

操作系统 LAB1

如何启动一个 OS

Report

姓名： _____ 肖丹妮 _____

学号： _____ 201220199 _____

邮箱： _____ 3165372798@qq.com _____

目录

1. 15 个 exercise.....	1
2. 2 个 task.....	5
3. 1 个 challenge.....	6

一、Exercises

exercise1:

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

Scrt1.o:      file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET      TYPE      VALUE
0000000000000012 R_X86_64_REX_GOTPCRELX __libc_csu_fini-0x0000000000000004
0000000000000019 R_X86_64_REX_GOTPCRELX __libc_csu_init-0x0000000000000004
0000000000000020 R_X86_64_REX_GOTPCRELX main-0x0000000000000004
0000000000000026 R_X86_64_GOTPCRELX    __libc_start_main-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET      TYPE      VALUE
0000000000000020 R_X86_64_PC32        .text
```

反汇编 Scrt1.o, 如猜想所说, main 在 .text 段中偏移量为 0x20 的位置, 也就是 Scrt1.o 里的 _start 函数会跳转到 main 函数。

exercise2

指令为 `[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b`

- 电脑开机第一条指令的地址是 0xffff0, 位于 BIOS ROM 区域;
- 电脑启动时 CS 寄存器的值是 0xf000, IP 寄存器的值是 0xffff0;
- 第一条指令是 ljmp, 将 CS 寄存器的值设为 0xf000, IP 寄存器的值设为 0xe05b。电脑的 BIOS 的物理地址范围固定设计为 0x000f0000-0x000fffff, 处理器复位的时候将 CS:EIP 的值设为上述第一条指令的地址, 开机之后 CPU 执行第一条指令, 就可以保证 BIOS 获得机器的控制权。

exercise3

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

gdb:
  gdb -n -x ./gdbconf/.gdbinit
```

- make qemu-nox-gdb:
利用 QEMU 模拟 80386 平台, Debug 自制的操作系统镜像 os.img, 选项 -nographic 保证不弹窗, 选项 -s 在 TCP 的 1234 端口运行一个 gdbserver, 选项 -S 使得 QEMU 启动时不运行 80386 的 CPU。
- make gdb:
启动 gdb, 选项 -n 不执行 .gdbinit 中的任何命令, 选项 -x 从 .gdbinit 中读取 gdb 命令。

exercise4

看见了开机从 f000:fff0H 开始执行, 长跳转到 f000:e05b 执行以及后面一系列 BIOS 的执行内容, 如下图所示:

```
The target architecture is set to "i8086".
[[f000:ffff] 0xffff0: jmp $xf000,$xe05b
(gdb) si
[[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x70c8
(gdb) si
[[f000:e062] 0xfe062: jne 0xfd414
(gdb) si
[[f000:e066] 0xfe066: xor %dx,%dx
(gdb) si
[[f000:e068] 0xfe068: mov %dx,%ss
(gdb) si
[[f000:e06a] 0xfe06a: mov $0x7000,%esp
(gdb) si
[[f000:e070] 0xfe070: mov $0xf2d4e,%edx
(gdb) si
[[f000:e076] 0xfe076: jmp 0xffff00
(gdb) si
[[f000:ff00] 0xffff00: cli
(gdb) si
[[f000:ff01] 0xffff01: cld
(gdb) si
[[f000:ff02] 0xffff02: mov %eax,%ecx
(gdb) si
[[f000:ff05] 0xffff05: mov $0x8f,%eax
(gdb) si
[[f000:ff0b] 0xffff0b: out %al,$0x70
(gdb) si
[[f000:ff0d] 0xffff0d: in $0x71,%al
(gdb) si
[[f000:ff0f] 0xffff0f: in $0x92,%al
(gdb) si
[[f000:ff11] 0xffff11: or $0x2,%al
(gdb) si
[[f000:ff13] 0xffff13: out %al,$0x92
(gdb) si
[[f000:ff15] 0xffff15: mov %ecx,%eax
(gdb) si
[[f000:ff18] 0xffff18: lidt %cs:0x70b8
(gdb) si
[[f000:ff1e] 0xffff1e: lgdtw %cs:0x7078
(gdb) si
[[f000:ff1e] 0xffff1e: lgdtw %cs:0x7078
(gdb) si
[[f000:ff24] 0xffff24: mov %cr0,%ecx
(gdb) si
[[f000:ff27] 0xffff27: and $0xffffffff,%ecx
(gdb) si
[[f000:ff2e] 0xffff2e: or $0x1,%ecx
(gdb) si
[[f000:ff32] 0xffff32: mov %ecx,%cr0
(gdb) si
[[f000:ff35] 0xffff35: jmp $0x8,$0xffff3d
(gdb) si
The target architecture is set to "i386".
=> 0xffff3d: mov $0x10,%ecx
(gdb) si
=> 0xffff42: mov %ecx,%ds
(gdb) si
=> 0xffff44: mov %ecx,%es
(gdb) si
```

exercise5

- 硬件识别异常和中断源的方式称为向量中断方式，在这种方式下，异常和中断处理程序的首地址称为中断向量，所有中断向量存放在一个表中，这个表就是中断向量表。每个异常和中断都被设定一个中断类型号，中断向量存放的位置与对应的中断类型号相关，故可以根据类型号快速找到对应的处理程序。
- 定义一个长度为 13 的字符串 message: "Hello, World!\n\0"，将字符串的长度和首地址压栈，然后调用 displayStr 函数，displayStr 函数传递相关参数如重复输出字符的次数等，利用 int \$0x10 指令打印字符串。

exercise6

段的大小与偏移地址的长度有关，在 8086 中，偏移地址用 16 位的二进制数表示，范围为 0x0000~0xffff，共有 2^{16} 种取值，在内存中可占 2^{16} 个单元，即 64KB，故段的大小最大为 64KB。

exercise7

不可以，因为 `mbr.elf` 不满足大小为 512 字节且 512 字节的最后两个字节为 `0x55` 和 `0xaa` 的条件。

exercise8

- 第一条指令：`-m` 是模拟仿真链接器，`elf_i386` 使得输出为 `elf` 的目标格式，`i386` 是目标平台，`-e start` 是指定 `start` 作为程序执行的起始点，`-Ttext 0x7c00` 是指定 `elf` 文件第一个字节的地址为 `0x7c00`，`-o mbr.elf` 指定输出文件的名称为 `mbr.elf`。
- 第二条指令：`-S` 是不将 `mbr.elf` 文件中的重定位信息和符号信息拷贝到输出文件中去，`-j .text` 是只将 `.text` 段拷贝到输出文件，`-O binary` 指定输出文件格式为二进制文件。

exercise9

- `genboot.pl` 在检查文件是否大于 510 字节，如果是，输出“ERROR: boot block too large: \$n bytes (max 510)\n”，如果不是，输出“OK: boot block is \$n bytes (max 510)\n”，并且在文件后面填充 510-n 个字节的 0，末尾还剩 $512-n-(510-n)=2$ 个字节填上 `0x55` 和 `0xaa`。
- 这样做是为了使文件大小正好为 512 字节且文件最后两字节是起标识作用的魔数。

exercise10

使用 `nasm` 工具反汇编 `mbr.bin` 得到 `mbr.asm`

```
oslab@oslab-VirtualBox:~/OS2022$ ndisasm mbr.bin > mbr.asm
```

`mbr.asm` 部分内容如下所示，说明 `mbr.bin` 与 `mbr.elf` 相比，多了 `0x41~0x1FC` 字节的 0 以及最后两字节的 `0x55` 和 `0xaa`，`mbr.bin` 整个文件的大小是 512 字节。

Open ▾	🔍	mbr.asm ~/OS2022
00000000	8CC8	mov ax,cs
00000002	8ED8	mov ds,ax
00000004	8EC0	mov es,ax
00000006	8ED0	mov ss,ax
00000008	B807D0	mov ax,0x7d00
0000000B	89C4	mov sp,ax
0000000D	6A0D	push byte +0xd
0000000F	68177C	push word 0x7c17
00000012	E81200	call 0x27
00000015	EBFE	jmp short 0x15
00000017	48	dec ax
00000018	656C	gs insb
0000001A	6C	insb
0000001B	6F	outsw
0000001C	2C20	sub al,0x20
0000001E	57	push di
0000001F	6F	outsw
00000020	726C	jc 0x8e
00000022	64210A	and [fs:bp+si],cx
00000025	0000	add [bx+si],al
00000027	55	push bp
00000028	67B442404	mov ax,[dword esp+0x4]
0000002D	89C5	mov bp,ax
0000002F	67B4C2406	mov cx,[dword esp+0x6]
00000034	B80113	mov ax,0x1301
00000037	B80C00	mov bx,0xc
0000003A	BA0000	mov dx,0x0
0000003D	CD10	int 0x10
0000003F	5D	pop bp
00000040	C3	ret
00000041	0000	add [bx+si],al
00000043	0000	add [bx+si],al
00000045	0000	add [bx+si],al
00000047	0000	add [bx+si],al
00000049	0000	add [bx+si],al
0000004B	0000	add [bx+si],al
0000004D	0000	add [bx+si],al
000001B5	0000	add [bx+si],al
000001B7	0000	add [bx+si],al
000001B9	0000	add [bx+si],al
000001BB	0000	add [bx+si],al
000001BD	0000	add [bx+si],al
000001BF	0000	add [bx+si],al
000001C1	0000	add [bx+si],al
000001C3	0000	add [bx+si],al
000001C5	0000	add [bx+si],al
000001C7	0000	add [bx+si],al
000001C9	0000	add [bx+si],al
000001CB	0000	add [bx+si],al
000001CD	0000	add [bx+si],al
000001CF	0000	add [bx+si],al
000001D1	0000	add [bx+si],al
000001D3	0000	add [bx+si],al
000001D5	0000	add [bx+si],al
000001D7	0000	add [bx+si],al
000001D9	0000	add [bx+si],al
000001DB	0000	add [bx+si],al
000001DD	0000	add [bx+si],al
000001DF	0000	add [bx+si],al
000001E1	0000	add [bx+si],al
000001E3	0000	add [bx+si],al
000001E5	0000	add [bx+si],al
000001E7	0000	add [bx+si],al
000001E9	0000	add [bx+si],al
000001EB	0000	add [bx+si],al
000001ED	0000	add [bx+si],al
000001EF	0000	add [bx+si],al
000001F1	0000	add [bx+si],al
000001F3	0000	add [bx+si],al
000001F5	0000	add [bx+si],al
000001F7	0000	add [bx+si],al
000001F9	0000	add [bx+si],al
000001FB	0000	add [bx+si],al
000001FD	0055AA	add [di-0x56],di

exercise11

下面是 start.s 的指令

```
# 长跳转切换到保护模式
data32 ljmp $0x08, $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
movw %ax, %gs
movl $0x8000, %eax # setting esp
movl %eax, %esp
```

由 `ljmp` 的第一个操作数可知，代码段选择子是 `0x08`；
`0x08 >>> 3 = 1` 故 GDT 第二个描述符对应 `cs`；
由 `movw $0x10, %ax` 以及 `movw %ax, %ds` 可知，数据段选择子是 `0x10`；
`0x10 >>> 3 = 2` 故 GDT 第三个描述符对应 `ds`；
由 `movw $0x18, %ax` 以及 `movw %ax, %gs` 可知，图像段选择子是 `0x18`。
`0x18 >>> 3 = 3` 故 GDT 第四个描述符对应 `gs`；
所以三个段描述符按照 `cs`，`ds`，`gs` 的顺序排列。

exercise12

定义一个内容为 "Hello, World!\n\0" 的字符串 `message`，将长度 13 和地址压栈，传递给 `displayStr` 函数，`displayStr` 获得参数，将图像段的有效地址存进 `edi`，将 `ah` 设为 `0x0c` 即字符的属性为 `0x0c`，然后通过循环重复操作：将当前字符的 ASCII 编码存进 `al`，利用 `movw %ax, %gs:(%edi)` 指令将字符写到对应的内存地址，从而实现显示 `helloworld`。

exercise13

将 `bootloader` 里的 `bootloader.bin` 和 `app` 里的 `app.bin` 合并生成 `os.img`。

exercise14

不可以。因为 `bootloader` 开始的地址是 `0x7c00`，且 `bootloader` 占 512 字节，故 `0x7c00~0x7dff` 都应该是 `bootloader` 的内容，若将 `app` 读到 `0x7c20`，将会破坏 `bootloader` 的内容。

exercise15

- 1) CPU 执行地址为 `0x0ffff0` 第一条指令 `ljmp`，长跳转到 BIOS；
- 2) BIOS 进行开机自检，完成 POST 过程和自举过程，然后将主引导扇区 MBR 的 512 字节内容加载到地址为 `0x7c00` 的地方，并检查最后两个字节是否为魔数 `0x55` 和 `0xaa`；
- 3) CPU 跳转到 `CS:IP=0x0000:0x7c00` 执行加载程序 `bootloader`。`bootloader` 开启保

护模式，将 OS 的代码和数据从磁盘加载到内存，然后跳转到 OS 的起始地址，将控制权交给 OS。

二、Tasks

task1

- 1) 把 cr0 的低位设置为 1:
利用 eax 寄存器，先将 cr0 存入 eax，然后让 eax 的内容和 1 相或，即可使 eax 最低位置是 1，再将 eax 存回 cr0。

```
# TODO: 把cr0的最低位设置为1
movl %cr0, %eax
orl $0x1, %eax
movl %eax, %cr0
```

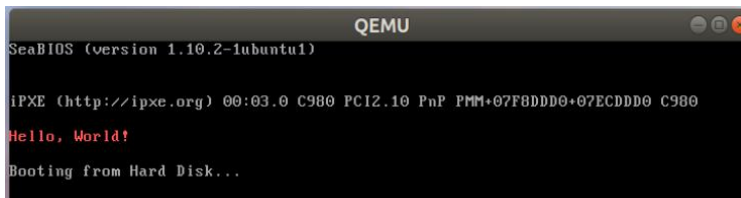
- 2) 填写 GDT:
Limit: 代码段、数据段、图像段的 limit 都是 0xfffff;
Base: 代码段和数据段的 base 都是 0, 由实验手册知图像段的 base 为 0xb800;
Type: 数据段和图像段的 type 都是 0x2, 代码段的 type 是 0x0a.
- 3) 显示 helloworld:
参照 app.s 里的方式。

```
# TODO: 编写输出函数, 输出"Hello World" (Hint:参考app.s!!!)
pushl $13
pushl $message
calll displayStr
loop:
    jmp loop

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

displayStr:
    movl 4(%esp), %ebx
    movl 8(%esp), %ecx
    movl $((80*5+0)*2), %edi
    movb $0x0c, %ah
nextChar:
    movb (%ebx), %al
    movw %ax, %gs:(%edi)
    addl $2, %edi
    incl %ebx
    loopnz nextChar # loopnz decrease ecx by 1
    ret
```

make clean 后再执行 make, 得到 os.img, 然后 make qemu 就可以显示字符串。



task2

- 1) 将 cr0 的低位设为 1 以及填写 GDT 的思路与 task1 一样, 利用 jmp 指令跳转到

bootMain 执行。

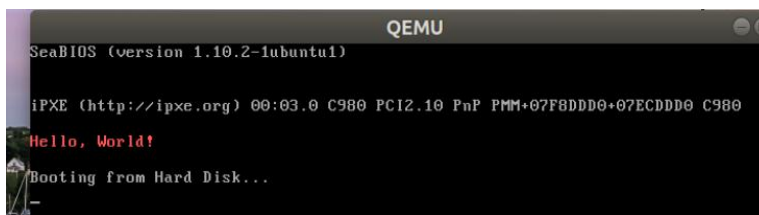
```
# TODO: 跳转到 jmp  
jmp bootMain
```

2) 填写 bootMain 函数:

首先调用 readSect 函数将 app 的从磁盘读出，读到物理地址为 0x8c00 的地方；再用内联汇编跳转到 0x8c00 执行。如下图

```
void bootMain(void) {  
    readSect((void *)0x8c00, 1);  
    __asm__ __volatile__("jmp 0x8c00");  
}
```

最后 make qemu 就可以看到下图所示的内容



三、Challenge

做法：第一种方法，用 C 语言代替 genboot.pl 文件，生成符合符合 mbr 格式的 mbr.bin。

如何实现：用 fopen() 打开 mbr.bin 文件，fread() 获得文件的字节数 n，if-else 语句判断字节数是否大于 512，如果大于则退出，否则用 fwrite() 在文件末尾追加 510-n 个 0 以及最后两字节的 0x55 和 0xAA。

最终结果为

