2022spring《操作系统》lab1实验报告

姓名：武雅琛
学号：201220102

一、PART1: 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）**

```
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
```

可以看出，在`Scrt1.o`中定义了`_start`符号，并且在`24`的位置上跳转到`main`函数执行。

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？**
    `0xffff0`，位于模拟物理空间的**BIOS区域**内。
  - **电脑启动时`CS`寄存器和`IP`寄存器的值是什么？**
    `CS`寄存器值为`0xf000`。`IP`寄存器的值为`0xfff0`。
  - **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）**
    `Ljmp`**指令。**
    因为16 * 段 + 偏移量 = 物理地址，得到`0xffff0`但是刚好位于模拟物理空间的BIOS区域的**最后16个bytes的首地址**，所以首先要跳转到BIOS区域的首地址0xf0000，从而**运行硬件自检代码BIOS系统**，然后由此跳转到开机引导程序`bootloader`。

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）**
  - `make qumu-noc-gdb`对应以下命令。

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

在实验手册中，我们得知了`qemu-system-i386 -s -S os.img`的含义。所以我们只需要了解`-nographic`的含义即可。使用`man qumu-system-i386`命令查阅手册可知该参数的含义是在启动`qemu`后不启用图形化界面而是单纯以命令行形式交互。模拟串口的输出将直接重定向到控制台。

- `make gdb`对应以下命令。

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

通过`man gdb`命令查阅手册。

`-n`参数：`Do not execute commands from any .gdbinit initialization files.`不运行`.gdbinit`初始化文件中的任何命令。

`-x`参数：`execute GDB commands from the file.`运行文件中的GDB命令。

粗略理解就是执行目录下`.gdbinit`的GDB命令来进行**GDB软件**的初始化。

ps：-n和-x为什么不会冲突？.gdbinit里的命令除了检测硬件模式还有什么功能？

exercise4

- **继续用`si`看见了什么？请截一个图，放到实验报告里。**(因为实在是太长了，这里直接用剪贴板复制过来了)
  - 初始化`ss`**段寄存器**和`esp`**栈顶寄存器**
  - 清空**中断位**
  - 设置`gdt`表和`idt`表首址寄存器
  - 设置`cr0`控制寄存器并开启**保护模式**

```
[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    0xfff00
0x0000e076 in ?? ()
(gdb) si
[f000:ff00]    0xfff00: cli
0x0000ff00 in ?? ()
```

```
(gdb) si
[f000:ff01]    0xfff01: cld
0x0000ff01 in ?? ()
(gdb) si
[f000:ff02]    0xfff02: mov    %eax,%ecx
0x0000ff02 in ?? ()
(gdb) si
[f000:ff05]    0xfff05: mov    $0x8f,%eax
0x0000ff05 in ?? ()
(gdb) si
[f000:ff0b]    0xfff0b: out    %al,$0x70
0x0000ff0b in ?? ()
(gdb) si
[f000:ff0d]    0xfff0d: in     $0x71,%al
0x0000ff0d in ?? ()
(gdb) si
[f000:ff0f]    0xfff0f: in     $0x92,%al
0x0000ff0f in ?? ()
(gdb) si
[f000:ff11]    0xfff11: or     $0x2,%al
0x0000ff11 in ?? ()
[f000:ff13]    0xfff13: out    %al,$0x92
0x0000ff13 in ?? ()
(gdb) si
[f000:ff15]    0xfff15: mov    %ecx,%eax
0x0000ff15 in ?? ()
(gdb) si
[f000:ff18]    0xfff18: lidtw  %cs:0x70b8
0x0000ff18 in ?? ()
(gdb) si
[f000:ff1e]    0xfff1e: lgdtw  %cs:0x7078
0x0000ff1e in ?? ()
(gdb) si
[f000:ff24]    0xfff24: mov    %cr0,%ecx
0x0000ff24 in ?? ()
(gdb) si
[f000:ff27]    0xfff27: and    $0x1fffffff,%ecx
0x0000ff27 in ?? ()
(gdb) si
[f000:ff2e]    0xfff2e: or     $0x1,%ecx
0x0000ff2e in ?? ()
(gdb) si
[f000:ff32]    0xfff32: mov    %ecx,%cr0
0x0000ff32 in ?? ()
(gdb) si
[f000:ff35]    0xfff35: ljmpl  $0x8,$0xfff3d
0x0000ff35 in ?? ()
(gdb) si
The target architecture is set to "i386".
=> 0xfff3d: mov    $0x10,%ecx
0x000fff3d in ?? ()
```

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》后，再简述一下示例MBR是如何输出helloworld的。**
  - 在**实地址模式**下存放异常和中断对应的异常和中断处理程序的**入口地址**存放在中断向量表中。
  - 中断向量表位于`0x0000`~`0x3ff`，共256组，每组占**四个字节**`CS::IP`。
  -

exercise6

- **为什么段的大小最大为64KB，请在报告上说明原因。**
  - 因为8086实模式下**地址总线20位**，寻址空间为1MB。寻址方式为物理地址 = `cs << 4 + offset`，则最大情形下`cs`寄存器决定物理地址的高四位，则偏移量的范围是`64KB`。

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?**

不可以。

第一文件没有指定从磁盘装载到内存的**地址**`0x7c00`，那么BIOS很可能在`0x7c00`根本**找不到MBR**。

其次文件末尾没有**魔数**，那么BIOS在“验明正身”的时候认为并非MBR程序，找不到启动设备。

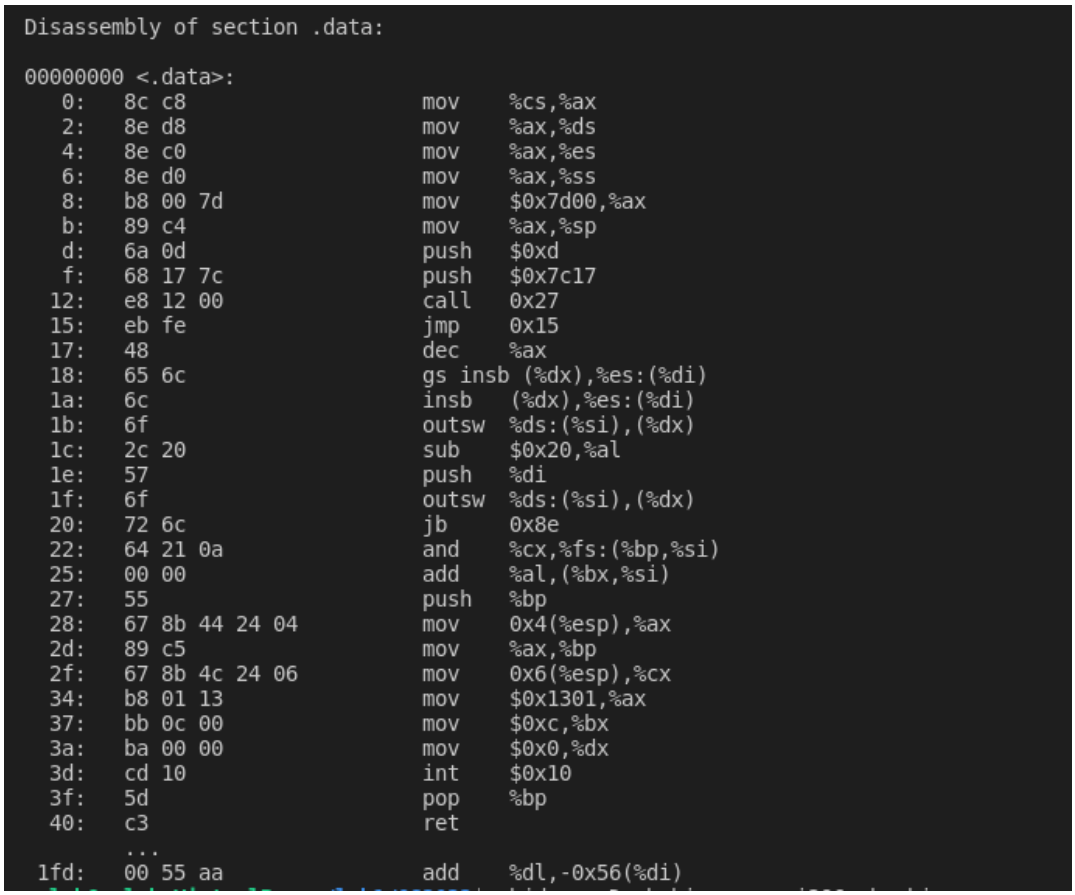
exercise8

- **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）**
  - ```
    $ld -m elf_i386 -e start -Ttext 0x7c00 mbr.o -o mbr.elf
    ```
    - `ld`是一种**链接器**。
    - `-m elf_i386 -m` 。`-m`指的是为链接器选择一种预定的仿真内部链接脚本来实现应对**不同硬件架构和操作系统的链接动作**。其次，我们选择了`elf_i386`的参数，则链接依据linux`elf`**文件格式**和`i386`**的处理器架构**的仿真链接脚本实现的。
    - `-e start`。`-e`是用于制订**程序的执行起点**，可以是符号或者地址。此处指定了链接得到的程序的执行起点的符号是`start`
    - `-Ttext 0x7c00`**指定节在输出文件中的绝对地址**。此处指定在`0x7c00`，即文件从磁盘上读取后会装载到内存的这个地址，**符合**`MBR`**文件**的特征。
  - ```
    `$objcopy -S -j .text -O binary mbr.elf mbr.bin`
    ```
    - `objcopy`是一种将目标文件的内容**复制**到另一种类型的目标文件的工具。
    - `-S`指的是在拷贝时忽略**重定位信息**和**符号信息**。
    - `-j .text`指的是在拷贝时只保留**只读代码节**。
    - `binary`指定拷贝为**binary格式**。

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。**
  - 粗浅观之，创建了一个`buf`变量，大小为`510 - 文件长度`，内容一律置为`\0`。在末尾又添加了两个字节`\0x55\0xAA`,也就是**BIOS**识别**MBR**的**“魔数”**。最后把这个变量`print`到了之前的文件中。

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**
- be like：
  - 

```
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)
```

- 初始化**段寄存器**`ds`,`es`,`ss`。
- **栈顶寄存器**`sp`，**栈基底寄存器**`bp`
- 然后在寄存器中传入参数：字符串的地址和长度，最后触发**10号系统调用**使得BIOS输出字符串`“Hello World!”`。

exercise11

- **请回答为什么三个段描述符要按照cs，ds，gs的顺序排列?**

  因为在初始化段选择子`sreg`的时候，`index`**在cs，ds，gs中依次置为1，2，3。**则在通过段选择子查找段描述符时对应了段描述符表的1，2，3项的内容。

exercise12

- **请回答app.s是怎么利用显存显示helloworld的。**

```
.code32

.global start
start:
    pushl $13
    pushl $message
    calll displayStr
loop:
    jmp loopS

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
```

- 首先将输出字符串的**长度和地址**压入栈桢作为参数调用**输出函数**`displayStr`。
- 然后将在当前的VGA模式下想要**输出的位置**存放在`edi`**寄存器**中，将输出字符**期望的字符属性**放入`ah`**寄存器**中。
- 最后进入循环以此输出字符，每次输出位置随之递增以达到连续输出而不发生覆盖的目的，当`ecx`**为0**时说明所有字符依次显示。最后进入`loop`**死循环**界面。

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。**
  - `cat`命令如果追加多个文件并重定向，则作用时将文件内容由前到后依次**拼接合并**在一次导入到`os.img`。

exercise14

- **如果把app读到**`0x7c20`**，再跳转到这个地方可以吗？为什么？**
- 不可以哦。
- 首先，我尝试了一下，不能成功跳转到OS输出字符串。
- 然后，我考虑到时BIOS将MBR装载到内存的`0x7c00`地址处执行，这里也就是我们的`bootloader`，然后在运行`bootloader`的时候读取位于第一扇区的OS，但是如果将OS读取到`0x7c20`极大可能性会直接**将**`bootloader`**的代码内容覆盖**导致运行混乱。

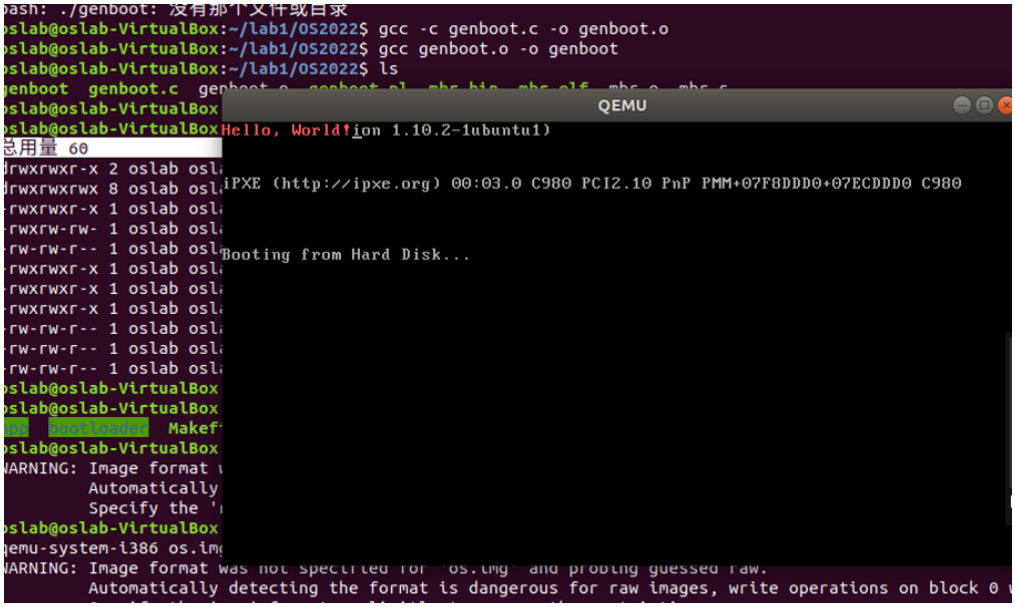
exercise15

- **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。**
  - **BIOS。**首先CPU运行的第一条指令是硬件厂商提供的BIOS指令：在这里初始化栈帧，加载IDTR和GDTR等一些基本的初始化工作并且并且进行开机硬件自检。
  - **MBR。**BIOS最后会将磁盘的0号扇区也就是MBR装载到内存0x7c00的内存地址上，并且识别“魔数”，将控制交付给bootloader。

    在这里进行更多的初始化工作，段寄存器，栈帧，开启保护模式。最后将操作系统搬到内存上，将控制交给操作系统执行。

二、PART2:challenge

challenge1

- **请尝试使用其他方式，构建自己的MBR，输出“Hello，world!”**
  - 首先分析了`genboot.pl`文件，可以分为以下几个部分：**分析命令行参数**，填写用于写入文件的**buf字节流**，将**buf写入文件**，关闭文件。同时包括对命令行参数个数的判断和对文件大小的判断。
    - 实现以上功能可以利用c语言的文件操作和解析命令行参数的操作：先判断`argc`->含有一个命令行参数->`fopen`打开`mbr.bin`文件->`ftell`得到文件长度->计算得到`buf`长度->用`memset`填写`buf`->`fwrite`将`buf`写入文件->`close`
    - 

三、PART3:task

task1

- **把cr0的低位设置为1。**,因为i386没有提供直接对cr0计算操作的指令，所以要通过通用寄存器实现。

```
movl %cr0,%eax
orl   0x1,%eax
movl %eax,%cr0
```

- **填写GDT。**

```
# TODO：代码段描述符，对应cs
    .word 0xffff, 0
    .byte 0,(0x90 | 0xa),(0xc0 | 0xf),0

# TODO：数据段描述符，对应ds
    .word 0xffff, 0
    .byte 0, (0x90 | 0x2), (0xc0 | 0xf), 0

# TODO：图像段描述符，对应gs
    .word 0xffff, 0x8000
    .byte 0x0b,(0x90 | 0x2), (0xc0 | 0xf), 0
```

- 填写cs、ds段寄存器

  查阅在linux扁平模式下，base为`0x0`，limit为`0xfffff`。另外根据助教老师在手册上提供的TYPE含义表得出，代码段TYPE为`1010B`，数据段是`0010B`。

bit 3	Data/Code	0 (data)
bit 2	Expand-down	0 (normal) 1 (expand-down)
bit 1	Writable	0 (read-only) 1 (read-write)
bit 0	Accessed	0 (hasn't) 1 (accessed)

bit 3	Data/Code	1 (code)
bit 2	Conforming	0 (non-conforming) 1 (conforming)
bit 1	Readable	0 (no) 1 (readable)
bit 0	Accessed	0 (hasn't) 1 (accessed)

- 填写gs段寄存器

  结合实验手册显存设置在内存寻址空间从`0xb8000`开始的区域，所以base为`0xb8000`，同时对于显存为可修改段，结合数据段的TYPE为`0x2`。

- **显示helloworld。**

  这一部分比较容易，参考`app.s`， 把显示部分的代码搬过来就好，没什么新增的，不赘述了。

task2

- **把上一节保护模式部分搬过来。**

  腾挪过来就可以，没啥。

  就是添加一条汇编指令

```
jmp bootMain
```

  因为我们不知道bootMain在程序中的具体地址。这里表明在链接时回去符号表查找地址并填写。

- **填写bootMain函数。**

```
void bootMain(void) {
    readSect((void*)0x8c00, 1);
    asm volatile(
        "movl $0x8c00, %eax\n\t"
        "jmp *%eax"
    );
}
```

  首先使用`readSect`函数读取`disk`第一个扇区的内容填写到从`0x8c00`开始的指定的内存地址上。

  然后进行内联汇编，跳转到指定的绝对地址即可。