OS Lab1 实验报告

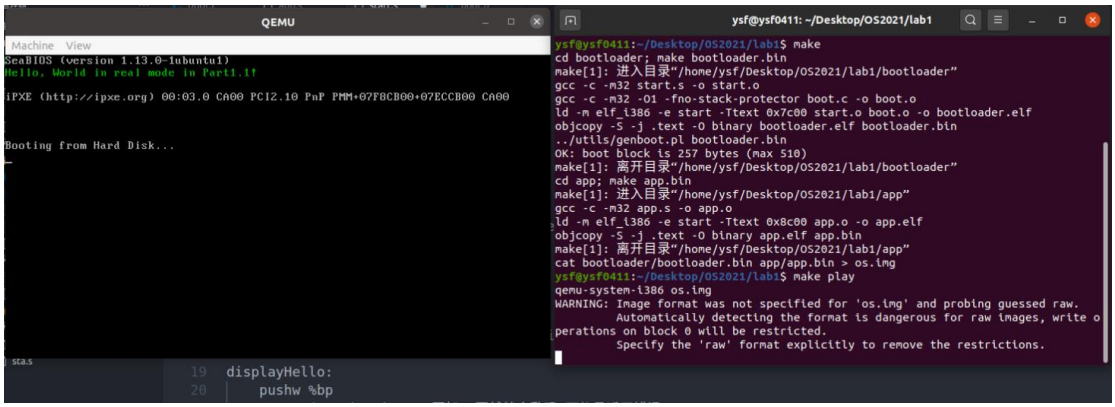
2 班 191300073 AI 杨斯凡 769226877@qq.com

一、试验进度

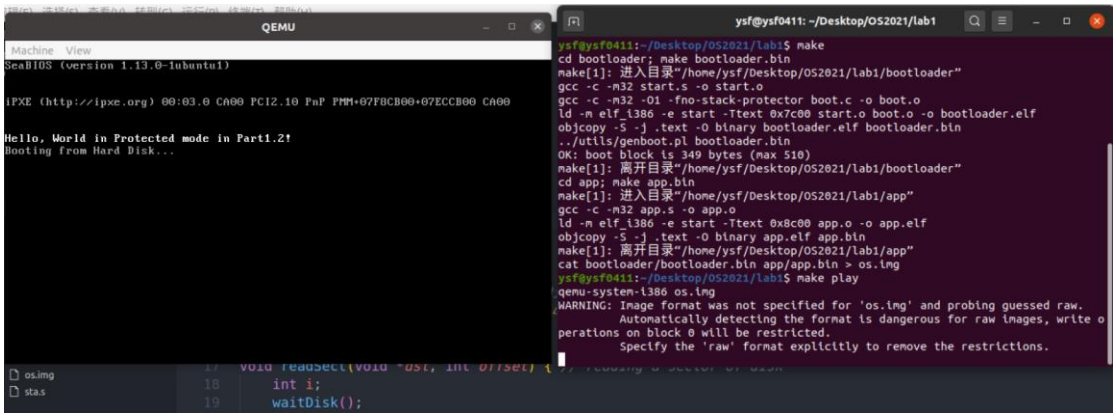
我完成了所有内容

二、实验结果

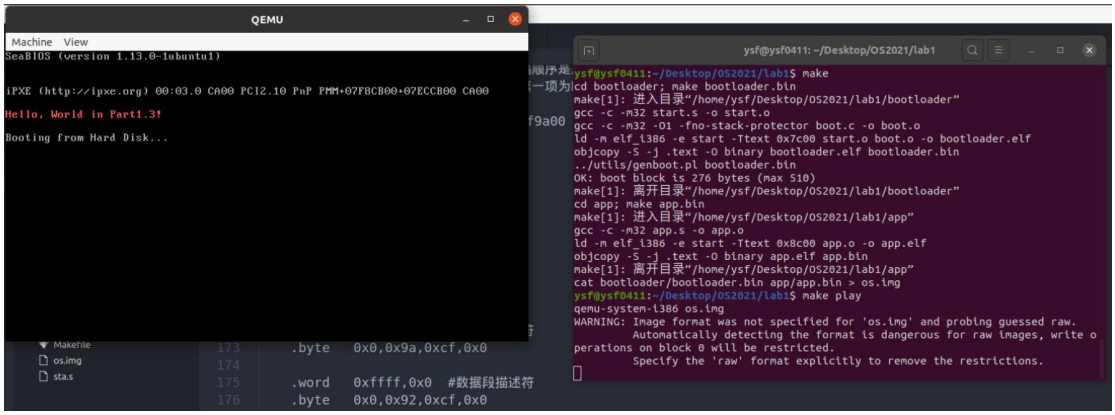
为了区分不同的模块，我修改了打印出的字符内容和颜色，使得每个任务的字符更加明朗。



实模式下打印字符，我将颜色设置为了绿色



保护模式下打印字符，我将颜色设置成了白色，并且闪烁



保护模式下加载程序（之后改了回去）

三、修改代码位置

首先阅读整个框架代码后，我对整个的流程有了一个大致的认知，并且对每个函数的的功能进行了了解，app 文件夹下存放的是加载的程序，app.s 是加载的程序的汇编指令，BootLoader 中存放的是加载程序，start.s 中分有 16 位和 32 位的模式。在 1.1 中，我仿照 index.md 中的指令修改了第一个部分的代码，利用了系统中断进行字符的打印加上了输出到终端的部分，并且为了区分不同模块，我想对颜色进行修改。

在网上查阅相关资料之后，学习了实模式下字符打印的过程，理解了每个寄存器中存储的数据的作用，我按照颜色规定对输出的字符的颜色和位置进行了更改。

第二部分我在阅读了教程的情况下，完成了保护模式开启的指令，关闭中断，打开 A20 数据总线，加载 GDTR ，设置 CR0 的 PE 位，长跳转设置 CS 进入保护模式，初始化一部分寄存器，由于中断的关闭，不能像 1.1 中的那样直接输出到终端，于是我模仿 app.s 中的指令完成了到终端的输出，并且将字体修改为白色，并且这里出了一个小 bug，在多次阅读教程之后得到了解决。

在做 1.3 的时候，当时并不知道有教程，因此只靠着阅读 i386 手册和 Google 来逐步的进行做，期间出了一系列 bug，在询问助教之后得到了解决。

四、系统启动过程

在阅读了 i386 手册和查阅相关资料之后可以得知：

系统在启动的时候，首先会在实模式下工作，CPU 只能访问 1MB 内存，这 1M 空间最上面的 0xF0000 到 0xFFFFF 这 64K 映射给 ROM，ROM 存有了 BIOS 的代码，第一条指令就会指向 0xFFFF0

然后会进行 BIOS 的自检工作，如果每个硬件都存在并且都可以正常工作，并且建立中断向量表和中断服务，然后根据 BIOS 中设置的系统启动顺序来搜索用于启动系统的驱动器，接着就加载主引导扇区的前 512 个字节到内存地址 0x7c00 处。

然后进行跳转，进入 BootLoader 时期，加载 core.img。然后再切换到保护模式，并且开启分段和分页机制，然后加载操作系统并启动内核执行。

五、问题解答

CPU 是计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。CPU 对计算机的所有硬件资源进行控制调配、执行通用运算。CPU 是计算机的运算和控制核心。计算机系统中所有软件层的操作，最终都将通过指令集映射为 CPU 的操作。简单来说，CPU 的主要功能是解释计算机指令以及处理计算机软件中的数据。

内存用于暂时存放 CPU 中的运算数据，与硬盘等外部存储器交换的数据。它是外存与 CPU 进行沟通的桥梁，计算机中所有程序的运行都在内存中进行，

BIOS 是英文"Basic Input Output System"的缩写，也就是"基本输入输出系统"。它是一组固化到计算机内主板上一个 ROM 芯片上的程序，它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，它可从 CMOS 中读写系统设置的具体信息。其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。此外，BIOS 还向作业系统提供一些系统参数。系统硬件的变化是由 BIOS 隐藏，程序使用 BIOS 功能而不是直接控制硬件。

磁盘是指利用磁记录技术存储数据的存储器。磁盘是计算机主要的存储介质，可以存储大量的二进制数据，并且断电后也能保持数据不丢失。

主引导扇区位于硬盘的 0 磁头 0 柱面 1 扇区，包括硬盘主引导记录和分区表。其中主引导记录是检查分区表是否正确以及确定哪个分区为引导分区，并在程序结束时把该分区的

启动程序。总共占 512 个字节。

加载程序是操作系统的一部分，主要功能是程序的加载，步骤包括，读取可执行文件，将可执行文件的内容写入存储器中，之后开展其他所需的准备工作，准备让可执行文件运行。当加载完成之后，操作系统会将控制权交给加载的代码，让它开始运行。

操作系统是管理计算机硬件与软件资源的计算机程序，同时也是计算机系统的内核。操作系统需要处理管理与配置内存、决定系统资源供需的优先次序、控制输入与输出设备、操作网络与管理文件系统等基本事务，并且提供一个让用户与系统交互的操作界面。

主要关系：

系统在启动的时候，首先会在实模式下工作，CPU 只能访问 1MB 内存，这 1M 空间存有 BIOS 的代码，第一条指令就会指向 0xFFFF0，然后 CPU 开始执行，这也是 0 号进程。然后会进行 BIOS 的自检工作，如果每个硬件都存在并且都可以正常工作，并且建立中断向量表和中断服务，然后根据 BIOS 中设置的系统启动顺序来搜索用于启动系统的驱动器，接着就加载主引导扇区的前 512 个字节到内存地址 0x7c00 处，然后会按照用户程序的需求对程序加载运行。

六、代码解读

Lab1 的实验框架主要由三个部分组成: app，bootloader，utils。

在本次实验的框架中使用了 qemu 的模拟的硬件进行指令的执行。

阅读 Makefile 之后，可以发现，整个实验的框架是先执行 BootLoader 中的 make 和 app 中的 make，然后再把得到的.bin 文件拼接形成 os.img 镜像文件。

BootLoader 中的 Makefile 会首先对目录下的各个文件进行编译得到.o 文件，然后再将.o 文件进行连接生成 elf 文件，app 目录下的 Makefile 的作用也是如此。

而 genboot.pl 这个脚本文件首句是打开 bootloader.bin 这个文件，读取 1000 个字节，最多读取 510 个字节，然后输出，不足的用 0 补上，0x55，0xaa 是 MBR 的结束标志，执行完毕后 bootloader.bin 的大小为 512 字节，模拟了磁盘 0 号扇区的 512 字节。

七、Bug 和解决

在写 1.1 的时候，我借鉴了 index.md 中的代码，但是不太清楚其中的原理，然后在 CSDN 上学习之后对其中的原理了解了，然后可以修改文字的位置和颜色，写完了汇编指令，我进行 make 的时候系统提示 4(%sp)是不合法的，我不是很能理解这个事情，于是我去 Stack Overflow 上查询之后，给出的解释是会产生覆盖的问题，因此得改为 esp，改过之后成功通过编译，然后我多次修改参数找到了最合适的位置。

但是在 1.2 中的时候，在写 GDT 的时候，我没有看教程，于是使用了 dword 这个类型进行编写，但是编译不能通过，我个人感觉是加了 m32 的原因，参考了教程之后，计算了每个段的 GDTR 之后，进行了数据的编写，此处要注意汇编中的数据存储的方式是反过来的，因此要先写低位，这个部分编写完成之后，输出却出现了乱码，多次修改之后我发现是 bootmain 的问题，完成了那个部分之后就可以正常显示。

在 1.3 中，调用 elf 这个函数指针即可完成显示文字程序的加载和文字的输出。

八、心得体会

因为系统的启动过程在上学期 ICS 的课程中有所学习，并且通过上学期 ICS 的 PA 我对 VGA 有了个初步的认识，因此编写代码比较容易。但是刚开始的时候没有发现有实验教程那个文件，只依靠着 i386 手册和 Google 进行编写，难度比较大。因此，在下次开始写 lab 之前一定会对教程进行充分学习后再开始。

但是，对于本次实验，还是有一个疑问，gdt 的部分，为什么全部使用 word 进行存储，而是要使用 byte。（我初步的想法是觉得会让结构更清晰？）