

**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)

答

```
oslab@oslab-VirtualBox: /usr/lib/x86_64-linux-gnu$ objdump -r -d Scrt1.o

Scrt1.o:      文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <_start>:
 0:  31 ed                xor     %ebp,%ebp
 2:  49 89 d1             mov     %rdx,%r9
 5:  5e                  pop     %rsi
 6:  48 89 e2             mov     %rsp,%rdx
 9:  48 83 e4 f0          and     $0xfffffffffffffff0,%rsp
 d:  50                  push    %rax
 e:  54                  push    %rsp
 f:  4c 8b 05 00 00 00 00  mov     0x0(%rip),%r8      # 16 <_start+0x16>
                        12: R_X86_64_REX_GOTPCRELX __libc_csu_fini-0x4
16:  48 8b 0d 00 00 00 00  mov     0x0(%rip),%rcx    # 1d <_start+0x1d>
                        19: R_X86_64_REX_GOTPCRELX __libc_csu_init-0x4
1d:  48 8b 3d 00 00 00 00  mov     0x0(%rip),%rdi    # 24 <_start+0x24>
                        20: R_X86_64_REX_GOTPCRELX main-0x4
24:  ff 15 00 00 00 00    callq  *0x0(%rip)        # 2a <_start+0x2a>
                        26: R_X86_64_GOTPCRELX __libc_start_main-0x4
2a:  f4                  hlt
```

可以看到确实跳转到了 main 函数

**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:

1. 电脑开机第一条指令的地址是什么, 这位于什么地方?

0xfffff0: 1 jmp \$0xf000, \$0xe05b

0x0000ffff in ?? ()

2. 电脑启动时 CS 寄存器和 IP 寄存器的值是什么?

(1) eip 0xffff0 0xffff0

(2) cs 0xf000 61440

3. 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)

1 jmp \$0xf000, \$0xe05b

因为为确保 BIOS 在开机后率先获得控制权, 便将其物理地址范围固定在了 0xf0000 到 0xfffff。

qemu 也模拟了这个过程, 并通过设置 CS 和 IP 得到分段地址 0xf000:f00, 翻译后得到物理地址 0xfffff0。

xfffff0 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, 就跳走了~

**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)

首先查阅资料获得参数的含义如下：

```
qemu-nox-gdb:
    qemu-system-i386 -nographic -s -S os.img
-s                shorthand for -gdb tcp::1234
-S                freeze CPU at startup (use 'c' to start execution)
```

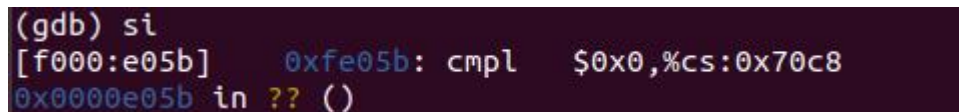
```
gdb:
    gdb -n -x ./gdbconf/.gdbinit
-n, -nx          禁止执行任何“.gdbinit”初始化文件中的命令。

-x file          执行指定文件中的 gdb 命令。
```

两个都是通过 gdb 开始调试“os.img”文件，加了“nographic”可以不用弹窗

**Exercise4:** 继续用 si 看见了什么？请截一个图，放到实验报告里。

可发现发生了跳转。



```
(gdb) si
[f000:e05b]    0xfe05b: cmpl    $0x0,%cs:0x70c8
0x0000e05b in ?? ()
```

**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。

中断向量表：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。

**MBR:**

我们在 mbr.s 中通过 int \$0x10，调用了 BIOS 0x10 中断，进而进行了屏幕上的显示。

我们通过汇编代码 mbr.s 中写入有关显示字符的各种操作，然后经过一系列的编译链接，得到了一个 mbr.elf 文件。再 objcopy 减少了 mbr 程序的大小。然后通过了 genboot.pl 文件，并在其中写入规格化 mbr.min 的代码，执行后是的 mbr 文件满足了格式要求，然后便执行使得 helloworld 得以显示。

**Exercise6:** 为什么段的大小最大为 64KB，请在报告上说明原因。

$(2^6) * (2^{10}B) = 2^{16} B$ ，而 8086 的寄存器也为 16 位，故最多只能给出 64KB 以内的物理地址

**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？

不可以，我们需要让文件变成标准的 MBR 的格式，即 512 字节，并且末尾两个字节需为 0x55 和 0xaa

#### Exercise8:

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

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

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）

-m elf\_i386: 使生成的可执行文件格式为 elf\_i386。

-e start: 指定程序入口为"start"

-Ttext 0x7c00: 因为 BIOS 依次将设备的首扇区加载到内存 0x7c00 的位置, 所以此处让代码段从 0x7c00 开始。

mbr.o -o mbr.elf: 链接 mbr.o 输出至 mbr.elf.

-S : 即 --strip-all, 清除所有符号和重定位信息

-j .text : 只保留.text 节

-O binary: 使得输出文件格式为二进制

**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。

如果大于 510 字节，则会报错终止；

如果小于 510 字节，则会在后面补充 '\0' 至 510 个，然后在末尾加上 0x55 和 0xaa 两个魔数。

目的是为了 MBR 符合 512 个字节的标准格式，并且加入魔数后 BIOS 可以识别出 MBR。

**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

可以看到 mbr 的代码部分，并且有魔数 "0x55" "0xaa" 被翻译成了指令

```

oslab@oslab-VirtualBox:~/桌面/lab/lab1/OS2022$ objdump -D mbr.bin -b binary -m i386

mbr.bin:      文件格式 binary

Disassembly of section .data:

00000000 <.data>:
 0:  8c c8          mov     %cs,%eax
 2:  8e d8          mov     %eax,%ds
 4:  8e c0          mov     %eax,%es
 6:  8e d0          mov     %eax,%ss
 8:  b8 00 7d 89 c4  mov     $0xc4897d00,%eax
 d:  6a 0d          push    $0xd
 f:  68 17 7c e8 12  push    $0x12e87c17
14:  00 eb          add     %ch,%bl
16:  fe 48 65       decb    0x65(%eax)
19:  6c             insb    (%dx),%es:(%edi)
1a:  6c             insb    (%dx),%es:(%edi)
1b:  6f             outsl   %ds:(%esi),(%dx)
1c:  2c 20          sub     $0x20,%al
1e:  57             push    %edi
1f:  6f             outsl   %ds:(%esi),(%dx)
20:  72 6c          jb      0x8e
22:  64 21 0a       and     %ecx,%fs:(%edx)
25:  00 00          add     %al,(%eax)
27:  55             push    %ebp
28:  67 8b 44 24     mov     0x24(%si),%eax
2c:  04 89          add     $0x89,%al
2e:  c5 67 8b       lds     -0x75(%edi),%esp
31:  4c             dec     %esp
32:  24 06          and     $0x6,%al
34:  b8 01 13 bb 0c  mov     $0xcbb1301,%eax
39:  00 ba 00 00 cd 10 add     %bh,0x10cd0000(%edx)
3f:  5d             pop     %ebp
40:  c3             ret
...
1fd:  00 55 aa       add     %dl,-0x56(%ebp)

```

**Exercisel1:** 请回答为什么三个段描述符要按照 cs, ds, gs 的顺序排列?

```

# 长跳转切换到保护模式
data32 ljmp $0x08, $start32

.code32
start32:
    movw $0x10, %ax # setting data segment selector
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %fs
    movw %ax, %ss
    movw $0x18, %ax # setting graphics data segment selector

```

我们可以读出 CS、DS、GS 的段选择子，分别 0x08,0x10,0x18,而作为在 GTD 中的编号的是前 13 位，分别是 1, 2, 3。所以 CS 是第一项、DS 是第二项、GS 是第三项

**Exercisel2:** 请回答 app.s 是怎么利用显存显示 hello world 的。

首先传入参数，再跳转至 displayStr 执行字符显示。

在 displayStr 中，ebi 首先存储了  $(80*5+0)*2$ ，后续每打印一个字符++2，实际上就是存储了在图像段的相对位置，一个字符又两个字节存储。

然后由%gs 给出高位，和 ebi 进行拼接便得到了我们需要显示字符应该存储的地址空间，如此便实现了字符的显示。

**Exercisel3:** 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.bin 和 app.bin 里的内容依次装载至 os.img 中

**Exercisel4:** 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，因为我们在“写一个自己的 MBR”时说过，会将 512 字节的 MBR 加载至 0x7c00 处，如果将 app 读至 0x7c20 则会产生覆盖

**Exercisel5:** 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：电脑加电之后，经第一条指令跳转至 BIOS 固件进行开机自检，然后将 MBR 加载至 0x7c00 处。接着通过类似于“bootloader”进行开启保护模式的一系列操作，最后通过 bootmain 函数，将 app.bin 的内容读至 0x8c00 处，并跳转至 0x8c00 处，此时便成功加载 app 即 OS。

**Challenge:** 第一种（基础）：还是使用提供的汇编语言，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

思路：int main(int argc, char\*\*argv) 以得到传入参数，然后采用文件读取操作把文件的内容存入到一个 buf[512] 中，然后便在 buf 后面补“\0”，并且最后两个字节改为“0x55”和“0xaa”，即可，然后再将 buf 写回文件即可。