

操作系统 lab1 系统引导

邱梓豪

141130077

0.配置实验环境

按照实验讲义上的要求配置实验环境即可。

```
$sudo apt-get update
$sudo apt-get install qemu-system-x86
$sudo apt-get install vim
$sudo apt-get install gcc
$sudo apt-get install gdb
$sudo apt-get install binutils
$sudo apt-get install make
$sudo apt-get install perl
```

在这里要注意一点：在安装 `qemu` 之前一定要先 `update`，我最开始安装时没有注意这点，然后在安装 `qemu` 时提示我有几个包未能获取，导致安装失败。

1.1 在实模式下在终端打印 Hello World！

实模式中，程序的逻辑地址就是物理地址，所以只要直接执行打印 `Hello World` 的汇编程序即可，这里我将该程序嵌入 `bootloader` 的 `start.s` 文件中：

```
code16

.global start
start:

movw $message, %ax
movw %ax, %bp
movw $13, %cx
movw $0x1301, %ax
movw $0x000c, %bx
movw $0x0000, %dx
int $0x10

message:
.string "Hello, World!"

.code32
start32:

.p2align 2
gdt:
.word 0,0 # empty entry
.byte 0,0,0,0

gdtDesc:
.word (gdtDesc - gdt - 1)
.long gdt
```

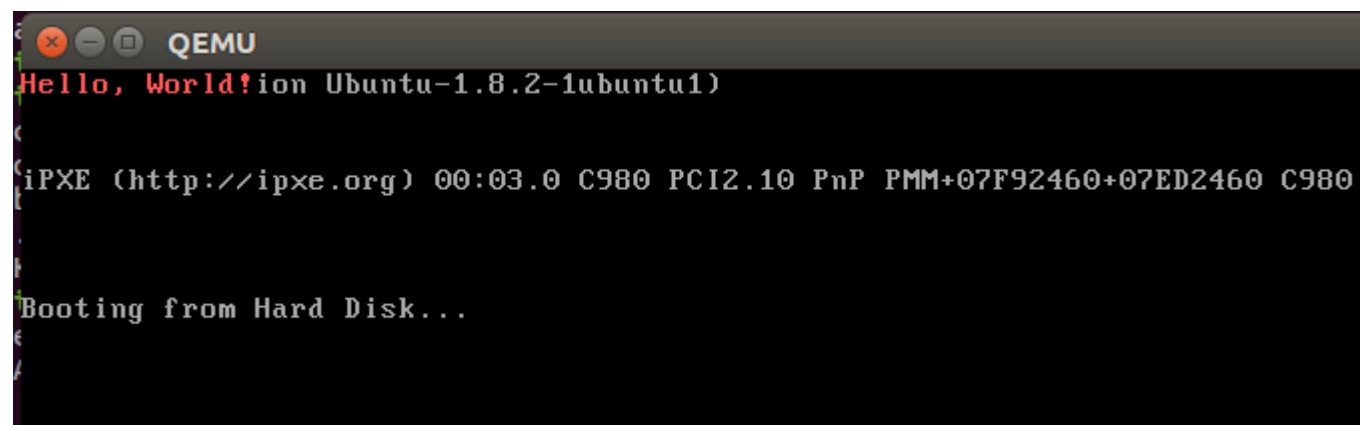
然后再修改 bootloader 目录下的编译脚本：

```
# to display 'Hello world' in phase 1
bootloader.bin: start.s
    gcc -c -m32 start.s -o start.o
    ld -m elf_i386 -e start -Ttext 0x7c00 start.o -o bootloader.elf
    objcopy -S -j .text -O binary bootloader.elf bootloader.bin
    ../utils/genboot.pl bootloader.bin

# to display 'Hello world' in phase 1
play:
    qemu-system-i386 bootloader.bin

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

这样打印 Hello World 的程序就被放入了操作系统的引导程序当中，使用 qemu 运行 bootloader.bin 就可以了：



```
QEMU
Hello, World!ion Ubuntu-1.8.2-1ubuntu1)
(iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980
Booting from Hard Disk...
```

1.2 在保护模式下在终端打印 Hello World！

首先进入保护模式，主要做如下工作：关中断，启动 A20 总线，加载 GDTR，启动保护模式，最后利用 ljmp 跳转至保护模式：

```
.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
    orl $0x1, %eax
    movl %eax, %cr0

    data32 ljmp $0x08, $start32        #长跳转切换至保护模式
```

因为进入保护模式之前关闭了中断，所以不能通过 BIOS 中断打印字符，只能通过写显存的方法打印字符。

```
.p2align 2
gdt:
    .word 0,0                # empty entry
    .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
```

由于视频段描述符在 GDT 的第三项，所以在写显存之前要将该段的下标赋给 %gs:

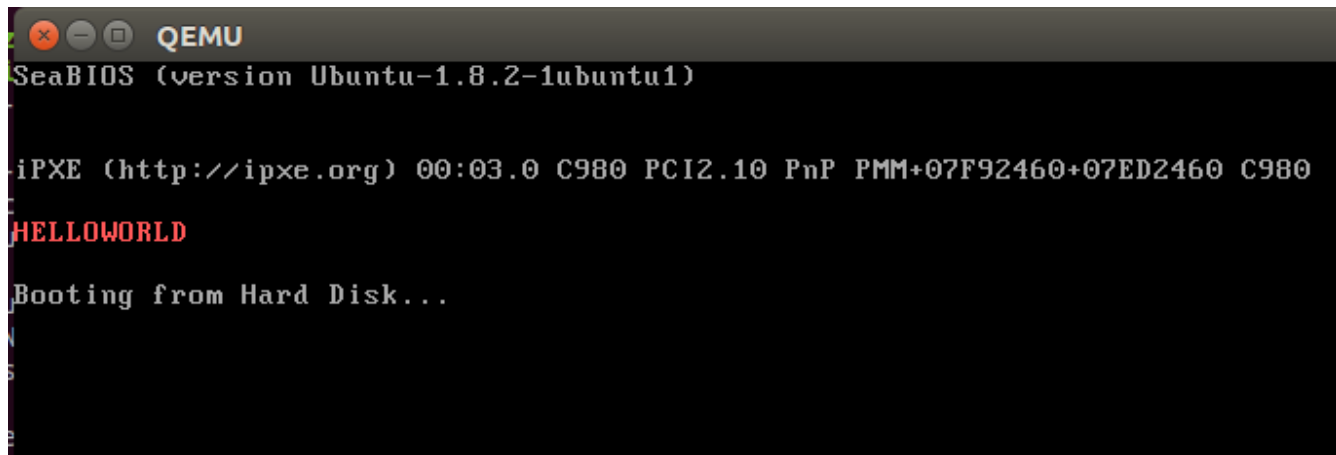
```
movw $0x18, %ax
movw %ax, %gs
```

之后我将每个字符依次写入显存:

```
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               #42为H的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+1)*2), %edi    #在第5行第1列打印
movb $69, %al               #42为E的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+2)*2), %edi    #在第5行第2列打印
movb $76, %al               #42为L的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+3)*2), %edi    #在第5行第3列打印
movb $76, %al               #42为L的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+4)*2), %edi    #在第5行第4列打印
movb $79, %al               #42为O的ASCII码
movw %ax, %gs:(%edi)        #写显存
```

```
movl $((80*5+5)*2), %edi    #在第5行第5列打印
movb $87, %al               #42为W的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+6)*2), %edi    #在第5行第6列打印
movb $79, %al               #42为O的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+7)*2), %edi    #在第5行第7列打印
movb $82, %al               #42为R的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+8)*2), %edi    #在第5行第8列打印
movb $76, %al               #42为L的ASCII码
movw %ax, %gs:(%edi)        #写显存
movl $((80*5+9)*2), %edi    #在第5行第9列打印
movb $68, %al               #42为D的ASCII码
movw %ax, %gs:(%edi)        #写显存
```

最后成功在保护模式下输出了 HELLOWORLD:

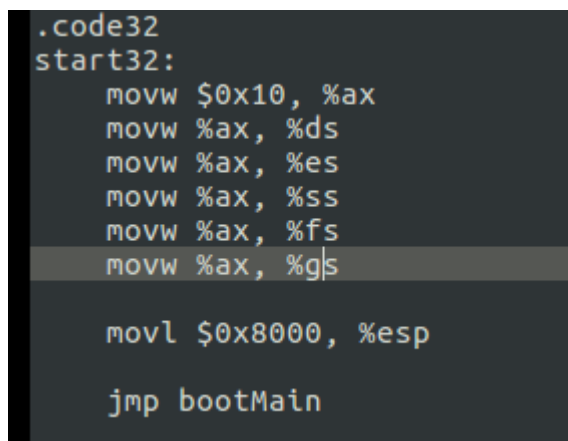


```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980
HELLOWORLD
Booting from Hard Disk...
```

1.3 在保护模式下加载磁盘中的 Hello World 程序运行

系统在进入保护模式之后先进行一些寄存器的初始化，然后跳转进入 bootMain

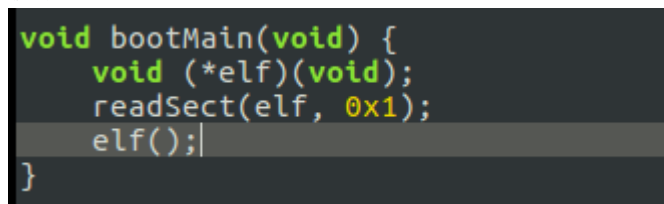


```
.code32
start32:
    movw $0x10, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %ss
    movw %ax, %fs
    movw %ax, %gs

    movl $0x8000, %esp

    jmp bootMain
```

在 bootMain () 中，所做的工作就是从磁盘的 1 号扇区读取程序进入内存的相应位置，然后转入执行。具体的做法就是首先声明一个函数指针 elf，然后从 1 号扇区读取内容读入 elf 所指向的内存空间，elf 实际上存储的就是这段代码在内存中的首地址。之后执行 elf ()，就是跳转到这段程序的首条指令处，开始执行。



```
void bootMain(void) {
    void (*elf)(void);
    readSect(elf, 0x1);
    elf();
}
```

在 app.s 中我使用写显存的方式来输出 Hello World，具体的代码如下：

```

.global start
start:

    movw $0x18, %ax
    movw %ax, %gs
    movl $((80*1+0)*2), %edi
    movb $0x0c, %ah
    movb $72, %al                # display 'H'
    movw %ax, %gs:(%edi)

    movl $((80*1+1)*2), %edi
    movb $101, %al              # display 'e'
    movw %ax, %gs:(%edi)

    movl $((80*1+2)*2), %edi
    movb $108, %al              # display 'l'
    movw %ax, %gs:(%edi)

    movl $((80*1+3)*2), %edi
    movb $108, %al              # display 'l'
    movw %ax, %gs:(%edi)

    movl $((80*1+4)*2), %edi
    movb $111, %al              # display 'o'
    movw %ax, %gs:(%edi)

```

```

    movl $((80*1+5)*2), %edi
    movb $32, %al                # display ' '
    movw %ax, %gs:(%edi)

    movl $((80*1+6)*2), %edi
    movb $87, %al                # display 'W'
    movw %ax, %gs:(%edi)

    movl $((80*1+7)*2), %edi
    movb $111, %al              # display 'o'
    movw %ax, %gs:(%edi)

    movl $((80*1+8)*2), %edi
    movb $114, %al              # display 'r'
    movw %ax, %gs:(%edi)

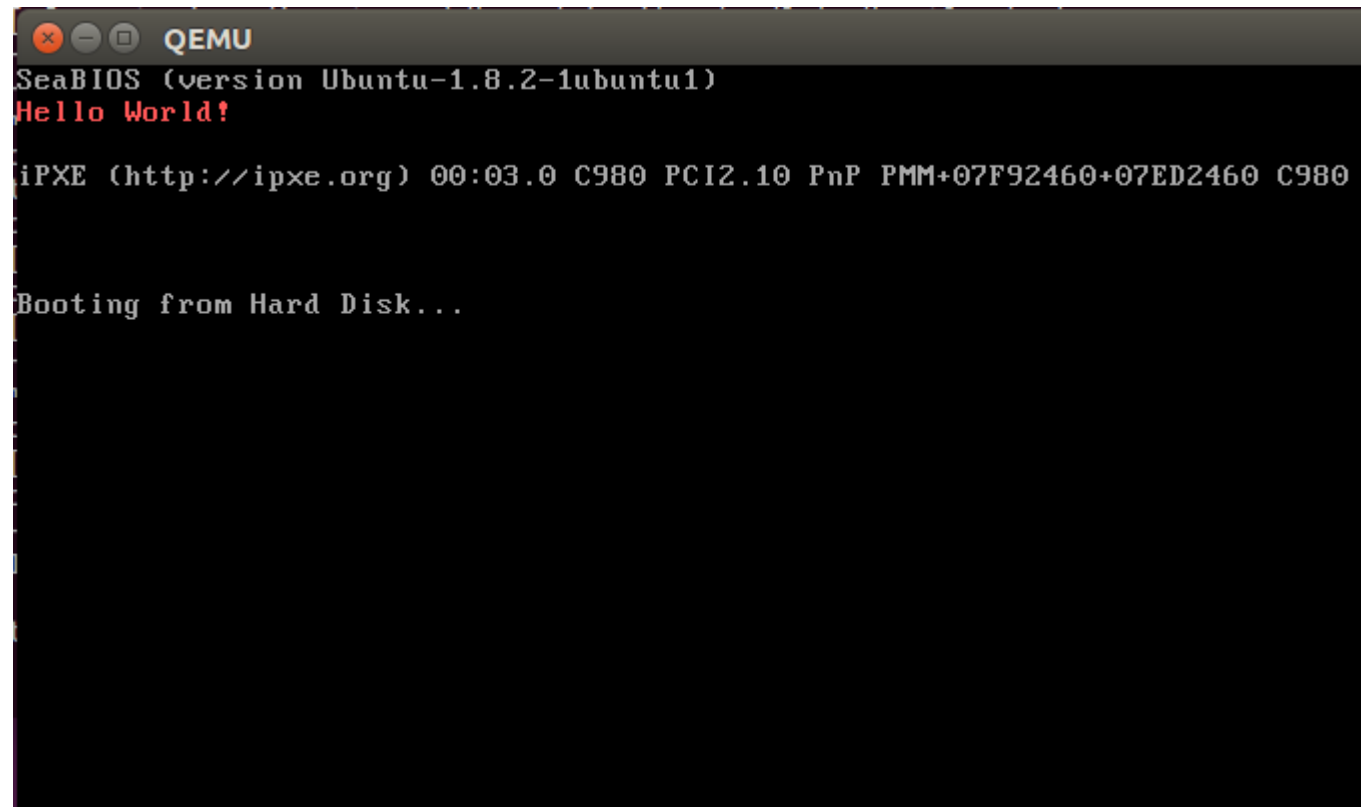
    movl $((80*1+9)*2), %edi
    movb $108, %al              # display 'l'
    movw %ax, %gs:(%edi)

    movl $((80*1+10)*2), %edi
    movb $100, %al              # display 'd'
    movw %ax, %gs:(%edi)

    movl $((80*1+11)*2), %edi
    movb $33, %al                # display '!'
    movw %ax, %gs:(%edi)

```

写完后在 lab1 根目录下运行 `make os.img; make play`, 就可以看到输出的字符。
最后执行的效果如下:



```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)
Hello World!

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

Booting from Hard Disk...
```