操作系统lab1实验报告

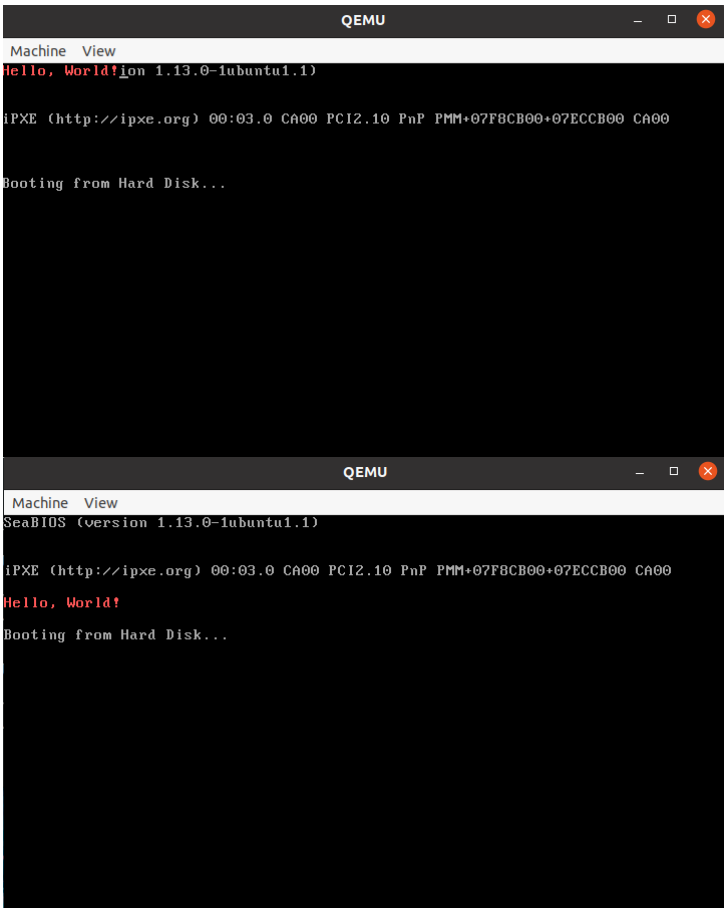
191220156　计算机科学与技术系 张桓　2659428453@qq.com

实验进度

完成了实模式下输出`hello world`，保护模式下输出`hello world`，保护模式下运行磁盘上的`hello world`程序的所有内容。

实验结果

- 上：实模式输出结果
- 下：保护模式输出结果



修改内容

实模式输出

- `bootloader/start.s`中添加了利用中断`int 0x10`输出内容，编写了`displayStr`函数，详情参见该文件的注释内容。

保护模式输出

- `bootloader/start.s`中添加了启动保护模式，设置`GDT`表的汇编代码，从而实现实模式向保护模式的切换。开启保护模式代码：

```
#TODO: Protected Mode Here
    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 $0x01, %eax
    movl %eax, %cr0         #设置CR0的PE位（第0位）为1
    data32 ljmp $0x08, $start32         #长跳转切换至保护模式
```

设置`GDT`表如下：

```
#GDT definition here
    .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
```

- 参照`app.s`文件内容编写输出函数`displayStr`和函数`nextChar`，详情参见该文件注释部分代码。

保护模式加载程序

- 开启保护模式设置`GDT`表同上
- `bootloader/boot.c`中完成了`bootMain`函数，利用`readSect`函数实现将磁盘上的`app`程序读到内存的特定地址，即`0x8c00`处，并且要**跳转执行**该程序，利用内敛汇编即可实现。

```
void bootMain(void) {
    readSect((void *)0x8c00, 1);//load the hello word function in 0x8c00
    asm volatile("jmp 0x8c00");
}
```

思考题

> 阐述小结标题中各种名词的含义和它们之间的关系

- 名词含义：
  - CPU是中央处理器，功能主要是用于解释计算机指令和处理计算机软件产生的数据
  - 内存用于暂时存放CPU中的运算数据，是外存和CPU沟通的桥梁
  - BIOS是电脑启动时加载的第一个软件，是一组固化到主板上ROM芯片上的程序，保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序等等
  - 磁盘是存储数据的设备，并且断电后能保持数据不丢失
  - 主引导扇区是位于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`
  - 加载程序用于将操作系统代码和数据加载到内存，并且跳转到操作系统的起始地址
  - 操作系统是管理计算机硬件和软件资源的程序
- 关系

  本次实验关注计算机启动后操作系统是如何开始运行的，因此这里主要阐述这一部分工作各个部件的简单运行模式：计算机启动后，会首先加载 `BIOS` 程序（`BIOS` 存放在一个断电后不会丢失内容的 `ROM` 中），`BIOS` 程序进行硬件自检，如果硬件有问题启动中止，如果没有问题 `BIOS` 会加载磁盘上的主引导扇区（位于0号柱面，0号磁头，0号扇区对应的扇区）到内存 `0x7c00` 的位置，并且根据末尾两个字节的魔数 `0x55` 和 `0xaa` 来判断是否成功，如果魔数不等于则去其他设备加载。加载好 `MBR` 后 `BIOS` 会跳转到 `0x7c00` 的位置执行 `MBR` 代码（实际上是加载程序代码），加载程序将操作系统代码和数据加载好后切换成保护模式，然后将控制权交给操作系统，至此 `OS` 成功启动。

> 现代操作系统中，主引导扇区和加载程序往往不一样，请思考一下为什么？

- 前面的分析我们知道了计算机启动后大概流程是先执行 `BIOS`，`BIOS` 会加载 `MBR` 然后执行它，`MBR` 再去加载 `OS` 代码，但查阅资料得知，`MBR` 是包含三部分内容（引导程序、分区表及分隔标识，`MBR` 总计512字节；其中引导程序最多占446个字节，`BIOS` 自检后，就会将 `MBR` Load进内存。也就意味着引导程序被激活，分区表信息已经加载到内存，同时也意味着对系统的控制权从 `BIOS` 过渡到 `GRUB`。`GRUB` 是 `GRand Unified Bootloader` 的缩写，它是一个多重操作系统启动管理器。用来引导不同系统。`GRUB` 是一个系统引导程序，分为两个阶段第一个阶段代码就算MBR中的引导程序部分，第二阶段需要到 `/boot` 分区读系统内核和配置文件。

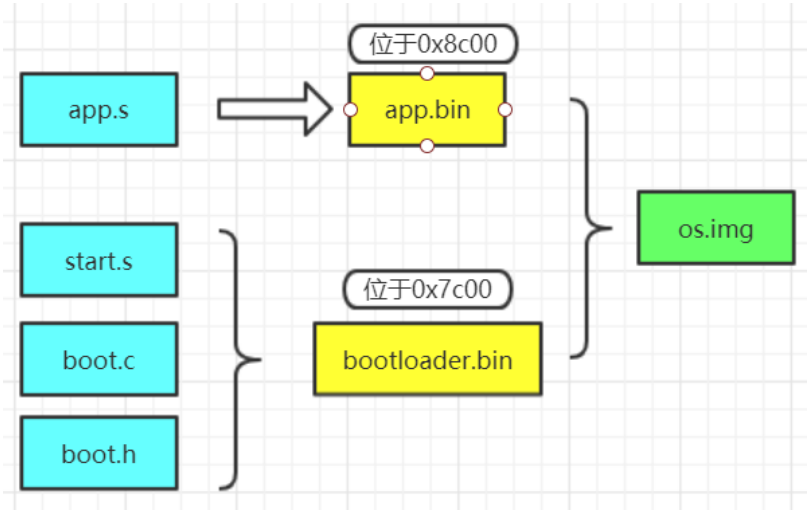
  可以看到，这些工作是相对比较复杂的，全给引导扇区去做 `512` 字节可能不够，因此要在中间加一个东西，它没有512字节的限制，把那些工作交给它来做，`MBR` 则负责加载它并将控制权交给它。所以，`BootLoader` 应运而生了。更进一步讲，`MBR` 的产生也是由于 `BIOS` 太小，工作能力有限导致的，一级一级加载，最终实现操作系统的加载。

实验心得

- 这次做实验之前虽然感觉自己对加载操作系统的过程理论上已经有个大概认识了，但是开始看到那几个文件时还是一头雾水，不知道每个文件的实际用途和它们之间的关系。后来在浏览 `guide` 时看到

  > 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）

  点开看了一下 `Makefile` 文件，发现文件里清楚地描述了 `app.bin` 文件的由来，当然包括链接成 `app.elf` 时确定的入口**0x8c00**。于是我又点开了 `bootloader` 文件夹里生成 `bootloader.bin` 的 `Makefile` 以及总文件夹里生成 `os.img` 的 `Makefile`，分析之后文件之间的关系变得很清楚：



我的工作也定位到实现`bootloader`中的`boot.c`和`start.s`文件。这也启示我遇到不熟悉的项目时，可以通过`Makefile`文件来快速了解文件的关系结构。

- 在开启保护模式时不知道怎么开启`A20`，查阅资料解决，参考文章。

设置`GDT`表时，遇到的问题是不知道怎么设置段描述符，写这个的时候想到之前`PA`也是实现的`i386`机器，虽然`kernel`没有让我们亲自实现，但是一定可以有些参考。于是我找到了之前的PA代码，发现的确有对各个段描述符的设置代码，但是是用C语言实现的。如下：

```
#define MAKE_NULL_SEG_DESC   \
        .word 0, 0;              \
        .byte 0, 0, 0, 0
#define MAKE_SEG_DESC(type,base,lim)                        \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
gdt:
        MAKE_NULL_SEG_DESC                      # empty segment
        MAKE_SEG_DESC(0xA, 0x0, 0xffffffff)  # code
        MAKE_SEG_DESC(0x2, 0x0, 0xffffffff)  # data
```

结合为视频段的基地址为`0xb8000`，经过计算顺利设置成功。这也启示我之前的先导课程一定不能和现在割裂开来，它们都是计算机系统的一部分，应该融会贯通，更何况本实验本身是在实现`OS`之前的工作，其实更贴近`ICS`的教学内容。

- 在用`readSec`函数将`hello world`程序加载到内存的`0x8c00`处后，应该跳转到那里执行，我的实现是利用内敛汇编`asm volatile("jmp 0x8c00");`加载完后直接跳转到那个地方执行。和同学的交流过程中得知，其实也可以用纯C语言解决，就是函数指针，如下

```
void (*my_function)(void);//定义函数指针my_function
my_function = (void(*)())(0x8c00);//将0x8c00强制类型转换成函数指针赋值给my_function
my_function();//执行函数
```

由于很久没用过函数指针，还参考了一篇文章，总结的很全面：参考文章

建议

- 这次实验刚接手时还是很蒙的，之前的`PA`课程虽说有点基础，但是让我直接写汇编代码设置`GDT`表还是有些困难，还有`A20`也是全新的概念，以及内联汇编也是我翻看之前的`PA`框架学到的，如果可以的话可以在教程里添加一些比较权威的参考文章，当然不提供也确实能锻炼到我的`STFW`能力😄