

Operating System: lab01

任课教师：叶保留 助教：尹熙喆，水兵

学院	计算机科学与技术系	专业	计算机科学与技术
学号	201220154	姓名	王紫萁
Email	201220154@smail.nju.edu.cn	开始/完成时间	2022. 3.5/2022.3.15

1. exercises

• exercise 1

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)

为了验证 exercise1 中链接无法通过的猜想, 我们反汇编 Scrt1.o 文件, 发现其重定位表项中有 main 函数, 进一步证实了在连接时 gcc 将该文件也链接进来并从 _start 转到 main 函数执行

```
1 Scrt1.o:      文件格式 elf64-x86-64
2
3 RELOCATION RECORDS FOR [.text]:
4 OFFSET          TYPE          VALUE
5 0000000000000012 R_X86_64_REX_GOTPCRELX __libc_csu_fini-0x0000000000000004
6 0000000000000019 R_X86_64_REX_GOTPCRELX __libc_csu_init-0x0000000000000004
7 0000000000000020 R_X86_64_REX_GOTPCRELX main-0x0000000000000004
8 0000000000000026 R_X86_64_GOTPCRELX __libc_start_main-0x0000000000000004
9
10 RELOCATION RECORDS FOR [.eh_frame]:
11 OFFSET          TYPE          VALUE
12 0000000000000020 R_X86_64_PC32      .text
13
```

• exercise2

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)

```

oslab@oslab-VirtualBox:~/oslab/lab1$
oslab@oslab-VirtualBox:~/oslab/lab1$ make qemu-nox-gdb
qemu-system-i386 -nographic -s -S os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write
        operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.

of GDB. Attempting to continue with the default i8086 settings.

The target architecture is set to "i8086".
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] 0xfffff0: ljmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
(gdb) █

```

1. 根据终端信息我们可以看到，在开机后的第一条指令的地址是 `0xfffff0`，在内存地址中处于 BIOS ROM 区，里面装着 ROM 中的用于开机的第一条代码。

```

(gdb) info registers
eax            0x0            0
ecx            0x0            0
edx            0x663          1635
ebx            0x0            0
esp            0x0            0x0
ebp            0x0            0x0
esi            0x0            0
edi            0x0            0
eip            0xfffff0       0xfffff0
eflags         0x2            [ ]
cs             0xf000         61440
ss             0x0            0
ds             0x0            0
es             0x0            0
fs             0x0            0
gs             0x0            0
(gdb)

```

2. 查看 `cs` 和 `eip` 的值我们可以猜出下一步将要执行的指令会转到 `0xf000: 0xfffff0` 的地址执行。
3. `ljmp` 指令，用于装载段寄存器并跳转执行指令，在实模式下，
 $paddr = segbase \times 16 + offset$ ，于是得到的物理地址 $paddr = 0xfffff0$ ，验证了我们在 2 中的猜想

```

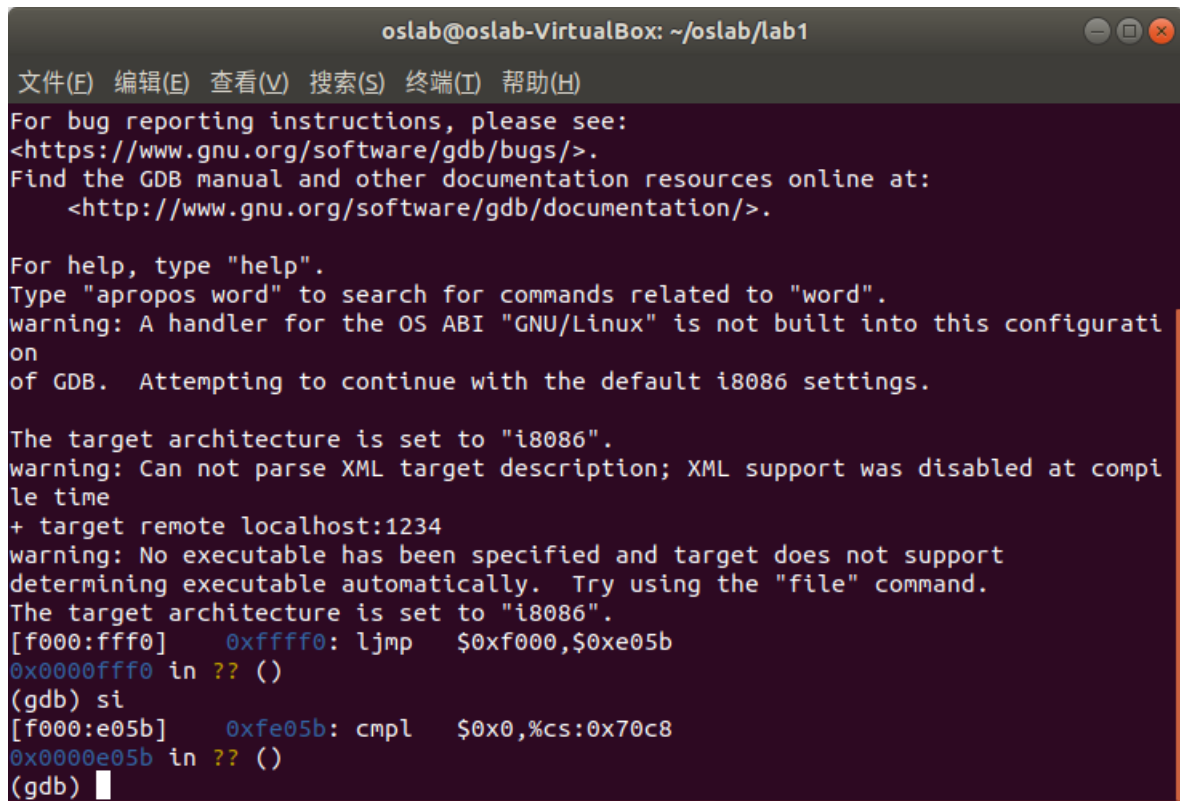
1 |          +-----+
2 |          |          |
3 |          +-----+
4 |          |   .....   |
5 |          +-----+ <---0x00100000
6 |          |  BIOS ROM  |
7 |          +-----+ <---0x000fffff
8 |          |   .....   |
9 |          +-----+
10|

```

• exercise3

- 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-gdb` 就是执行 `qemu-system-i386 -s -S os.img`, 在开机不启用CPU且开启 `gdb` 端口为 1234 的情况下运行 `qemu`。
- `make qemu-nox-gdb` 就是在上一条指令基础上加了 `-nographic` 参数使得运行 `qemu` 不会唤出图形界面

• exercise4



```
oslab@oslab-VirtualBox: ~/oslab/lab1
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
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.

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]    0xffff0: jmp     $0xf000,$0xe05b
0x0000fff0 in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x70c8
0x0000e05b in ?? ()
(gdb) █
```

(该阶段发生事待补充)

• exercise5

- 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](#)是一组保存中断服务程序(interrupt handler)入口地址的一种数据结构。中断请求号和服务例程的入口地址一一对应, 其本质上是一种 `dispatch table`。他有三种最常见的方式找到中断服务例程的入口地址:

- **Predefined**: 设置 PC, 将程序转移到目的地址执行
- **Fetch**: 与前者最大的不同是该方法用某一个中断向量中给出的地址拉出另一个中断向量中的地址
- **Interrupt acknowledge**: 外部设备通过中断请求号查表。

与之相似的还有[中断描述符表](#), 常见的调用访问也有三种: `hardware interrupts`, `software interrupts`, `processor exceptions`。

示例 MBR 通过的第一步与[exercise12](#)中的第一步异曲同工。但 `displayStr` 的实现开始就不太一样了。首先，该函数先在 `ax` , `bx` , `cx` , `dx` 寄存器中准备参数，`int $10` 进入[BIOS中断调用](#)，此类服务包括设置显示模式，字符和字符串输出，和基本图形（在图形模式下的读取和写入[像素](#)）功能。于是就能在BIOS屏幕上看到打印的字符串了

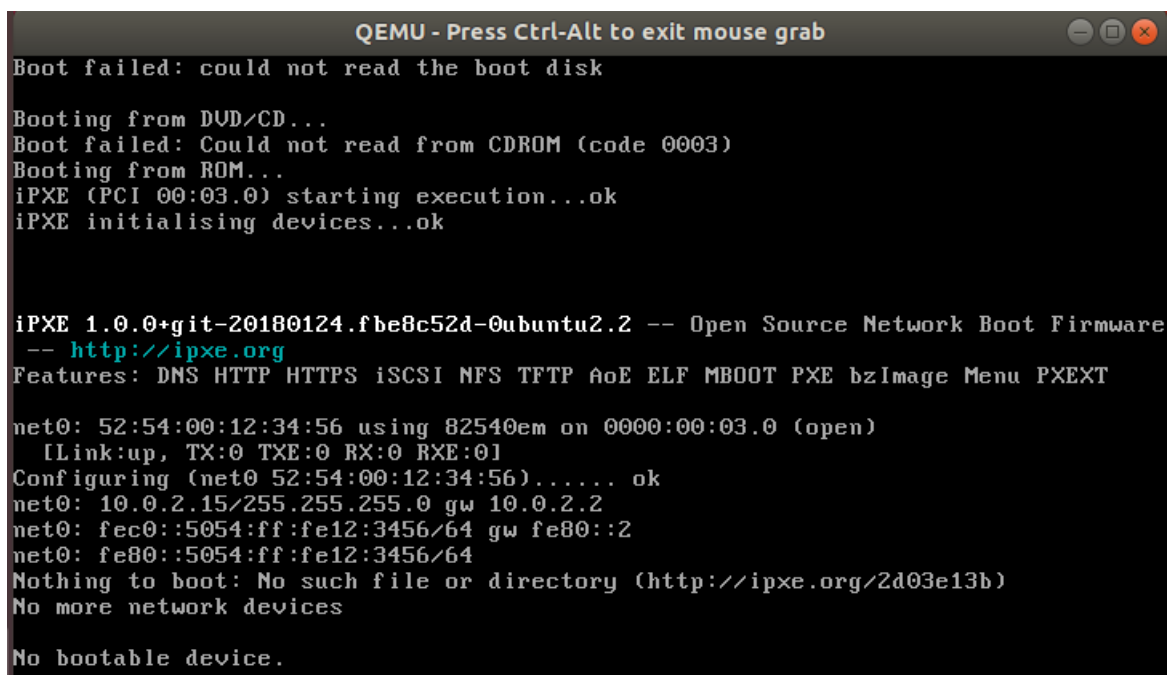
• exercise6

• exercise6：为什么段的大小最大为64KB，请在报告上说明原因。

因为8086地址线，寄存器都只有16位，最大只支持 2^{16} B大小的寻址空间

• exercise7

• 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？



```
QEMU - Press Ctrl-Alt to exit mouse grab
Boot failed: could not read the boot disk

Booting from DVD/CD...
Boot failed: Could not read from CDROM (code 0003)
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok

iPXE 1.0.0+git-20180124.fbe8c52d-0ubuntu2.2 -- Open Source Network Boot Firmware
-- http://ipxe.org
Features: DNS HTTP HTTPS iSCSI NFS TFTP AoE ELF MBOOT PXE bzImage Menu PXEXT

net0: 52:54:00:12:34:56 using 82540em on 0000:00:03.0 (open)
[Link:up, TX:0 TXE:0 RX:0 RXE:0]
Configuring (net0 52:54:00:12:34:56)..... ok
net0: 10.0.2.15/255.255.255.0 gw 10.0.2.2
net0: fec0::5054:ff:fe12:3456/64 gw fe80::2
net0: fe80::5054:ff:fe12:3456/64
Nothing to boot: No such file or directory (http://ipxe.org/2d03e13b)
No more network devices

No bootable device.
```

并不能，我们尝试一下这条命令，发现没有可供boot的设备，原因也很简单，因为BIOS需要魔数 `0x55 0xaa` 确定这一段代码为 `bootloader`，而我们的ELF文件末尾并没有设置魔数，自然也就不会成功运行这段代码。

• exercise8

```
1 $ld -m elf_i386 -e start -Ttext 0x7c00 mbr.o -o mbr.elf
2 $objcopy -S -j .text -O binary mbr.elf mbr.bin
```

• 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` 指令：分为四段，用`elf_i386`链接 `mbr.o` 到 `mbr.elf` 。
 - `-m elf_i386`：指定链接模拟器
 - `-e start`：指定程序入口函数为 `start`
 - `-Ttext 0x7c00`：设置代码段的起始地址即程序执行的第一条指令的地址为 `0x7c00` 并存放在`mbr.o`文件中
 - `-o mbr.elf`：将链接完成的程序输出到 `mbr.elf` 。

- `objcopy` 指令：分为两段，将 `mbr.elf` 的 `.text` 节以二进制形式复制到 `mbr.bin`
 - `-S -j .text`：不复制 `.relocate` 节和 `.symbol` 节，并指定唯一复制的节为 `.text` 节
 - `-O binary mbr.elf mbr.bin`：以二进制形式将 `.elf` 复制到 `.bin` 文件

• exercise9

- 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。

```
1 $buf .= "\0" x (510-$n);
2 $buf .= "\x55\xAA";
```

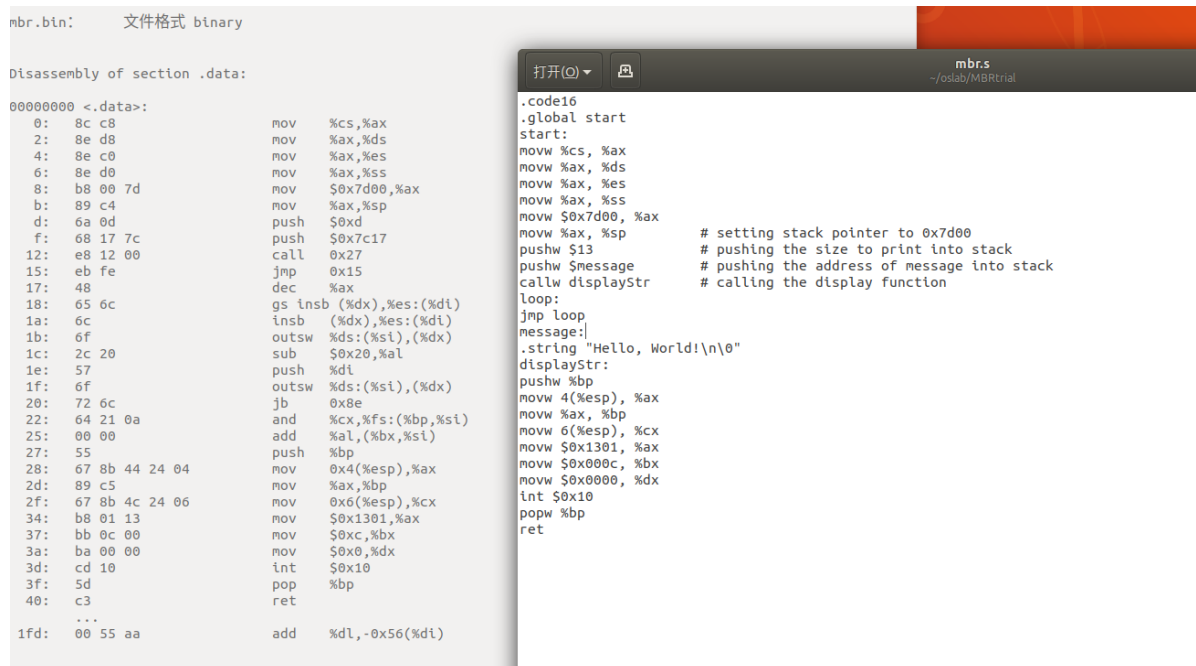
上面是检查合格后的关键两步，会将不满510字节的部分置为0，并在末尾加上 `0x55 0xaa` 作为 boot loader 的魔数。目的是告诉 BIOS 磁盘上的这一段程序是 `bootloader`

• exercise10

- 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

键入指令：

```
1 $ objdump -D -b binary -m i8086 mbr.bin > mbr.dis
```



发现了以下几点：

- 对应的前几条指令指令对应着 `mbr.s` 中的前几条指令
- 惊奇的发现，从 `0x17` 行开始到 `0x25` 字节就是“Hello, World!\n\0”的ASCII码对应的值
- 从 `0x27` 字节开始的指令又和 `.s` 文件对应起来了
- 在510字节处有 `55 aa` 的魔数之前的省略号就是 `genboot.pl` 补充的后缀0

• exercise11

exercise11: 请回答为什么三个段描述符要按照cs, ds, gs的顺序排列?

```
1  data32 jmp $0x08, $start32
2  .code32
3  start32:
4      movw $0x10, %ax # setting data segment selector
5      movw %ax, %ds
6      movw %ax, %es
7      movw %ax, %fs
8      movw %ax, %ss
9      movw $0x18, %ax # setting graphics data segment selector
10     movw %ax, %gs
```

在上面一段代码中我们发现

cs 段选择子为 0x08, ds, es, fs, ss 段选择子为 0x10, gs 的段选择子为 0x18, 那么段寄存器中 idx 域的大小顺序为 cs < ds < gs。因此在段描述符表的先后顺序也应该遵循这个顺序才能索引到正确的表项。

• exercise12

exercise12: 请回答app.s是怎么利用显存显示helloworld的。

```
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
16     movl 8(%esp), %ecx
17     movl $((80*5+0)*2), %edi # where to print
18     movb $0x0c, %ah # additional font configuration
19     #####
20 nextChar:
21     movb (%ebx), %al
22     movw %ax, %gs:(%edi)
23     addl $2, %edi
24     incl %ebx
25     loopnz nextChar # loopnz decrease ecx by 1
26     ret
27
```

app.s 的执行分为三个部分（已在代码中划分）：第一部分准备 displayStr 阶段的参数将字符串长度和首地址压栈；第二部分确定在屏幕上开始输出的位置以及字体的颜色信息，最后一部分扫描字符串将2字节（额外信息：ASCII码）的字符放到 gs : offset 确定的内存上去，屏幕左上角(0,0)点位置为 0xb8000 因此 gs 的 base 段就是该值，详细信息参考手册[显存控制](#)一节。

• exercise13

exercise13: 请阅读项目里的3个Makefile，解释一下根目录的Makefile文件里

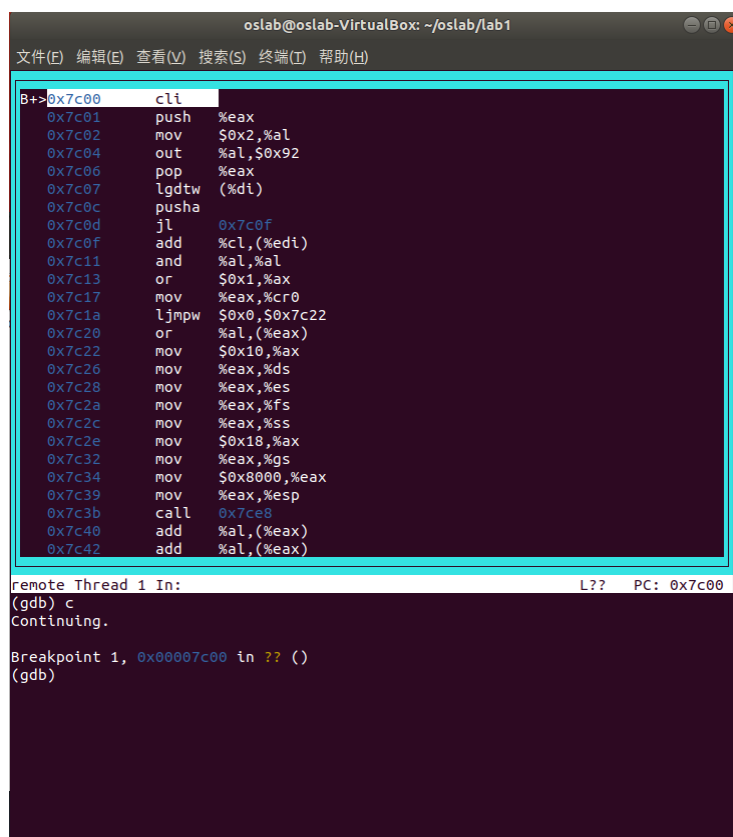
```
1 | cat bootloader/bootloader.bin app/app.bin > os.img
```

这行命令是什么意思。

这一行命令是我们完成 task2 的关键，表示将 app.bin 文件直接拼接在 bootloader.bin 文件的尾部构成os.img文件。因此 boot_main 函数就是将1号扇区（app.bin）的内容搬到主存的 0x8c00 地址上去。

• exercise14

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



```
oslab@oslab-VirtualBox: ~/oslab/lab1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

B+> 0x7c00 cli
0x7c01 push %eax
0x7c02 mov $0x2,%al
0x7c03 out %al,$0x92
0x7c04 pop %eax
0x7c05 lgdtw (%di)
0x7c06 pusha
0x7c07 jl 0x7c0f
0x7c08 add %cl,(%edi)
0x7c09 and %al,%al
0x7c0a or $0x1,%ax
0x7c0b mov %eax,%cr0
0x7c0c ljmpw $0x0,$0x7c22
0x7c0d or %al,(%eax)
0x7c0e mov $0x10,%ax
0x7c0f mov %eax,%ds
0x7c10 mov %eax,%es
0x7c11 mov %eax,%fs
0x7c12 mov %eax,%ss
0x7c13 mov $0x18,%ax
0x7c14 mov %eax,%gs
0x7c15 mov $0xb800,%eax
0x7c16 mov %eax,%esp
0x7c17 call 0x7ce8
0x7c18 add %al,(%eax)
0x7c19 add %al,(%eax)
0x7c1a
0x7c1b
0x7c1c
0x7c1d
0x7c1e
0x7c1f
0x7c20
0x7c21
0x7c22
0x7c23
0x7c24
0x7c25
0x7c26
0x7c27
0x7c28
0x7c29
0x7c2a
0x7c2b
0x7c2c
0x7c2d
0x7c2e
0x7c2f
0x7c30
0x7c31
0x7c32
0x7c33
0x7c34
0x7c35
0x7c36
0x7c37
0x7c38
0x7c39
0x7c3a
0x7c3b
0x7c3c
0x7c3d
0x7c3e
0x7c3f
0x7c40
0x7c41
0x7c42

remote Thread 1 In: L?? PC: 0x7c00
(gdb) c
Continuing.

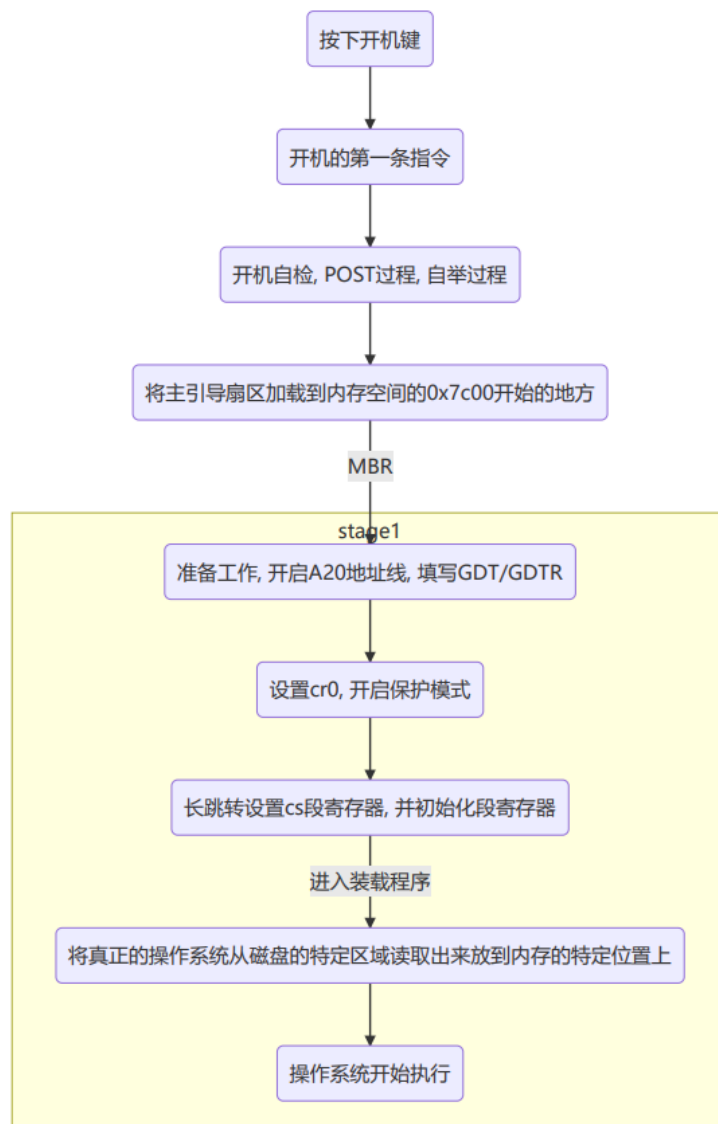
Breakpoint 1, 0x00007c00 in ?? ()
(gdb)
```

不可以，从上图我们可以看到 0x7c20 是 bootloader 的执行代码，并不空闲，加载到该地址会有将代码，段描述符表甚至魔数都覆盖掉的风险，并且app.bin的长度已经足够覆盖到 GDT 再跳转到这里执行会发生段错误且可能会因为 bootloader 错误导致下一次系统启动失败。

• exercise15

- 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。不用太详细, 把每一部分是做什么的说清楚就好了。

为了使逻辑流更清晰, 我们用一张流程图如简要概括一下



2. challenge

• challenge1

- challenge1: 请尝试使用其他方式, 构建自己的MBR, 输出“Hello, world!”

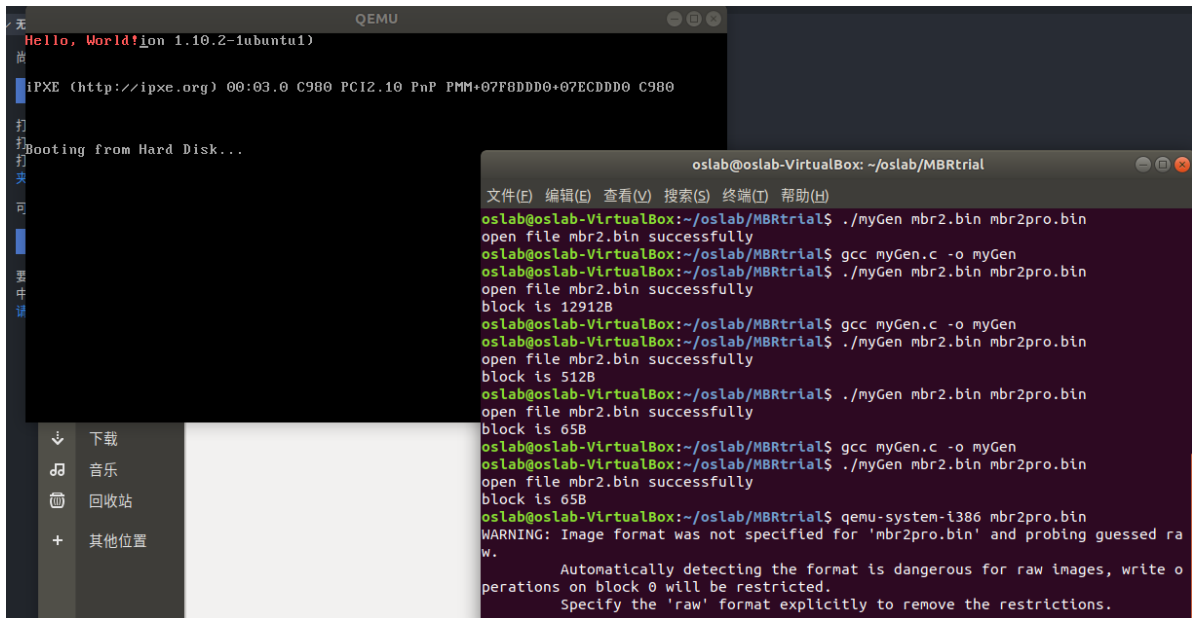
我们先尝试一下最简单的方法 (等变强了再试试解析ELF☺), 编写一个.c文件替代 `genboot.pl`, 思路就是当生成的65B .bin 文件读入到512B缓冲区, 直接在缓冲区尾部加上魔数即可。该小程序接受至少一个参数, 一个是转换源文件 (如“`mbr.bin`”), 另一个可选参数是转换好后将要写入的目标文件名称 (若该参数缺省则直接覆盖源文件写入)。代码如下:


```

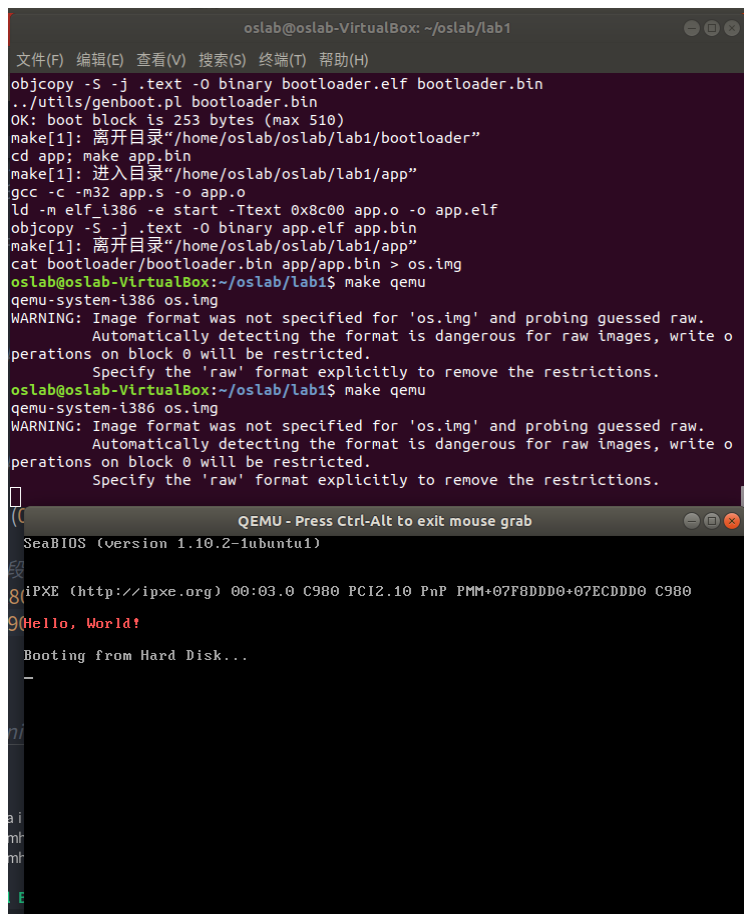
1 // @file name : myGen.c
2 // @author : wayne-ziqi
3 // @setup time : 22/3/6
4
5 // "transfer mbr.bin to the format of boot loader"
6 #include <stdio.h>
7 #include <string.h>
8 #include <memory.h>
9 #include <sys/stat.h>
10
11 char buf[512];
12 char outname[128];
13
14 int main(int argc, char **args)
15 {
16     FILE *fp = fopen(args[1], "r");
17     if (fp == NULL)
18     {
19         printf("no such file %s in directory\n", args[1]);
20         return -1;
21     } else{
22         printf("open file %s successfully\n", args[1]);
23     }
24     struct stat statbuf;
25     stat(args[1], &statbuf);
26     int fsize = statbuf.st_size;
27
28     memset(buf, 0, 512 * sizeof(char));
29     buf[510] = 0x55;
30     buf[511] = 0xaa;
31     printf("block is %dB\n", fsize);
32     fread(buf, sizeof(char), fsize, fp);
33     fclose(fp);
34
35     memset(outname, 0, 128 * sizeof(char));
36     if (argc > 2)
37         memcpy(outname, args[2], strlen(args[2]) * sizeof(char));
38     else
39         memcpy(outname, args[1], strlen(args[1]) * sizeof(char));
40
41     FILE *fout = fopen(outname, "w");
42     if(fout == NULL){return -2;}
43     fwrite(buf, sizeof(char), 512, fout);
44     fclose(fout);
45     return 0;
46 }

```

我们用生成好的 mbr2pro.bin 尝试执行指令（不小心暴露了debug的过程🙈🙈）



2. task2成果展示



task2在task1基础上实现了加载了一个小小的操作系统模拟程序打印出了hello word，理论上可以打出任何。

3. 实验收获

1. 了解了真正的操作系统的装在过程，并初步认识到了引导操作系统的系统BIOS
2. 重新复习了实地址模式和保护模式的实现机制

3. 重新熟悉了汇编指令的编写逻辑
4. 了解了更多gdb的黑科技，不再会轻易落入非IDE敲不出代码的窘境。