



南開大學  
Nankai University

# Operational System Lab Report

## Lab1: Booting a PC

专    业: 智能科学  
姓    名: 柳鹏阳  
学    号: 1511305  
指导老师: 宫晓利

2017 年 10 月 21 日

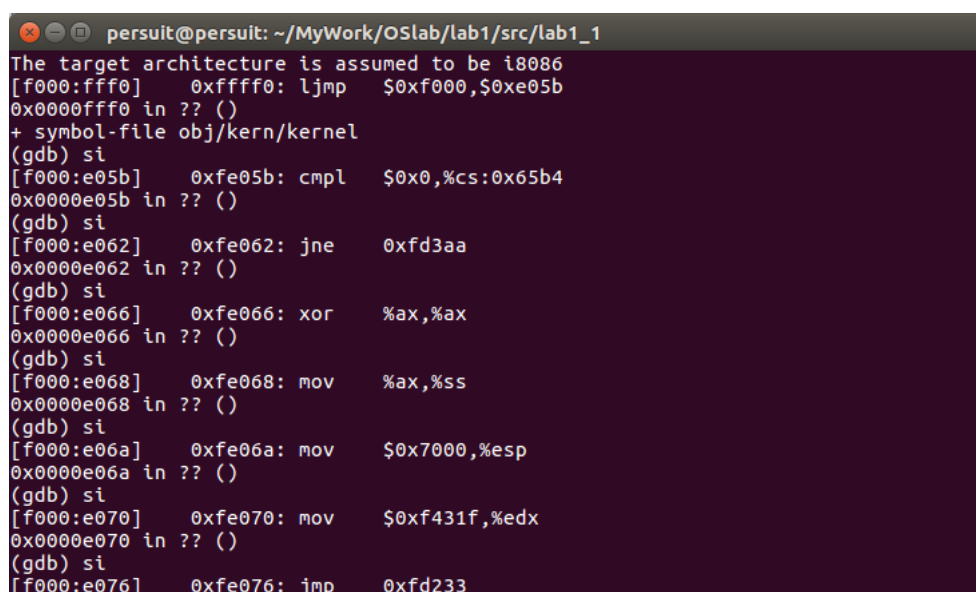
# 目录

<b>第一章</b>	<b>PC Bootstrap</b>	<b>2</b>
1.1	exercise 2 . . . . .	2
<b>第二章</b>	<b>The Boot Loader</b>	<b>4</b>
2.1	exercise 3 . . . . .	4
2.2	exercise 4 . . . . .	5
2.3	exercise 5 . . . . .	6
2.4	exercise 6 . . . . .	6
<b>第三章</b>	<b>Kernel</b>	<b>7</b>
3.1	exercise 7 . . . . .	7
3.2	exercise 8 . . . . .	7
3.3	exercise 9 . . . . .	9
3.4	exercise 11 . . . . .	10
3.5	exercise 12 . . . . .	11

# 第一章 PC Bootstrap

## 1.1 exercise 2

使用 GDB 的 si 指令，我们跟踪了 ROM BIOS 的前八条指令，如图 1.1 所示。接下来我们简单分析以下这八条指令的意义。



```
persuit@persuit: ~/MyWork/OSlab/lab1/src/lab1_1
The target architecture is assumed to be i386
[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) si
[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x65b4
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xfd3aa
0x0000e062 in ?? ()
(gdb) si
[f000:e066] 0xfe066: xor %ax,%ax
0x0000e066 in ?? ()
(gdb) si
[f000:e068] 0xfe068: mov %ax,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a] 0xfe06a: mov $0x7000,%esp
0x0000e06a in ?? ()
(gdb) si
[f000:e070] 0xfe070: mov $0xf431f,%edx
0x0000e070 in ?? ()
(gdb) si
[f000:e076] 0xfe076: jmp 0xfd233
```

图 1.1: ROM BIOS 的前八条指令

1. 0xffff0: ljmp \$0xf000, \$0xe05b

第一条命令是一个跳转指令，跳转到 0xfe05b, 地址推断方法为断寄存器值左移四位，然后和段内地址相加。

2. 0xfe05b: cmpl \$0x0, \$cs:0x6ac8

cmpl 指令: 将立即数 0x0 与 \$cs:0x6ac8 所代表的内存地址的值进行比较。cs 为代码段寄存器

3. 0xfe062: jne 0xfd2e1

jne 指令: 如果 ZF 标志位不为 0 时跳转的 0xfd2e1, 即上一条指令中两者数值不等时跳转。

4. 0xfe066: xor %dx, %dx

xor 指令: 异或位运算, 相同位不同为 1, 相同为 0, 此条指令实际在清零 dx 寄存器

```
5. 0xfe068:  mov  %dx %ss
6. 0xfe06a:  mov  $0x7000, %esp
7. 0xfe070:  mov  $0xf34d2, %edx
8. 0xfe076:  jmp  0xfd15c
```

上述指令同样是在做一些寄存器的赋值以及地址的跳转

通过继续对指令的分析, 我们可以看出正如实验指导上所说的, BIOS 主要是在做一些初始化的, 检测硬件设备, 以及最重要的加载 bootloader 等。

## 第二章 The Boot Loader

### 2.1 exercise 3

按照实验要求，在追踪阅读了 boot/boot.S，并结合 obj/boot/boot.asm 及相关代码后，我们接下来回答以下问题。

#### 1. 处理器什么时候开始执行 32 位代码？如何完成从 16 位到 32 位模式的切换？

答：在运行 `ljmp $PROT_MODE_CSEG, $protcseg` 指令后，开始执行 32bit 代码。在 boot.S 中，PC 首先处于 real mode，以 .code16 即 16bit 运行，在 real mode 使能高于 20 的地址线，完成 real mode 到 protected mode，从而完成 .code16 到 .code32 的转化。可参考 boot.S 中的代码及注释。

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
lgdt    gdt_desc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg

.code32                                # Assemble for 32-bit mode
protcseg:
# Set up the protected-mode data segment registers
movw    $PROT_MODE_DSEG, %ax         # Our data segment selector
movw    %ax, %ds                     # -> DS: Data Segment
NORMAL boot/boot.S                               asm 71% 65
```

图 2.1: boot.S 中由 16bit 跳转到 32bit 的部分代码

#### 2. 引导加载程序 bootloader 执行的最后一个指令是什么？加载的内核第一个指令是什么？

答：bootloader 由 boot/boot.S 和 main.c 组成。追踪到最后一个指令：

```
((void (*)(void)) (ELFHDR->e_entry))();
```

加载的内核的第一个指令在 kern/entry.S:

```
movw    $0x1234, 0x472                # warm boot
```

### 3. 内核的第一个指令在哪里？

答:kern/entry.S

4. 引导加载程序如何决定为了从磁盘获取真个内核读取多少扇区？在哪里可以找到这些俄信息？

答: 在 Program Header Table 中的每一个表项分别对应操作系统的段，内容包括段的大小，段起始地址偏移等信息，由此表我们可以确定内核占用多少扇区。其中表存放在操作系统内核映像的 ELF 头部信息中。

## 2.2 exercise 4

通读 K & R 书中 5.1 (指针和地址) 到 5.5 (字符指针和函数) 的内容。然后下载 pointers.c 的代码，运行它，并确保你了解所有打印值的来源。特别是，请确保你理解第 1 行和第 6 行中指针指向哪里的地址，第 2 行到第 4 行的值是如何被写入的，以及为什么第 5 行中打印的值看起来像是错乱的。

答: 如下图为 pointers.c 的运行结果。我们主要来看一下第 1, 5, 6 行的显示。

首先来看第 1 行的三个地址：

- a: 输出的是数组 a 的首地址，是在程序的栈中分配的。
- b: 输出的是指针 b 所指向的由操作系统在堆中分配的空间的起始地址。
- c: 输出的是未定义的指针变量的值

接下来看第 5 行的显示：

a[1] 看起来乱码的原因是这句命令：c = (int \*) ((char \*) c + 1); 这条语句把 int\* 强制转化为 char\* 并加 1 然后在强制转化为 int\*, 0xbfb4bf89 0xbfb4bf8c, 不再等于 (a+1):0xbfb4bf88 0xbfb4bf8b, 所以会出现类似”乱码”现象。

最后第 6 行的显示参考以上分析。

```
persuit@persuit:~$ cd MyWork/OSlab/DOC/
persuit@persuit:~/MyWork/OSlab/DOC$ gcc -o pointers pointers.c
persuit@persuit:~/MyWork/OSlab/DOC$ ./pointers
1: a = 0x7fffe8f008b0, b = 0x13fc010, c = 0x7465675f6f736476
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7fffe8f008b0, b = 0x7fffe8f008b4, c = 0x7fffe8f008b1
persuit@persuit:~/MyWork/OSlab/DOC$
```

图 2.2: 编译并运行 pointers.c 的结果

## 2.3 exercise 5

跟踪 bootloader 程序的前几个指令，找到开始使用链接地址的第一条指令，即，如果你使用了错误的链接地址，那么执行到这里的时候就必须停下来，否则就会发生错误。然后将 boot/Makefrag 中的链接地址更改为一个错误的地址，运行 make clean，用 make 命令重新编译实验，然后再次跟踪到引导加载程序，看看会发生什么。不要忘了改变链接地址后要再次执行 make clean！

答：有前边所讲，链接地址即通过编译器链接器处理形成的可执行程序中指令的地址，加载地址是可执行文件真正被装入内存后运行的地址。在 bootloader 运行时仍处于实模式，此时链接地址等于加载地址。根据题目要求，我们要改动 bootloader 的链接地址 0xc7c00。重新 make，比较前后的 obj/boot/boot.asm，如下图：可见两者的链接地址不同。

<pre> globl start start: .code16 cli 7c00: fa cld 7c01: fc  # Set up the important data segment registers (DS xorw  %ax,%ax 7c02: 31 c0 </pre>	<pre> .globl start start: .code16 cli 7e00: fa cld 7e01: fc  # Set up the important data segment registers (DS, ES, SS). xorw  %ax,%ax 7e02: 31 c0 </pre>
(a) 更改链接地址前的 boot.asm	(b) 更改链接地址后的 boot.asm

考虑到 BIOS 默认把 bootlader 装入 0x7c00 处，当我们再次设置断点在 0x7c00 时，继续用 si 单步调试，发现在加载全局描述表寄存器 GDTR 时出现如图错误。显然 GDTR 的值不应该为 0。

## 2.4 exercise 6

复位机器（退出 QEMU / GDB 并再次启动）。在 BIOS 进入引导加载程序的那一刻停下来，检查内存中 0x00100000 地址开始的 8 个字的内容，然后再次运行，到 bootloader 进入内核的那一点再停下来，再次打印内存 0x00100000 的内容。为什么这 8 个字的内容会有所不同？第二次停下来时，打印出来的内容是什么？（你不需要使用 QEMU 来回答这个问题，思考即可）

答：在进入 bootloader 前，0x00100000 内容全为 0。在进入内核时，此时 bootloader 已经将内核的各个程序段送入内存地址 0x00100000，所以现在存放的就是内核的某一段的内容。

## 第三章 Kernel

### 3.1 exercise 7

使用 QEMU 和 GDB 跟踪到 JOS 内核并停止在 `movl%eax, %cr0`。查看内存中在地址 `0x00100000` 和 `0xf0100000` 处的内容。下面，使用 GDB 命令 `stepi` 单步执行该指令。指令执行后，再次检查 `0x00100000` 和 `0xf0100000` 的内存。确保你明白刚刚发生的事情。新映射建立后的第一条指令是什么，如果映射配置错误，它还能不能正常工作？注释掉 `kern/entry.S` 中的 `movl%eax, %cr0`，再次追踪到它，看看你的理解是否正确。

答: 由之前的学习得到 `entry` 的入口地址 `0x10000c`, 在运行 `movl%eax,%cr0` 之前, `0x00100000` 和 `0xf0100000` 处的内容如下图, 可见采用分页管理后, `0xf0100000` 处的内容已经映射到 `0x100000`

```
(gdb) x/4x 0x100000
0x100000:      0x1badb002      0x00000000      0xe4524ffe      0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000
x00000000
(gdb) si
=> 0x100025:      mov      %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4x 0x100000
0x100000:      0x1badb002      0x00000000      0xe4524ffe      0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000
```

图 3.1: turn on page 前后 0xf0100000 处的内容对比

### 3.2 exercise 8

阅读 `kern/printf.c`, `lib/printfmt.c` 和 `kern/console.c`, 并确保你了解他们的关系。我们省略了一小段代码, 使用 “%o” 形式的模式打印八进制数字所需的代码。查找并补全此代码片段

`kern/printf.c` 是最高层的调用, 其中 `cprintf` 调用了 `lib/printfmt.c` 中的 `vprintfmt` 子程序, `putch` 调用了 `kern/console.c` 的 `cputchar` 子程序。前者在调整输出格式, 主要是 `int`。而 `console.c` 则主要为之完成 CGA 的单个字符显示, 同时提供 ‘High’-level console I/O 接口。



### 解释一下 console.c 文件中，下面这段代码的含义

```
if (crt_pos >= CRT_SIZE) {
    int i;
    memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
    for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
        crt_buf[i] = 0x0700 | ' ';
    crt_pos -= CRT_COLS;
}
```

在 console.c 的注释中我们知道，采用的是 Text-mode CGA/VGA display output。上网查阅相关资料，80\*25 的 text-mode 就是每页最多可显示 80\*25 个字符，当我们要显示某个字符，我们需要指定显示的字符，位置给 CGA。所以本段代码是在处理当前显示位置超过 CRT\_SIZE 即 80\*25 的情况，memcpy 将 1-79 行的内容复制到 0-78 行，for 循环则是将 79 行置为空格，然后更新 crt\_pos 的位置。

### 跟踪以下代码并单步执行:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

### 回答下列问题:

- 再调用 cprintf() 时，fmt 是什么意思，ap 是什么意思？
- 按照执行的顺序列出所有对 cons\_putc, va\_arg, 和 vprintf 的调用。对于 cons\_putc, 列出它所有的输入参数。对于 va\_arg 列出 ap 在执行完这个函数后的和执行之前的变化。对于 vprintf 列出它的两个输入参数的值。

答:fmt 即指向格式显示字符串的指针。ap 是指向参数列表的字符型指针变量。

调用顺序:

```
—>vprintf("x %d, y %x, z %d\n",[1,2,3])
—>cons\_putc('x')
—>va\_arg():ap=ap+sizeof(int)
—>con\_putc(1)

—>cons\_putc('y')
—>va\_arg():ap=ap+sizeof(int)
—>con\_putc(3)

—>cons\_putc('z')
—>va\_arg():ap=ap+sizeof(int)
—>con\_putc(4)
```

### 运行下面的代码

```
unsigned int i = 0x00646c72;
cprintf("H%x W%s", 57616, &i);
```

输出: **He110, World**, 原因%x 是按照 16 进制输出 57616 显示为 e110.int i 分配四个字节, 按照小端存储, 四个字节分别存储 0x72('r'), 0x6c('l'), 0x64('d'), 0x03('\0'), cprintf 按照 i 的内存地址开始逐字节遍历并显示, 正好输出"world"。

看下面代码, y= 后会出现什么, 为什么?

```
printf("x=%d y=%d", 3);
```

结果为: **x=3 y=-267380452**, 由于 y 并没有参数被指定, 所以会输出一个不确定的值。

假设 GCC 更改了它的调用约定, 以声明的顺序将参数压入栈中, 这样会使最后一个参数最后被压入。你将如何更改 cprintf 或其接口, 以便仍然可以传递一个可变数量的参数? 第一个想法是修改 va\_list 变量, 增加一个指向最后一个变量的指针。va\_arg 实现由栈顶到栈底的访问。

挑战增强控制台的能力以允许以打印不同颜色的文本。传统的方法是使它解析嵌入在待打印的文本字符串中的 ANSI 转义序列, 但是你可以使用任何你喜欢的机制。在实验的参考内容和网络上其他地方有对 VGA 显示硬件进行编程的大量信息。如果你真的喜欢挑战, 也可以尝试将 VGA 硬件切换到图形模式, 并使控制台将文本绘制到图形帧缓冲区上

### 3.3 exercise 9

确定内核在哪里完成了栈的初始化, 以及栈所在内存的确切位置。内核如何为栈保留空间? 栈指针初始化时指向的是保留区域的“哪一端”

答: 在 entry.S 中我们可以看到执行了 movl \$0x0,%ebp 和 movl \$(bootstacktop),%esp 时已经处于虚拟地址。通过反汇编我们得到 bootstacktop 值为 0xf0110000, 在定义 bootstacktop 之前, 分配了 KSTKSIZE=8\*PGSIZE=8\*4kB=32kB, 所以 bootstack 分配的空间为 0xf0108000-0xf0110000, 对应物理内存为 0x108000-0x110000。

内核如何给堆栈保留内存空间

如上分析, 在 entry.S 的数据段声明了大小为 32kB 的空间作为堆栈使用, 从而为内核保留一块空间。

堆栈指针又是指向这块被保留的区域的哪一端的呢?

因为堆栈是向下的, 堆栈指针指向最高地址即 bootstacktop。

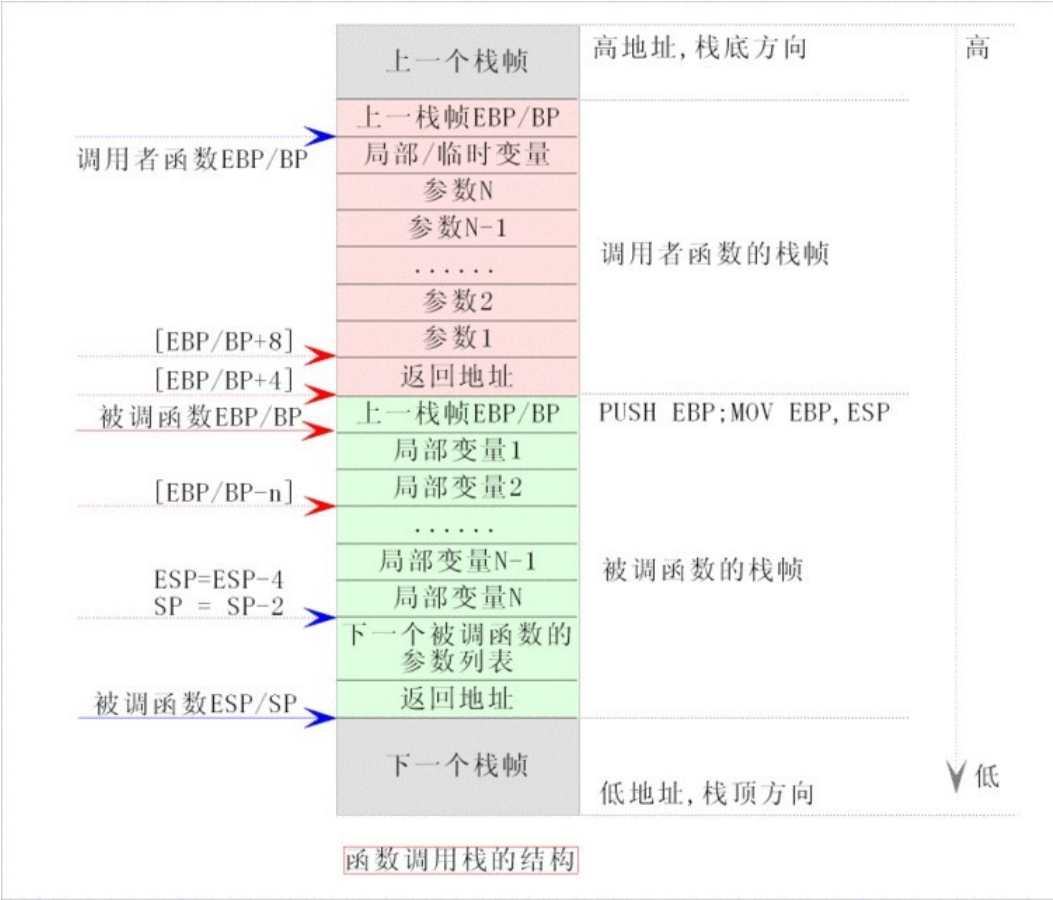
要熟悉 x86 上 C 语言函数的调用约定, 请在 obj/kern/kernel.asm 中找到 test\_backtrace 函数的地址, 在其中设置一个断点, 并检查在内核启动后每次这个函数被调用时会发生什么。每一级的 test\_backtrace 在递归调用时, 会在栈上压入多少个 32 位的字, 这些字的内容是什么?

会压入 4 个 32 位的字包括: 返回地址, 调用者的 ebp, 寄存器 ebx, 传递参数

3.4 exercise 11

实现如上所述的回溯功能。请使用与示例中相同的格式，否则打分脚本将会出错。当你认为你的工作正确的时候，运行 `make grade` 来看看它的输出是否符合我们的打分脚本所期待的，如果没有，修正发现的错误。在你成功提交实验 1 的作业后，欢迎你以任何你喜欢的方式更改回溯功能的输出格式。

由 exercise 10 的分析可以得知函数在调用子函数对栈的操作  
结合反汇编代码: 易得本题 code:



```
f0100040: 55                push    %ebp
f0100041: 89 e5             mov     %esp,%ebp
f0100043: 53                push    %ebx
f0100044: 83 ec 0c          sub     $0xc,%esp
f0100047: 8b 5d 08           mov     0x8(%ebp),%ebx
```

图 3.2: 调用 testbaktrace 子函数的部分反汇编代码

```
uint32_t *ebp=(uint32_t*)read_ebp();
uint32_t eip=ebp[1];
cprintf("Stack backtrace:\n");
while(ebp){
```

```
    cprintf(" ebp %08x eip %08x args ",ebp,eip);
    int i=2;
    for (i;i<7;++i){
        cprintf("%08x ",ebp[i]);
    }
    ebp=(uint32_t*) ebp[0];
    eip=ebp[1];
}
cprintf("\n");
```

### 3.5 exercise 12

修改你的堆栈回溯功能，为每个 eip 显示与该 eip 对应的函数名称，源文件名和行号  
这个问题没看懂，代码是 copy 网上的。