

OSLAB1

191830064 姜纪文

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)

我们可以看到入口_start 函数, 并且调用了与 main 相关的函数

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

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?

电脑开机第一条指令的地址是 0xfffff0, 这位于 bios 代码的最尾部?

2. 电脑启动时 CS 寄存器和 IP 寄存器的值是什么?

CS:F000

IP:FFF0

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)

第一条指令是 `ljmp $0xf000, $0xe05b`。因为 bios 的物理内存地址范围按规定为 0x000f0000-0x000fffff。又因为 `cpu reset` 后按规定 `cs, ip` 寄存器有相应的状态即 `CS:F000; IP:FFF0`, 正好在 bios 地址范围的最后面, 于是设计 `ljmp` 指令改变 IP 寄存器使得 `CS: IP` 为 bios 开始工作的代码地址。

```
oslab@oslab-VirtualBox:~/桌面/lab1$ make gdb
gdb -n -x ./gdbconf/.gdbinit
GNU gdb (GDB) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i386 settings.

--Type <RET> for more, q to quit, c to continue without paging--
The target architecture is set to "i386".
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 "i386".
[f000:fff0] 0xfffff0: jmp 0xf000,0xe05b
0x0000fff0 in ?? ()
(gdb) si
[f000:e05b] 0xfe05b: cmpl 0x0,%cs:0x70c8
0x0000e05b in ?? ()
(gdb) 
```

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）

```
qemu:
    qemu-system-i386 os.img

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

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

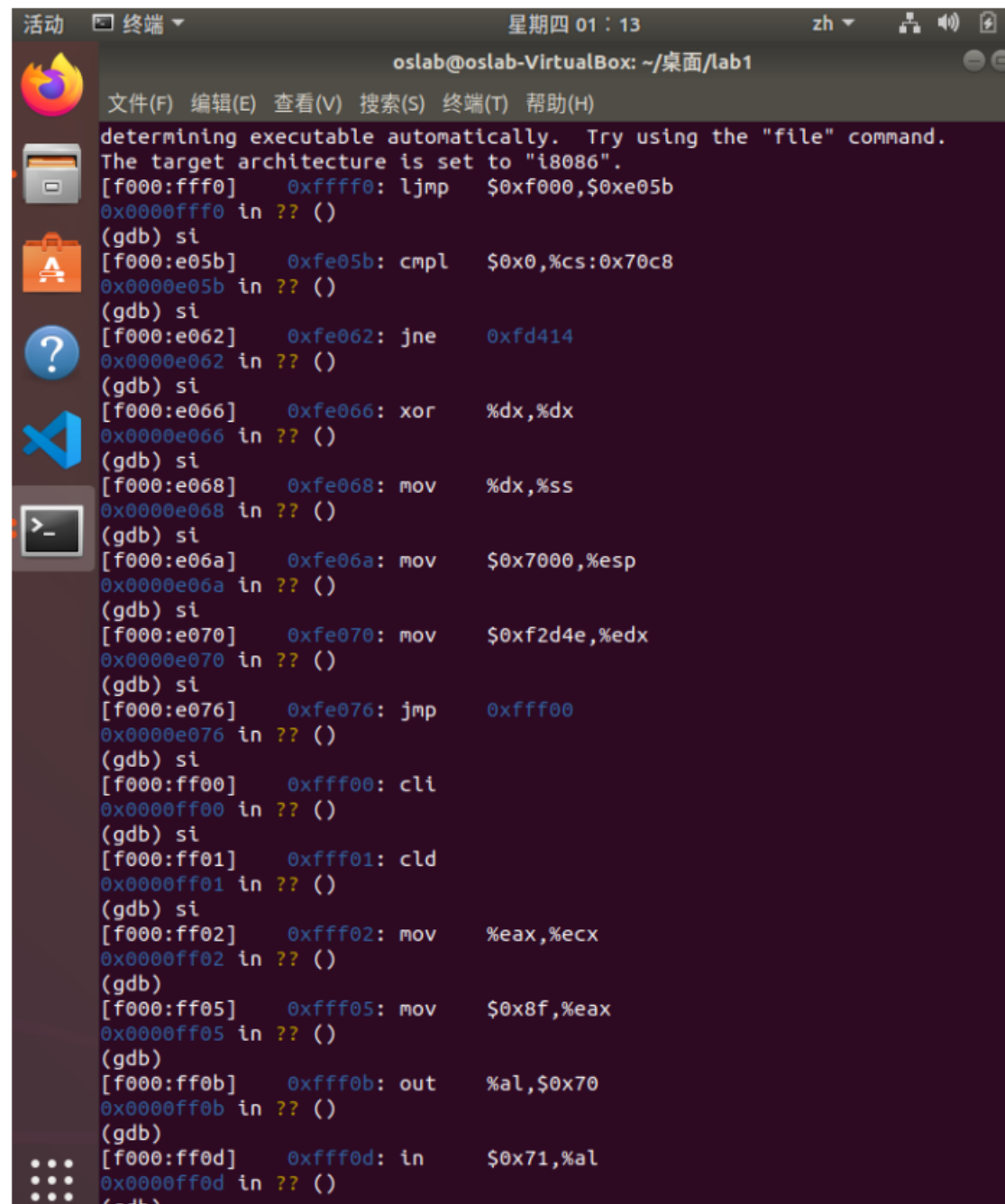
gdb:
    gdb -n -x ./gdbconf/.gdbinit
```

如上图所示，make qemu-nox-gdb 为 qemu 选定为 i386 硬件平台，-nographic 选项为不显示图形界面，-s 选项为 qemu 机器开启 1234 号端口以 tcp 协议与 gdb 通信，即 gdb 调试 qemu；-S 选项是指 qemu 虚拟机停留在第一条指令上。os.img 指加载的操作系统。

而 make gdb 中 -n 选项指明不执行任何在初始化文件中能找到的命令，-x 选项指明后面文件才是为 gdb 的初始化文件，然后执行该文件的 gdb's script。

exercise4: 继续用 si 看见了什么？请截一个图，放到实验报告里。

继续 si 可以看到进行了一些中断设置 (cli)，与 IO 端口的通信等等 (in&out)



```
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) si
[f000:e070] 0xfe070: mov $0xf2d4e,%edx
0x0000e070 in ?? ()
(gdb) si
[f000:e076] 0xfe076: jmp 0xffff00
0x0000e076 in ?? ()
(gdb) si
[f000:ff00] 0xff00: cli
0x0000ff00 in ?? ()
(gdb) si
[f000:ff01] 0xff01: cld
0x0000ff01 in ?? ()
(gdb) si
[f000:ff02] 0xff02: mov %eax,%ecx
0x0000ff02 in ?? ()
(gdb) si
[f000:ff05] 0xff05: mov $0x8f,%eax
0x0000ff05 in ?? ()
(gdb) si
[f000:ff0b] 0xff0b: out %al,$0x70
0x0000ff0b in ?? ()
(gdb) si
[f000:ff0d] 0xff0d: in $0x71,%al
0x0000ff0d in ?? ()
(gdb)
```

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 (IVT)，在操作系统内核代码中实现，当中断到来时，会将机器的控制权交给

OS 的中断处理程序来解决。IVT 是一个结构数组，包含相应中断程序的入口地址。硬件检测到中断后，回去利用相应寄存器提前保存的信息（索引）去寻找 IVT 相应结构元素，再利用结构体中的保存的中断程序的入口地址来处理问题。

MBR 是通过 bios 上的关于屏幕显示器的系统调用来对 helloworld 的输出的，提前已经设置好了参数，例如字符串的地址，字符串的长度，都保存在栈上；字体显示的参数在寄存器中保存。

```
1 .code16
2 .global start
3 start:
4 movw %cs, %ax
5 movw %ax, %ds
6 movw %ax, %es
7 movw %ax, %ss
8 movw $0x7d00, %ax
9 movw %ax, %sp      # setting stack pointer to 0x7d00
10 pushw $13          # pushing the size to print into stack
11 pushw $message     # pushing the address of message into stack
12 callw displayStr   # calling the display function
13 loop:
14 jmp loop
15 message:
16 .string "Hello, World!\n\0"
17 displayStr:
18 pushw %bp
19 movw 4(%esp), %ax
20 movw %ax, %bp
21 movw 6(%esp), %cx
22 movw $0x1301, %ax
23 movw $0x000c, %bx
24 movw $0x0000, %dx
25 int $0x10
26 popw %bp
27 ret
```

exercise6: 为什么段的大小最大为 64KB，请在报告上说明原因。

因为在实地址寻址下，即 cs:ip，二者都是 16 位寄存器。如果看作 cs 选择段，ip 为 offset，那么一个段最大为 ip 的寻址数量大小即 $2^{16} \text{ bits} = 64 \text{ KB}$

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？

不能，因为 mbr.elf 为 elf 文件，内部包含了许多与程序运行无关的代码（调试、定位），

即有许多文本的编码，无法直接以指令运行，需要加载。

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

ld 链接器，输入文件为 mbr.o，输出文件为 mbr.elf。选项 -m 指明硬件平台为 i386，-e 选项指明入口函数，-Ttext 选项指明代码段从 0x7c00 开始

objcopy，-j 选项指明指拷贝输入文件的 .text 段，-S 选项指明不拷贝重定位相关信息，并且删除调试相关节，-O binary 选项说明以二进制来构建输出文件，mbr.elf 为处理前的源文件，mbr.bin 为处理后的输出文件

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

```
#!/usr/bin/perl
open(SIG, $ARGV[0]) || die "open $ARGV[0]: $!";
$n = sysread(SIG, $buf, 1000);
if($n > 510){
    print STDERR "ERROR: boot block too large: $n bytes (max 510)\n";
    exit 1;
}
print STDERR "OK: boot block is $n bytes (max 510)\n";
$buf .= "\0" x (510-$n);
$buf .= "\x55\xAA";
open(SIG, ">$ARGV[0]") || die "open >$ARGV[0]: $!";
print SIG $buf;
```

文件大于 510 字节后输出错误提示信息，并且退出程序。因为 mbr 一般认为是第一个扇区，512 个字节。文件不大于 512 字节情况下则将剩下的位置以 \0 填充，512 字节最后两个字节填充为 \x55\xAA。因为要保持文件的大小为 512 字节并且要有标识的字节，即最后两个字节来提示已经找到正确的扇区。

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

```

oslab@oslab-VirtualBox:~/桌面/lab1/os2022$ objdump -m i386 -b binary -D mbr.bin
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(%sl),%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     $0xcbb1301,%eax
39:  00 ba 00 00 cd 10    add     %bh,0x10cd0000(%edx)
3f:  5d                   pop     %ebp
40:  c3                   ret
...
1fd:  00 55 aa             add     %dl,-0x56(%ebp)
oslab@oslab-VirtualBox:~/桌面/lab1/os2022$

```

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

因为我们通过段选择子去寻找段描述符时是按索引去找的。从下图可以知道, gs 段选择子为 0x18, ds 段选择子为 0x10。根据段选择子的结构可知, gs 段描述符是数组索引为 3 的元素, ds 段描述符在 gdt 数组索引为 2 处。

```

.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
    movw %ax, %gs
    movl $0x8000, %eax # setting esp
    movl %eax, %esp

```

exercisel2: 请回答 app. s 是怎么利用显存显示 helloworld 的。

app.s 对寄存器 ah 传入 0x0c, 即设置字体背景颜色为黑底红字, 然后将字体的 ascii 码 (1 字节) 传入 al 寄存器, 然后将 ax 寄存器 (ah:al) mov 到相应显存位置来显示 helloworld。

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。

利用 cat 程序将 bootloader.bin 和 app.bin 整合输出到 os.img 文件中, 从而使得 qemu 能够执行二进制文件 os.img。

exercise14: 如果把 app 读到 0x7c20, 再跳转到这个地方可以吗? 为什么?

bootloader 程序从地址 0x7c00 处开始, 如果将 app 读到 0x7c20, 会修改 bootloader 程序导致运行不正常。

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。

电脑加电后 cpu reset 会去执行 firmware 映射到内存的程序, 一般为 bios 程序。bios 最开始将进行中断的屏蔽, A20 地址线设置, 栈寄存器的初始化, gdt 的设置等工作; 接着是 bios 的 post 过程, 对硬件进行扫描与设置以及中断服务的设置等等; 接着 bios 自举过程将 mbr 读到内存 0x7c00 处并将控制交给 mbr 中的 bootloader; 紧接着 bootloader 将操作系统加载进内存再将控制权交给操作系统。

Task1:

Cr0 最后一位置 0。

先将 cr0 赋值到其他寄存器, 然后对其进行 or 0x1 操作, 再赋值回 cr0 寄存器。

填充 gdt。

代码段, 数据段的起始地址为 0, 界限为 4G。只不过二者的可读可写的权限有所不同。

gs 段描述符则起始地址为 0xb800。其他位可参考具体 descriptor table 结构

显示 helloworld。

参考 app.s 代码, 在 statr.s 相应位置填入 app.s 代码即可。

Task2:

完善 bootMain。

为函数指针赋值 0x8c00。调用 readSec 函数即可。gcc 内联汇编, 向寄存器传入 0x8c00, 再利用 jmp 指令跳转该地址执行。

修改 start.s

Task1 中的显示 helloworld 代码去掉，用 `jmp bootMain` 指令替换，其他关于保护模式的代码留下，即可调用 boot.s 的 bootMain 函数并且 load app.s 从而显示 hello world。

Challenge:mbr.s

想法是利用 nasm 支持填充预留相应的字节数从而达到使 mbr.bin 文件大小为 512 字节。核心代码与手册一致，只不过改成 nasm 的语法，section 从 7c00 处开始，在代码最后面计算相应的偏移量来填充 0 以及末尾的魔数以构造完整的 mbr。