

# 南京大学本科生实验报告

课程名称：操作系统

任课教师：叶保留

助教：尹熙喆/水兵

学院	计算机科学与技术	专业（方向）	计算机科学与技术
学号	201220062	姓名	黄子睿
Email	201220062@smail.nju.edu.cn	开始/完成日期	3/8-3/9

## 1. 实验名称：

OS LAB 1

## 2. 实验目的：

安装完善并熟悉实验环境与相关软件，包括实验平台 linux, git & github 以及相关汇编与 gbd 使用方法。通过该实验，希望能过回顾 ics 相关保护模式的知识，并且了解 BIOS 与 MBR 的运行机理。

## 3. 实验内容：

### 1.回答 Exercise

**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：反汇编可见下述代码：

/usr/lib/gcc/x86\_64-linux-gnu/7/../../../../x86\_64-linux-gnu/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    $0xffffffffffff0,%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

```

从中可见语句：

```

24:  ff 15 00 00 00 00    callq *0x0(%rip)      # 2a <_start+0x2a>
                                26: R_X86_64_GOTPCRELX    __libc_start_main-0x4

```

就是 start 进入 main 函数的入口指令，因此“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”的想法得到了验证。

## 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）

答：1.电脑开机的第一条指令地址在 0xffff0。

2.执行第一条指令时，CS 的值是 0xf000，而 IP 的值是 0xffff0。根据实模式下物理寻址的规则，此时物理地址为 $(\%cs) \ll 4 + (\%ip) = 0xffff0$ ，正是 BIOS 区域的开始处。

3.第一条指令是[f000:fff0] 0xffff0: ljmp \$0xf000,\$0xe05b

通过查询 ics 手册，可以该长跳转指令的结果是将 cs 赋值为 0xffff0，ip 赋值为 0xe05b。

## 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）

```

1. The operations after several 'si' instructions:
2. (gdb) si
3. [f000:e05b]    0xfe05b: cmpl    $0x0,%cs:0x70c8
4. . . . . .
5. 0x0000e066 in ?? ()
6. (gdb) si
7. [f000:e068]    0xfe068: mov     %dx,%ss #set ss register
8.                #设置 ss
9. 0x0000e068 in ?? ()
10. (gdb) si
11. [f000:e06a]    0xfe06a: mov     $0x7000,%esp #set esp register
12.                #设置 esp
13. 0x0000e06a in ?? ()

```

```

14. (gdb) si
15. [f000:e070]    0xfe070: mov    $0xf2d4e,%edx
16. 0x0000e070 in ?? ()
17. (gdb) si
18. [f000:e076]    0xfe076: jmp    0xffff00
19. 0x0000e076 in ?? ()
20. (gdb) si
21. [f000:ff00]    0xffff00: cli    #cli block interrupt
22.                                     #cli 关中断
23. 0x0000ff00 in ?? ()
24. (gdb) si
25. [f000:ff01]    0xffff01: cld
26.     #The instructions after cld is related to in and out
27.     #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)
28. 0x0000ff01 in ?? ()
29. (gdb) si
30. . . . . .
31. (gdb) si
32. [f000:ff15]    0xffff15: mov    %ecx,%eax
33. 0x0000ff15 in ?? ()
34.     #然后用 lidt 与 lgdt 加载 IDTR 与 GDTR (ICS 学过, 跟保护模式有关)
35. (gdb) si
36. [f000:ff18]    0xffff18: lidt   %cs:0x70b8
37. 0x0000ff18 in ?? ()
38. (gdb) si
39. [f000:ff1e]    0xffff1e: lgdtw  %cs:0x7078
40. 0x0000ff1e in ?? ()
41. (gdb) si
42. [f000:ff24]    0xffff24: mov    %cr0,%ecx

```

具体的截图为：

```
oslab@oslab-VirtualBox: ~/lab1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
warning: Can not parse XML target description; XML support was disabled at compile time
+ target remote localhost:1234
warning: No executable has been specified and target does not support determining executable automatically. Try using the "file" command.
The target architecture is set to "i8086".
[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
(gdb) si
[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x70c8
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xfd414
0x0000e062 in ?? ()
(gdb) si
[f000:e066] 0xfe066: xor %dx,%dx
0x0000e066 in ?? ()
(gdb) si
[f000:e068] 0xfe068: mov %dx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a] 0xfe06a: mov $0x7000,%esp
0x0000e06a in ?? ()
(gdb) █
```

Exercise 5: 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：关于中断向量表：

IA-32 采用中断向量模式，这是一种通过硬件识别中断与异常源的方式。在这种方式下异常或中断程序的首地址成为中断向量，所有中断向量放在一个表中，称为中断向量表。每个异常与中断都被设定一个中断类型号，中断向量存放的位置就与对应的中断类型号相关。

对于实例 MBR，其利用向量中断方式输出的方式是：

1. **displayStr:**
2. **pushw %bp**
3. **movw 4(%esp), %ax** #此时 mem[%esp+4]是 message 的首地址，此时的栈是 2 字节的
4. **movw %ax, %bp** #此时 bp 指向 message 的首地址
5. **movw 6(%esp), %cx** #%cx 存放 13，是字符串 message 的长度
6. **movw \$0x1301, %ax** #
7. **movw \$0x000c, %bx** #
8. **movw \$0x0000, %dx** #
9. **int \$0x10** #调用 0x10 号中断，
10. **popw %bp**
11. **ret**

注释中给出了对关键汇编 displayStr 的解释。主要完成的任务是准备调用中断需要的各个参数，存储在约定好对的寄存器中。虽有通过软中断 int \$0x10 进入 IBIOS 对屏幕及显示器所提供的服务程序。

**Exercise 6：为什么段的大小最大为 64KB，请在报告上说明原因。**

答：实模式下作为段内地址的 ip 寄存器是 16bit 的，所以它能表示的最大空间就是  $2^{16}\text{bit} = 64\text{KB}$ 。

**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？**

答：不行。由于一个扇区的大小是 512B，并且 mbr 文件需要在末尾加上 0x55,0xaa 的魔数，所以一来这个文件由于缺少魔数无法被识别为 BIOS 文件，并且在 300B 到 512B 中间的内存可能是乱码，使得它无法运行。

**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）**

**\$ld -m elf\_i386 -e start -Ttext 0x7c00 mbr.o -o mbr.elf**

1.-m emulation

模拟仿真链接器。

2.-e entry Use entry as the explicit symbol for beginning execution of your program, rather than the default entry point.

指示程序的入口地址，而非使用默认的入口地址。

3.-Ttext 指 elf 中的代码段

4.-o output 指示输出文件

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

1.-S --strip-all Do not copy relocation and symbol information from the source file. 指在拷贝时省略源文件 elf 的重定位信息与符号信息。

2. -j sectionpattern

--only-section=sectionpattern

Copy only the indicated sections from the input file to the output file.

指仅仅拷贝指令中指示的段，这里是.text 代码段。

3.-O bfdname

--output-target=bfdname

Write the output file using the object format bfdname.

指输出文件，并以 bfdname 方式接受输入与生成输出。(bfdname 是 BFD 库中描述的标准格式名。如果不指明源文件格式，objcopy 会自己去分析源文件的格式，然后去和 BFD 中描述的各种格式比较，从而得知源文件的目标格式名)

**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。**

答：在 genboot.pl 中不难发现：

```
1. $n = sysread(SIG, $buf, 1000);  
2. if($n > 510){  
3.     print STDERR "ERROR: boot block too large: $n bytes (max 510)\n";  
4.     exit 1;  
5. }
```

如果文件的大小超过 510B，512B 的扇区空间中就没有 2 字节的魔数的位置，因此程序将报错。

**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**

答：用 i8086 反汇编 .bin 文件，即指令 `obidump -D -b binary -m i8086 OS2022/mbr.bin > mymbr/mbr.asm`，可以得到正确的反汇编结果。

不难看出 mbr.bin 事实上就是 mbr.elf 的 .text 部分，但是在文件的末尾，mbr.bin 增加了两字节的魔数 0x55aa，并且对多余部分填充了 0。

具体的结果如下图所示：

```
OS2022/mbr.bin: 文件格式 binary
Disassembly of section .data:
00000000 <.data>:
0: 8c c8      mov     %cs,%ax
2: 8e d8      mov     %ax,%ds
4: 8e c0      mov     %ax,%es
6: 8e d0      mov     %ax,%ss
8: b8 00 7d   mov     $0x7d00,%ax
b: 89 c4      mov     %ax,%sp
d: 6a 0d      push    $0xd
f: 68 17 7c   push    $0x7c17
12: e8 12 00   call    0x27
15: eb fe      jmp     0x15
17: 48         dec     %ax
18: 65 6c      gs insb (%dx),%es:(%di)
1a: 6c         insb    (%dx),%es:(%di)
1b: 6f         outsw   %ds:(%si),(%dx)
1c: 2c 20      sub     $0x20,%al
1e: 57         push    %di
1f: 6f         outsw   %ds:(%si),(%dx)
20: 72 6c      jb      0x8e
22: 64 21 0a   and     %cx,%fs:(%bp,%si)
25: 00 00      add     %al,(%bx,%si)
27: 55         push    %bp
28: 67 8b 44 24 04 mov     0x4(%esp),%ax
2d: 89 c5      mov     %ax,%bp
2f: 67 8b 4c 24 06 mov     0x6(%esp),%cx
34: b8 01 13   mov     $0x1301,%ax
37: bb 0c 00   mov     $0xc,%bx
3a: ba 00 00   mov     $0x0,%dx
3d: cd 10      int     $0x10
3f: 5d         pop     %bp
40: c3         ret
...
1fd: 00 55 aa   add     %dl,-0x56(%di)
```

Exercise11: 请回答为什么三个段描述符要按照 cs, ds, gs 的顺序排列?

答: 通过以下两段代码, 可以看出 cs, ds, gs 的值:

1. # 长跳转切换到保护模式
2. # During the ljmp, index of cs is set to 1
3. data32 ljmp \$0x08, \$start32
- 4.
5. movw \$0x10, %ax # setting data segment selector
6. movw %ax, %ds #index in ds is 2
7. movw %ax, %es
8. movw %ax, %fs
9. movw %ax, %ss
10. movw \$0x18, %ax # setting graphics data segment selector
11. movw %ax, %gs #index in gs is 3
12. movl \$0x8000, %eax # setting esp
13. movl %eax, %esp



首先通过长跳转命令 `ljmp ptr16:ptr:32 cs:eip` 将 `cs` 中的 `index` 部分设置成了 1，再在下面的代码中通过 `mov` 指令将 `ds`, `gs` 的 `index` 部分分别设置成了 2 跟 3，而段选择符中的 `index` 部分正是 GDT 中的下标，因此 GDT 数组中，除开 `GDT[0] = NULL`，剩余元素分别对应 `cs`, `ds`, `gs`。

## Exercise12: 请回答 `app.s` 是怎么利用显存显示 `helloworld` 的。

答：通过分析 `app.s` 文件：

```
1. .code32
2.
3. .global start
4. start:
5.     pushl $13
6.     pushl $message
7.     calll displayStr
8. loop:
9.     jmp loop
10.
11. message:
12.     .string "Hello, World!\n\0"
13.
14. displayStr:
15.     movl 4(%esp), %ebx #message 的首地址存储在%ebx 中
16.     movl 8(%esp), %ecx #message 的长度存储在%ecx 中
17.     movl $((80*5+0)*2), %edi #%edi 中存储了 message 的第一个字符在
    VGA 部分距离 VGA 首地址的偏移量，也是光标的初始偏移量
18.     movb $0x0c, %ah #VGA 中一个字符的显示需要 2 字节，高字节存放 0x0c
    表示输出红色字符
19. nextChar:
20.     movb (%ebx), %al #VGA 的低字节存放对应的 ASCII 码
21.     movw %ax, %gs:(%edi) #将存放在%ax 中的 VGA 字符信息移动到由%gs:
    (%edi)表示的 VGA 内存中
22.     addl $2, %edi #每在 VGA 中输出一个字符，光标偏移量向后移动两个字节
23.     incl %ebx #%ebx 增加，访问下一个 message 字符
24.     loopnz nextChar # loopnz decrease ecx by 1
25.     ret
```

这里分析一下利用 `%gs:(%edi)` 向 VGA 内存输出的逻辑：首先，VGA 内存的基地址存储在 `gs` 指示的 GDT 描述符的 `base` 部分，应当是 `0x8c00` (具体参考 task 1 中在 `start_protected` 分支下修改的 `start.s` 代码)。光标的初始偏移量是  $(80*5+0)*2$ ，其含义为每行有 80 个字符，从第六行行首开始输出第一个字符，同时  $*2$  意味着每个字符占 2 bytes 的内存。以后每次输出一个字符，光标都要相应向后偏移 2 字节的空间，直到输出完成。

**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。

答: cat 是 concatenate 的简写, 这句命令的含义是将 `bootloader/bootloader.bin` `app/app.bin` 这两个文件连接起来依次合并成一个文件, 并将其输出到 `os.img` 中。这样一来, 合并后的文件就可以看作是一个磁盘空间, 现在有两个扇区, 0 扇区是 `bootloader/bootloader.bin`, 而 1 号扇区是 `app/app.bin`, 在 task 2 中利用这一点, 可以调用 `readSect((void*)0x8c00, 1)` 将 `app.bin` 读到 `0x8c00` 的内存中。

**Exercise14:** 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。BIOS 程序占 512 个字节并且从 `0x7c00` 的位置开始。如果在 `0x7c20` 的内存位置在读入一个磁盘扇区的大小, 即 512 字节, 就会覆盖 BIOS 的原有代码, 使得程序无法正确执行。

**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。

答: 电脑开机时, CPU 处于实模式下, 此时内存的计算方式是段基址  $\ll 4$  + 段内偏移。CPU 第一条指令的地址通过 `CS:ip` 来获得, `cs` 初始值为 `0xf000`, `ip` 初始为 `0xffff`, 这是通过硬件设置的。所以最开始执行的指令地址就是 `0xffff0`, 这个内存地址映射在主板的 BIOS ROM (只读存储区) 中。

ROM 中的程序会检测 RAM、键盘、显示器、软硬磁盘是否正确工作。同时会从地址 0 开始设置 BIOS 的中断向量表。ROM 中的程序继续执行, 将启动设备磁盘 0 磁道 0 扇区, 也就是 `mbr`, 一个 512 字节的扇区读到内存 `0x07c00` 处。通过 `ljmp` 指令设置 `cs:ip = 0x7c00:offset`, ROM 中的程序执行结束, 转到 `0x07c00` 处开始执行。MBR 中存有的程序称为 `bootloader`, 将由它设置 GDT 与 LDT 等信息, 并加载操作系统, 转入保护模式运行。即 BIOS 负责从 MBR 中取出 `bootloader`, 加载到 `0x7c00` 处, 再由 `bootloader` 启动操作系统, 进入保护模式。

之后操作系统开始执行。

## 2. 阐述 Challenge:

首先，选择方法一，尝试用 C 语言写了一个与 genboot.pl 同功能的脚本程序，程序逻辑与 genboot.pl 相同，以用 argv 中的文件名打开文件后，进行相关处理即可，具体代码参见 start\_protected 分支下 ./challenge/genboot-c1.c，生成可执行文件后，运行即可。

对于方法二，脚本处理了 objcopy 后的文件。只要利用 ics 课上对 elf 文件的学习，可以通过分析 elf 的程序头表直接将 objcopy 的功能加入脚本中，让脚本可以直接处理 elf 文件。这一方法没有具体实现，不过思路并不复杂。

对于方法三，直接利用 nasm 的伪指令生成 mbr 格式。具体思路与 app.s 相同，主要增加的代码是以下几句：

```
1. message: db "Hello, World!",10,13
2. msglen: equ $ -message
3. times 510 - ($ - $$) db 0          ;fill the .bin file with 0 to 510 B
4. dw 0aa55h                        ;set magic number 0x55aa in little-endian
```

几条伪指令的含义是：

\$-\$\$：经常被用到，表示本行距离程序开始处的相对距离。

times：重复汇编，在 times 后跟着的表达式会被重复汇编指定次数。

db 定义字节类型变量，一个字节数据占 1 个字节单元，读完一个，偏移量加 1

dw 定义字类型变量，一个字数据占 2 个字节单元，读完一个，偏移量加 2

dd 定义双字类型变量，一个双字数据占 4 个字节单元，读完一个，偏移量加 4

通过命令 nasm boot.asm -o boot.bin，可以直接得到符合条件的 bin 文件。具体参见 start\_protected 分支下 ./challenge/mbrasm.asm 文件。