

1. 实验要求

本实验通过实现一个简单的引导程序，介绍系统启动的基本过程

1.1 在实模式下实现一个Hello World程序

在实模式下在终端中打印 `Hello, world!`

1.2 在保护模式下实现一个Hello World程序

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!`

1.3 在保护模式下加载磁盘中的Hello World程序运行

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. 相关资料

2.1 CPU、内存、BIOS、磁盘、主引导扇区、加载程序、操作系统

最开始先讨论这样一个问题：

CPU在加电之后，它的第一条指令在哪？

我们知道，CPU在电源稳定后会将内部的寄存器初始化成某个状态，然后执行第一条指令。第一条指令 在哪？答案是**内存**

内存是用来存数据的，但是有过了解的同学都知道，断电后内存中的内容会丢失。那上哪找第一条指令？

其实内存除了我们说的内存条这种RAM，还有ROM；在i386机器刚启动时，内存的地址划分如下：

- **基本内存** 占据0 ~ 640KB地址空间
- **上位内存** 占据640KB ~ 1024KB地址空间。分配给显示缓冲存储器、各适配卡上的ROM和系统 **ROM BIOS**。（这个区域的地址分配给ROM，相应的384KB的RAM被屏蔽掉）
- **扩展内存** 占据1MB以上地址空间



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。

查看磁盘的MBP

在Linux中, 你可以很容易地查看磁盘的MBR(需要root权限):

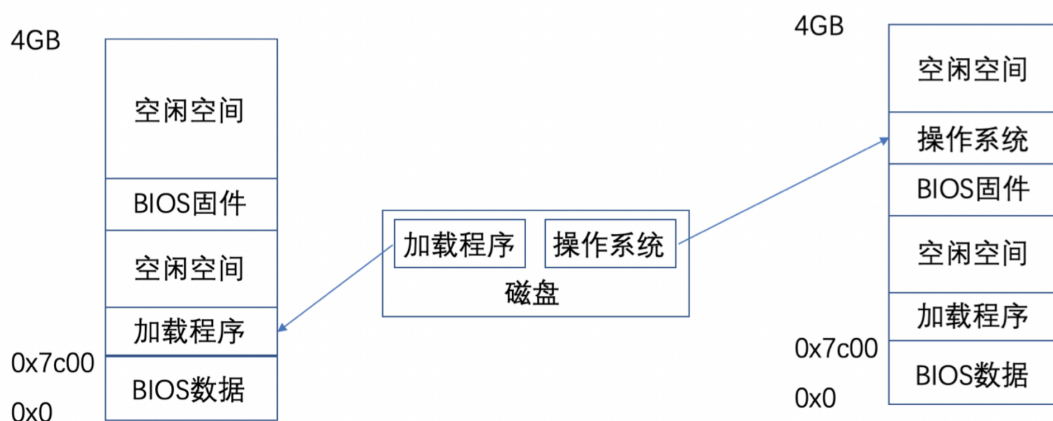
```
$head -c 512 /dev/sda | hd
```

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
- 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。

Ex1: 你弄清楚本小结标题中各种名词的含义和他们间的关系了吗? 请在实验报告中阐述。

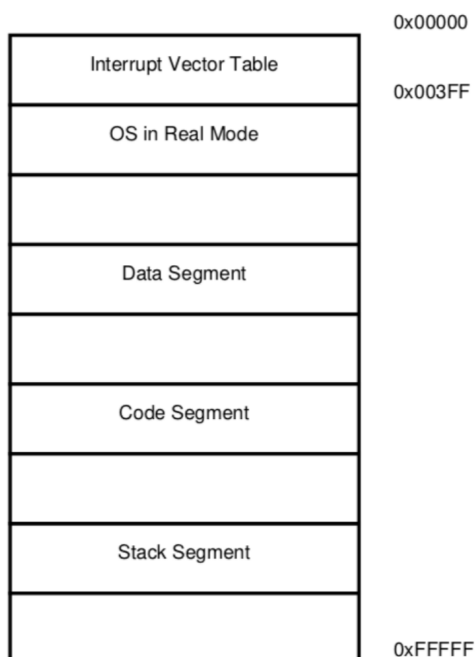
2.2 IA-32的存储管理

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；

2.2.1 实模式简介

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

- 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
- 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
 - 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



- 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

- 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

8086的中断处理是交给BIOS完成的，这也是为什么我们看到**2.1节**内存0x0处是BIOS数据。、

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。

Ex2：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。

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
 - 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

Ex3：为什么段的大小最大为64KB，请在报告上说明原因

2.2.2 保护模式

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，并由Bootloader迅速完成从实模式向保护模式的切换

- **保护模式带来的变化**
 - 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

	8086 寄存器	80386 寄存器
通用寄存器	AX, BX, CX, DX SP, BP, DI, SI	EAX, EBX, ECX, EDX ESI, EDI, EBP, ESP
段寄存器	CS, DS, SS, ES	CS, DS, SS, ES, FS, GS
段描述符寄存器	无	对程序员不可见
状态和控制寄存器	FLAGS, IP	EFLAGS, EIP CR0, CR1, CR2, CR3
系统地址寄存器	无	GDTR, IDTR, TR, LDTR
调试寄存器	无	DR0, ..., DR7
测试寄存器	无	TR0, ..., TR7

• 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

全局描述符表 (Global Descriptor Table) 即 GDT，GDT 中由一个个被称为段描述符的表项组成，表项中定义了段的起始 32 位物理地址，段的界限，属性等内容；

段描述符的结构如下图所示：

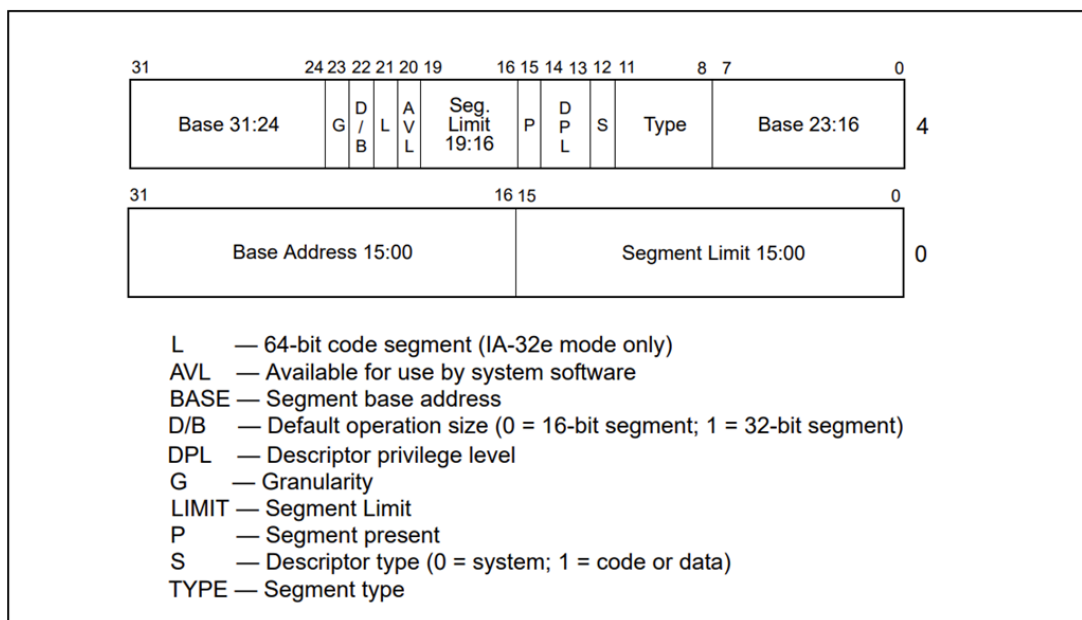


Figure 3-8. Segment Descriptor

你可以把段描述符当做一个结构体，这样，段表就相当于一个**结构数组**

这里是段描述符每个部分的含义：

- 每个段描述符为 8 个字节，共 64 位
- 段基址为第 2, 3, 4, 7 字节，共 32 位
- 段限长为第 0, 1 字节及第 6 字节的低 4 位，共 20 位，表示该段的最大长度
- G代表粒度，说明段限长的单位是什么（4KB或者1B）。当属性 G 为 0 时，20 位段限长为实际段的最大长度（最大为 1MB）；当属性 G 为 1 时，该 20 位段限长左移 12 位后加上 0xFFF 即为实际段的最大长度（最大为 4GB）。
- D/B: 对于不同类型段含义不同
 - 在可执行代码段，该位叫做 D 位，D 为 1 时，使用 32 位地址和 32/8 位操作数，D 为 0 使用 16 位地址和 16/8 位操作数。
 - 在向下扩展的数据段中，该位叫做 B 位，B 为 1 段的上界为 4GB，B 为 0 段的上界为 64KB。
 - 在描述堆栈段的描述符中，该位叫做 B 位，B 为 1 使用 32 位操作数，堆栈指针用 ESP，B 为 0 使用 16 位操作数，堆栈指针用 SP。
- AVL: Available and Reserved Bit，通常设为 0。
- P: 存在位，P 为 1 表示段在内存中。

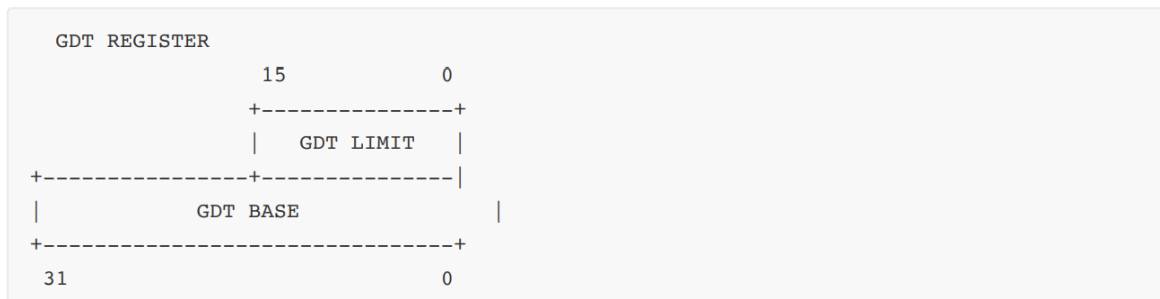
- DPL: 描述符特权级, 取值 0-3 共 4 级; 0 特权级最高, 3 特权级最低, 表示访问该段时 CPU 所处于的最低特权级, 后续实验会详细讨论。
- S: 描述符类型标志, S 为 1 表示代码段或数据段, S 为 0 表示系统段 (TSS, LDT) 和门描述符。
- TYPE: 当 S 为 1, TYPE 表示的代码段, 数据段的各种属性如下表所示:

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)

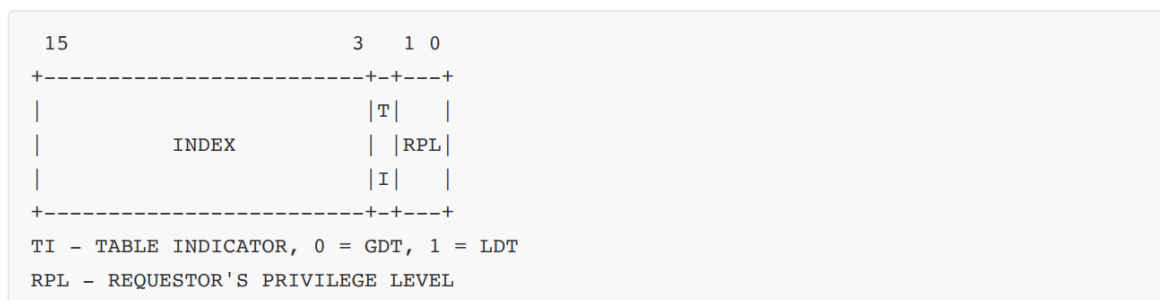
bit 3	Data/Code	1 (code)
bit 2	Conforming	0 (non-conforming) 1 (conforming)
bit 1	Readable	0 (no) 1 (readable)
bit 0	Accessed	0 (hasn't) 1 (accessed)

段选择子如何查找段描述符

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**。

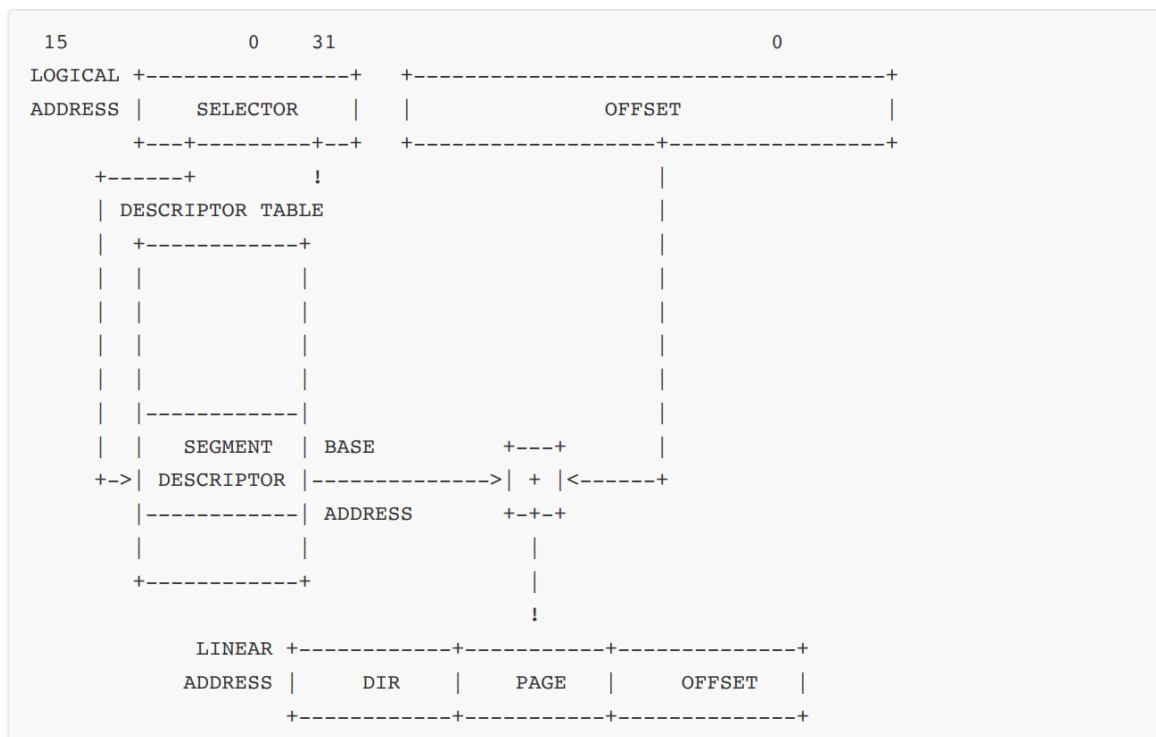


由 GDTR 访问 GDT 是由 **段选择子（理解成段选择子里包含数组下标）** 来完成的；为访问一个段，需将段选择子存储入 **段寄存器**，比如数据段选择子存储入 `DS`，代码段选择子存储入 `CS`；其数据结构如下：



TI 位表示该段选择子为全局段还是局部段，RPL 表示该段选择子的特权等级，13位 Index 表示描述符表中的编号（下标）。

Selector:Offset 表示的**逻辑地址**可如下图所示转化为**线性地址**，倘若不采用分页机制，则该**线性地址**即**物理地址**。



总结地址转换过程

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。

2.2.3 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。

关于CR0寄存器

CR0寄存器的结构如下图所示：

63

32

Reserved, MBZ

31

30

29

28

19

18

17

16

15

6

5

4

3

2

1

0

P

C

N

Reserved

A

R

W

Reserved

N

E

T

S

E

M

P

E

G

D

W

M

P

E

Bits

Mnemonic

Description

R/W

63–32

Reserved

Reserved, Must be Zero

31

PG

Paging

R/W

30

CD

Cache Disable

R/W

29

NW

Not Writethrough

R/W

28–19

Reserved

Reserved

18

AM

Alignment Mask

R/W

17

Reserved

Reserved

16

WP

Write Protect

R/W

15–6

Reserved

Reserved

5

NE

Numeric Error

R/W

4

ET

Extension Type

R

3

TS

Task Switched

R/W

2

EM

Emulation

R/W

1

MP

Monitor Coprocessor

R/W

0

PE

Protection Enabled

R/W

CR0的PE位 (protection enable) 即保护位表示是否开启了段级保护,一旦置为1,就表示开启了保护模式,而开机加电时默认是置为0的. 同理如果要开启分页机制,就需要把PG位,即分页标志位置为1

此次不要求分页，对CR0寄存器感兴趣的同学可以课后深入探索。

2.3 显存映射

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。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。**

3. 实验过程

lab1.1任务：请按照下面操作，在自己的电脑上完成实验，并在报告中附上实验结果(截图，下同)。

ps：下面给出了lab1.1的代码，同学们需要按照 4.提交格式 中的 Makefile工程格式，动手实践下，最终在根目录依次执行 `make os.img` 和 `make play` 即可得到输出结果（lab1.2和lab1.3同）。关于Makefile用法请查看实验课主页以及自己查找相关资料。

lab1.1 是 lab1.2 和 lab1.3 的基础，**最终上交的作业报告只需附上 lab1.3 的代码即可。**

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：

```
$mkdir os2022
$cd os2022
```

将我们所下发的文件存放到该文件夹下，打开 lab1-STUID 文件夹，可以看到

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**

可以看出, 在 start.s 中存在三大部分:

- 第一部分就是我们写的实模式下的 hello world, 这一部分只需要将对应的注释关闭就可以编译运行

```
/* Real Mode Hello world */
...

```

- 第二部分需要同学们自己手动实现, 对应保护模式下的 hello world

```
/* Protected Mode Hello world */
...

```

- 第三部分同样，是保护模式下加载 hello world APP 功能模块

```
/* Protected Mode Loading Hello world APP */
```

```
...
```

3.1 在实模式下实现一个Hello World程序 (lab1.1)

我们以第一个例子为实验导引，粗略了解一下整个实验过程的大致流程：

首先新建一个引导代码文件 `mbr.s`，对应的代码已经写好在 `start.s` 的第一部分中，你可以将注释关闭后，直接拷贝到 `mbr.s` 中：

```
.code16
.global start
start:
    movw %cs, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %ss
    movw $0x7d00, %ax
    movw %ax, %sp                # setting stack pointer to
0x7d00
    pushw $13                    # pushing the size to print
into stack
    pushw $message               # pushing the address of
message into stack
    callw displayStr             # calling the display function
loop:
    jmp loop

message:
    .string "Hello, world!\n\0"

displayStr:
    pushw %bp
    movw 4(%esp), %ax
```

```

    movw %ax, %bp
    movw 6(%esp), %cx
    movw $0x1301, %ax
    movw $0x000c, %bx
    movw $0x0000, %dx
    int $0x10                # 8086 interrupts are handled
by the BIOS
    popw %bp
    ret

```

接下来使用gcc编译得到mbr.s文件：

```
$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：

```
$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。

```

$ls -al

drwxrwxr-x 2 oslab oslab 4096 9月 20 03:24 .
drwxr-xr-x 3 oslab oslab 4096 9月 20 03:07 ..
-rwxrwxr-x 1 oslab oslab 3588 9月 20 03:24 mbr.elf
-rw-rw-r-- 1 oslab oslab 656 9月 20 03:23 mbr.o
-rwxrw-rw- 1 oslab oslab 610 9月 19 19:22 mbr.s

```

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

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：

```
$ 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。

```
$ ls -al mbr.bin  
  
-rwxrwxr-x 1 oslab oslab 65 9月 20 03:26 mbr.bin
```

然后我们需要将这个mbr.bin真正做成一个 MBR，这一过程需要我们提供的工具。将 lab1-STDID/utlis/genboot.pl 拷贝到你当前的目录下，并且给文件可执行权限

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

Ex4: 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字节等等。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。在实验报告中简述一下。

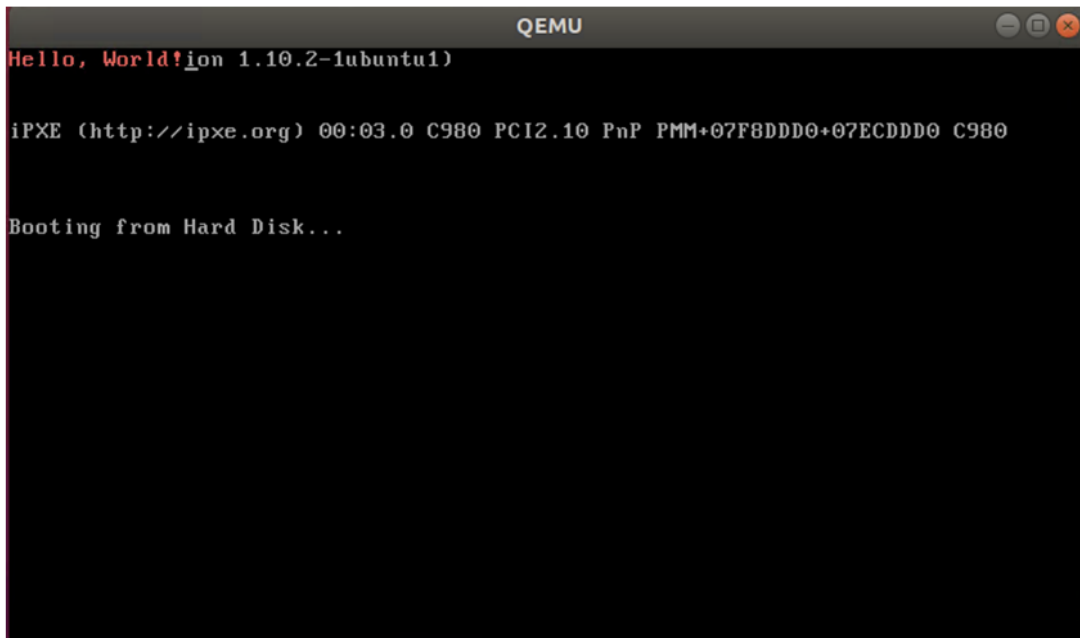
然后利用genboot.pl生成一个MBR，再次查看mbr.bin，发现其大小已经为512字节了

```
$/genboot.pl mbr.bin  
OK: boot block is 65 bytes (max 510)  
$ ls -al mbr.bin  
-rwxrwxr-x 1 oslab oslab 512 9月 20 03:28 mbr.bin
```

一个MBR已经制作完成了，接下来就是查看我们的成果

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

会弹出这样一个窗口：



输出 `hello world` 成功。

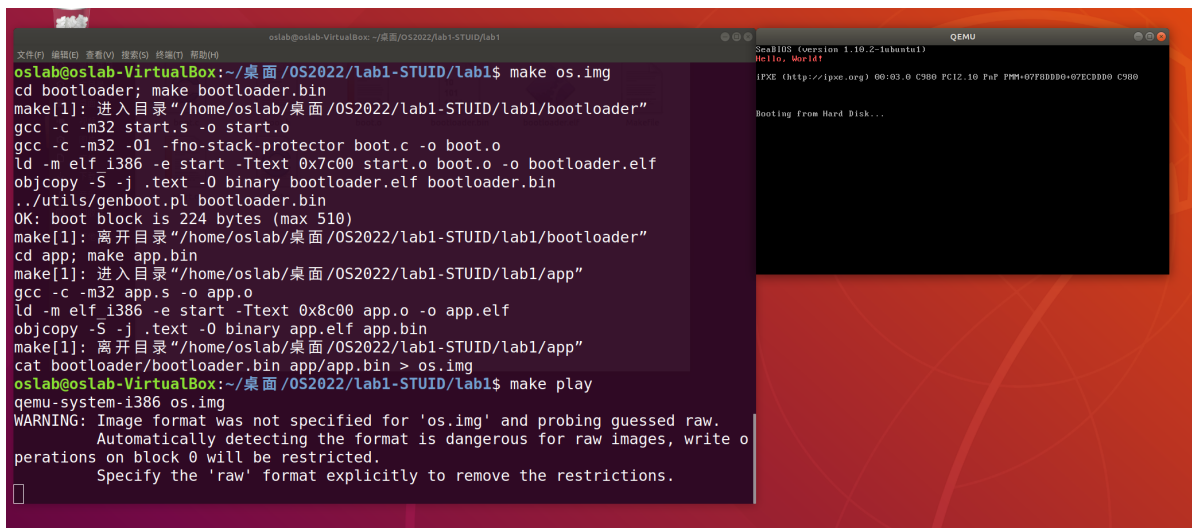
我们看到，每次编写代码后，如果都要进行手动编译链接，整个过程单调却又极其繁琐，而且容易出错，我们可以将其封装到一个自动化脚本中：

如果同学们顺利完成缺失的代码逻辑，在根目录下依次执行 `make os.img` 和 `make play`，屏幕上会输出 `hello, world`

```
$ make os.img
$ make play
```

例如：

假设你正在完成 `lab1.1`，在 `start.s` 写完对应的代码后（其实就是把 `lab1.1` 的代码注释关闭啦，因为 `lab1.1` 我们已经写好了），退回 `labx-STUID` 目录下，执行上述脚本指令，你就可以看到类似于这样的场景（我把 `hello, world` 换了一行）



```
oslab@oslab-VirtualBox: ~/桌面/OS2022/lab1-STUID/lab1
oslab@oslab-VirtualBox:~/桌面/OS2022/lab1-STUID/lab1$ make os.img
cd bootloader; make bootloader.bin
make[1]: 进入目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/bootloader"
gcc -c -m32 start.s -o start.o
gcc -c -m32 -O1 -fno-stack-protector boot.c -o boot.o
ld -m elf_i386 -e start -Ttext 0x7c00 start.o boot.o -o bootloader.elf
objcopy -S -j .text -O binary bootloader.elf bootloader.bin
./utils/genboot.pl bootloader.bin
OK: boot block is 224 bytes (max 510)
make[1]: 离开目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/bootloader"
cd app; make app.bin
make[1]: 进入目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/app"
gcc -c -m32 app.s -o app.o
ld -m elf_i386 -e start -Ttext 0x8c00 app.o -o app.elf
objcopy -S -j .text -O binary app.elf app.bin
make[1]: 离开目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/app"
cat bootloader/bootloader.bin app/app.bin > os.img
oslab@oslab-VirtualBox:~/桌面/OS2022/lab1-STUID/lab1$ make play
qemu-system-i386 os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
[]
```

```
SeaBIOS (version 1.10.2-Inhntn1)
Hello, World!
IPXE (http://ipxe.org) 00:03:0 C800 FC12:10 PaP PPM-07F00D00-07EC0000 C900

Booting from Hard Disk...
```

之后的实验我们同样提供了 `makefile` 脚本

3.2 实模式切换保护模式 (lab1.2)

我们已经在 3.1 节中手把手的领略了一次开发的完整过程是什么，之后的代码以及编译运行就需要同学们自己完成了。

lab1.2 任务：以下任务点是你需要在 lab1.2 中需要完成的（修改 start.s，代码中已通过TODO注释）

- 填写GDT
- 把cr0的低位设置为1
- 显示helloworld

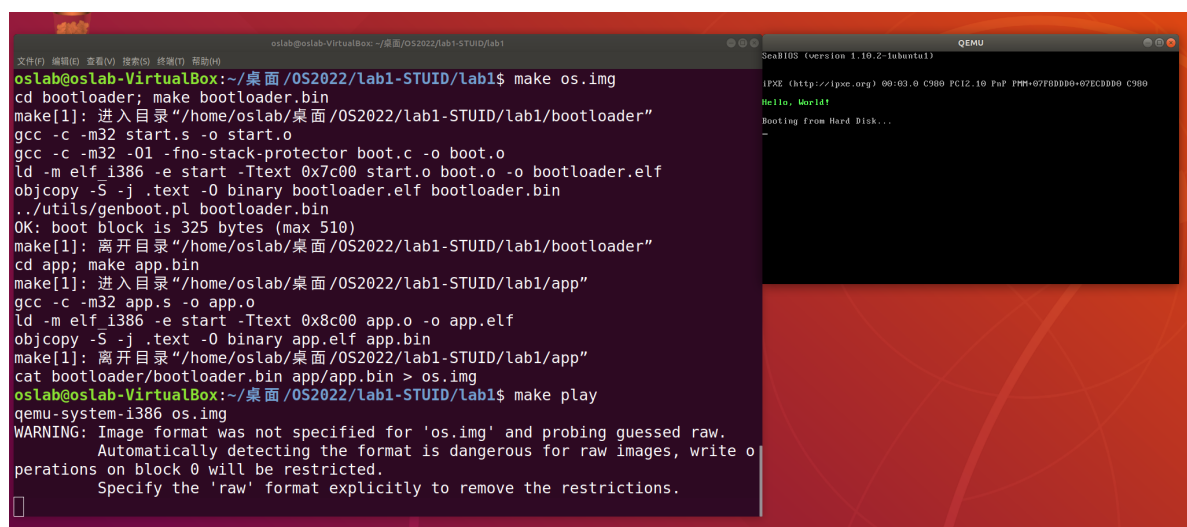
为了开启保护模式，我们要做如下几步：

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

我们需要实现的地方只有三处：

- 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
- 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
- 编译运行，输出 `helloworld!`

如果顺利完成，你能够看见下面的场景



```

oslab@oslab-VirtualBox: ~/桌面/OS2022/lab1-STUID/lab1
oslab@oslab-VirtualBox:~/桌面/OS2022/lab1-STUID/lab1$ make os.img
cd bootloader; make bootloader.bin
make[1]: 进入目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/bootloader"
gcc -c -m32 start.s -o start.o
gcc -c -m32 -O1 -fno-stack-protector boot.c -o boot.o
ld -m elf_i386 -e start -Ttext 0x7c00 start.o boot.o -o bootloader.elf
objcopy -S -j .text -O binary bootloader.elf bootloader.bin
../utils/genboot.pl bootloader.bin
OK: boot block is 325 bytes (max 510)
make[1]: 离开目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/bootloader"
cd app; make app.bin
make[1]: 进入目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/app"
gcc -c -m32 app.s -o app.o
ld -m elf_i386 -e start -Ttext 0x8c00 app.o -o app.elf
objcopy -S -j .text -O binary app.elf app.bin
make[1]: 离开目录"/home/oslab/桌面/OS2022/lab1-STUID/lab1/app"
cat bootloader/bootloader.bin app/app.bin > os.img
oslab@oslab-VirtualBox:~/桌面/OS2022/lab1-STUID/lab1$ make play
qemu-system-i386 os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 60:63:6 C580 PC12.10 PaP PMM+07F0DDB0+07ECDB0 C580
Hello, World!
Booting from Hard Disk...
  
```

3.3 在保护模式下加载OS (lab1.3)

lab1.3: 以下任务点是在本节需要完成的 (完成修改 start.s 和 boot.c 代码):

- 把上一节保护模式部分适配到本节，请注意 lab1.3 不同于 lab1.2 的部分，
 - 仔细观察：lab1.2 最后是陷入 loop 循环中，而 lab1.3 最后却是 jmp bootMain
 - 注意：如果你想要运行 lab1.3，需要将 start.s 中 lab1.1 和 lab1.2 的部分都注释掉
- 填写 bootMain 函数。

本次实验是在保护模式下加载 app (即微型OS), 由于中断关闭, 无法通过陷入磁盘中断调用BIOS进行磁盘读取, 本次实验提供的代码框架中实现了 readSec(void *dst, int offset) 这一接口 (定义于 bootloader/boot.c 中), 其通过读写 (in, out 指令) 磁盘的相应端口 (Port) 来实现磁盘特定扇区的读取。

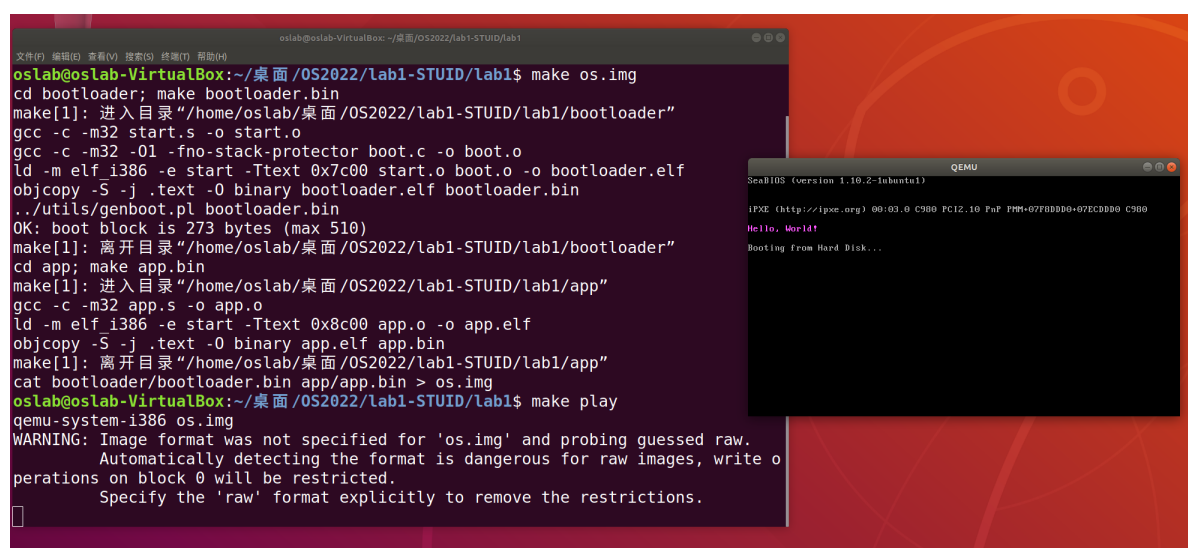
通过上述接口读取磁盘MBR之后扇区中的程序至内存的特定位置并跳转执行 (注意代码框架 app/Makefile 中设置的该Hello World程序入口地址)

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)

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

如果你成功加载app, 即可通过 `app.s` 在屏幕上打印 `hello, world`。



```
oslab@oslab-VirtualBox: ~/桌面/OS2022/lab1-STUID/lab1$ make os.img
cd bootloader; make bootloader.bin
make[1]: 进入目录 "/home/oslab/桌面/OS2022/lab1-STUID/lab1/bootloader"
gcc -c -m32 start.s -o start.o
gcc -c -m32 -O1 -fno-stack-protector boot.c -o boot.o
ld -m elf_i386 -e start -Ttext 0x7c00 start.o boot.o -o bootloader.elf
objcopy -S -j .text -O binary bootloader.elf bootloader.bin
./utils/genboot.pl bootloader.bin
OK: boot block is 273 bytes (max 510)
make[1]: 离开目录 "/home/oslab/桌面/OS2022/lab1-STUID/lab1/bootloader"
cd app; make app.bin
make[1]: 进入目录 "/home/oslab/桌面/OS2022/lab1-STUID/lab1/app"
gcc -c -m32 app.s -o app.o
ld -m elf_i386 -e start -Ttext 0x8c00 app.o -o app.elf
objcopy -S -j .text -O binary app.elf app.bin
make[1]: 离开目录 "/home/oslab/桌面/OS2022/lab1-STUID/lab1/app"
cat bootloader/bootloader.bin app/app.bin > os.img
oslab@oslab-VirtualBox: ~/桌面/OS2022/lab1-STUID/lab1$ make play
qemu-system-i386 os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
[]
```

```
QEMU
SeaBIOS (version 1.10.2-ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C900 FC12:10 FaP PMM-07F00DD0-07EC0DD0 C900
Hello, World!
Booting from Hard Disk...
```

Ex5: 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。简略描述即可。

4. 作业规范与提交

4.1 作业规范

- **学术诚信:** 如果你确实无法完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数; 但若发现抄袭现象, 抄袭双方(或团体)在本次实验中得0分.
- 实验源码提交前需清除编译生成的临时文件, 虚拟机镜像等无关文件
- 请你在实验截止前务必确认你提交的内容符合要求(格式, 相关内容等), 你可以下载你提交的内容进行确认. 如果由于你的原因给我们造成了不必要的麻烦, 视情况而定, 在本次实验中你将会被扣除该次实验得分的部分分数, 最高可达50%
- 实验不接受迟交, 一旦迟交按**学术诚信**给分
- **本实验给分最终解释权归助教所有**

4.2 提交格式

本次实验提供一个示范代码框架, 你所提交的文件应该具有如下目录:

```
lab1-STUID #待修改
├── labx
│   ├── Makefile
│   ├── app
│   │   ├── Makefile
│   │   └── app.s      #用户程序
│   ├── bootloader
│   │   ├── Makefile
│   │   ├── boot.c     #加载磁盘上的用户程序
│   │   ├── boot.h     #磁盘I/O接口
│   │   └── start.s    #引导程序
│   └── utils
│       └── genboot.pl  #生成MBR
└── report
    └── STUID.pdf      #你的实验报告, STUID待替换
```

- 你所提交的代码中应该包含两个部分:

- labX 文件夹下是你的工程代码文件, X代表这是第几次实验
- report 文件夹下存放实验报告, 要求为 pdf 格式
- 在提交作业之前先将 STUID 更改为**自己的学号**, 例如 `mv lab1-STUID lab1-191220000`
- 然后用压缩工具将源文件夹压缩成一个zip包
- 压缩包以**学号**命名, 例如 `lab1-191220000.zip` 是符合格式要求的压缩包名称
- 为了防止出现编码问题, 压缩包中的所有文件名都不要包含中文
- 我们只接受pdf格式, 命名只含学号的实验报告, 不符合格式的实验报告将视为没有提交报告. 例如 `191220000.pdf` 是符合格式要求的实验报告, 但 `191220000.docx` 和 `191220000张三实验报告.pdf` 不符合要求
- 作业提交网站 <http://cslabcms.nju.edu.cn>

5. 实验报告内容

你**必须**在实验报告中描述以下内容

- **姓名、学号、邮箱**等信息, 方便我们及时给你一些反馈信息
- **实验进度**。简单描述即可, 例如"我完成了所有内容", "我只完成了xxx"。缺少实验进度的描述, 或者描述与实际情况不符, 将被视为没有完成本次实验
- **实验结果**。贴图或说明都可, 不需要太复杂, 确保不要用其他同学的结果, 否则以抄袭处理
- **实验修改的代码位置**, 简单描述为完成本次实验, 修改或添加了哪些代码。不需要贴图或一行行解释, 大致的文件和函数定位就行

你可以**自由选择**报告的其它内容。你不必详细地描述实验过程, 但我们鼓励你在报告中描述如下内容:

- **你遇到的问题和对这些问题的思考**
- 对讲义或框架代码中某些**思考题**的看法
- 或者你的其它想法, 例如**实验心得**, 对提供帮助的同学的感谢等

认真描述实验心得和想法的报告将会获得分数的奖励；思考题选做，完成了也不会得到分数的奖励，但它们是经过精心准备的，可以加深你对某些知识的理解和认识。如果你实在没有想法，你可以提交一份不包含任何想法的报告，我们不会强求。但请**不要**

- 大量粘贴讲义内容
- 大量粘贴代码和贴图, 却没有相应的详细解释(让我们明显看出来是凑字数的)

来让你的报告看起来十分丰富，编写和阅读这样的报告毫无任何意义，你也不会因此获得更多的分数，同时还可能带来扣分的可能。