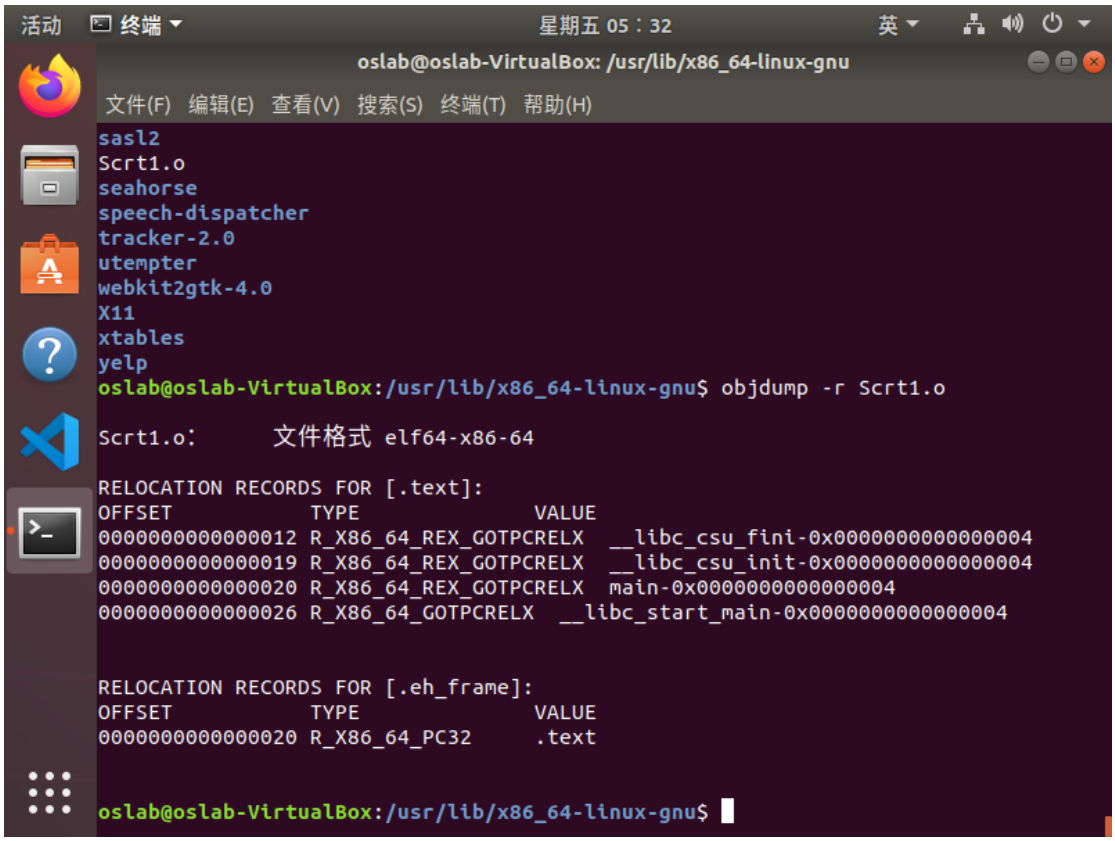
OSlab1 实验报告 201220180 李全昊

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



这里找到路径蛮难的，同学告诉我说直接复制省略号就行，很神奇。这里可以看到确实有 main 函数存在

二．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）

答：第一条指令是　ljmp　$0xf000,$0xe05b；位于内存中 0xFFFF0 的位置；

CS 设置为 0xf000, IP 设置为 0xfff0。

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 0xfff0。

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。由此便可以找到开机后执行的第一条指令的位置 0xFFFF0

这样设计的好处是：在第一条指令的后面只有 16 个字节，啥也干不了。所以，不会在实地址模式中执行额外的程序，起到保护的作用

三．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）

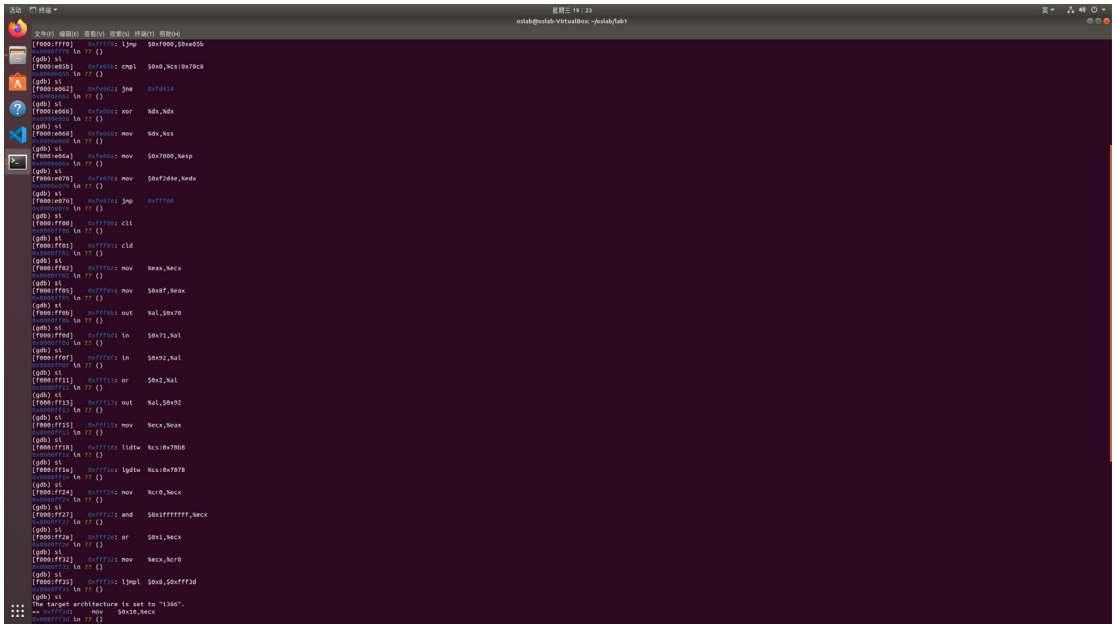
答：qemu-gdb: qemu-system-i386 -s -S os.img

qemu-nox-gdb: qemu-system-i386 -nographic -s -S os.img

两者都是开启 os.img 的调试功能。-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 的链接。-S: 一个暂时不让 qemu 启动的选项

后者加了-nographic 参数，这样就可以不让窗口弹出

四．继续用 si 看见了什么？请截一个图，放到实验报告里



五．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。

答：1) 中断向量表：中断源的识别标志，可用来形成相应的中断服务程序的入口地址或存放中断服务程序的首地址称为中断向量。是指中断服务程序入口地址的偏移量与段基址，一个中断向量占据 4 字节空间。中断向量表是 8086 系统内存中最低端 1K 字节空间，它的作用就是按照中断类型号从小到大的顺序存储对应的中断向量，总共存储 256 个中断向量。在中断响应过程中，CPU 通过从接口电路获取的中断类型号（中断向量号）计算对应中断向量在表中的位置，并从中断向量表中获取中断向量，将程序流程转向中断服务程序的入口地址。

2) 如何输出 helloworld:

关键在于：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.

```
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
```

首先将栈指针设置为 0x7d00 的位置，然后将想输出的 message 信息和长度（就是汇编代码中的.str: hello,world!）压栈。最后调用输出的函数。

在输出函数中，我们可以看到将系统调用号和其他参数存放到相应寄存器，最后执行 int 0x10 来开启中断，输出字符串到屏幕，达成最后的效果。

补充：在汇编代码转换到 elf 文件之后，elf 文件体积过大，超过一个扇区，因此需要压缩，并且还需要在后面添加魔数，以便让 BIOS 认出这就是 MBR 的代码段

六．为什么段的大小最大为 64KB，请在报告上说明原因

答：一个段中每个内存单元的地址表示为 段地址:偏移地址

其中，段的长度是偏移地址可以取的数值规定的，在 8086cpu 中，偏移地址使用一个

16 位的二进制数表示，其表示范围是（0000H:0FFFFH），总共有 2^16(2 的 16 次方)=64K 个不同的取值，一个内存单元使用 1 个偏移地址，故一个段的大小是 64K

七．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？

答：我觉得不可以，因为下文讲 mbr.bin 压缩后还要扩展至 512byte 的时候才可以执行，说明必须要达到扇区空间也就是 512byte，所以不可以直接执行 300byte 的 elf 文件，还需要扩展，并且加上最后两个字节的魔数

八．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）

答：1. ld -m elf_i386 -e start -Ttext 0x7c00 mbr.o -o mbr.elf

这里-m 参数是在选择硬件模拟，这里显示使用 elf_i386 作为硬件；-e 参数表示从什么 symbol 开始执行我的程序，这里表明从 start 这个位置开始执行；-Ttext 表示，当创建一个 ELF 可执行文件的时候，将 text 段的第一个字节放在什么位置，这里放在 0x7c00 处，把 MBR 的内容从磁盘调入地址为 0x7c00 的地方。最后三个单词的意思就是将给定的.o 文件链接成为给定名称的 elf 文件

2. objcopy -S -j .text -O binary mbr.elf mbr.bin

-S 参数去除了那些包含了调试信息的部分；-j 表示将指定的段从 input 文件 copy 到 output 文件中，这里只复制.text 段；-O 指定输出的格式，这里是二进形式；最后两个单词分别是 input 文件和 output 文件

九．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。

答：如果大于 510 比特，会输出一个报错信息，并且直接退出程序。

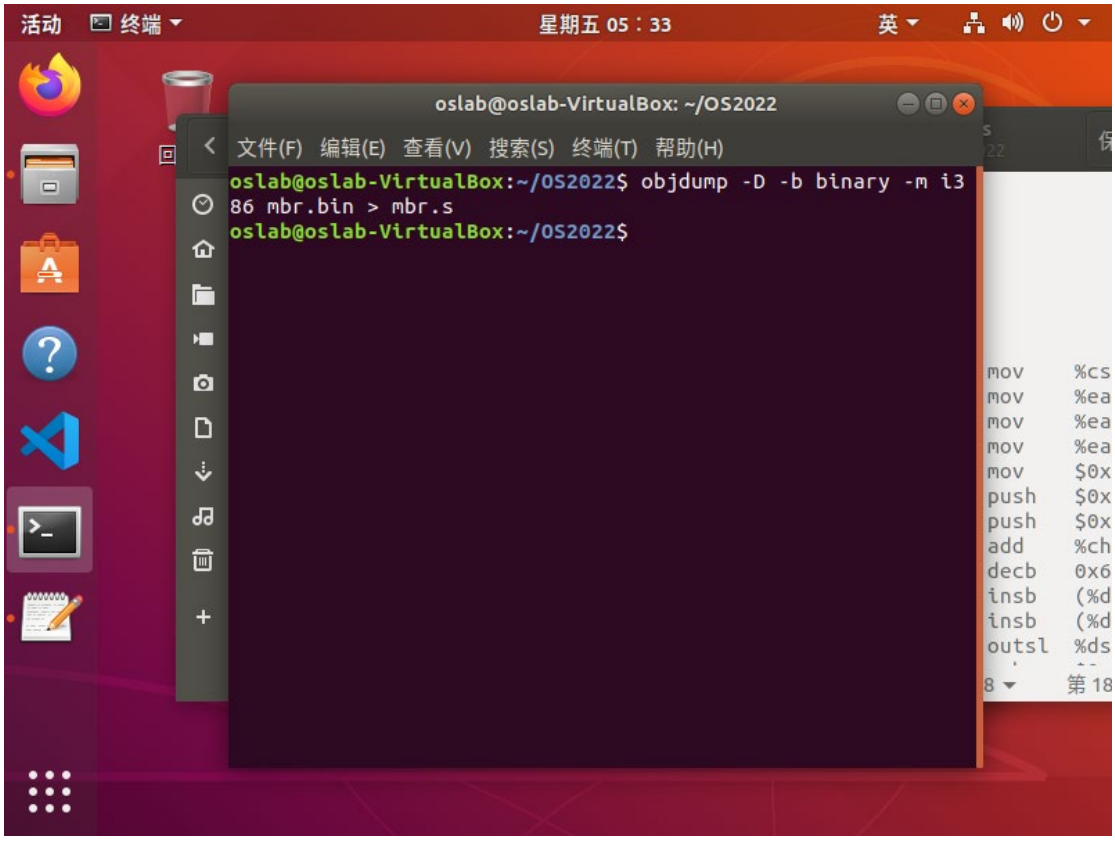
否则会输出一个表示大小正确的信息，然后在该文件后面补上（510-$n）个"\0"和"\x55\xAA"两个字符，凑成 512 个字节。

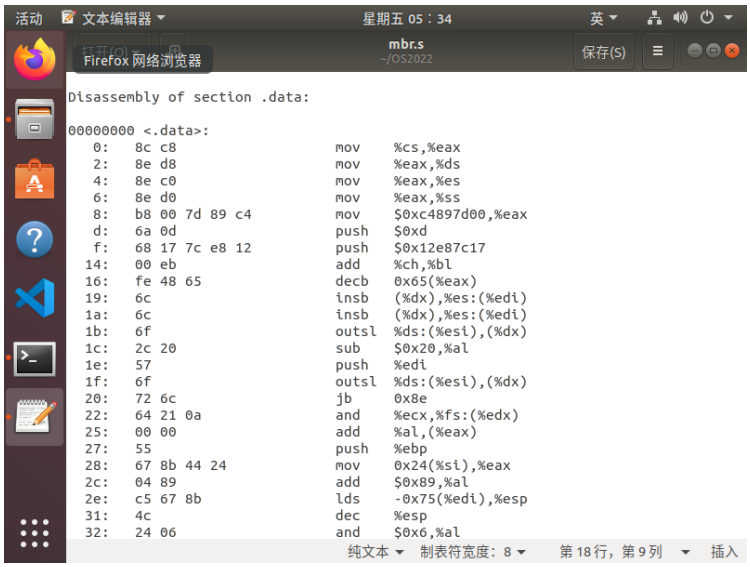
因为如果不检查它是否大于 510 字节，就没有空间加上那两个魔数，以对 BIOS 表明他是 MBR，所以必须进行检查

十．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

答： 使用 objdump -D -b binary -m i386 mbr.bin > mbr.s 进行反汇编

-D 表示对全部文件进行反汇编，-b 表示二进制，-m 表示指令集架构，mbr.bin 就是我们要反汇编的二进制文件





以上内容就是我们输入的 mbr.s 中的汇编代码

Challenge 1:

我选择采用第一种方法:

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

代码整体思路其实非常简单，就是用 fopen 打开该二进制文件，然后

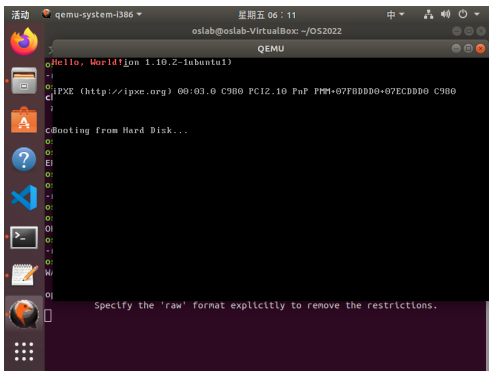
用 fseek(fp, 0, SEEK_END); 定位到文件末 再用 n = ftell(fp); 计算文件长度

再接着执行原本 genboot.pl 文件的功能，判断文件长度是否大于 510 字节，如果大于，就

输出报错信息；否则就补上'\0'字节以及两个字节的魔数形成 512 字节的最后的文件

```
oslab@oslab-VirtualBox:~/OS2022$ ls -al mbr.bin
-rwxrwxr-x 1 oslab oslab 65 3月   4 06:00 mbr.bin
```

```
oslab@oslab-VirtualBox:~/OS2022$ ./challenge
OK: boot block is 65bytes(max 510)
oslab@oslab-VirtualBox:~/OS2022$ ls -al mbr.bin
-rwxrwxr-x 1 oslab oslab 512 3月   4 06:00 mbr.bin
oslab@oslab-VirtualBox:~/OS2022$
```



操作过程如上所示，可以输出 hello, world!

十一. 请回答为什么三个段描述符要按照 cs，ds，gs 的顺序排列?

```
 15                        3   1 0
+--------------------------+-+---+
|                           |T|   |
|           INDEX           | |RPL|
|                           |I|   |
+--------------------------+-+---+
TI - TABLE INDICATOR, 0 = GDT, 1 = LDT
RPL - REQUESTOR'S PRIVILEGE LEVEL
```

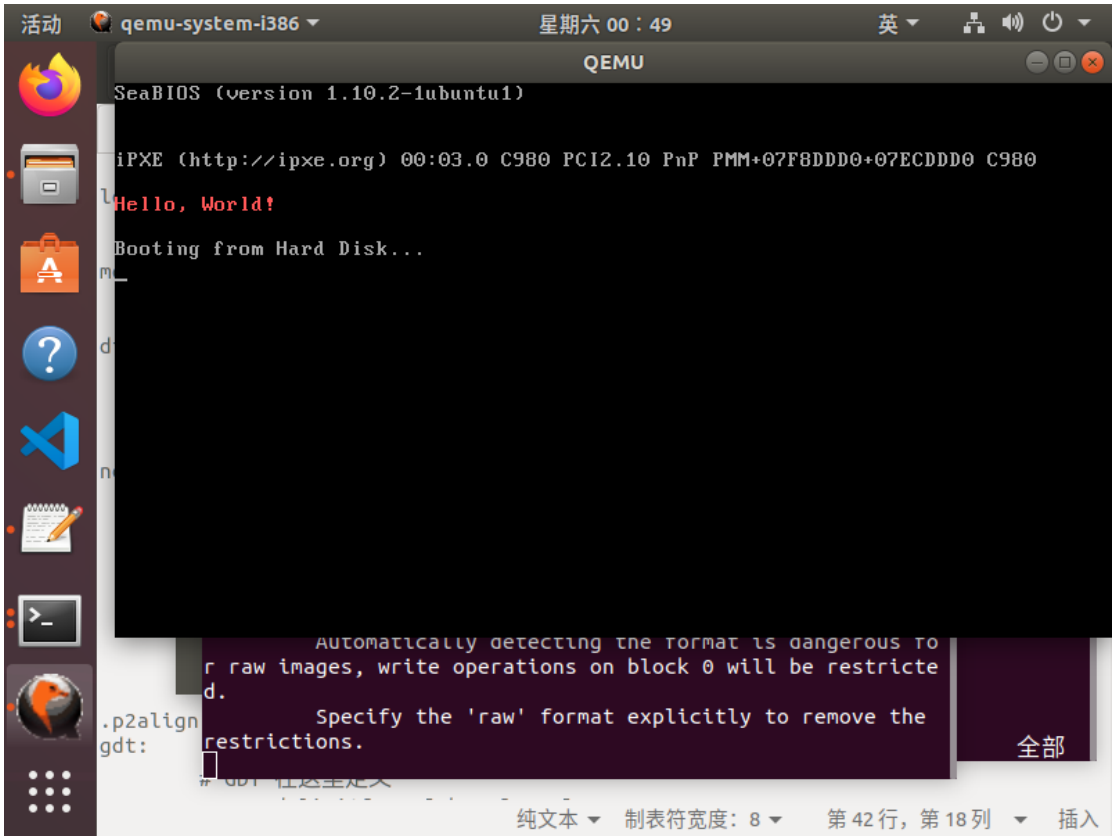
汇编代码中 ds 寄存器是 0x10，gs 寄存器是 0x18，由此可以得出 Ds 的序号是 2，gs 的序号是 3

至于 CS 寄存器比较难看出来：ljmp 指令把第一个操作数 0x08 存入 CS 寄存器，然后再跳到 start32 处，因此 cs 的序号是 1

十二. 请回答 app.s 是怎么利用显存显示 helloworld 的。

答： 首先他将字符串长度以及字符串压栈，然后调用 displaystr 函数；

在这个函数中，我们可以看到将字符串长度存到 ecx 寄存器，将字符串首地址存到 ebx 寄存器，然后将我们想要在显存中显示的位置存到 edi 寄存器；movb $0x0c, %ah 的意思是黑底红字。执行完之后，我们进入会循环执行的 nextChar 函数，这个函数会挨个将每个字符输出到给定的显存位置%gs:(%edi)，然后将 edi,ebx 与 ecx 更新。最终我们就可以得到输出 hello world 的效果。



效果见上图

十三. 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。

答： 将 bootloader 目录下的 bootloader.bin 与 app 目录下的 app.bin 合并为一个文件，名为 os.img

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

答： 不可以。因为 MBR 被加载到 0x7c00 开始的区域，如果 MBR 再将 app 也加载到 0x7c20 的内存区域的话，会把原来 MBR 的代码段（0 号扇区）给冲掉，发生不可预测的异常

十五. 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。

答： 在加电后的第一条指令都是跳转到 BIOS 固件进行开机自检。开机之后从 0FFF:0000H 处开始执行 BIOS，然后 jmp far 到 POST 过程，经过 int 19，自举之后，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 处，然后跳转到 0x7c00 开始执行 MBR。加载完 boot loader 之后，转入保护模式，然后 boot loader 就会调用 bootMain 函数，将操作系统加载到内存中并执行

（一些补充：qemu-system-i386 指令其实就是把 i386 机器用 QEMU 模拟之后，接收一个 os.img 文件，将他当作自己的磁盘，然后将这上面的 MBR 转入内存。

至于为什么 app.bin 在第一个扇区是有原因的：关键点就在于 os.img 是如何形成的，它是由 bootloader.bin 与 app.bin 拼接而成，这里 bootloader.bin 已经是 512 bytes，那么如果将 os.img 作为自己的磁盘的话，那么 app.bin 自然就在磁盘中的第 1 号扇区之中了）

两个 task 的思路：

Task1:

要转到保护模式中需要进行一些步骤
将 cr0 最低位置 1，这个已经给出提示，就是把它存储到 eax 再把 eax 末位置为 1，再切换回去即可。
填写 GDT 只要按照给定的格式，然后上网查查每个段的 base 与 limit 和 type 字段即可。
最后的 helloworld 更加简单，直接把 app.s 中的汇编代码复制过来就行
Task2:
首先把 task1 的开启保护模式搬过来，然后直接写 jmp bootMain 就可以转到 bootmain 函数执行，将 app 加载到内存的 0x8c00 处，然后开始执行对应的 app 程序
其中 bootMain 函数的写法有两种，都需要先调用 readSect 将 app.bin 装到内存 0x8c00 的地方。然后有两种做法：其一是用函数指针指向这个内存地址，然后调用这个函数，其二是直接采用内联汇编的办法在 asm 中直接写 jmp 到这个内存地址即可。