 BIOS。
BIOS 进行系统自检等工作，**之后 BIOS 会读取磁盘的**

MBR(Master Boot Record, 磁盘的 0 号柱面, 0 号磁头, 0 号扇区的 512 字节)到内存的 0x7C00(被装入的程序一般称为 **Bootloader, 启动引导程序)**, 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****。

而 **start.s** 和 **boot.c** 中的程序就是本次实验中的 **bootloader**。其中, **start.s** 的作用是将 80386 处理器从实模式切换至 32 位的保护模式, 并完成各个段寄存器以及栈顶指针 ESP 的初始化, 而 **boot.c** 的作用是将存储在 MBR 之后的磁盘扇区中的程序加载至内存的特定位置并跳转至那处来执行。

于是下面我们来查看 **start.s** 和 **boot.c** 的代码:

```
/* Protected Mode Hello World */
.code16
.global start
start:                                     # personal comprehension: doing necessary stuffs to boot the sys
    movw %cs, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %ss
    cli                                     # clear interruption:关闭中断
    inb $0x92, %al                         # 打开A20地址线
    orb $0x02, %al
    outb %al, $0x92
    data32 addr32 lgdt gdtDesc             # loading gdtr, data32, addr32: 加载GDTR寄存器
    movl %cr0, %eax
    orb $0x01, %al                         # set last bit of cr0 to 1 : 设置CR0寄存器末位为1, 表示进入保护模式
    movl %eax, %cr0                       # setting cr0
    data32 ljmp $0x08, $start32           # 长跳转进入保护模式代码: reload code segment selector and ljmp to st
```

图 5: **start.s** 代码及注释局部: 该图显示的为实模式的代码

```

#保护模式代码:
.code32
start32:
    movw $0x10, %ax      # setting data segment selector
    movw %ax, %ds        # 初始化 DS, SS, ES, FS, GS 这些段寄存器
    movw %ax, %es
    movw %ax, %fs
    movw %ax, %ss
    movw $0x18, %ax      # setting graphics data segment selector
    movw %ax, %gs

    movl $0x8000, %eax    # setting esp:初始化esp寄存器
    movl %eax, %esp

```

```

# jmp to bootMain in boot.c: 应该将下述代码修改为跳转到boot.c中的bootMain函数去执行
pushl $13                #字符串中字符个数
pushl $message            #字符串首地址
calll displayStr          #调用字符串打印函数
loop32:                   #这个死循环很重要
    jmp loop32

message:                  #字符串
    .string "Hello, World!\n\0"

displayStr:               #函数体
    movl 4(%esp), %ebx
    movl 8(%esp), %ecx
    movl $((80*5+0)*2), %edi
    movb $0x0c, %ah

nextChar:                 #打印字符的循环
    movb (%ebx), %al
    movw %ax, %gs:(%edi)
    addl $2, %edi
    incl %ebx
    loopnz nextChar       # loopnz decrease ecx by 1
    ret

```

图 6: start.s 代码及注释局部：该图显示的为保护模式的代码

由图 6 中的 start.s 的代码及我个人加的注释可知，start.s 内除了完成从实模式到保护模式的切换以及各个段寄存器及 ESP 的初始化工作以外，还完成了输出字符串“Hello world!”的工作。在初始化完 esp 后，程序将要输出的字符串的长度、首地址压栈并调用 displayStr 函数；调用完该函数后

陷入一个死循环 loop32。displayStr 函数读取要输出的字符串的长度和地址，用 nextChar 循环输出字符串。nextChar 循环实质上是不断地往显存中特定位置（用 %edi 来存储位置。displayStr 中的 movl \$((80*5+0)*2), %edi 语句就是把要输出的位置赋给 edi 寄存器）写入字符，以此完成输出工作。

```
void bootMain(void) {
    void (*elf)(void);
    elf = (void*)(void)0x8c00;

    readSect((void*)elf, 1); //将sector1中的程序加载到内存的0x8c00处 load
    elf();                  //跳转到内存指定位置执行程序：也即执行app.s ju
}
```

图 7: boot.c 中部分代码

同时，如图 7 所示，boot.c 中的 bootMain 函数的功能是将磁盘的 1 号扇区中的程序（此处即为 app.s，也即以后的操作系统程序）加载到内存的 0x8c00 处，然后跳转到 0x8c00 处执行程序。

本实验要求在保护模式下加载磁盘中的 Hello World 程序并运行，故综上分析可知，**只需将输出 hello world 的代码放到 app.s 里，并在 start.s 内相关的功能语句（从实模式转到保护模式、初始化寄存器）执行完后让控制转移到 bootMain 处，让 bootMain 把控制转到 app.s 里的程序即可。**

故最终的实现如下：

```

.code32
.global start

message:
    .string "Hello, Wordddld!"

start:
    pushl $15
    pushl $message
    calll displayStr

loop32:
    jmp loop32

displayStr:
    movl 4(%esp), %ebx
    movl 8(%esp), %ecx
    movl $((80*5+0)*2), %edi
    movb $0x0c, %ah

nextChar:
    movb (%ebx), %al
    movw %ax, %gs:(%edi)
    addl $2, %edi
    incl %ebx
    loopnz nextChar    # loopnz decrease ecx by 1
    ret

```

图 8: app.s 的最终代码：只需将原来的 start.s 的输出字符串的代码剪切过来即可完成。此处将输出的字符串的长度变为 15，输出的内容变为 “Hello, Wordddld!”

```

    movw %ax, %gs

    movl $0x8000, %eax    # setting esp
    movl %eax, %esp

    # jmp to bootMain in boot.c: 应该将下述代码修改为跳转到boot.c中的bootMain函数去执行
    jmp bootMain

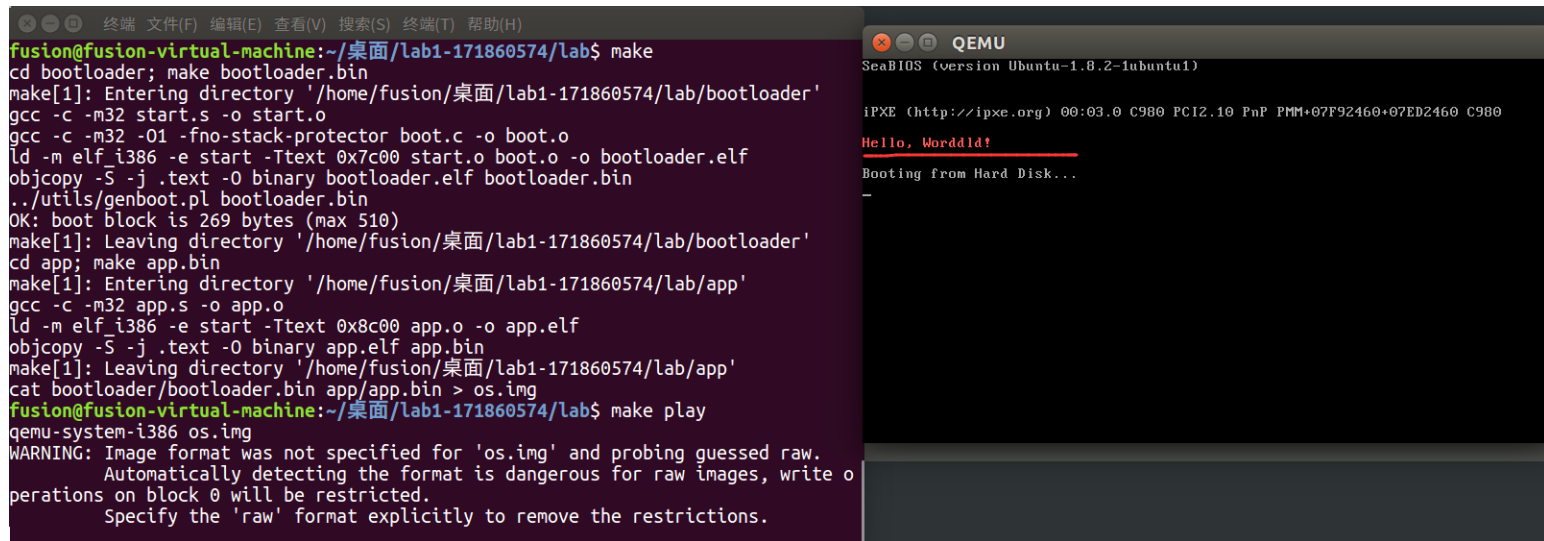
#   pushl $13            #字符串中字符个数
#   pushl $message       #字符串首地址
#   calll displayStr     #调用字符串打印函数
#loop32:                #这个死循环很重要
#   jmp loop32

.p2align 2
gdt:                    # 8 bytes for each table entry, at least 1 entry
    .word 0,0           # empty entry

```

图 9: start.s 的最终代码：将原来的输出字符串的代码删去，添加 jmp bootMain 语句

最终实现效果：



The image shows two side-by-side windows. The left window is a terminal with a dark background and light text, showing the commands and output for building a bootloader. The right window is a QEMU emulator window with a light background and dark text, showing the boot process of a virtual machine.

```
fusion@fusion-virtual-machine:~/桌面/lab1-171860574/lab$ make
cd bootloader; make bootloader.bin
make[1]: Entering directory '/home/fusion/桌面/lab1-171860574/lab/bootloader'
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
OK: boot block is 269 bytes (max 510)
make[1]: Leaving directory '/home/fusion/桌面/lab1-171860574/lab/bootloader'
cd app; make app.bin
make[1]: Entering directory '/home/fusion/桌面/lab1-171860574/lab/app'
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
make[1]: Leaving directory '/home/fusion/桌面/lab1-171860574/lab/app'
cat bootloader/bootloader.bin app/app.bin > os.img
fusion@fusion-virtual-machine:~/桌面/lab1-171860574/lab$ make play
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 operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

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

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

Booting from Hard Disk...
```

可见成功输出“Hello, Wordddld!”。

至此本次实验已完成。