

Copy of OSlab

😁 操作系统lab1【如何启动一个OS】

lab1的介绍

在一段时间的学习之后，相信大家已经知道操作系统大概是个啥（但我相信是一知半解）。我们的课程实验就是让大家深入到实际编程中，来探索 OS 的奥妙。我们一共分成5次实验，一步一步地带领大家从入门到入土（x）。

下面的一些介绍可能会比较啰嗦，但这应该会帮助大家更好地从 ICS 过渡到 OS。没学好 ICS 的同学也不要着急，我相信在你进行第一次实验过后，你不仅会复习到很多 ICS 的知识，也会学到新的东西！

本次实验需提交的内容：

- 15个exercise（编号从1到15），答案按照编号顺序写在实验报告里即可。
- 2个task，即代码任务，提交时请切换到load-os分支，打包时候记得把.git文件夹带上。并通过注释或在实验报告里说明思路。
- 1个challenge，需要写代码，请把代码提交在report文件夹里，并在报告里说明你的做法。
- 实验报告，提交在report文件夹里。

必备的工具

没有基础知识是没办法开工的，所以第一步是先熟悉一下我们要用到的工具。放轻松，这个部分不需要写什么代码。这一部分是帮助我们了解**实验平台**和**一些有用的工具**（注意，我们常说的是系统，你可以理解成系统包括了硬件和系统软件两个部分，而操作系统则属于系统软件）。同时，我们也需要了解 QEMU 和 GDB 的使用，来帮助我们更好地进行实验。

了解x86汇编文件

好，下面让我们看官方手册学习汇编：[PC Assembly Lanugage \(mit.edu\)](https://www.cs.cmu.edu/~rmoore/15-445/lectures/01-assembly.html)。

哈哈，开玩笑的！实际上都是些上学期学的比较基础的指令。

以下面这段代码为例：

```
1 # test1.s
2 .section .data
3 .section .text
4 .globl _start
5 _start:
```

```
6     movl $1, %eax
7     movl $3, %ebx
8     int $0x80
```

这段汇编代码，我相信大家能看懂一些，但我还是要详细地解释一下。

我们先对它进行汇编操作：

```
1 gcc -c test1.s -o test1.o
```

然后再进行链接操作：

```
1 gcc test1.o -o test1
```

是不是懵B了？为什么会报错？从报错信息中，我们可以看出两个错误的原因：

- `_start` 函数出现了双重定义
- `_start` 函数出现了未定义的引用 `main`

```
1 /usr/bin/ld: test1.o: in function `_start':
2 (.text+0x0): multiple definition of `_start'; /usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_
3 /usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/Scrt1.o: in function
4 (.text+0x24): undefined reference to `main'
5 collect2: error: ld returned 1 exit status
```

实际上，`_start` 函数便是链接器默认的**程序入口**。也就是说，通过链接器 `ld` 链接的ELF可执行文件运行后执行的第一条指令就是从 `_start` 开始的第一条指令！！！而联系到我们平时写的C语言代码，都是从 `main` 函数开始执行的。那么，我们推测，通过 `gcc` 默认链接的ELF可执行文件，一定有一段从 `_start` 跳转到 `main` 的代码。

 exercise1：请反汇编 `Scrt1.o`，验证下面的猜想（加 `-r` 参数，显示重定位信息）

所以结合报错（**报错的第二行有个 `Scrt1.o`**），我们不难猜测出错误发生的原因：`gcc` 在链接的时候，默认会把 `Scrt.o` 这个ELF可重定位文件同时链接到我们的程序里，而这个 `Scrt1.o` 里面包含了 `_start` 函数，这个函数的作用正是**跳转到 `main` 函数**。而我们的代码里面也出现了一个 `_start` 函数，并且不存在 `main`，所以导致了两个报错！！

好，回到正题。干脆不用 `gcc` 链接了，直接改用进行 `ld` 链接：

```
1 njucs> ld test1.o -o test1
2 njucs> ./test1
3 njucs> echo $?
4 3
```

在执行编译好的程序之后，我们 `echo $?` 来查看程序返回值，发现是3。我们在 ICS 里面学到，`int`

`$0x80` 是执行系统调用（System Call）的软中断。而 `%eax` 中的1是系统调用号，通过简单查阅可知，1是 `_exit` 系统调用的系统调用号，作用是结束当前进程，并返回。`%ebx` 里面存放了退出时返回的值。所以会显示出3。

这就是我们这个简单的汇编程序执行的全过程。那么借助这个程序，我来解释几个关键的符号：

- 通过一个“点”开头的东西比如 `.section` 和 `.data` 叫做汇编语言地**伪指令**，用来指导汇编器进行布局和汇编。
- `.section` 把这个程序划分为不同的段（`section`），比如代码段，数据段，相信大家都了解。
- `.data` 是声明一个**可读可写数据段**，存放的数据比如全局变量。（在这个代码中，没有全局变量，所以是空的！）。
- `.text` 段是声明一个**代码段**，不可以写，只可以读和执行，程序的代码部分在这个段里。
- `.globl` 说明这是一个**全局函数**或者**全局变量**，决定是否可以被其他模块引用（有`globl`的就可以被其他模块引用）。

好吧，上面这段文字就是给大家介绍一下汇编语言的文件是怎么写的。`ICS` 更多偏重于学习一些汇编指令，而本阶段的 `OS` 则是会用到一些真正的汇编文件。当然，这些不是最全的，不过我相信经过讲解，大家遇到问题都可以合理去搜索，实践并且分析出答案。

这个是**GCC官网**，我们可以查阅相关内容（全英文有点难，可以挑战一下自己）。还可以通过 `man gcc` 或者 `gcc --help` 来学习。

下面推荐另外一套工具链：<https://asmtutor.com/#top>

QEMU+GDB

QEMU

我们的实验要用到一个计算机硬件的模拟器，叫做 `QEMU`。听到这个东西，是不是很熟悉？我们在 `ICS` 课上写过一个叫 `NEMU` 的东西，来模拟计算机的硬件，同时，我们记得 `NEMU` 里面还有一个叫做 `Kernel` 的东西。在操作系统课程的实验中，`QEMU` 和 `NEMU` 类似，都是完全通过软件来模拟硬件的执行。但不同的是，`QEMU` 这个系统十分庞大，功能非常强大。

实验过程中写出的操作系统，是可以在实际的硬件上运行的，但是如果直接在我们的电脑上运行，有以下两点坏处：

- 我们糟糕的代码直接运行在硬件上可能会损坏硬件
- 没办法进行调试

所以，我们采用 `QEMU` 模拟器来运行我们的操作系统，并通过 `GDB` 来进行 `Debug`，具体操作在下面会解释。

GDB

GDB 我知道大家学过了，但是估计只学过一些比较简单的。下面教给大家一点黑科技。（暂时了解即可，在遇到需要GDB的地方时再回头看）。

利用 QEMU 模拟80386平台，运行自制的操作系统镜像 `os.img`

```
1 $ chmod +x ./utils/genboot.pl #给予genboot.pl权限，为什么暂时不用管，后面章节会解释
2 $ make                        #生成os.img
3 $ qemu-system-i386 os.img     #运行os.img
```

利用 QEMU 模拟80386平台，Debug 自制的操作系统镜像 `os.img`，选项 `-s` 在 TCP 的1234端口运行一个 `gdbserver`（不用懂，反正就是 QEMU 会把在其上运行的操作系统的执行信息发送给 GDB），选项 `-S` 使得 QEMU 启动时不运行80386的 CPU。

```
1 $ qemu-system-i386 -s -S os.img
```

再另外开一个 `shell`，启动 GDB，连接上述 `gdbserver`，在程序计数器 `0x7c00` 处添加断点，运行 80386的 CPU，显示寄存器信息，单步执行下一条指令。

```
1 $ gdb #启动gdb
2 $(gdb) target remote localhost:1234 #gdb远程连接上gdbserver
3 ...
4 $(gdb) b *0x7c00 #打断点
5 $(gdb) c #运行
6 ...
7 $(gdb) info registers #查看寄存器的值
8 ...
9 $(gdb) si #逐句执行，执行下一步
10 ...
```

由于跟 GDB 有关的黑科技太多，为了让文档不太长，下面列出一些文档，你们可以参考：

- [GDB入门教程_Dablelv的博客专栏-CSDN博客_gdb菜鸟教程](#)：这里面有一些比较好用的 GDB 指令，比如查看源码，反汇编等等，请自行查阅相关指令。
- [GDB Documentation \(sourceware.org\)](#)：这是GDB官方用户手册，有能力的人可以参考。

补充

你可能会比较疑惑，我们是怎么知道 `qemu-system-i386 -s -S os.img` 要用这两个参数。其实这个可以通过 `man` 命令来搜索，输入 `man qemu-system-i386` 就可以进入到 `qemu` 的手册里，这里有任何关于 `qemu` 的知识。输入后，首先进入到 `qemu` 的手册界面。

然后输入斜杠 / 加上关键词来搜索，比如我想要用 gdb 调试 qemu 上的代码，就输入 /gdb，敲回车。这时，会显示有 gdb 的内容。输入 n 查看下一个，输入 N 查看上一个！然后你就发现有一个什么 -s 参数会开启端口为1234的 gdb server。它调用了 -gdb 参数。它让 qemu 等待 gdb 的连接。

```
Disable set*uid/gid system calls
-gdb dev
Wait for gdb connection on device dev. Typical connections will likely be TCP-based, but also UDP, pseudo
TTY, or even stdio are reasonable use case. The latter is allowing to start QEMU from within gdb and
establish the connection via a pipe:

(gdb) target remote | exec qemu-system-x86_64 -gdb stdio ...

-s Shorthand for -gdb tcp::1234, i.e. open a gdbserver on TCP port 1234.
```

搜索"gdb"关键词

这时你会发现，还没等你连上 gdb，qemu 就把代码跑完了。所以我们需要一个暂时不让 qemu 启动的选项。搜索关键词 start，发现 -S 参数可以做到这点。

```
ping...
-S Do not start CPU at startup (you must type 'c' in the monitor).
```

-S

好了，参数找到了，开启qemu，再打开gdb远程连接即可！

Git管理项目

啥是git

Git is a [free and open source](#) distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

我想很多同学之前听说过 git 这个东西，但基本不会用。git 是一种**版本控制软件**，可以让代码管理和多人协作变得简单方便。这里教大家一些基础的 git 技能。

在你写代码的时候，有没有这样一种体验？改一个bug或者加一个功能的时候，把原本工整干净的代码弄得乱七八糟，这时你想变回原来的代码，结果发现已经物是人非，再也回不去了。传统的笨方法就是，你在增加新功能之前，把代码文件复制一份新的，然后再对其中一份进行代码的修改，这样做相当笨拙。而 git 可以完美解决这个问题，为你提供了“**后悔药**”，可以让你随时穿梭在你代码的不同版本之间。（当然，这只是强大的 git 的其中一个功能。）下面我们来配置一下git吧。

开始玩一下

（注意！！！这个是学习git用的，请在别处另建一个文件夹，不要在实验框架代码里进行操作！！！）

鉴于我们的实验环境已经由镜像（或自己配置）配好了，git 也已经安装好。先通过命令 `git --version` 来检查一下版本。出现版本号说明成功安装。

```
1 $ git version 2.25.1
```

然后进行一些基本的配置工作，设置下名字或者邮箱：

```
1 $ git config --global user.name "Zhang San" # your name
2 $ git config --global user.email "zhangsan@foo.com" # your email
```

这时，我们就可以使用 `git` 了。先切换到你开发的项目的根目录（该目录下的所有内容都会被 `git` 管理），然后输入：

```
1 $ git init
2 ...
3 $ git add .
4 ...
5 $ git commit -m 'first commit'
```

我来解释一下上面三条命令是啥意思：

- `git init` 是初始化 `git` 仓库，会在该目录下添加一个 `.git` 文件夹，这里存放了和 `git` 有关的信息，包括你的每一次提交记录，每次提交的代码是什么样的.....
- `git add .` 意思是把该文件夹下的所有内容都添加到当前的 `git` 版本里面。那个“点”意思是当前文件夹，你也可以添加单独的文件，比如 `git add file.txt`。
- `git commit -m 'first commit'` 这个命令的意思是告诉 `git`，把刚刚添加的文件提交到 `git` 仓库。其中 `-m` 表示要添加一段说明，即 `first commit`。如果你修复了一个 `bug` 之后进行一个提交，可以用 `git commit -m 'fix a bug...'`。这个说明的作用就是帮助你分辨代码的不同版本。
- 补充：那 `git add` 和 `git commit` 有啥区别呢？不都是添加到什么什么 `git` 仓库吗？做个类比：你是一个种橘子树的果农，你一箱一箱地采摘橘子装到车里，装满了才能发车。一箱箱装车的过程就是一次次 `git add` 的过程；最后发车就是 `git commit` 的过程；加个 `-m` 相当于给这车橘子贴个标签“xx年xx月xx日采摘，采摘地点xx”；车装满可以发车就相当于 `bug` 修复好了可以进行 `commit`。

在经过几次 `commit` 之后，输入 `git log`，我们会看见类似下面这种，其中记录了每个 `commit` 的作者，提交时间，以及你加的那句说明。`commit` 右边那串长长的串是版本号。

```
commit d531b1b1d6b7696dfd9695c1d560e3df53e615c5 (HEAD -> master, tag: 2.3.0, origin/master, origin/HEAD)
Author: Lef Ioannidis <elefthei@mit.edu>
Date: Thu Sep 6 22:11:54 2018 -0400

    Apply OSX fixes, test first on linux

commit 1f73b5e0fbe1b27f1a5d3f2e1aeb2253e2529c29
Author: Cody Cutler <ccutler@csail.mit.edu>
Date: Wed Sep 7 07:16:23 2016 -0400

    fix build
```

这时，如果你想回退到之前的某一个版本，就用 `git reset` 指令。以上图为例，我想把代码回退到 `fix build` 对应的版本。只需要输入：

```
1 $ git reset --hard 1f73b
```

其中 1f73b 是我们想要回退的版本的版本号的前几位！（不管几位，只要不和其他版本冲突就行）

以上就是 git 最最基础的使用，**非常建议**你在构建代码的过程中使用 git，会减少很多不必要的麻烦。如果你害怕你的虚拟机崩掉，.git 文件和代码一起丢失的话，也可以尝试远程仓库，把代码放在 gitee 或 github 上。

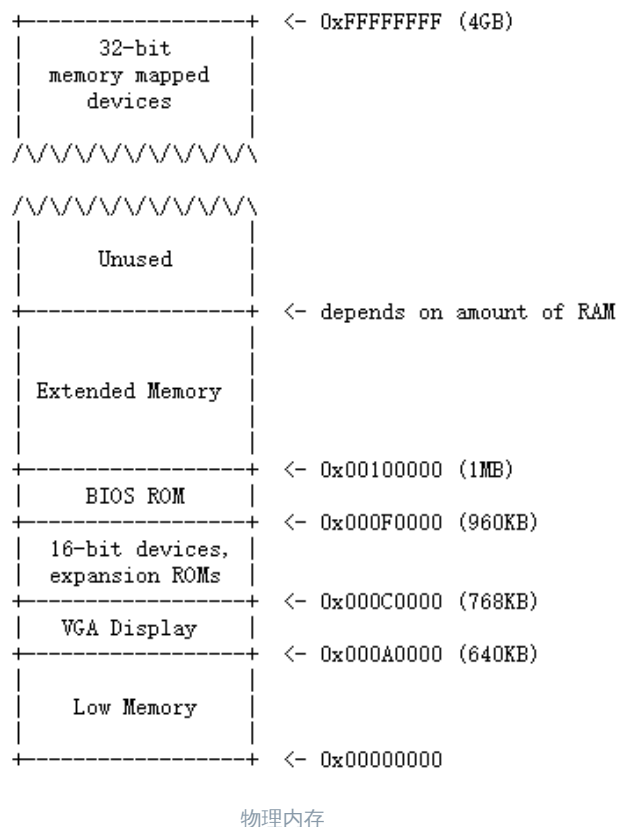
有关 git 使用的比较全面和通俗的介绍，请查看廖雪峰的教程：[Git教程 - 廖雪峰的官方网站 \(liaoxuefeng.com\)](http://liaoxuefeng.com)。这里面会教你如何使用远程仓库和其他一些实用的指令。

基础知识

没有基础知识是很难完成实验的，下面我来介绍一些基础知识。这其中会穿插一些问题~

计算机的物理地址空间

机器刚刚启动的时候，物理内存是按照下面这样划分的：



第一代 PC 基于16位 Intel 8088 处理器，只能寻址1MB的物理内存。所以早期PC的物理地址空间将从 0x00000000 开始，到 0x000FFFFF 结束，而不是 0xFFFFFFFF（32位）。标记为 Low Memory 的 640KB 空间是早期PC能够使用的唯一随机访问内存（RAM）。

从 0x000A0000 到 0x000FFFFF 的 384KB 区域（也就是 640KB 到 1MB 之间的区域）由**硬件预留**，用于特殊用途，如视频显示缓冲区（显存）和一些系统固件。这个保留区域中最重要的部分是 **Basic**

Input/Output System (BIOS) (下面会介绍这个概念)，它占用了从 `0x000F0000` 到 `0x000FFFFF` 的 64KB 区域。

在早期的 PC 中，BIOS 保存在真正的只读存储器 (ROM) 中，但现在的 pc 将 BIOS 存储在可更新的闪存中。**BIOS 主要负责对系统进行基本的初始化操作**，如激活显卡、检查内存安装量等。执行这个初始化之后，BIOS 从一些适当的位置 (比如硬盘) 加载操作系统，并将机器的控制权传递给操作系统。(划重点，BIOS 的作用)。

随着技术的发展，Intel 最终使用 80286 和 80386 处理器突破了 1MB 的寻址，它们分别支持 16MB 和 4GB 的物理地址空间，但 PC 架构师仍然保留了原始的 1MB 物理地址空间布局，以确保与现有软件的**向后兼容性**。因此，现代 pc 在物理内存中有一个从 `0x000A0000` 到 `0x00100000` 的一个洞，将 RAM 划分为“low memory”或“conventional memory” (前 640KB) 和“extended memory” (其他的部分)。

最近的 x86 处理器可以支持超过 4GB 的物理 RAM，所以 RAM 可以扩展到 `0xFFFFFFFF` 以上。在这种情况下，BIOS 必须安排在系统 RAM 的 32 位可寻址区域的顶部留下第二个洞，为这些 32 位设备的映射留下空间 (实际上是虚拟内存)。

开机的第一条指令

你们有没有想过这样一个问题：为什么我按了电脑的开机键，电脑就会开机？这个问题看起来似乎很蠢，但我相信很少有人能够回答出来。

CPU 会在接通电源后执行第一条指令，而我们知道，我们的程序代码需要放在内存里才能被 CPU 读取。那么接通电源后的第一条指令在哪里呢？**还是内存**。

学过 ICS 的同学对内存应该都有一定的概念，它就相当于程序运行的草稿纸。一般来说，断电之后，内存的内容会丢失。那么，按照这种说法，在加电之前，内存中应该是空的，所以加电之后，第一条指令需要被 CPU 搬到内存中去；但是内存中没有指令，CPU 又不会运行。这样就出现了先有鸡还是先有蛋的问题！

其实，内存中除了我们常说的 RAM (断电易失)，还有 **ROM (可以长久保存)**。

不管是 i386 (采用 intel 80386 架构) 还是 i386 之前的芯片，在加电后的第一条指令都是跳转到 BIOS 固件进行开机自检，然后将磁盘的主引导扇区 (Master Boot Record, MBR, 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`) 加载到 `0x7c00`。

补充

BIOS (Basic Input Output System) 是硬件厂商在硬件上自带的一段启动的代码，它的作用是进行一些硬件检查，并且跳转到引导程序 `bootloader`。

😁 一个小实验

先在实验根目录里输入 `make`，然后输入 (下面这行意思是不弹窗)：

```
1 $ make qemu-nox-gdb
```


或者（下面这行意思是有弹窗）：

```
1 $ make qemu-gdb
```

这时就会开启 qemu 里的 gdb server。

然后在实验根目录再打开一个终端，输入 make gdb，观察一下。

i exercise2：根据你看到的，回答下面问题

我们从看见的那条指令可以推断出几点：

- 电脑开机第一条指令的地址是什么，这位于什么地方？
- 电脑启动时 CS 寄存器和 IP 寄存器的值是什么？
- 第一条指令是什么？为什么这样设计？（后面有解释，用自己话简述）

QEMU为什么会这样启动呢？

Intel 就是这样设计 8088 处理器的（不同CPU架构下，BIOS可能不一样，这里只是举了x86架构的例子）。

因为电脑的 BIOS 是“天生的”物理地址范围 `0x000f0000-0x000fffff`，这种设计可以确保机器的 BIOS 总是在开机之后先获得机器控制权（位置固定）。QEMU 模拟器自带自己的 BIOS，它将 BIOS 放置在处理器模拟物理地址空间的这个位置。处理器复位时，（模拟）处理器进入实模式，并将 CS 设置为 `0xf000`，IP 设置为 `0xffff0`。

那么分段地址 `0xf000:ffff0` 如何变成物理地址？

为了回答这个问题，我们需要了解一些关于实际模式寻址的知识。在实模式下(即 PC 刚启动的模式)，地址转换的工作公式为: 物理地址 = `16 * 段 + 偏移量`。

因此，当计算机将 CS 设置为 `0xf000`，IP 设置为 `0xffff0` 时，所引用的物理地址为:

```
1 16 * 0xf000 + 0xffff0 # in hex multiplication by 16 is
2 = 0xf0000 + 0xffff0   # easy--just append a 0.
3 = 0xfffff0
```

`0xfffff0` 是 BIOS 结束前16个字节(`0x100000`)。在第一条指令的后面只有16个字节，啥也干不了。所以，.....

i exercise3：请翻阅根目录下的 makefile 文件，简述 make qemu-nox-gdb 和 make gdb 是怎么运行的（.gdbinit 是 gdb 初始化文件，了解即可）



了解BIOS

BIOS的初始化

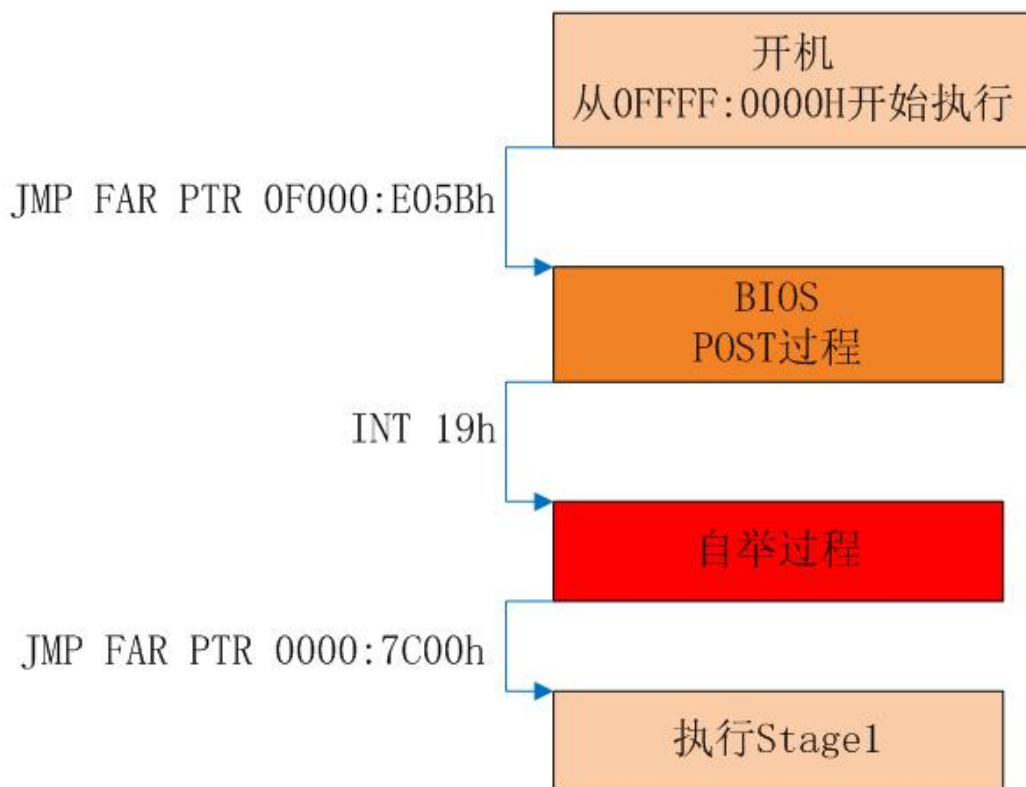
刚刚发现，BIOS 的第一条指令会跳转到某个更低地址的部分。那么，他会做什么呢？请用 `si` 往下看！

i exercise4：继续用 `si` 看见了什么？请截一个图，放到实验报告里。

这里稍微解释一下。

- 首先，设置了 `ss` 和 `esp` 寄存器
- 然后通过 `cli` 指令屏蔽了中断
- `cld` 和后面的 `in` 和 `out` 指令相关（暂时不用管）
- 然后通过 `in`，`out` 指令和 IO 设备交互，进行初始化，打开 A20 门（暂时不用管）
- 然后用 `lidtw` 与 `lgdtw` 加载 IDTR 与 GDTR（ICS 学过，跟保护模式有关）
- 最后开启保护模式，长跳转到 BIOS 的主要模块进行执行

上面是BIOS执行的最开始部分。而下图是BIOS执行的全过程，在`jmp far`之后，关闭保护模式，开启实模式，跳转到Stage1执行，其中Stage1就是我们前面说的**MBR（即bootloader）**！！！！



POST（不必了解）

至于POST过程，其实不必了解，但我还是提一下，如下：

- 1) 初始化各种主板芯片组
- 2) 初始化键盘控制器8042
- 3) 初始化中断向量，中断服务例程.
- 4) 初始化 VGA BIOS 控制器
- 5) 显示BIOS的版本和公司名称
- 6) 扫描软驱和各种介质容量
- 7) 读取CMOS的启动顺序配置，并检测启动装置是否正常
- 8) 调用INT 19h启动自举程序

自举过程

这个实际上就是把MBR的内容从磁盘调入地址为0x7c00的地方。

MBR & Bootloader

在开机的第一条指令这个模块，我们提到了 MBR。MBR 的全称叫做 Master Boot Record，是磁盘里的某个扇区，但绝大多数放在0号柱面，0号磁头，0号扇区对应的扇区。MBR 占一个扇区，共512字节。末尾有一个标识的东西，叫“魔数”（magic number），作用是告诉 BIOS：“这里是 MBR，你找对了。把我加载上去就可以启动操作系统”了。

MBR是一个扇区，而里面的代码，我们常常叫做Bootloader，大概翻译过来就叫**启动加载器**。从名字就可以看出，它可以把真正的OS（Operating System，后面OS即指“操作系统”）加载到内存中，然后把控制权交给OS！

先看看MBR长什么样子

在Linux中，我们可以很容易地查看磁盘的MBR（记得 sudo）：

```
1 head -c 512 /dev/sda | hd
```

```
[sudo] password for njucs:
00000000 eb 63 90 10 8e d0 bc 00 b0 b8 00 00 8e d8 8e c0 |.C.....|
00000010 fb be 00 7c bf 00 06 b9 00 02 f3 a4 ea 21 06 00 |...|.....!..|
00000020 00 be be 07 38 04 75 0b 83 c6 10 81 fe fe 07 75 |....8.u.....u|
00000030 f3 eb 16 b4 02 b0 01 bb 00 7c b2 80 8a 74 01 8b |.....|...t..|
00000040 4c 02 cd 13 ea 00 7c 00 00 eb fe 00 00 00 00 00 |L.....|.....|
00000050 00 00 00 00 00 00 00 00 00 00 00 80 01 00 00 00 |.....|.....|
00000060 00 00 00 00 ff fa 90 90 f6 c2 80 74 05 f6 c2 70 |.....t...p|
00000070 74 02 b2 80 ea 79 7c 00 00 31 c0 8e d8 8e d0 bc |t...y|..1....|
00000080 00 20 fb a0 64 7c 3c ff 74 02 88 c2 52 bb 17 04 |...d|<.t...R...|
00000090 f6 07 03 74 06 be 88 7d e8 17 01 be 05 7c b4 41 |...t...}...|A|
```

```

000000a0 bb aa 55 cd 13 5a 52 72 3d 81 fb 55 aa 75 37 83 ..U..ZRr=..U.u7.
000000b0 e1 01 74 32 31 c0 89 44 04 40 88 44 ff 89 44 02 ..t21..D.@.D..D.
000000c0 c7 04 10 00 66 8b 1e 5c 7c 66 89 5c 08 66 8b 1e ....f..\\f..f..
000000d0 60 7c 66 89 5c 0c c7 44 06 00 70 b4 42 cd 13 72 `|f..D..p.B..r
000000e0 05 bb 00 70 eb 76 b4 08 cd 13 73 0d 5a 84 d2 0f ...p.v....s.Z...
000000f0 83 d0 00 be 93 7d e9 82 00 66 0f b6 c6 88 64 ff .....}...f....d.
00000100 40 66 89 44 04 0f b6 d1 c1 e2 02 88 e8 88 f4 40 @f.D.....@
00000110 89 44 08 0f b6 c2 c0 e8 02 66 89 04 66 a1 60 7c .D.....f..f..|
00000120 66 09 c0 75 4e 66 a1 5c 7c 66 31 d2 66 f7 34 88 f..uNf..\\f1.f.4.
00000130 d1 31 d2 66 f7 74 04 3b 44 08 7d 37 fe c1 88 c5 .1.f.t.;D.)7....
00000140 30 c0 c1 e8 02 08 c1 88 d0 5a 88 c6 bb 00 70 8e 0.....Z....p.
00000150 c3 31 db b8 01 02 cd 13 72 1e 8c c3 60 1e b9 00 .1.....r....
00000160 01 8e db 31 f6 bf 00 80 8e c6 fc f3 a5 1f 61 ff ...1.....a.
00000170 26 5a 7c be 8e 7d eb 03 be 9d 7d e8 34 00 be a2 &Z|..}....}.4...
00000180 7d e8 2e 00 cd 18 eb fe 47 52 55 42 20 00 47 65 }.....GRUB .Ge
00000190 6f 6d 00 48 61 72 64 20 44 69 73 6b 00 52 65 61 om.Hard Disk.Rea
000001a0 64 00 20 45 72 72 6f 72 0d 0a 00 bb 01 00 b4 0e d. Error.....
000001b0 cd 10 ac 3c 00 75 f4 c3 31 d7 8b 03 00 00 80 20 ...<.u..1.....
000001c0 21 00 83 fe ff ff 00 08 00 00 00 f0 7f 02 00 00 !.....
000001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
*
000001f0 00 00 00 00 00 00 00 00 00 00 00 00 55 aa |.....U.
00000200

```

你可以在输出结果的末尾看到魔数 0x55 和 0xaa。

有了这个魔数，BIOS 就可以很容易找到可启动设备了：BIOS 依次将设备的首扇区加载到内存 0x7c00 的位置，然后检查末尾两个字节是否为 0x55 和 0xaa。

0x7c00 这个内存位置是 BIOS 约定的，如果你希望知道为什么采用 0x7c00，而不是其他位置，[这里](#)可以给你提供一些线索。

如果成功找到了魔数，BIOS将会跳到 0x7c00 的内存位置，执行刚刚加载的启动代码，这时BIOS已经完成了它的使命，剩下的启动任务就交给 MBR了；如果没有检查到魔数，BIOS将会尝试下一个设备；如果所有的设备都不是可启动的，BIOS将会发出它的抱怨：“找不到启动设备”。

BIOS加载主引导扇区后会跳转到 CS:IP=0x0000:0x7c00 执行加载程序，这就是我们操作系统实验开始的地方。在我们目前的实验过程中，主引导扇区（MBR）和加载程序（bootloader）其实代表一个东西。但是现代操作系统中，他们往往不一样，请思考一下为什么？主引导扇区中的加载程序的功能主要是将操作系统的代码和数据从磁盘加载到内存中，然后跳转到操作系统的起始地址。其实真正的计算机的启动过程要复杂很多，有兴趣请自行了解。

补充

MBR和Bootloader的区别是什么？MBR是一块磁盘扇区，而Bootloader是这块扇区里面的一小段代码。

这块扇区里面不一定非得是Bootloader，可以是你写的任何程序。我们会在后面给予你直观的印象。

IA-32的存储管理

在IA-32下，CPU有两种工作模式：**源于8086的实模式**与**源于80386的保护模式**。

下面我们分别介绍一下这两种模式。

PS：此部分为ICS内容，如果掌握的好可以快点读。（不可跳过，因为里面有问题要问。）

😊 实模式

8086为16位CPU，有16位的寄存器（Register），16位的数据总线（Data Bus），20位的地址总线（Address Bus），寻址能力为1MB。

1. 8086 的寄存器集合

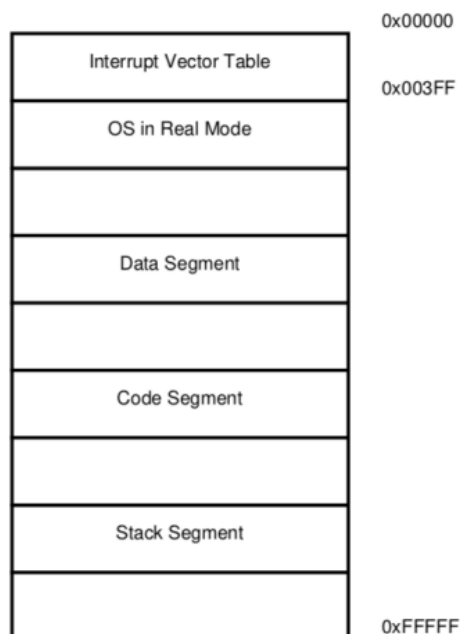
- 通用寄存器（16 位）：AX，BX，CX，DX，SP，BP，DI，SI
- 段寄存器（16 位）：CS，DS，SS，ES
- 状态和控制寄存器（16 位）：FLAGS，IP

2. 寻址空间与寻址方式

- 采用实地址空间进行访存，寻址空间为 2^{20}
- 物理地址 = 段寄存器 $\ll 4$ + 偏移地址
- 举个例子：CS=0x0000:IP=0x7C00 和 CS=0x0700:IP=0x0C00 以及 CS=0x7C0:IP=0x0000 所寻地址是完全一致的

3. 8086的中断

- 中断向量表存放在物理内存的开始位置(0x0000 至 0x03FF)
- 最多可以有 256 个中断向量
- 0x00 至 0x07 号中断为系统专用
- 0x08 至 0x0F，0x70 至 0x77 号硬件中断为 8259A 使用



- 一个实模式下用户程序的例子
 - 各个段在物理上必须是连续的
 - 装载程序在装入程序时需要按照具体的装载位置设置 CS，DS，SS

一个实模式下用户程序例子

8086的中断处理是交给BIOS完成的，实模式下可以通过 int \$0x10 中断进行屏幕上的字符串显示，具体细节请参考BIOS中断向量表或自行查找资料。



exercise5：中断向量表是什么？你还记得吗？请查阅相关资料，并在报告上说明。做完《[写一个自己的MBR](#)》这一节之后，再简述一下示例MBR是如何输出helloworld的。

实模式或者说8086本身有一些缺点：

- 安全性问题
 - 程序采用物理地址来实现访存，**无法实现对程序的代码和数据的保护**（划重点）
 - 一个程序可以通过改变段寄存器和偏移寄存器访问并修改不属于自己的代码和数据
- 分段机制本身的问题
 - 段必须是连续的，从而无法利用零碎的空间
 - 段的大小有限制（最大为 64KB），从而限制了代码的规模



exercise6：为什么段的大小最大为64KB，请在报告上说明原因。

因为一些问题，我们需要开启保护模式.....请看下节。

保护模式

变化

80386开始，Intel处理器步入32位CPU；80386有32位地址线，其寻址空间为 $2^{32}=4\text{GB}$ ；为保证兼容性，实模式得以保留，PC启动时CPU工作在实模式（BIOS运行过程中可能会开启保护模式），并由Bootloader迅速完成从实模式向保护模式的切换。

下面是保护模式的一些不同。

1. 保护模式下的寄存器

- 通用寄存器(从 16 位扩展为 32 位): EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- 段寄存器(维持 16 位): CS, DS, SS, ES, FS, GS
- 状态和控制寄存器(32/64 位): EFLAGS, EIP, CR0, CR1, CR2, CR3
- 系统地址寄存器: GDTR, IDTR, TR, LDTR
- 调试与测试用寄存器: DR0, ..., DR7, TR0, ..., TR7

2. 寻址方式的变化

- 在保护模式下，分段机制是利用一个称作段选择子（Selector）的偏移量到全局描述符表中找到需要的段描述符，而这个段描述符中就存放着真正的段的物理首地址，该物理首地址加上偏移量即可得到最后的物理地址。
- 一般保护模式的寻址可用 0xMMMM:0xNNNNNNNN 表示，其中 0xMMMM 表示段选择子的取值，16

位（其中高 13 位表示其对应的段描述符在全局描述符表中的索引，低 3 位表示权限 等信息），0xNNNNNNNN 表示偏移量的取值，32 位。

- 段选择子为 CS，DS，SS，ES，FS，GS 这些段寄存器。

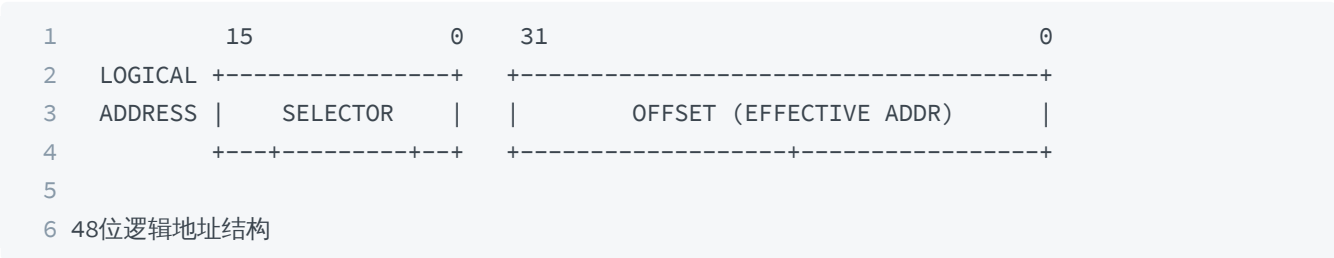
寻址过程

在保护模式下，寻址方式会产生变化。简单来说，程序给出的32位地址不再直接解释为物理地址，而是相对于某一个段的偏移量（offset）。真正的物理地址由下式给出：

```
1 physical address = base address + offset
```

其中的 base address 是一个32位的地址，对应某个段的基地址；而 offset 则是程序给出的32位段内偏移量。基地址存储在**段描述符**里面，而**段描述符需要通过Selector来寻找**！

当开启保护模式后，QEMU中运行的程序在访问内存时给出的就不再是简单的32位物理地址了，而是由一个16位的段选择符加上32位的段内偏移量（有效地址）所构成的48位的逻辑地址！

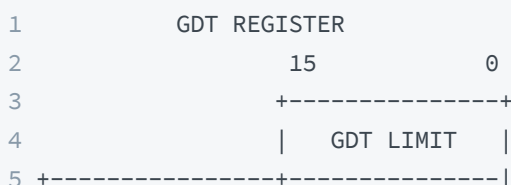


这里是段描述符每个部分的含义：

- | | | |
|-------|-------------|---------------------------------|
| bit 3 | Data/Code | 0 (data) |
| bit 2 | Expand-down | 0 (normal)
1 (expand-down) |
| bit 1 | Writable | 0 (read-only)
1 (read-write) |
| bit 0 | Accessed | 0 (hasn't)
1 (accessed) |

TYPE位

为进入保护模式，需要在内存中开辟一块空间存放GDT表；80386提供了一个寄存器 **GDTR** 用来存放 **GDT** 的32位物理基地址以及表长界限；在将GDT设定在内存的某个位置后，可以通过 **LDGT** 指令将GDT的入口地址装入此寄存器。



结合虚拟地址、段选择符和段表的相关概念，在分段机制中，将虚拟地址转换成线性地址（此时即为物理地址）的过程可描述如下：

1. 根据段选择子中的 TI 位选择GDT或LDT；
2. 根据段选择子中的 index 部分到GDT中找到对应位置上的段描述符；
3. 读取段描述符中的 base 部分，作为32位段基址，加上32位段内偏移量获取最终的物理地址。

Reference

本节部分内容摘自《计算机系统基础》PA-3-2的手册。

😬 从实模式切换到保护模式

在实模式下，操作系统需要初始化段表（如 GDT）和描述符表寄存器（如 GDTR）。在初始化完成后，操作系统通过将0号控制寄存器（CR0）中的 PE 位置为1的方式，来通知机器进入保护模式。在此之前，CR0 中的 PE 初始化为0。CR0 寄存器的结构请自行参阅i386手册的相关内容。

（具体内容会在后面实验里有直观体验。）

显存管理

前文我们提到，在实模式下可以通过BIOS中断在屏幕上显示文字，切换到保护模式后有一个令人震惊的事实：切换到32位保护模式的时候，我们不能再使用 BIOS 了。☹️

切换到32位（开启保护模式之后）碰到的第一个问题是如何在屏幕上打印信息。之前我们请求 BIOS 在屏幕上打印一个 ASCII 字符，但是它是如何做到将合适的像素展示在计算机屏幕恰当的位置上的呢？

目前，只要知道显示设备可以用很多种方式配置成两种模式：文本模式和图像模式。屏幕上展示的内容只是某一特定区域的内存内容的视觉化展示。所以为了操作屏幕的展示，我们必须在当前的模式下管理内存的某特定区域。显示设备就是这样子的一种设备，和内存相互映射的硬件。

当大部分计算机启动时候，虽然它们可能有更先进的图像硬件，但是它们都是先从简单的视频图像数组（VGA, video graphics array）颜色文本模式，尺寸80*25，开始的（数电和ICS都接触过）。在文本模式，编码人员不需要为每个字符渲染每一个独立的像素点，因为一个简单的字体已经在VGA显示设备内部内存中定义了。每一个屏幕上字符单元，在内存中通过两字节表示，第一个字节展示字符的ASCII编码，第二个字节包含字符的一些属性，比如字符的前景色和背景色，字符是否应该闪烁等。

所以，如果我们想在屏幕上展示一个字符，那么我们需要为当前的VGA模式，在正确的内存地址处设置一个ASCII码值，通常这个地址是0xb8000。

开始实验

刚刚介绍了有关BIOS的知识，我们可以开启真正的实验了。

写一个自己的MBR

在这一部分的第一小节，我们会DIY一个主引导扇区，然后用 `qemu` 启动它，并在屏幕上输出 `Hello, World !`

在第二小节，我们会对第一小节的操作进行解释。

操作一波

一系列操作

在配置好实验环境之后，先建立一个操作系统实验文件夹存放本实验代码：

```
1 $mkdir OS2022
```

进入创建好的文件夹，创建一个mbr.s文件：

```
1 $cd OS2022
2 $touch mbr.s
```

然后将以下内容保存到 mbr.s 中：

```
1 .code16
2 .global start
3 start:
4 movw %cs, %ax
5 movw %ax, %ds
6 movw %ax, %es
7 movw %ax, %ss
8 movw $0x7d00, %ax
9 movw %ax, %sp          # setting stack pointer to 0x7d00
10 pushw $13             # pushing the size to print into stack
11 pushw $message        # pushing the address of message into stack
12 callw displayStr      # calling the display function
13 loop:
14 jmp loop
15 message:
16 .string "Hello, World!\n\0"
17 displayStr:
18 pushw %bp
19 movw 4(%esp), %ax
20 movw %ax, %bp
```

```

22 movw $0(%esp), %ax
23 movw $0x000c, %bx
24 movw $0x0000, %dx
25 int $0x10
26 popw %bp
27 ret

```

接下来使用gcc编译得到mbr.s文件：

```
1 $gcc -c -m32 mbr.s -o mbr.o
```

文件夹下会多一个mbr.o的文件，接下来使用ld进行链接：

```
1 $ld -m elf_i386 -e start -Ttext 0x7c00 mbr.o -o mbr.elf
```

我们会得到mbr.elf文件，查看一下属性。

```

1 $ls -al
2 ...
3 -rwxr-xr-x 1 abc abc 3588 2月 15 19:50 mbr.elf
4 -rw-r--r-- 1 abc abc 656 2月 15 19:46 mbr.o
5 -rw-r--r-- 1 abc abc 594 2月 15 19:43 mbr.s

```

我们发现 mbr.elf 的大小有 3588byte，这个大小超过了一个扇区，不符合我们的要求。

i exercise7：假设mbr.elf的文件大小是300byte，那我是否可以直接执行qemu-system-i386 mbr.elf这条命令？为什么？

不管是i386还是i386之前的芯片，在加电后的第一条指令都是跳转到BIOS固件进行开机自检，然后将磁盘的主引导扇区（Master Boot Record, MBR；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）加载到0x7c00。

所以我们使用objcopy命令尽量减少mbr程序的大小：

```
1 $ objcopy -S -j .text -O binary mbr.elf mbr.bin
```

再查看，发现mbr.bin的大小小于一个扇区。

```

1 $ ls -al mbr.bin
2 -rwxr-xr-x 1 kingxu kingxu 65 2月 15 20:03 mbr.bin

```

然后我们需要将这个mbr.bin真正做成一个MBR，新建一个genboot.pl文件


```
1 $touch genboot.pl
```

将以下内容保存到文件中

```
1 #!/usr/bin/perl
2 open(SIG, $ARGV[0]) || die "open $ARGV[0]: $!";
3 $n = sysread(SIG, $buf, 1000);
4 if($n > 510){
5     print STDERR "ERROR: boot block too large: $n bytes (max 510)\n";
6     exit 1;
7 }
8 print STDERR "OK: boot block is $n bytes (max 510)\n";
9 $buf .= "\0" x (510-$n);
10 $buf .= "\x55\xAA";
11 open(SIG, ">$ARGV[0]") || die "open >$ARGV[0]: $!";
12 print SIG $buf;
```

给文件可执行权限

```
1 $chmod +x genboot.pl
```

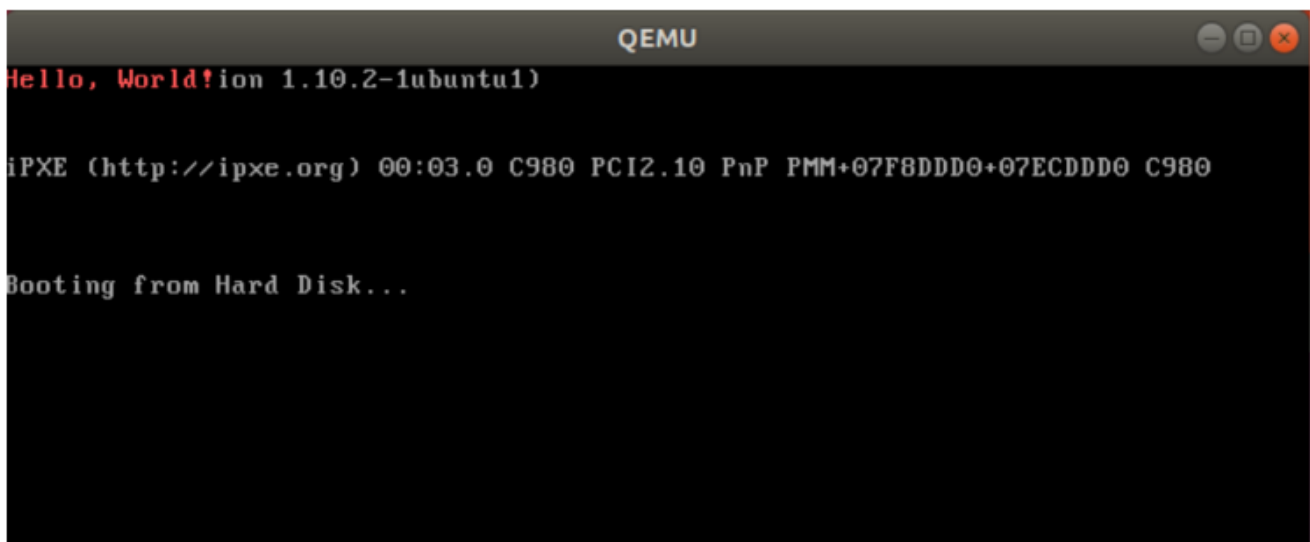
然后利用 `genboot.pl` 生成一个 MBR，再次查看 `mbr.bin`，发现其大小已经为512字节了

```
1 $./genboot.pl mbr.bin
2 OK: boot block is 65 bytes (max 510)
3 $ls -al mbr.bin
4 -rwxr-xr-x 1 kingxu kingxu 512 2月 15 20:11 mbr.bin
```

一个MBR已经制作完成了，接下来就是查看我们的成果

```
1 $qemu-system-i386 mbr.bin
```

会弹出这样一个窗口



输出hello world成功

😬 分析一下

上一小节，大家一脸懵逼的进行了操作，下面来解释一下。

先编写 `mbr.s`，然后通过gcc编译成可链接目标文件`mbr.o`（注意-m32参数）

然后用ld进行链接，然后用objcopy进行一些操作：

```
1 $ld -m elf_i386 -e start -Ttext 0x7c00 mbr.o -o mbr.elf
2 $objcopy -S -j .text -O binary mbr.elf mbr.bin
```

- ① exercise8：面对这两条指令，我们可能摸不着头脑，手册前面..... 所以请通过之前教程教的内容，说明上面两条指令是什么意思。（即解释参数的含义）

在执行这两段命令之后，我们发现 `mbr.bin` 的大小为65字节，远远少于之前的3k多。

下面使用的genboot.pl其实是一个脚本程序，虽然我们没学过这种脚本语言，但可以大概看出来，它先打开 `mbr.bin`，然后检查文件是否大于510字节，然后.....

- ① exercise9：请观察genboot.pl，说明它在检查文件是否大于510字节之后做了什么，并解释它为什么这么做。

好的，qemu可以成功运行我们这个简易版的mbr了！

深入探索

大家有没有发现，我们并没有使用elf格式的 `mbr.elf`，而是使用了bin格式的 `mbr.bin`？

- ① exercise10：请反汇编 `mbr.bin`，看看它究竟是什么样子。请在报告里说出你看到了什么，并附上截图

实际上，我们发现elf格式并不是执行代码的必备条件，它可以是exe文件，可以是elf文件，甚至是很原始的bin文件。

连接角度

ELF 头
program header table 程序头部表（可选）
section 1（节1）
.....
section n （节 n ）
.....
section header table （节头部表）

运行角度

ELF 头
program header table 程序头部表
segment 1（段1）
segment 2（段2）
.....
section header table （节头部表，可选）

但是为什么我们常常使用elf格式的文件呢？

想想elf文件里都有什么，elf头，程序头表.....这些正是导致elf格式文件体积膨大的原因。我们通过elf头中的信息找到各种不同的符号和信息，从而在链接和加载的时候，按照这样固定的流程进行解析和重组。并且，我们可以通过强大的工具链（Tool chain，比如gdb、objdump）来对程序进行解析和重组。

也就是说，如果可以的话，我们可以制定一种自己的程序文件格式。编写一套自己的工具链来解析、构建可执行文件！！

challenge !!!

🚨 challenge1：请尝试使用其他方式，构建自己的MBR，输出“Hello，world!”

这是对我们的第一个挑战，目的是帮助大家更深刻的理解MBR是如何执行的。我这里给出几种可能的方法。如果你有自己的方法就更好了！

- 第一种（基础）：还是使用我们提供的汇编语言，通过编写一段C语言或者Python代码来代替genboot.pl文件，来生成符合mbr格式的mbr.bin
- 第二种（基础）：使用C语言或Python自己解析elf文件格式，来代替objcopy和genboot.pl，来生成符合mbr格式的mbr.bin
- 第三种（进阶）：直接使用汇编语言编写代码（在汇编文件里面编写指令填充剩余字节和添加魔数），通过nasm编译器直接编译成512字节的mbr.bin
- 第四种（困难）：像上一节一样，编写汇编代码，但用nasm编译成其他目标格式（如exe，macho32等），最后抽取出代码部分！

- 你自己的想法.....

我们鼓励你去尝试高难度的操作，但是为了避免卷怪，以上方法任选一种去实现就好了（分数几乎不会有差别），并且要在实验报告里面说明你选择的方法是什么，和如何实现的（并在challenge文件夹里面附上你的代码）。

补充材料

gcc输出的可执行文件格式都是Unix传统的ELF格式，而nasm汇编器（注意是汇编器）可以输出多种目标格式（比如我们熟知的windows对应的exe，macos对应的macho32，原始的bin文件.....）。

上面第三种思路的同学可以直接把汇编代码编译成512字节的bin格式，第四种思路的同学可以先编译成任意其他格式，再通过某种办法从中把可执行代码抽取出来.....

对于选择第三种思路的同学，这里提供nasm汇编的资料：<https://asmtutor.com/#top>

你们甚至可以在网上找到第三种思路的代码，可以借鉴，但是一定要理解！

加载真正的操作系统

重头戏来了！

在前面的介绍中，我们熟悉了 BIOS 和 MBR 。这一部分，我们体验一下OS的加载过程。

本部分分为两小节：

- 第一小节是开启保护模式
- 第二小节是加载一个微型“操作系统”

每小节按照下面这样的结构安排：

1. 实验任务：具体要改哪些地方
2. （准备工作）：进行本节之前，需要做哪些准备
3. 框架理解：帮助理解框架，并给予提示

下面是一个黑科技，实验结束可以研究一下：

[C 如何编译出一个不需要操作系统的程序？ - 知乎 \(zhihu.com\)](#)

开启保护模式

实验任务

i task1：以下任务点是我们在本节需要完成的（代码中已通过TODO注释）

- ✓ 把cr0的低位设置为1。
- ✓ 填写GDT。
- ✓ 显示helloworld。

框架理解

我们先要开启保护模式，现在我们可以打开给出的代码文件。

文件的结构是这样的：

- bootloader（这个文件夹编译生成bootloader）
 - start.s（通过它开启保护模式）
 - boot.c（通过它来加载app，app即我们的微型“OS”）
 - boot.h（包含辅助函数，与硬件交互，了解即可，不必掌握）
 - Makefile（编译bootloader的makefile）
- app（微型os）
 - app.s（显示hello，world）
 - Makefile（编译app）
- utils
 - genboot.pl
- Makefile（总体的makefile，生成os.img）

我们目前需要修改的文件是**start.s**。

为了开启保护模式，我们要做如下几步：


1. 关中断
2. 开启A20地址线（请自行搜搜看A20地址线是什么）
3. 加载GDTR（请看一下代码中的gdtDesc是什么）
4. 把cr0的最低位设置成1（翻阅一下前面的教程，会告诉你为什么要置为1）
5. 长跳转切换到保护模式

对于长跳转指令，不知道大家还有没有印象。它有两个操作数，第一个操作数是**代码段选择子**，第二个操作数是**跳转地址**，具体是做什么的，请回顾PA学过的内容~

我们需要实现的地方只有三处：

- 把cr0的最低位置为0
 - 思路：可以借助eax作为中转寄存器，先把cr0存入eax，然后把eax最低位置为1，最后存回cr0
 - 当然，如果有自己的思路更好
- 填写GDT
-

- GDT的第一个描述符都是0，请自行搜索为什么。
- base和limit的大小请参考Linux的实现。
- 请先思考，为什么我要把三个段描述符按照cs，ds，gs的顺序排列？（Hint：结合汇编代码，看看段选择子是多少 # 段选择子如何查找段描述符）
- 在注释里我已经把公式给出，请结合 # GDT 来填写它。
- 输出helloworld
 - 这个提示的已经太明显了 😄

 exercise11：请回答为什么三个段描述符要按照cs，ds，gs的顺序排列？

 exercise12：请回答app.s是怎么利用显存显示helloworld的。

加载“OS”

这部分只要写两行代码

实验任务

我们最后一步，是加载app（即OS）。

 task2：以下任务点是在本节需要完成的：

- ☒ 把上一节保护模式部分搬过来。
- ☒ 填写bootMain函数。

切换到本节实验

 切换前请commit来把task1的代码提交！！

请在完成《开启保护模式》之后，在实验根目录命令行里输入，切换到本节的分支：

```
1 $ git checkout load-os
```

框架理解

boot.c里面的bootMain函数的作用有如下两个：

1. 把app.bin里面的内容读到0x8c00（在本实验我们这样规定，实际上不一定非是0x8c00）

2. 跳转到0x8c00 (Hint : 使用内联汇编)

readSector函数是将第offset块磁盘读出，读到物理地址为dst的内存中。

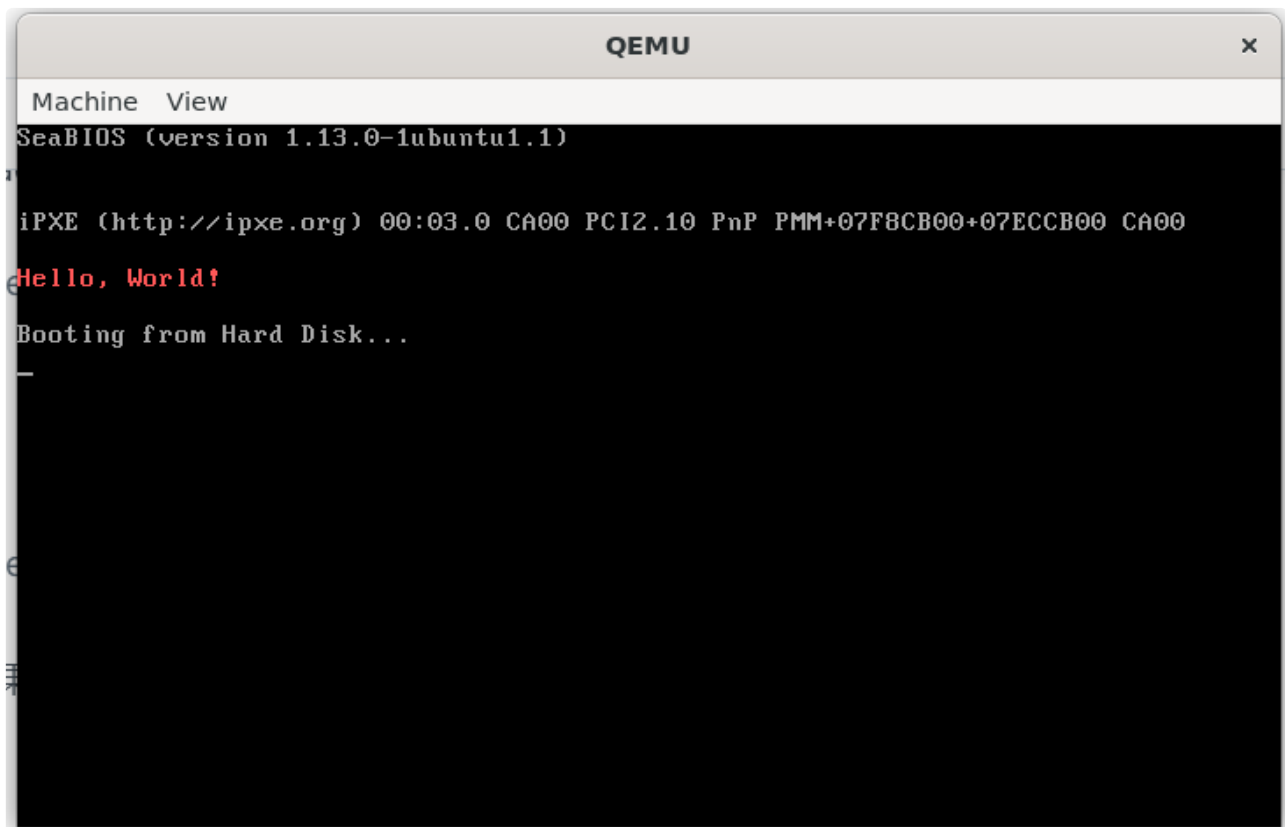
i exercise13 : 请阅读项目里的3个Makefile，解释一下根目录的Makefile文件里

`cat bootloader/bootloader.bin app/app.bin > os.img`

这行命令是什么意思。

i exercise14 : 如果把app读到0x7c20，再跳转到这个地方可以吗？为什么？

如果你成功加载app，就可以看到下图。恭喜你，成功完成了实验！



成功了

i exercise15 : 最终的问题，请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。

不用太详细，把每一部分是做什么的说清楚就好了。