南京大学本科生实验报告

课程名称：**操作系统实验**　　任课教师：**叶保留**

学院	**计算机科学与技术**	专业（方向）	**计算机科学与技术**
学号	**201220156**	姓名	**余越**
Email	**yuyue2002163@163.com**	完成日期	**2022/3/13**

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
```

由反汇编结果可知，会调用 callq 指令，执行_start+0x2a 处地址，说明_start 是默认的程序入口。

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）

地址为 0x000ffff0，位于 BIOS ROM 的末尾。
cs 寄存器为 0xf000,ip 寄存器为 0xfff0 。
指令如图所示。
设计原因：当 PC 加电后，CPU 的寄存器（cs,ip 等）被设为某些特定值（0xf000,0xfff0）。保证处理器复位后，第一条指令直接执行跳转。而该指令由于位于 BIOS 末尾，只能进行跳转，到达 0xfe05b 地址，进入 BIOS POST 过程。

```
[f000:fff0]     0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
```

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）

如图所示：

```
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
```

make gdb 操作:

参数:

-x：从指定文件（.gdbinit 文件）中执行 GDB 指令

-n：不从任何.gdbinit 初始化文件中执行命令。通常情况下，这些文件中的命令是在所有命令选项和参数处理完后才执行。

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 -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。

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

```
[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
```

```
[f000:ff02]    0xfff02: mov    %eax,%ecx
0x0000ff02 in ?? ()
(gdb) sis
Undefined command: "sis".  Try "help".
(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 ?? ()
(gdb) si
[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
```

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。

8086 系统是把所有的中断向量集中起来，按中断类型号从小到大的顺序存放到存储器的某

一区域内，这个存放中断向量的存储区叫做中断向量表，即中断服务程序入口地址表。

在示例 MBR 中：

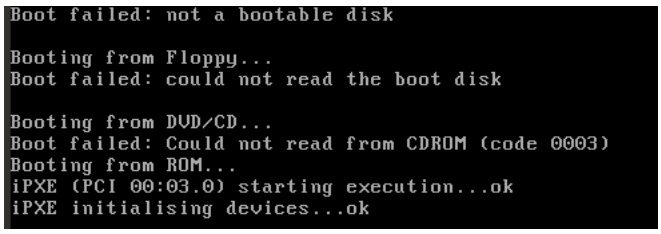
```
    首先将参数（字符串长度，字符串地址）入栈：
pushw $13             # pushing the size to print into stack
pushw $message        # pushing the address of message into stack
    再调用 displayStr：
callw displayStr      # calling the display function
    准备功能参数：
movw $0x1301, %ax     #ah 为 0x13，表示显示字符
                      #al 为 0x01，光标跟随移动
movw $0x000c, %bx     #bl 为 0x00，bx 为 0x0c，页表为 0
movw $0x0000, %dx     #起始行列为 0
movw 6(%esp), %cx     #串长度为 (%esp + 0x6)，存储着存入的字符串长度 13
    调用 10 号中断（BIOS 对屏幕及显示器提供的服务程序）
int $0x10             #打印"Hello, World!\n\0"
```

exercise6：为什么段的大小最大为64KB，请在报告上说明原因。

采用物理地址（20 位） = 段寄存器 << 4 + 偏移地址（16 位）的寻址方法，导致段内偏移量最大为 0x10000，即 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？

不能直接执行，mbr.elf 不是真正的 mbr 文件，末尾没有魔数 0x55,0xaa。qemu 检查无法识别，会出现类似的警告：



找不到启动设备！

```
$ld -m elf_i386 -e start -Ttext 0x7c00 mbr.o -o mbr.elf
$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: elf_i386： 输出为 elf 的目标格式，386 是目标平台。

-m：设置仿真
-e start：设置 start（Mbr 中的第一个函数）为起始地址
-Ttext 0x7c00：设置.text 节地址为 0x7c00
mbr.o -o mbr.elf：将 mbr.o 生成 mbr.elf

objcopy:
-S：删除所有符号和重定向信息
-j .text：只拷贝.text 文件
-O binary：生成的输出文件为二进制
mbr.elf mbr.bin：生成 mbr.bin 二进制文件

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。

```
if($n > 510){
print STDERR "ERROR: boot block too large: $n bytes (max 510)\n";
exit 1;
#如果大于 510 字节，则退出并打印错误信息
}
print STDERR "OK: boot block is $n bytes (max 510)\n";
#打印目标文件长度信息
$buf .= "\0" x (510-$n);
$buf .= "\x55\xAA";
open(SIG, ">$ARGV[0]") || die "open >$ARGV[0]: $!";
#用“    ...    55aa”填充文件，直至 mbr.bin 文件达到 510 字节。
print SIG $buf;
#打印文件内容
```

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

符合.bin 格式，没有分段信息，本来应该归于.code 段的内容被识别成了.data 段
在 mbr.bin 最后部分被 0 填充，并加上魔码 0x55,0xaa。

```
oslab@oslab-VirtualBox:~/mylab/OS2022$ objdump -D -b binary -m i386 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(%si),%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)
```

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

在前面的代码中，段描述符的初始化是按照 cs,ds,gs 的顺序执行的。

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

```
pushl $13
pushl $message
....
movl 4(%esp), %ebx          #ebx 寄存器存入 message 地址
movl 8(%esp), %ecx          #ecx 寄存器存入 message 长度
movl $((80*5+0)*2), %edi    #edi 寄存器内为 0x320
movb $0x0c, %ah             #eax 寄存器中 ah 为 0x0c
#这是循环将字符送入显存
nextChar:
    movb (%ebx), %al        #ax 寄存器 := 0x0c + 字符
    movw %ax, %gs:(%edi)    #将 ax 存入显存（0xb8000 + edi）地址中
    addl $2, %edi           #edi 寄存器加 2
    incl %ebx               #ebx 寄存器加 1
    loopnz nextChar         #循环每次 ecx 寄存器减一，直至为 0
```

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 将 app.bin 文件（位于 app 目录中）附加在 bootloader.bin 文件（位于 bootloader 目录中），形成一个新的文件：os.img

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

可以在 qemu 上运行。因为在内存划分中 0x7c00 到 0x7e00 的内容被规划到 mbr 主引导区了。而本次实验中 app 属于 bootloader 内容(归于 mbr)

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。

加电之后，CS 被设置为 0xf000，IP 被设置为 0xfff0，执行第一条指令：跳转到 0xfe05b 处，执行接下来的其他 BIOS 指令（设置 ss 和 esp 寄存器，通过 cli 指令屏蔽了中断
通过 in，out 指令和 IO 设备交互，进行初始化，打开 A20 门，然后用 lidtw 与 lgdtw 加载 IDTR 与 GDTR。最后开启保护模式，长跳转到 BIOS 的主要模块进行执行）。
接下来进行 BIOS 的 POST 过程，其中包括加电自检，检测 CPU 各寄存器、计时芯片、中断芯片、DMA 控制器，等操作，然后执行 INT 19h 指令，执行自举过程。
在自举过程中，将 MBR 的内容调入地址为 0x7c00 的地方，期间会识别磁盘中末尾的魔数（0x55,0xAA（0101...便于识别））。最后跳转到 0x7c00 处执行指令。主引导扇区中（已经装入 0x7c00 处）的加载程序的功能主要是将操作系统的代码和数据从磁盘加载到内存中，然后跳转到操作系统的起始地址。