

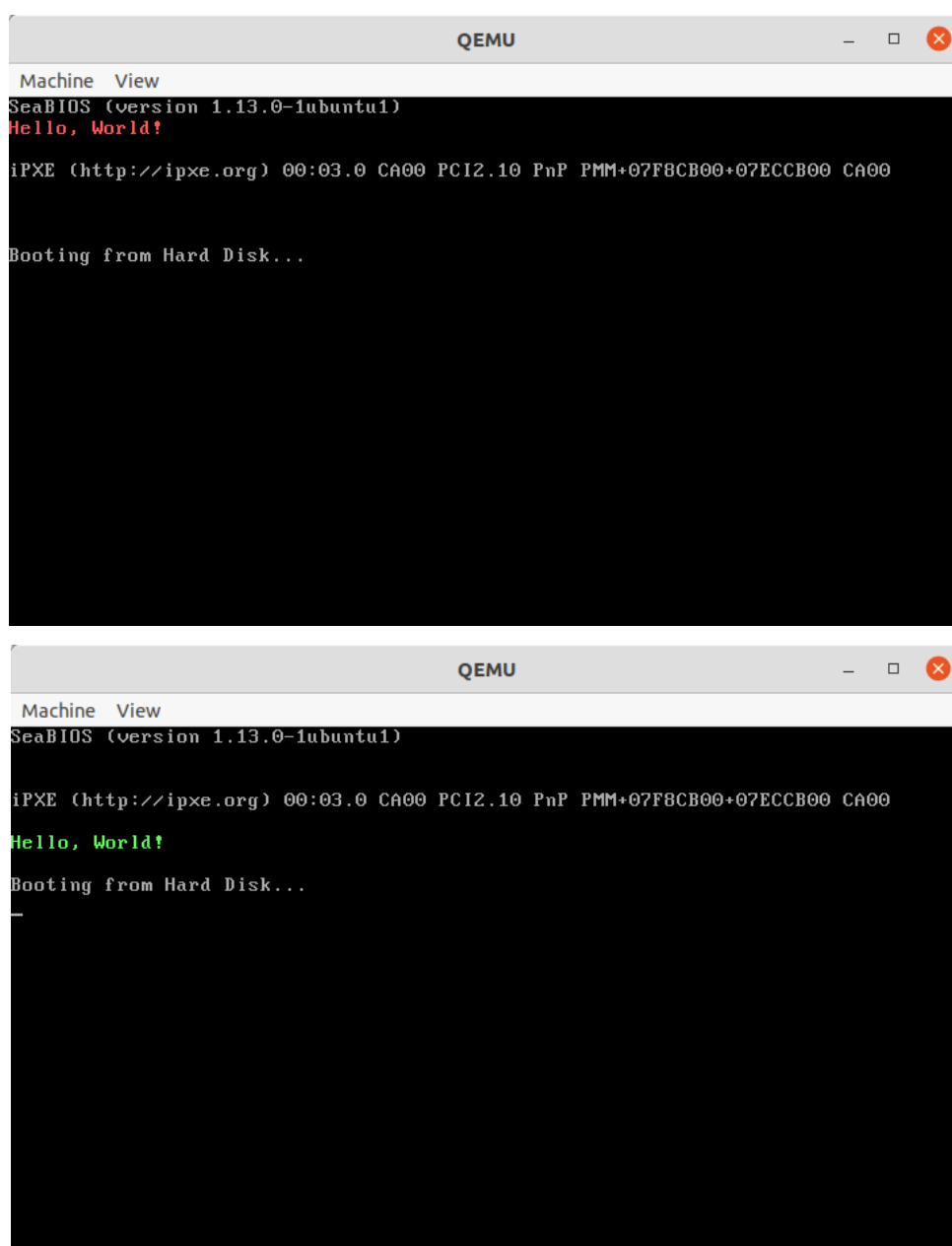
# lab1 实验报告

李培佳 191300029 [191300029@smail.nju.edu.cn](mailto:191300029@smail.nju.edu.cn)

## 1.实验进度

完成了所有内容，即实模式下在终端中打印 Hello, World!，从实模式切换至保护模式，并在保护模式下在终端中打印 Hello, World!，以及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!

## 2.实验结果



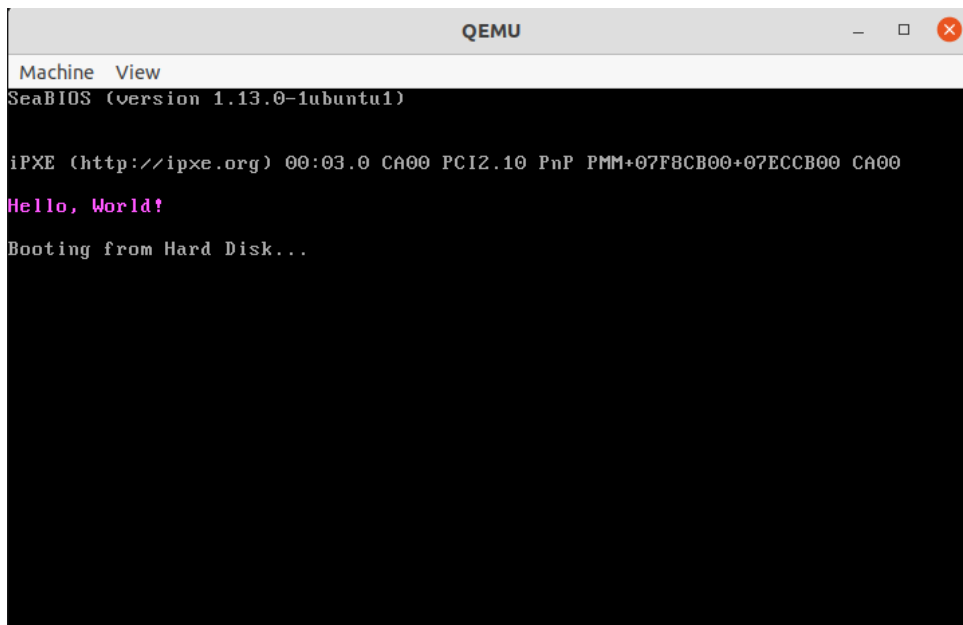
The image displays two screenshots of a QEMU virtual machine terminal window. The window title is "QEMU". The terminal output shows the following sequence of events:

```
Machine View
SeaBIOS (version 1.13.0-1ubuntu1)
Hello, World!

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
```

The first screenshot shows the initial boot sequence, including the SeaBIOS version, the "Hello, World!" message, the iPXE version, and the booting from the hard disk. The second screenshot shows the same sequence, but with the "Hello, World!" message appearing in green text, indicating successful execution in protected mode.



三张图片分别对应三个任务，并且从字体颜色上进行了区分，实模式下 `%bx` 寄存器控制颜色，并且修改了 `%dx` 进行换行，保护模式下通过修改 `%ah` 改变了颜色，没有修改用于控制位置的 `%edi`

## 3.修改的代码位置

### 3.1 Real Mode

在原代码框架下在 `start.s` 中 `/*Real Mode Hello World*/` 代码块里将字符串长度 `13` 和字符串 `message` 入栈，调用 `displayStr` 进行打印，`displayStr` 的实现仿照了实验课给出的资料 `index.md` 中的实现，首先构建栈帧，设置好控制字体颜色，字体位置的寄存器，使用 `int $0x10` 进行中断打印。

### 3.2 Protect Mode

在原代码框架下在 `start.s` 中 `/*Protect Mode Hello World*/` 代码块里

根据lab1指导文件中的描述

“关闭中断,打开 `A20` 数据总线,加载 `GDTR`,设置 `CR0` 的PE位(第0位)为 `1b`，通过长跳转设置 `CS` 进入保护模式,初始化 `DS`, `ES`, `FS`, `GS`, `SS`”

进行了实现，在 `.code16` 代码段中，关闭中断已经给出即 `ccli`，打开 `A20` 总线使用 `System Port 0x92`，加载 `GDTR` 也已经给出，将 `CR0` 的PE位置为1，将 `CR0` 的值先移入 `eax`，通过 `or` 指令将 `eax` 最低位置为1，最后移回 `CR0`。

在 `.code32` 代码段中，此时已经进入保护模式，设置 `gs` 和 `esp`，剩余步骤为打印字符，由于进入保护模式后，无法使用 `bios` 中断进行打印，考虑到 `app.s` 中的程序也是运行在保护模式下的，将 `app.s` 中的 `displayStr` 放在这里最终成功打印出 `Hello, World!`

具体实现方法为将每个字符放入 `al` 中，并使用 `movw %ax, %gs:(%edi)` 来写入显存，循环直到字符串结束。

注：修改了字体颜色

### 3.3 Protect Mode Loading

进入保护模式的方法和3.2中相同，不同的是3.2直接打印字符，3.3中需要跳到 `boot.c` 中的 `bootmain`，在 `bootmain` 定义一个函数指针，指向 `app.s` 的入口，可以在 `/app/Makefile` 中看到即 `0x8c00`，并利用 `readSect` 函数读取1号扇区，最终执行 `app.s` 中的程序，打印字符的具体过程和3.2相同。

### 3.4 关于GDT描述符

在保护模式下，需要引入全局描述符GDT，本次实验的框架代码中使用了 `.word, .byte` 分别用于生成字和字节

这种表示GDT的方式的规则如下：

```
.word (((lim) >> 12) & 0xffff), ((base) & 0xffff);  
  
.byte (((base) >> 16) & 0xff), (0x90 | (type)), (0xC0 | (((lim) >> 28) & 0xf)),  
(((base) >> 24) & 0xff)
```

`type` 为某个描述符的权限类型，每个类型都有其对应的16进制数值。

以代码段为例，代码段的基地址 `base` 为 `0x0`，`lim` 为 `0xffff`

`.word` 计算出来为 `0xffff, 0x0`，代码段的属性为可执行和可读，`.byte` 计算出来为 `0, 0x9a, 0xcf, 0`

数据段和视频段同理。

### 4.一些感想

开学前我组装过一台电脑，装好开机，主板上的boot指示灯长亮，最终给主板刷了两次bios解决了。由于这件事，在第一节操作系统课上，我产生了一个疑问，操作系统和bios的关系是什么？当时只是有一个想法，没有深入思考，没有想到第一次实验就涉及到了这个问题，随着这次实验做完，也让我对pc加电启动的过程有了一定的了解。当然，现代的操作系统更加复杂，我所了解的不过是冰山一角。希望在接下来的学习中，能够更加深入的了解操作系统和硬件的联系。

**以上是本次实验报告的全部内容，感谢阅读！**