操作系统Lab1实验报告

211300082 谭荣熙 rongxitan@smail.nju.edu.cn

1. 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

1.1 在实模式下实现一个Hello World程序

1.2 在保护模式下实现一个Hello World程序

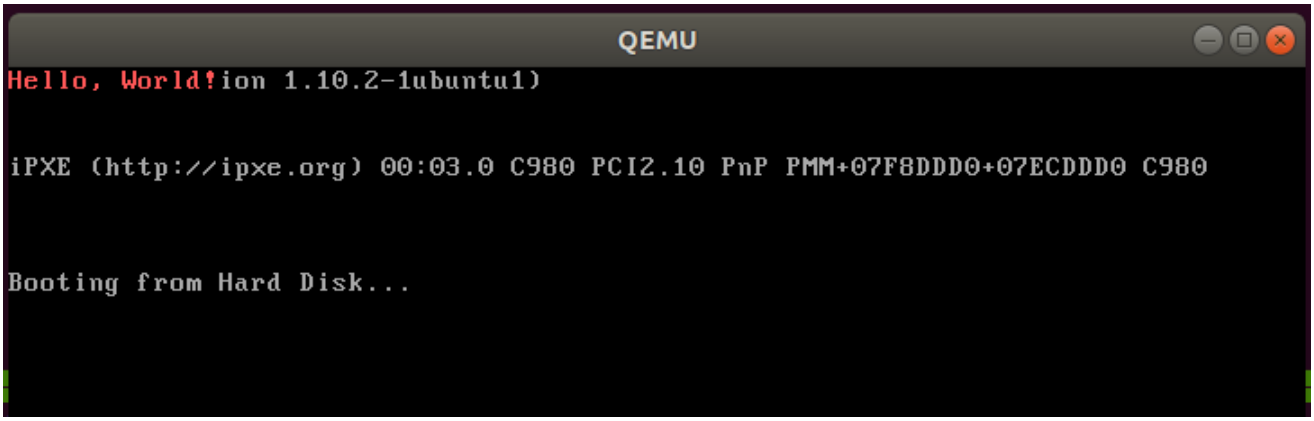
1.3 在保护模式下加载磁盘中的Hello World程序运行

2. 实验过程

在Lab1中, 我完成了实验的全部内容, 接下来我将从三个阶段来介绍完成实验的过程.

2.1 Lab1.1的实现

- 第一次输出`Hello world`: 在理解了`mbr`的实现逻辑后, 在/bootloader/start.s文件中取消注释, 我能通过生成的`mbr.bin`可执行文件, 在终端输出`Hello world!`
- 弥补`Makefile`逻辑上的缺失: 在生成`os.img`的过程中, 会发现有报错信息. 通过阅读报错信息, 发现这是源代码`Makefile`中逻辑有缺失, 导致/utils/genboot.pl脚本缺乏读写权限. 通过在/bootloader/Makefile的`bootloader.bin`规则倒数第二行中插入`chmod +x ../utils/genboot.pl`指令, 便可解决此问题, 并在终端弹窗中输出`Hello world!`



2.2 Lab1.2的实现

- 开启A20地址线: 通过阅读手册与查阅资料, 我了解到A20地址线开启的实现主要有两种方式, 一种是通过系统自陷, 另一种是通过`in/out`端口对0x92端口的1位对A20地址线进行控制. 我采用的是第二种方法

```
inb $0x92, %al                    #启动A20总线
orb $0x2, %al
outb $0x92, %al
```

- 填写GDT表项中每个段的初始值: 根据手册中段描述符的结构解释和手册中有关内容, 对GDT中每个表项进行填写. 其中, 每个段的limit值都设为0xfffff, 即为4G的大小, 除视频描述符外, base值都设为0, 其他参数根据描述来填写

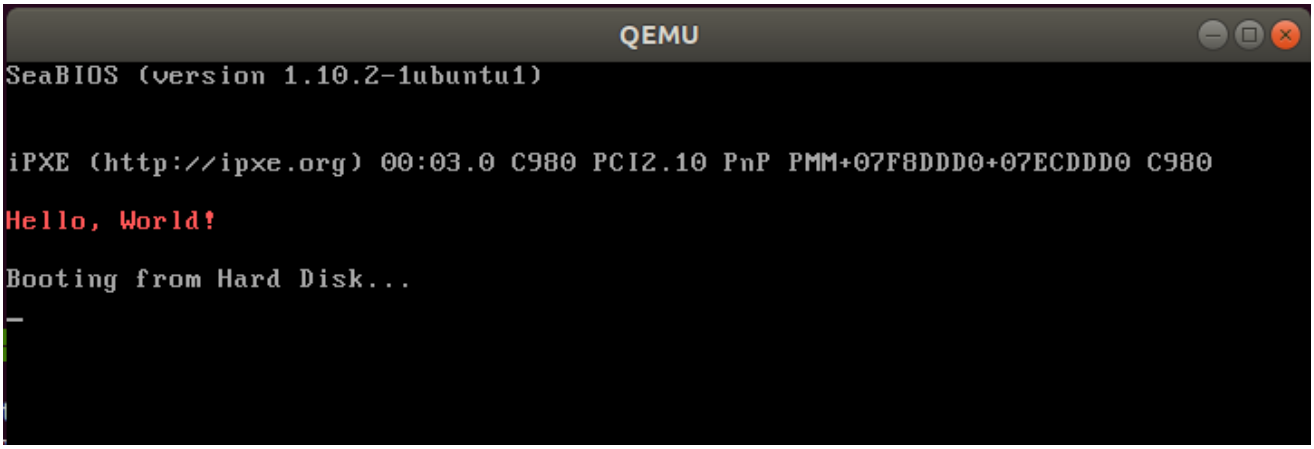
```
gdt:
    .word 0, 0                    #GDT第一个表项必须为空
    .byte 0, 0, 0, 0
    .word 0xffff, 0               #代码段描述符
    .byte 0, 0x9a, 0xcf, 0
    .word 0xffff, 0               #数据段描述符
    .byte 0, 0x92, 0xcf, 0
    .word 0xffff, 0x8000          #视频段描述符
    .byte 0xb, 0x92, 0xcf, 0
```

- 把cr0寄存器的低位设置为1: 具体实现依据指引中的提示, 使用eax寄存器作为中转寄存器.

```
movl %cr0, %eax                   #设置CR0的PE位(第0位)为1
orl $0x1, %eax
movl %eax, %cr0
```

- 初始化cs, ds, es, fs, gs, ss寄存器与栈顶指针esp: 段寄存器中存储的是段选择子, 是寄存器GDTR访问GDT的中介. 根据其结构进行填写, 段选择子0x10存入ds, es, fs, ss寄存器中, 数据段选择子0x18存入gs寄存器, 将栈顶寄存器esp设置到0x7c00.

实现完全后, 建立`os.img`并运行, 能正确输出`Hello world!`



2.3 Lab1.3的实现

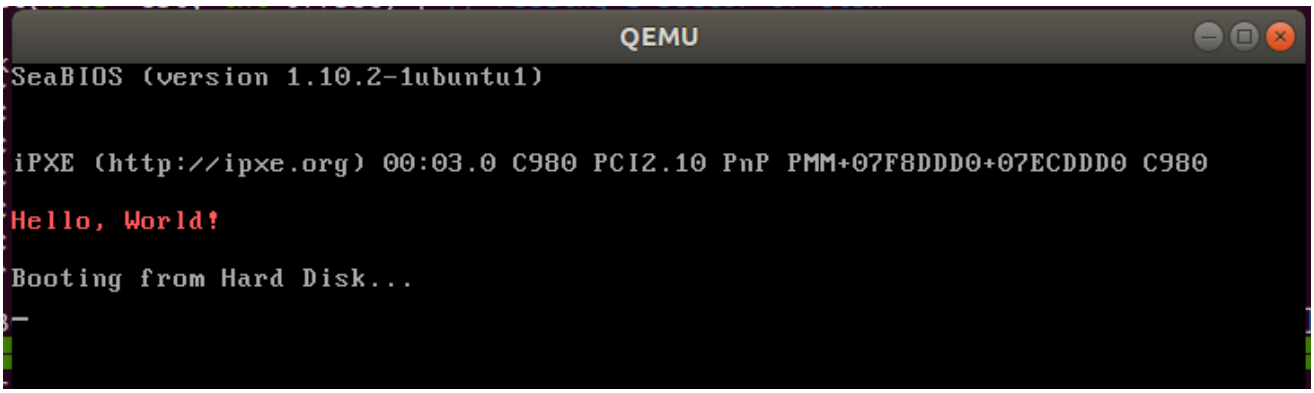
- 将Lab1.2中保护模式的部分适配到Lab1.3的代码中: 具体实现为将上一节中提到的实现内容复制粘贴到1.3代码中即可. 观察到1.3代码中使用了`jmp bootMain`指令, 这代表程序会跳转到该函数中.

- 填写bootMain函数: 具体地, 依据指引, 利用readSect函数将app.bin里面的内容读到0x8c00的地址上. 由于在保护模式执行到jmp bootMain时, 已经填写好GDT表和相关段寄存器, 故可直接调用readSect函数读取, 同时采用内联汇编的方式, 将程序跳转到0x8c00地址执行.

```
void bootMain(void) {
    readSect((void *)0x8c00, 1);
    // ((void (*) (void))0x8c00)();
    asm volatile(
        "movw $0x8c00, %ax\n\t"
        "jmp %ax"
    );
}
```

> *在这一部分的实现中, 我先后采用了两种实现方法: 一种是基于函数指针调用的跳转(上代码框中注释掉的部分), 另一种是使用内联汇编. 两种方法都能正确完成任务, 但经过思考后, 我认为第一种写法经过编译后会用call指令的形式跳转, 执行完后会回到bootMain函数的栈帧中, 而内联汇编中直接jmp, 不会返回. 根据实验要求, 第二种写法应该更符合要求.*

经过正确实现后, 能正确打印出Hello world!



3. 问题回答

3.1 实验手册2.1小节中各名词的含义和其间联系.

1. CPU是中央处理器, 负责执行指令, 进行数据处理和控制计算机的各个部件.
2. 内存是计算机中用于存储数据和指令的临时存储器件.
3. BIOS是基本的输入输出系统, 负责在计算机启动时进行硬件检测和初始化, 并加载操作系统.
4. 磁盘是用于存储数据的一种永久性存储器件, 通过读写磁盘上的扇区来进行数据的读取和写入.
5. 主引导扇区是硬盘中的第一个扇区, 其中存放着分区表, 引导程序等启动操作系统所必需的信息.
6. 加载程序是操作系统的一个子功能, 负责将操作系统的代码从硬盘中加载到内存中, 并开始执行操作系统的初始化程序.
7. 操作系统是控制计算机硬件和软件资源的一组程序, 为用户提供各种服务.

8. BIOS在计算机启动时负责加载操作系统，从存放在磁盘上的主引导扇区中获取引导程序及操作系统启动所需的其他信息，CPU通过执行引导程序,将操作系统的代码从磁盘上的指定位置加载到内存中，并开始执行操作系统的初始化程序，对输入输出设备进行控制等.

3.2 中断向量表是什么?

中断向量表是操作系统中用于应对中断操作的数据结构. 表中的每个表项对应一个中断向量，用于存储对应中断处理程序的入口地址. CPU在接收到中断信号时，会暂停当前正在执行的指令，查找中断向量表中对应的表项，并跳转到对应的中断处理程序的入口地址，执行中断处理程序. 中断向量表从0号单元开始，是用来保存各个中断程序入口地址的一段内存单元，查询可知，其大小为1k.

3.3 为什么段的大小最大为64KB?

- 在早期的计算机系统中，CPU采用16位寻址模式，最多只能够寻址64KB个内存单元. 为了利用CPU此特性，当时计算机内存管理在设计分段模式时，规定每个段的大小最大为64KB.
- 如今CPU的寻址能力大幅增加，远超64KB，但内存管理中依然选择最大64KB的分段方式. 这有两方面的原因:
    - 若一个段过大，访问段内内存单元时，段内偏移地址过大，会造成访问效率的降低
    - 要管理较大规模的内存时，使用分页的模式来实现，一个页面包含了很多段

3.4 说明genboot.pl在检查文件是否大于510字节之后做了什么，并解释为什么这么做.

- 检查文件是否大于510字节之后，脚本做了:
    - 如果文件大于510字节则输出错误信息并退出程序，否则输出提示信息继续执行.
    - 将$buf字符串末尾填充‘\0’直到总长度为510字节.
    - 在$buf字符串末尾添加两个字节0x55和0xAA，这是标准的引导扇区结尾标记.
    - 打开文件，并将修改后的$buf字符串写入该文件中.
- 这么做的原因是: 标准的合法引导扇区的大小为512字节，其中最后两个字节0x55和0xaa是魔数. 在文件末尾添加‘\0’字符填位与魔数，能把指定文件转换成一个合法的标准引导扇区.

3.5 简述电脑从加电开始，到OS开始执行为止，计算机是如何运行的.

- 计算机硬件和BIOS程序进行自检，检查功能是否正常
- CPU从内存中获取第一条指令，启动BIOS
- BIOS从磁盘等存储设备中寻找主引导扇区，并启动引导程序，将引导程序读取到内存中
- CPU执行引导程序，加载操作系统
- 操作系统开始初始化并执行

4. 实验心得

本次实验让我了解了计算机系统从通电到启动操作系统所发生的一系列过程，特别是理解清楚实模式和保护模式的区别与意义，加载程序的概念和意义，同时通过阅读i386手册，也了解到Linux系统的一些具体实现. 这次实验的难点在于Lab1.2，读手册和理解GDT与段寄存器的关系是一个很难的任务，但最终能顺利完成该实验，我也收获满满.