操作系统实验报告

学号： 201220139

姓名： 艾泠州

邮箱： 201220139@smail.nju.edu.cn

日期： 2022年3月15日

---

1. Exercises

1. 请反汇编 `Scrt1.o`，验证下面的猜想（加 `-r` 参数，显示重定位信息）

```
0000000000000000 <_start>:   ### 默认程序入口
   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   ### 重定位到main函数
  ### 跳转到main函数
  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
```

2. 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？
   (2) 电脑启动时 CS 寄存器和 IP 寄存器的值是什么？
   (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）

   答：(1) `0x000fffff0`。(2) `0xf000:0xfff0`。(3) `ljmp $0xf000,$0xe05b`；i8086 的 BIOS 地址范围为 `0x000f0000-0x000fffff`，而 QEMU 的 BIOS 启动地址为固定 `0x000fff0`，剩余空间不足以保存 BIOS 必需的代码，因此 BIOS 代码保存在更低地址，要跳转到该地址执行 BIOS 代码。

3. 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。

   答：执行 `qemu-system-i386 -nographic -s -S os.img` 命令，让 QEMU 以无窗口、允许远程控制、在开始运行时不启动 CPU 的模式运行；执行 `gdb -n -x ./.gdbconf/.gdbinit` 命令，以通过 `.gdbinit` 文件初始化且不执行初始化命令的模式启动 GDB。

4. 继续用 `si` 看见了什么？

```
The target architecture is set to "i8086".
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x70c8
[f000:e062]    0xfe062: jne    0xfd414
[f000:e066]    0xfe066: xor    %dx,%dx
[f000:e068]    0xfe068: mov    %dx,%ss
[f000:e06a]    0xfe06a: mov    $0x7000,%esp
[f000:e070]    0xfe070: mov    $0xf2d4e,%edx
[f000:e076]    0xfe076: jmp    0xfff00
[f000:ff00]    0xfff00: cli
[f000:ff01]    0xfff01: cld
[f000:ff02]    0xfff02: mov    %eax,%ecx
[f000:ff05]    0xfff05: mov    $0x8f,%eax
[f000:ff0b]    0xfff0b: out    %al,$0x70
[f000:ff0d]    0xfff0d: in     $0x71,%al
[f000:ff0f]    0xfff0f: in     $0x92,%al
[f000:ff11]    0xfff11: or     $0x2,%al
[f000:ff13]    0xfff13: out    %al,$0x92
[f000:ff15]    0xfff15: mov    %ecx,%eax
[f000:ff18]    0xfff18: lidtw  %cs:0x70b8
[f000:ff1e]    0xfff1e: lgdtw  %cs:0x7078
[f000:ff24]    0xfff24: mov    %cr0,%ecx
[f000:ff27]    0xfff27: and    $0x1fffffff,%ecx
[f000:ff2e]    0xfff2e: or     $0x1,%ecx
[f000:ff32]    0xfff32: mov    %ecx,%cr0
[f000:ff35]    0xfff35: ljmpl  $0x8,$0xfff3d
The target architecture is set to "i386".
=> 0xfff3d: mov    $0x10,%ecx
=> 0xfff42: mov    %ecx,%ds
=> 0xfff44: mov    %ecx,%es
=> 0xfff46: mov    %ecx,%ss
=> 0xfff48: mov    %ecx,%fs
=> 0xfff4a: mov    %ecx,%gs
=> 0xfff4c: jmp    *%edx
=> 0xf2d4e: push   %ebx
=> 0xf2d4f: sub    $0x20,%esp
...
```

5. 中断向量表是什么？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 `Hello World!` 的。

   答：中断向量表是将中断类型码与中断处理程序关联起来的结构，根据中断向量表，程序可以跳转到对应的中断处理程序执行。MBR 输出 `Hello World!` 的原理为：先初始化各段寄存器，再调用 `displayStr(char*, size_t)` 函数，此函数通过中断向量表中的系统调用来输出字符串。

6. 为什么段的大小最大为 64 KB？请在报告上说明原因。

   答：因为 8086 的寄存器大小为 16 位，最多可以表示 64 KB 的地址。

7. 假设 `mbr.elf` 的文件大小是 300 byte，那我是否可以直接执行 `qemu-system-i386 mbr.elf` 这条命令？为什么？

答：不能。因为虚拟机刚上电时无法解析 elf 文件；且其所在扇区的末尾两字节均为 `0x00`，而扇区末尾必须为魔数 `0x55 0xaa` 才能被加载。

8. 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
```

答：`ld`：

`-m elf_i386`：使用 elf-i386 链接器进行链接；

`-e start`：设置程序入口函数为 `start`；

`-Ttext 0x7c00`：指定 `.text` 节的起始地址为 `0x7c00`。

`objcopy`：

`-S`：移除所有标志及重定位信息；

`-j .text`：抽取 `.o` 文件中的 `.text` 节信息；

`-O binary mbr.o mbr.bin`：二进制模式，源文件为 `mbr.o`，目标文件为 `mbr.bin`。

9. 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：若文件大小大于 510 字节，则直接退出程序；若小于等于 510 字节，则在文件尾部填充 `0x00` 至 510 字节，再在文件末尾添加 `0x55 0xaa` 魔数。因为扇区大小为 512 字节，且扇区末尾必须为魔数，因此要将文件填充至 512 字节以保证占满整个扇区。

10. 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。

```
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)
```

答：反汇编程序将本应是字符串的 `message` 节（即 `0x17` 至 `0x26` 字节）反汇编成了指令。因为 `.bin` 文件相比 `.o` 文件，移除了重定位、节头表等信息，没有 `elf` 文件那样的复杂的结构。而机器刚上电时，是无法解析 `elf` 文件的，只能运行简单的二进制文件。

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

    答：在 bootloader 阶段，`data32 ljmp $0x08, $start32` 将 `cs` 段寄存器设置为 `0x08`，之后将 `ds` `es` `fs` `ss` 段寄存器设置为 `0x10`，将 `gs` 段寄存器设置为 `0x18`。故 `CS` `DS` `GS` 的 `index` 分别为 1、2、3，对应第 2、3、4 个段描述符。

12. 请回答 `app.s` 是怎么利用显存显示 `Hello World!` 的。

    答：在 `displayStr(char*, size_t)` 函数中，将字符串首地址放入 `ebx` 寄存器，将字符串长度放入 `ecx` 寄存器，并将显存的起始地址放入 `edi` 寄存器（在这里为第 6 行的起始地址，由于每个字符占 2 个字节（其中高字节固定为 `0x0c`，是字符本身的一些属性，如前景色、背景色等），故地址为 (80 * 5 + 0) * 2）。通过循环，每次 `ebx` 寄存器加 1，`ecx` 寄存器减 1，`edi` 寄存器加 2。当 `ecx` 寄存器为 0 时，循环结束，函数返回。

13. 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/bootloader.bin` 和 `app/app.bin` 文件中的内容写入 `os.img` 中。

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

    不可以。因为在 `start.s` 中将栈顶设置为了 `0x8000`，而 `0x7c20` 在栈的内部，会破坏栈的结构。

15. 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：电脑开始加电时，从 BIOS 开始工作。首先从 `0xffff0` 地址处开始执行第一条指令，之后跳转到 BIOS 内的一个更低的地址执行一些操作。执行完毕后， CPU 进入保护模式，并设置段寄存器。之后，执行 bootloader 程序，将 OS kernel 装载到内存中。最后跳转到 OS，将控制权交给 OS kernel。

2. Tasks

1. 以下任务点是我们在本节需要完成的：把 `cr0` 的低位设置为1；填写 GDT；显示 `Hello World!`。

   (1) 将 `cr0` 的最低位设置为 1：

   ```
       movl %cr0,%eax
       orl $1,%eax
       movl %eax,%cr0
   ```

   (2) 填写 GDT：

   ```
       # 第一个描述符是NULL
       .word 0,0
       .byte 0,0,0,0
       # TODO: 代码段描述符，对应cs
       .word 0xffff,0
       .byte 0,0x9f,0xcf,0
       # TODO: 数据段描述符，对应ds
       .word 0xffff,0
       .byte 0,0x93,0xcf,0
       # TODO: 图像段描述符，对应gs
       .word 0xffff,0x8000
       .byte 0x0b,0x93,0xcf,0
   ```

   代码段和数据段的 base 字段均设置为 0，limit 字段设置为 `0xffffffff`；图像段则按照约定将 base 字段设置为 `0xb8000`。

   (3) 输出 `Hello World!`：

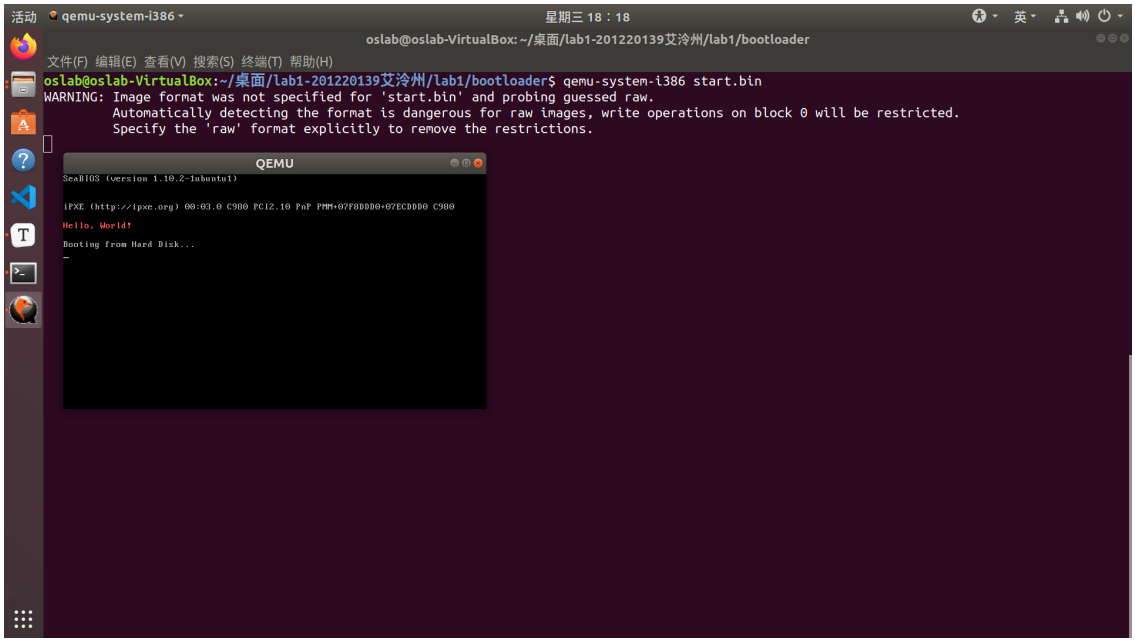
   ```
   .code32
   start32:
       movw $0x10, %ax # setting data segment selector
       movw %ax, %ds
       movw %ax, %es
       movw %ax, %fs
       movw %ax, %ss
       movw $0x18, %ax # setting graphics data segment selector
       movw %ax, %gs
       movl $0x8000, %eax # setting esp
       movl %eax, %esp
       # 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
```

编译、链接、生成 MBR 文件并运行，结果如图：



注意到，在 `displayStr()` 函数中，改变 `edi` 寄存器的初始值，将改变字符串输出的起始位置。

2. 以下任务点是在本节需要完成的：把上一节保护模式部分搬过来；填写 `bootMain` 函数。

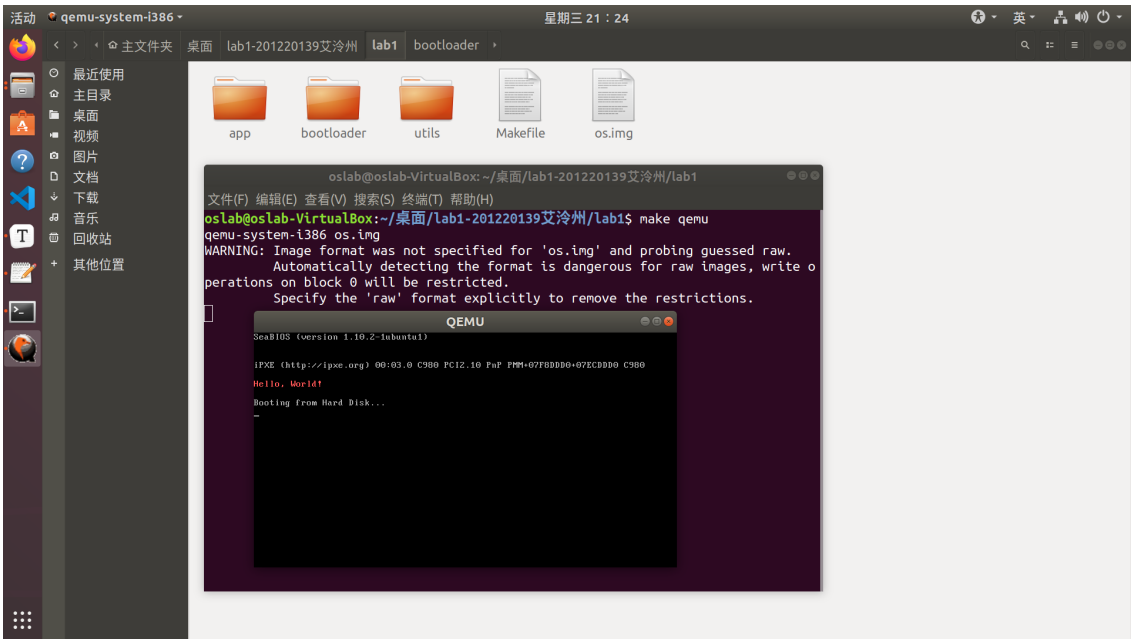
   保护模式代码如下：

```
.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
    # TODO: 跳转到bootMain
    call bootMain
loop:
    jmp loop
```

   `bootMain` 函数实现如下：

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

执行结果如图：



3. Challenges

请尝试使用其他方式，构建自己的 MBR，输出 `Hello, world!`

使用 C++ 语言，来代替 `genboot.pl`，来生成符合 MBR 格式的 `mbr.bin` 。文件位置：`report/challenge/my_genboot.cpp`。