exercise1.

_start 反汇编代码中地址为 0x24 有一条 call 指令，用 objdump 显示重定位信息可以看出在地址为 0x26 处的重定位地址为__libc_start_main 函数的入口地址

```
oslab@oslab-VirtualBox:~$ objdump -d /usr/lib/x86_64-linux-gnu/Scrt1.o

/usr/lib/x86_64-linux-gnu/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>
  16:   48 8b 0d 00 00 00 00    mov    0x0(%rip),%rcx        # 1d <_start+0x1d>
  1d:   48 8b 3d 00 00 00 00    mov    0x0(%rip),%rdi        # 24 <_start+0x24>
  24:   ff 15 00 00 00 00       callq  *0x0(%rip)        # 2a <_start+0x2a>
  2a:   f4                      hlt
```

```
oslab@oslab-VirtualBox:~$ objdump -r /usr/lib/x86_64-linux-gnu/Scrt1.o

/usr/lib/x86_64-linux-gnu/Scrt1.o:     文件格式 elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000012 R_X86_64_REX_GOTPCRELX  __libc_csu_fini-0x0000000000000004
0000000000000019 R_X86_64_REX_GOTPCRELX  __libc_csu_init-0x0000000000000004
0000000000000020 R_X86_64_REX_GOTPCRELX  main-0x0000000000000004
0000000000000026 R_X86_64_GOTPCRELX  __libc_start_main-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

exercise2

1.电脑开机的第一条指令的地址是 0xffff0, 它位于 BIOS 中。

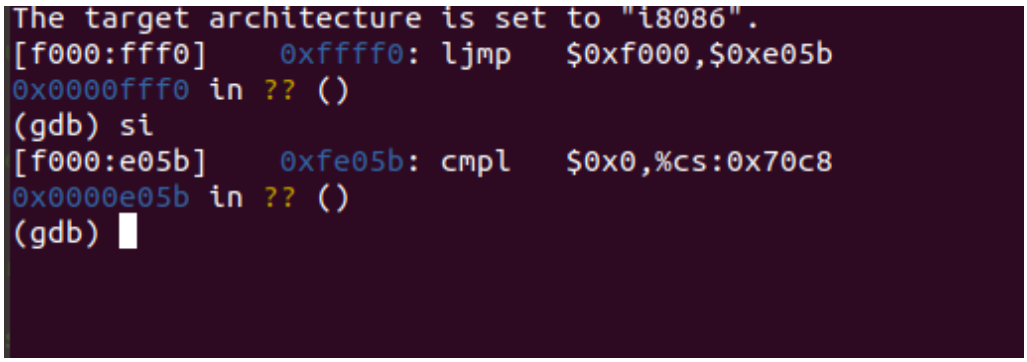
2.电脑启动时 CS:EIP=[0xf000:0xfff0]。

3.第一条指令是“ljmp　$0xf000,$0xe05b”。BIOS 物理地址范围是 0x000f0000-0x000fffff, 而 0x000ffff0 距离 BIOS 结束只有 16 个字节，无法执行复杂的指令逻辑。

exercise3

执行 make 后生成 os.img，执行 make qemu-nox-gdb 将会执行“qemu-system-i386 -s -S os.img”指令运行 os.img。 执行 make gdb 将会执行“gdb -n -x ./.gdbconf/.gdbinit”将会启动 gdb 运行。

exercise4



exercise5

1.中断向量表包含中断服务例程或异常处理函数的入口地址

2.输出 helloworld:

启动 qemu，由 qemu 的 BIOS 将 mbr.bin 加载到内存的 0x7c00 位置并执行该段代码

把字符串的长度放入%cx，字符属性放入%bx,字符串的地址放入%bp,字符在屏幕上的起始位置放入%dx,再通过 int $0x10 中断使用 BIOS 显示服务输出字符串

exercise6

因为 8086 段寄存器为 16 位，寻址方式为物理地址 = 段寄存器 << 4 + 偏移地址，16 位地址最大的可寻址地址空间为 64KB.

exercise7

不能。因为 BIOS 需要检查扇区末尾的两个字节（即第 511 和 512 个字节）来判断是否执行这段指令。

exercise8

ld：-m elf_i386 设置 elf_i386 仿真；-e start 使用 start 作为程序执行的入口标志；-Ttext 0x7c00 设置.text 节的地址；-o 设置输出文件名

objcopy:-S 不从源文件复制重定位和符号信息；-j .text 仅从源文件复制.text 节信息到目标文件；-O binary 以二进制格式写入目标文件

exercise9

如果文件超过 510 个字节，输出文件过大的提示信息并退出程序；否则，输出文件大小在 510 字节内的提示信息，并将文件扩充至 512 个字节——除了第 511，512 个字节用 0x55,0xaa 扩充外，其余用'\0'扩充。

因为 BIOS 需要通过确认读取的内容的最后两个字节内容是否为 0X55,0XAA 来确认是

否装入了正确的程序。

exercise10

```
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)
(END)
```

程序指令从 0 开始，0x0-0x40 是程序指令；0x41-0x1fd 是为扩充文件所加的'\0';最后两个字节是魔数。

exercise11

查找段描述的方式为[GDTR]+SELECTOR.index*8;从 start.s 中可以看出 cs，ds,gs 的段选择子分别为 0x8, 0x10, 0x18;索引分别为 1，2，3，因此三者顺序为 cs, ds, gs。

exercise12

app.s 的执行过程如下：

先将字符串的长度以及起始地址压栈，然后跳转到 displayStr 处执行

将字符串起始地址和长度分别放入 ebx,ecx 寄存器，并将和字符属性有关的信息

0x0c 放入%ah

从 ebx 寄存器指向的地址中取出字符的 ASCII 码，并将其放入%al 中，然后将%ax 中的内容（两个字节）放入显存，之后将 ebx 中的地址增加 1，循环取出要输出的字符并放入显存相应位置，直到将所有要输出字符放入显存为止

exercise13

将 bootloader.bin 和 app.bin 进行拼接，写入 os.img

exercise14

不可以。0x7c20 之后的地址包含 gdt 以及装入 app.bin 所需要的相关代码，如果把 app.bin 装入从 0x7c20 处开始的地址会把 gdt 以及那些相关代码破坏掉

exercise15

电脑开机后将 CS 设置为 0xf000, IP 设置为 0xfff0，在实地址模式下经地址转换后指向 BIOS 程序的入口地址

之后 CPU 开始执行 BIOS，BIOS 进行硬件检查并将 MBR 扇区中的 bootloader 装入内存中 0x7c00 的位置并执行该程序

然后由 bootloader 加载操作系统执行

Challenge

用 C 语言编写 genboot.c 来替换 genboot.pl 的功能，可执行文件为 genboot，接受需要改变格式的文件路径作为参数。

实现逻辑比较简单，打开文件并将其中的内容读入字符数组，如果其中的内容超过 510 字节，退出程序输出提示信息。否则用'\0'扩充至 510 个字节，再在末尾追加 0x55,0xaa 两个字节，然后将 buf 中的内容写入文件。