

OS-lab1

211275009 陈铭浩

email: 211275009@smail.nju.edu.cn

实验进度：我完成了所有内容

相关资料思考与总结

1. 磁盘的主引导扇区（MBR）：

- 首先，整个硬盘上一般有很多的盘片组成，沿着盘片半径的方向被划分成了很多同心圆，这就是磁道，每条磁道由很多的扇形区域组成，叫做扇区（扇区是从磁盘读出和写入信息的最小单位，通常大小为512字节），不同盘片上的同半径磁道组成了柱面，每个盘片一般有上下两面，分别对应1个磁头
- MBR就是0号柱面，0号磁头，0号（相对）扇区对应的扇区
- MBR末尾两字节为魔数 0x55 和 0xaa。每次执行系统引导代码时都会检查MBR主引导扇区最后2字节是否是"55 AA"，若是，则继续执行后续的程序，否则，则认为这是一个无效的MBR引导扇区，停止引导系统。
- MBR包含3个部分，引导信息、分区表、结束标志（魔数）。

2. 为什么在现代操作系统中，主引导扇区（MBR）和加载程序（bootloader）不一样？

- MBR需要做的事情主要有加载内核、做好进入保护模式的准备工作（设置GDT等），然后跳入保护模式，最后再转移到内核部分开始执行等。这些工作是相对比较复杂的，全给主引导扇区去做那么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。这就是加载程序（bootloader）

3. CPU、内存、BIOS、磁盘、主引导扇区、加载程序、操作系统之间的关系：

- 在计算机启动时，BIOS首先被执行，负责硬件的初始化和自检。

- BIOS之后会寻找启动设备（通常是硬盘），并读取主引导扇区（MBR）的内容到内存中。
- 主引导扇区包含了加载程序的代码，BIOS将控制权交给这个加载程序。
- 加载程序读取磁盘上的操作系统文件，将操作系统的内核加载到内存中，并开始执行。
- 操作系统接管对硬件的控制，管理内存、CPU、磁盘和其他设备，为应用程序运行提供环境。

4. CS=0x0000:IP=0x7C00 表示什么意思？

- 表示在x86架构的计算机中，代码段寄存器（Code Segment, CS）的值为0x0000，而指令指针寄存器（Instruction Pointer, IP）的值为0x7C00。这种表示方式用于确定CPU开始执行代码的物理内存地址。
- 实模式下，物理地址 = CS << 4 + IP

5. SS, SP, BP 三个寄存器

- SS:存放栈的段地址；
- SP:堆栈指针寄存器SP(stack pointer)，存放栈的偏移地址；
- BP:基数指针寄存器BP(base pointer)是一个寄存器，它的用途有点特殊，是和堆栈指针SP联合使用的，作为SP校准使用的，只有在寻找堆栈里的数据和使用个别的寻址方式时候才能用到
 比如说，堆栈中压入了很多数据或者地址，你肯定想通过SP来访问这些数据或者地址，但SP是要指向栈顶的，是不能随便乱改的，这时候你就需要使用BP，把SP的值传递给BP，通过BP来寻找堆栈里数据或者地址。一般除了保存数据外,可以作为指针寄存器用于存储器寻址,此时它默认搭配的段寄存器是SS-堆栈段寄存器。

6. 函数调用的汇编实现细节

- 关于栈的增长方向：栈是从高地址向低地址方向增长的。即栈底处于高地址，栈顶处于低地址，每次入栈时需要将栈指针减小，每次出栈时需要将栈指针增加。（从栈顶出入栈）
- 在 call 指令执行时，CPU 会自动将返回地址压栈。这个返回地址是 call 指令后的下一条指令的地址
- 在函数执行完毕，执行 ret 指令时，会发生以下步骤：
 - a. 弹出返回地址：ret 指令首先从当前堆栈顶部弹出（Pop）返回地址。这个地址是之前通过 call 指令自动压入堆栈的。

- b. 跳转执行：CPU 使用弹出的地址作为下一条执行指令的地址，即跳转回到调用函数的地点，继续执行后续指令。
- c. 调整堆栈指针：由于返回地址被弹出，堆栈指针（**sp** 或 **esp**，取决于是在16位还是32位模式下）会自动向上移动，指向堆栈中下一个可能的位置。

7. .word和.byte是什么？

- **.word**用于定义一个或多个“字（word）”大小的数据。在x86体系结构中，一个“字”通常是16位（2字节）长。当你使用**.word**指令时，你可以指定一个或多个16位的初始值来初始化内存中的空间。例如，**.word 0x1234**会在内存中分配2字节的空间，并将其初始化为**0x1234**。
 - **.byte**用于定义一个或多个“字节（byte）”大小的数据。每个字节是8位长。使用**.byte**指令时，你可以指定一个或多个8位的初始值来初始化内存中的空间。例如，**.byte 0xFF**会在内存中分配一个字节的空間，并将其初始化为**0xFF**。
-

实验内容与结果

1. 实模式Hello World程序

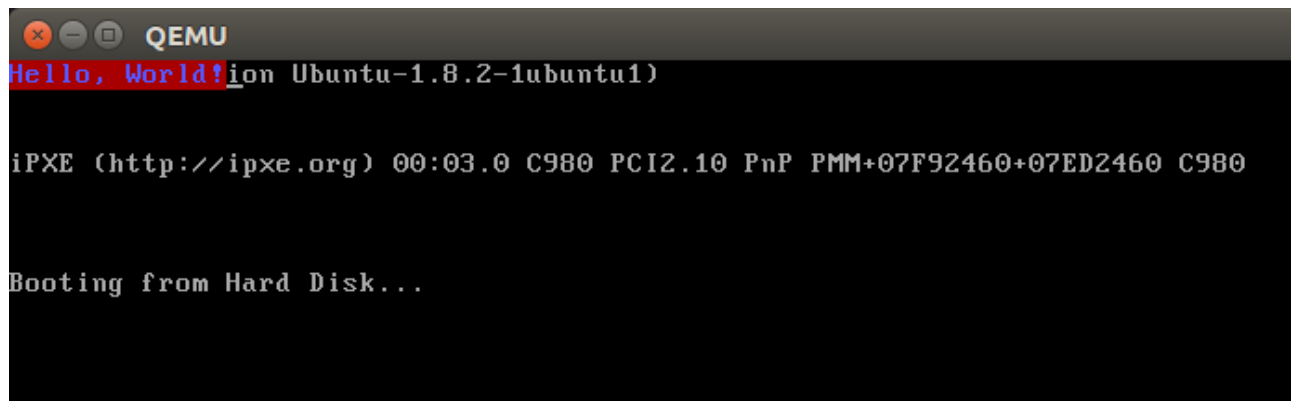
1.1. 基本思路

- 利用BIOS中断中的**int \$0x10**，**AH=13**显示字符串；**AL=0x01**，使光标跟随移动；**BL=0x49**，使字符显示为亮蓝色，背景为红色（<https://blog.csdn.net/cgzldfend/article/details/80325125>）（<https://blog.csdn.net/longintchar/article/details/70183677>）；**BH=0x00**，表示页号为0；**DX=0x0000**，表示起始行列都为0；在**bp**中存放串地址（通过**esp+4**得到，因为在函数调用时自动压入了放回地址，函数刚开始又压入了旧的**bp**值）；在**cx**中存放串的长度（通过**esp+6**得到，同理）
- 因此在代码中新增**printStr**函数，基于上述思路实现打印hello world，并在**start**中将串长度和串地址压入栈中，并调用**printStr**函数

1.2. 实验结果

在lab1文件夹下执行如下命令：

```
1  chmod +x utils/genboot.pl  # 需要首先将genboot.pl文件提权
2  make os.img
3  make play
```



2. 实模式切换保护模式，并在保护模式下打印hello world

2.1. 基本思路

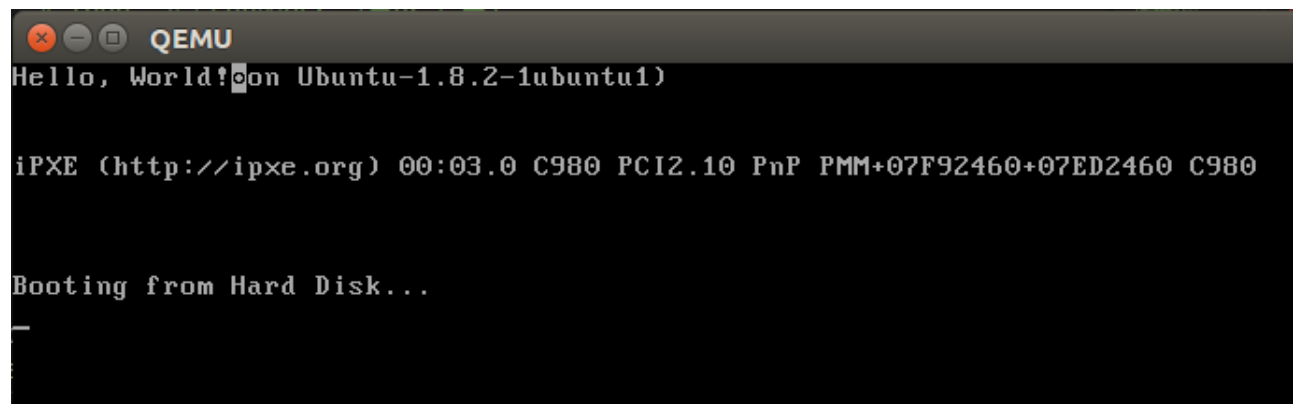
1. 关闭中断：在进入保护模式之前，应该关闭中断，以避免在模式转换过程中不可预测的中断处理。这可以通过清除中断标志（IF）来实现，使用指令cli（Clear Interrupt Flag）。
2. 设置CR0的PE位为1：在切换到保护模式之前，需要设置控制寄存器CR0的保护使能（PE）位。这可以通过读取CR0，修改PE位，然后写回CR0来完成。PE位是CR0的最低位（位0）。因为CR0有32位，故需要用movl。
3. GDT表：
 - a. 代码段与数据段的基地址都为 0x0，视频段的基地址为 0xb8000
 - b. 段界限统一设置为0xFFFFF
 - c. graphics看作一种特殊的数据
 - d. G=1, D/B=1, AVL=0, P=1, DPL=00，其余位按照是代码段还是数据段分别取对应的值

4. 打印hello world: 这一部分直接操作显存来输出"Hello, World!"字符串。文本模式下的显存通常位于物理地址0xB8000，其中每个字符占用两个字节：字符的ASCII码和字符属性（如颜色）。通过将字符串地址加载到ESI，显存的起始地址加载到EDI，然后通过一个循环，逐字符地将字符串复制到显存。每复制一个字符后，EDI递增2，跳过属性字节（只设置了字符码，没有设置属性，因此字符将以默认颜色显示）。一旦字符串结束，即AL为0（由test %al, %al检查），循环终止，并进入一个无限循环，等待系统重置或进一步的操作。

2.2. 实验结果

在lab1文件夹下执行如下命令：

```
1 make os.img
2 make play
```



3. 保护模式下加载磁盘中的hello world程序并运行

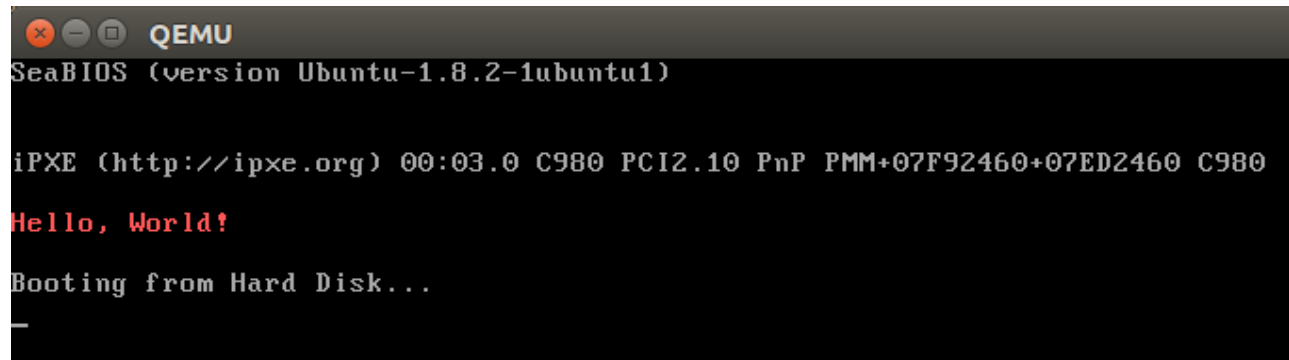
3.1. 基本思路

1. start.s中增加代码除打印hello world外均与任务2中相同，不再赘述
2. 从app文件夹下的makefile中可以看出app程序的入口地址是0x8c00，因此需要将磁盘MBR之后扇区（即1扇区）中的程序加载到0x8c00
3. 在bootMain中调用readSect函数后，执行该程序即可

3.2. 实验结果

在lab1文件夹下执行如下命令：

```
1 make os.img  
2 make play
```



The screenshot shows a terminal window titled "QEMU" with standard window controls. The text inside the terminal is as follows:

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