

# 《操作系统》实验一报告

201220069 周心同

## 一、实验目的

通过实际编程学习操作系统。

## 二、实验内容

本次实验需提交的内容：

- 15个exercise（编号从1到15），答案按照编号顺序写在实验报告里即可。
- 2个task，即代码任务，提交时请切换到load-os分支，打包时候记得把.git文件夹带上。并通过注释或在实验报告里说明思路。
- 1个challenge，需要写代码，请把代码提交在report文件夹里，并在报告里说明你的做法。
- 实验报告，提交在report文件夹里。

## 三、实验过程和结果

**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）

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

```
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     $0xffffffffffffffff,%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

```

其中 24: ff 15 00 00 00 00 callq \*0x0(%rip) 为从 \_start 跳转到 main 的代码。

## 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？  
地址是 0xffff0，位于 BIOS ROM。
- 电脑启动时 CS 寄存器和 IP 寄存器的值是什么？  
CS 为 0xf000，IP 为 0xffff0。
- 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 0xf000,0xe05b。

0xfffff0 是 BIOS 结束前16个字节(0x100000)。在第一条指令的后面只有16个字节，啥也干不了。所以通过ljmp跳转到更低的地址，让BIOS对系统进行基本的初始化工作。

```
oslab@oslab-VirtualBox:~/lab1$ make gdb
gdb -n -x ./gdbconf/.gdbinit
GNU gdb (GDB) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
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 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 ?? ()
```

验证cs和ip寄存器的值:

```
(gdb) info r
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                0xffff0            0xffff0
eflags             0x2                [ ]
cs                 0xf000             61440
ss                 0x0                0
ds                 0x0                0
es                 0x0                0
fs                 0x0                0
gs                 0x0                0
```

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

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

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 (QEMU会把在其上运行的操作系统的执行信息发送给GDB), 选项-S使得QEMU启动时不运行80386的CPU,选项-nographic使其不弹窗。

gdb:启动gdb, 选项-n不从任何.gdbinit初始化文件中执行命令,-x ../gdbconf/.gdbinit表示从初始化文件.gdbinit开始执行gdb。

#### exercise4:

继续用 `si` 看见了什么？请截一个图，放到实验报告里。

之后会关中断，打开A20线，加载GDTR和IDTR等。

```
[f000:e05b] 0xfe05b: cml $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 0xffff00
0x0000e076 in ?? ()
(gdb) si
[f000:ff00] 0xffff00: cli
0x0000ff00 in ?? ()
(gdb)
```

#### 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。

中断向量表按照中断类型号的顺序存储对应的中断向量，中断向量指向中断处理程序的地址。

首先将要输出的长度和字符串放入栈，调用函数displayStr，然后准备好中断需要的参数，通过int \$0x10中断进入内核态，INT 10H是由BIOS对屏幕及显示器所提供的服务程序，该程序会访问寄存器AL = 字符，BL = 前景色，然后显示在屏幕上。

### exercise6:

为什么段的大小最大为64KB，请在报告上说明原因。

因为寄存器为16位，所以偏移地址的范围是0000H-FFFFH共 $2^{16}$  (64K) 种取值，所以如果段的大小超过64KB，根据物理地址 = 段寄存器  $\ll$  4 + 偏移地址的算法，有些物理地址无法用段寄存器+偏移地址表示。

### 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？

不行。首先MBR要求是512byte，同时需要末尾有魔数。

### 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  
$objcopy -S -j .text -O binary mbr.elf mbr.bin
```

ld: GNU链接器

-m elf\_i386: 模拟elf\_i386链接器，是输出为 elf 的目标格式，386是目标平台。

-e start: 将start作为程序开始的符号

-Ttext 0x7c00: 定位绝对地址0x7c00作为MBR的起始地址

-o mbr.elf: 生成文件重命名为mbr.elf

objcopy: 把一种目标文件中的内容 (mbr.elf) 复制到另一种类型的目标文件 (mbr.bin) 中

-S: 移出源文件所有的标志及重定位信息

-j .text: 仅复制elf的.text节

-O binary: 以二进制形式写入输出文件

## 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。

如下两张图为mbr.bin文件前后的变化, 再结合genboot.pl脚本程序, 它发现文件没有大于510字节, 于是在最后两个字节补充魔数0x55aa, 并在前面补充0一直到文件大小为512字节, 这样使得文件符合MBR的条件。如果文件大于510字节, 那么就没有空间放两个字节的魔数并使得文件大小为512字节。

```
00000000: 8cc8 8ed8 8ec0 8ed0 b800 7d89 c46a 0d68 .....}..j.h
00000010: 177c e812 00eb fe48 656c 6c6f 2c20 576f .|....Hello, Wo
00000020: 726c 6421 0a00 0055 678b 4424 0489 c567 rld!...Ug.D$.g
00000030: 8b4c 2406 b801 13bb 0c00 ba00 00cd 105d .L$.....]
00000040: c3                                     .
~
```

```
00000000: 8cc8 8ed8 8ec0 8ed0 b800 7d89 c46a 0d68 .....}..j.h
00000010: 177c e812 00eb fe48 656c 6c6f 2c20 576f .|....Hello, Wo
00000020: 726c 6421 0a00 0055 678b 4424 0489 c567 rld!...Ug.D$.g
00000030: 8b4c 2406 b801 13bb 0c00 ba00 00cd 105d .L$.....]
00000040: c300 0000 0000 0000 0000 0000 0000 0000 .....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000070: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000110: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000130: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000140: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000150: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000160: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000170: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000180: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000190: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa .....U.
~
```



## 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

```
objdump -D -b binary -m i8086 mbr.bin
```

可以看出mbr.bin的反汇编结果和mbr.s的内容大致相同。

最后一行是魔数0x55aa。

```
mbr.bin:      文件格式 binary

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)
```

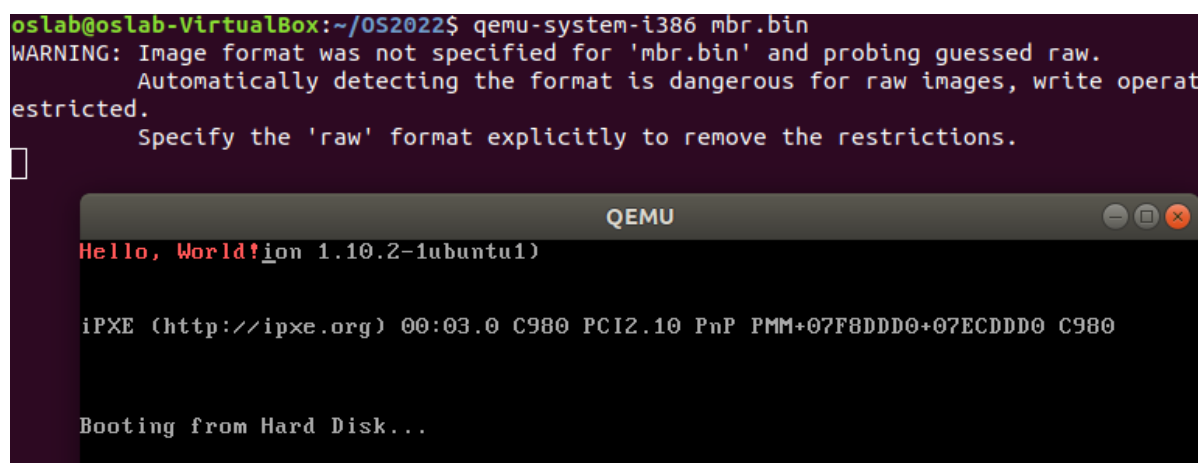


## challenge1:

请尝试使用其他方式，构建自己的MBR，输出“Hello, world!”

选择第一种：还是使用我们提供的汇编语言，通过编写 Python 代码来代替 `genboot.pl` 文件，来生成符合 `mbr` 格式的 `mbr.bin`

```
#!/usr/bin/python
import os
str = input()
size = os.path.getsize(str)
if size < 510:
    file = open(str, mode='ab+')
    for i in range(size, 510):
        file.write(chr(0).encode())
    file.write(chr(0x55).encode())
    # file.write(chr(0xaa).encode())
    # 这样写会多一个c2 即上面括号内为b'\xc2\xaa'
    file.write(b'\xaa')
    file.close()
```



The image shows a terminal window and a QEMU window. The terminal window displays the command `qemu-system-i386 mbr.bin` and a warning message: "WARNING: Image format was not specified for 'mbr.bin' and probing guessed raw. Automatically detecting the format is dangerous for raw images, write operation is restricted. Specify the 'raw' format explicitly to remove the restrictions." The QEMU window shows the output: "Hello, World! (on 1.10.2-1ubuntu1)", "iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980", and "Booting from Hard Disk..."

## task1:

以下任务点是我们在本节需要完成的（代码中已通过TODO注释）

- ☑ 把 `cr0` 的低位设置为1。

```
# TODO: 把cr0的最低位设置为1
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 0,0x9a,0xcf,0

# TODO: 数据段描述符, 对应ds
.word 0xFFFF,0
.byte 0,0x92,0xcf,0

# TODO: 图像段描述符, 对应gs
.word 0xFFFF,0x8000
.byte 0x0b,0x92,0xcf,0
```

✓ 显示helloworld。

直接把app.s复制过来

```
# 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
```

## exercise11:

请回答为什么三个段描述符要按照cs, ds, gs的顺序排列?

```
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
```

ds段选择子为0x10=10000b, gs段选择子为0x18=11000b, 而cs是代码段寄存器, 跟当前指令的地址有关, 在ljmp的时候变为0x08 (第一个操作数) =1000b。后三位都为0, 故前十三位为对全局描述符表的索引, cs,ds,gs的索引值分别为1, 2, 3。

## exercise12:

请回答app.s是怎么利用显存显示helloworld的。

显示设备将内存与0xb8000开头的一部分区域的内存内容相互映射, 屏幕上会根据这部分内容展示信息。

app.s通过在以0xb8000+%edi开头部分存入"Hello, World!\n\0"达到显示的目的。

## task2:

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。

```

void bootMain(void) {
    void * addr = (void *)0x8c00;
    readSect(addr,1);
    __asm__ __volatile__ ( "pushl %eax\n\t"
                           "movl $0x8c00,%eax\n\t"
                           "call %eax\n\t"
                           "popl %eax"
                           );
}

```

### 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。

将文件bootloader.bin和文件app.bin 合并（重定向）成一个文件os.img

### exercise14:

如果把app读到0x7c20，再跳转到这个地方可以吗？为什么？

不可以。因为MBR的内容是从0x7c00的512个字节，调入后会覆盖掉MBR的内容，致使程序无法正常执行。

### 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。

电脑加电后，由硬件设置好其执行的第一条指令，其首先会执行BIOS程序，BIOS程序会执行POST过程，然后将MBR的内容调入到0x7c00，跳转到那里后，关闭保护模式，开启实模式，执行MBR，然后关中断，开启A20地址线，加载GDTR，设置cr0最低位为1，再跳转到保护模式，再加载OS到指定位置并跳转。

## 四、实验的启示/意见和建议

这个操作系统实验比ics手册详细多了，很多地方都有提示和知识点复习，感觉非常好。

有些地方卡了一下，比如填GDT，基本上做起来很顺畅。