

# Lab 1: Booting a PC——Part1&2

罗宸、朱勋建、朱志成、赵审铎、罗思唯、吴昆默

2019 年 11 月 10 日

## 摘要

本次实验取自MIT 6.828 Lab1的前两个部分。第一部分主要熟悉x86汇编语言、QEMU的x86 模拟器和PC开机的引导过程。第二部分主要检查内核的引导加载程序。

关键词: `qemu`、`boot loader`、`JOS kernel`

## 1 实验环境与工具

本实验使用Ubuntu 18.04系统与qemu 6.828进行。

## 2 练习一 GNU assembler

MASM汇编微软定义的汇编语言。AT&T汇编是GNU的开发者定义的汇编语言。GNU汇编程序是GNU操作系统的默认汇编程序。它处理多个架构并支持多种汇编语言语法。主要用于汇编GNU c编译器的输出以供链接器使用，因此它可以被视为tigcc包的内部部分。然而，它可能被称为一个独立的程序，GNU汇编试图使as-assemble能够正确地组装同一台机器的其他汇编程序。任何异常都有明确的记录。GNU assambler可执行文件称为as，是UNIX汇编程序的基本名称。

## 3 问题一 Boot Loader

### 3.1

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

阅读boot.asm中的代码与注释，`".code16"`段为实模式，`".code32"`段为保护模式，由此可知，处理器在命令`ljmp $PROT_MODE_CSEG, $protcseg`之后开始执行32位代码。

这一切换过程是通过如下命令实现的

```

1  # Switch from real to protected mode, using a bootstrap GDT
2  # and segment translation that makes virtual addresses
3  # identical to their physical addresses, so that the
4  # effective memory map does not change during the switch.
5  lgdt    gdt_desc
6      7c1e:    0f 01 16                lgdtl    (%esi)
7      7c21:    64 7c 0f                fs  jl    7c33 <protcseg+0x1>
8  movl    %cr0, %eax
9      7c24:    20 c0                and     %al,%al
10  orl     $CR0_PE_ON, %eax
11      7c26:    66 83 c8 01            or      $0x1,%ax
12  movl    %eax, %cr0
13      7c2a:    0f 22 c0                mov     %eax,%cr0

```

其中加载了全局描述符表，然后将cr0中的PE位置1，从而实现从实模式到保护模式的转换。

## 3.2

引导加载程序boot loader 执行的最后一个指令是什么，加载的内核的第一个指令是什么？

boot loader执行的最后一条指令为`call *0x10018`，调用了ELF的头部，实现跳转到kernel。

内核执行的第一条指令从kernel.asm中找，如下

```

1  .globl entry
2  entry:
3      movw $0x1234,0x472                # warm boot

```

## 3.3

内核执行的第一条指令在哪？

调用`objdump -f obj/kern/kernel`，得到

```

1  $ objdump -f obj/kern/kernel
2

```

```

3 obj/kern/kernel:      file format elf32-i386
4 architecture: i386, flags 0x00000112:
5 EXEC_P, HAS_SYMS, D_PAGED
6 start address 0x0010000c
7 .....

```

由此可知内核的第一条指令的地址为0x0010000c。

### 3.4

boot loader如何决定为了从磁盘获取整个内核必须读取多少扇区？在哪里可以找到这些信息？

boot loader会从硬盘中读入ELF File Header，对应代码在boot/main.c的bootmain函数中：

```

1 // read 1st page off disk
2 readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
3
4 // is this a valid ELF?
5 if (ELFHDR->e_magic != ELF_MAGIC)
6     goto bad;
7
8 // load each program segment (ignores ph flags)
9 ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
10 eph = ph + ELFHDR->e_phnum;
11 for (; ph < eph; ph++)
12     // p_pa is the load address of this segment (as well
13     // as the physical address)
14     readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
15
16 // call the entry point from the ELF header
17 // note: does not return!
18 ((void (*)(void)) (ELFHDR->e_entry))();

```

从硬盘中读取ELFHDR，通过设定好的魔法数字来检验ELF头的有效性。

可以看到，由ELFHDR的地址+程序头表的文件偏移e\_phoff能得到开始其中保存的起始程序头的地址ph，eph = ph + ELF Header中总的程序头个数e\_phnum为结束地址。

利用ph和eph可遍历每一个程序头，并依次从中读取出kernel的内容。

## 4 问题二 Kernel

### 4.1

printf.c对console.c的接口是cprintf(const char \*fmt)，在console.c中需要输出一个字符串时直接把字符串作参数（可以以print格式添加变量）调用即可，返回输出字符串的长度。但事实上printf.c中仅提供一个封装的接口，在cprintf(const char \*fmt)被调用后，会调用定义在printfmt.c中的vprintfmt(void (\*putch)(int, void\*), void \*putdat, const char \*fmt, va\_list ap)进行逐字分析，以决定是处理控制符还是输出常量部分。

最后打印工作交由定义在console.c中的cputchar(int c)处理，cputchar(int c)会调用serial\_init(void)，lpt\_putc(int c)，cga\_init(void)三个函数来完成具体的打印工作。

### 4.2

```
1 if (crt_pos >= CRT_SIZE) {
2     int i;
3     memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *
4             sizeof(uint16_t));
5     for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
6         crt_buf[i] = 0x0700 | ' ';
7     crt_pos -= CRT_COLS;
8 }
```

这段代码的目的是缓冲区满时清除一部分留出空间，可以理解成一页写满时自动下拉一行。比如设一页有25行，每行可放80个字符，则CRT\_SIZE=25\*80=2000，CRT\_COLS=80。当检测到crt\_pos>CRT\_SIZE,即光标位置超出屏幕外时，将缓冲区(crt\_buf)中后24行复制到前24行，最后一行以' '填充。最后将光标位置上移一行。

## 5 作业一 printf()

补全输出"%o"格式字符串的代码。

首先分析kern/printf.c，lib/printfmt.c和kern/console.c三者的关系。由代码上方的注释可知kern/printf.c中的vcprintf，cprintf函数都调用了lib/printfmt.c中的vprintfmt函数：

```

1 int
2 vfprintf(const char *fmt, va_list ap)
3 {
4     int cnt = 0;
5     vprintfmt((void*)putch, &cnt, fmt, ap);
6     return cnt;
7 }

```

经查阅资料后可知它的四个输入参数中(void\*)putch(int, void\*)为函数指针，一般调用输出到屏幕上的函数。void \*putdat是输入字符要放的内存地址指针。const char \*fmt 是格式化字符串。va\_list ap为多个输入参数。

kern/printf.c中的putch函数调用kern/console.c的cputchar函数。lib/printfmt.c中也有putch函数。所以kern/printf.c 和lib/printfmt.c 依赖于kern/console.c。

之后去分析kern/console.c。

kern/console.c中的cputchar函数调用了cons\_putc函数：

```

1 // output a character to the console
2 static void
3 cons_putc(int c)
4 {
5     serial_putc(c);
6     lpt_putc(c);
7     cga_putc(c);
8 }

```

cons\_putc的功能是输出一个字符到控制台。由serial\_putc, lpt\_putc和cga\_putc这三个函数组成。

先分析serial\_putc函数：

```

1 static void
2 serial_putc(int c)
3 {
4     int i;
5     for (i = 0;
6         !(inb(COM1 + COMLSR) & COMLSR_TXRDY) && i < 12800;
7         i++)

```

```

8         delay ();
9         outb(COM1 + COM_TX, c);
10    }

```

它控制的是端口0x3F8，inb内联汇编函数读取的是COM1 + COM\_LSR = 0x3FD端口，outb内联汇编函数输出到了COM1 + COM\_TX = 0x3F8端口。在inb(COM1 + COM\_LSR) 之后，有& COM\_LSR\_TXRDY 这个操作。!(inb(COM1 + COM\_LSR) & COM\_LSR\_TXRDY)是为了查看读入的数据的第6位，也就PORTS.LST中03FD中提到的bit 5是否为1。如果为1，上面的语句结果就是0，停止for循环。这个bit 5是判断发送数据缓冲寄存器是否为空。outb 是将端口0x3F8 的内容输出到c。当0x3F8被写入数据，它作为发送数据缓冲寄存器，数据是要发给串口。所以serial\_putc是为了把一个字符输出到串口。

再分析lpt\_putc函数：

```

1  static void
2  lpt_putc(int c)
3  {
4      int i;
5      for (i = 0; !(inb(0x378+1) & 0x80) && i < 12800; i++)
6          delay ();
7      outb(0x378+0, c);
8      outb(0x378+2, 0x08|0x04|0x01);
9      outb(0x378+2, 0x08);
10 }

```

它的作用是将字符给并口设备。

最后分析cga\_putc函数：

```

1  static void
2  cga_putc(int c)
3  {
4      // if no attribute given, then use black on white
5      if (!(c & ~0xFF))
6          c |= 0x0700;
7      switch (c & 0xff) {
8      case '\b':
9          if (crt_pos > 0) {

```

```

10         crt_pos--;
11         crt_buf[crt_pos] = (c & ~0xff) | ' ';
12     }
13     break;
14 case '\n':
15     crt_pos += CRT_COLS;
16     /* fallthru */
17 case '\r':
18     crt_pos -= (crt_pos % CRT_COLS);
19     break;
20 case '\t':
21     cons_putc(' ');
22     cons_putc(' ');
23     cons_putc(' ');
24     cons_putc(' ');
25     cons_putc(' ');
26     break;
27 default:
28     crt_buf[crt_pos++] = c;    /* write the character */
29     break;
30 }
31 // What is the purpose of this?
32 if (crt_pos >= CRT_SIZE) {
33     int i;
34
35     memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE -
36         CRT_COLS) * sizeof(uint16_t));
37     for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
38         crt_buf[i] = 0x0700 | ' ';
39     crt_pos -= CRT_COLS;
40 }
41 /* move that little blinky thing */
42 outb(addr_6845, 14);
43 outb(addr_6845 + 1, crt_pos >> 8);
44 outb(addr_6845, 15);
45 outb(addr_6845 + 1, crt_pos);
46 }

```

其中!(c & ~ 0xFF) 用来检测是否在0 255 之间。\\ b是退格键，让缓冲区crt\_buf 的下标crt\_pos 减1。其他的同理，case都是格式操作。default是往缓冲区里写入字符c。当缓存超过CRT\_SIZE，就用memmove复制内存内容。最后四句代码是将缓冲区的内容输出到显示屏。

最后是去实现"%o"的格式化输出，在lib/printfmt.c中可以看到要填写的地方：

```
1 // (unsigned) octal
2 case 'o':
3     // Replace this with your code.
4     putch( 'X', putdat);
5     putch( 'X', putdat);
6     putch( 'X', putdat);
7     break;
```

参考上面case 'u' 中的写法，可以得出：

```
1 case 'u':
2     num = getuint(&ap, lflag);
3     base = 8;
4     goto number;
```

修改完以后保存，make clean之后运行，会发现启动以后，qemu里JOS启动时会出现：



```
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

图 5.1: `printf("%o")`

可见第一行完成了十进制数6828转八进制。

## 6 作业二

问题二要求实现`mon_backtrace()`函数，显示`ebp`，`eip`和`arg`信息。

### 6.1 涉及属性

`eip`（返回指令指针）：存储当前执行指令的下一条指令在内存中的偏移地址。

`ebp`（基址指针）：存储指向当前函数需要使用的参数的指针。

`esp`（栈指针）：存储指向栈顶的指针。

在程序中，如果需要调用一个函数，首先会将函数需要的参数进栈，然后将`eip`中的内容进栈，也就是下一条指令在内存中的位置，这样在函数调用结束后便可以通过堆栈中的`eip`值。返回调用函数的程序，如下图所示

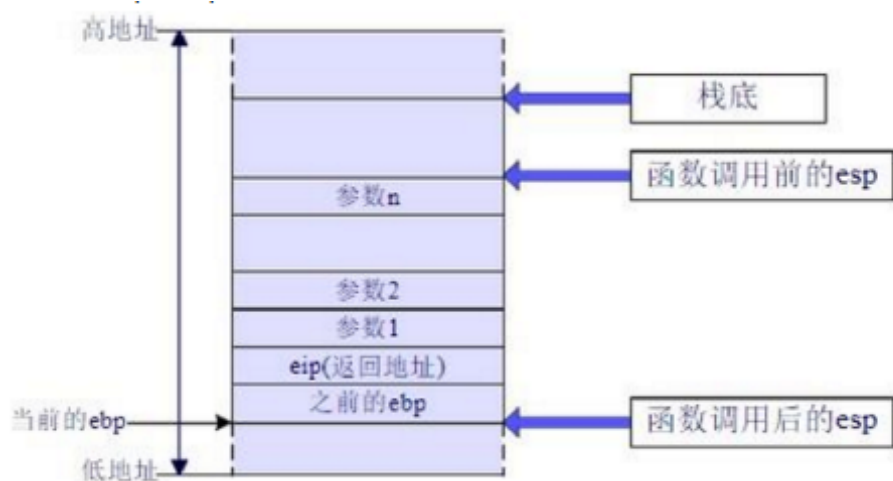


图 6.1: 栈

## 6.2 涉及函数

mon\_backtrace函数的原型在kern/monitor.c中

```

1 int
2 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
3 {
4     return 0;
5 }

```

调用mon\_backtrace的函数test\_backtrace在kern/init.c中

```

1 // Test the stack backtrace function (lab 1 only)
2 void
3 test_backtrace(int x)
4 {
5     cprintf("entering test_backtrace %d\n", x);
6     if (x > 0)
7         test_backtrace(x-1);
8     else
9         mon_backtrace(0, 0, 0);
10    cprintf("leaving test_backtrace %d\n", x);
11 }

```

kern/entry.s中提供的停止信息如下

```

1 relocated:
2 # Clear the frame pointer register (EBP)
3 # so that once we get into debugging C code,
4 # stack backtraces will be terminated properly.
5 movl    $0x0,%ebp      # nuke frame pointer

```

即，进入内核监控后，stack tracers会被中止，mon\_backtrace和test\_backtrace的作用域失效，可确定当ebp值为0时停止题目要求信息的打印。

read\_edp在inc/x86.h中

```

1 static __inline uint32_t
2 read_edp(void)
3 {
4     uint32_t ebp;
5     __asm __volatile ("movl %%ebp,%0" : "=r" (ebp));
6     return ebp;
7 }

```

则read\_edp的返回值类型为uint32\_t，对应可确定edp变量类型。

### 6.3 代码编写及执行情况

由上述描述，用格式符"%08x"进行8位16进制的格式控制,编写代码如下：

```

1 int
2 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
3 {
4     int i;
5     cprintf("Stack backtrace:\n");
6     uint32_t ebp = read_edp();
7     while ((int)ebp != 0)
8     {
9         cprintf("  ebp:0x%08x eip:0x%08x args:%08x %08x %08x %08x\n",
10                ebp,
11                *((uint32_t *)ebp+1),*((uint32_t *)ebp+2),*((uint32_t *)ebp+3),

```

```

11         *((uint32_t *)ebp+4),*((uint32_t *)ebp+5),*((uint32_t *)
           ebp+6));
12     cprintf("\n");
13     ebp = ((uint32_t *)ebp)[0];
14 }
15 return 0;
16 }

```

程序执行情况如下:

```

Stack backtrace:
  ebp:0xf010ff18 eip:0xf0100087 args:00000000 00000000 00000000 00000000 f010094
1  ebp:0xf010ff38 eip:0xf0100069 args:00000000 00000001 f010ff78 00000000 f010094
1  ebp:0xf010ff58 eip:0xf0100069 args:00000001 00000002 f010ff98 00000000 f010094
1  ebp:0xf010ff78 eip:0xf0100069 args:00000002 00000003 f010ffb8 00000000 f010094
1  ebp:0xf010ff98 eip:0xf0100069 args:00000003 00000004 00000000 00000000 00000000
0  ebp:0xf010ffb8 eip:0xf0100069 args:00000004 00000005 00000000 00010094 00010094
4  ebp:0xf010ffd8 eip:0xf01000ea args:00000005 00001aac 00000644 00000000 00000000
0  ebp:0xf010fff8 eip:0xf010003e args:00111021 00000000 00000000 00000000 00000000
0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.

```

图 6.2: 作业二结果

## 7 挑战作业

为每个eip显示源文件名函数名和行号。

为了输出eip的调试信息, 根据题中信息, 首先去kern/kdebug.h中查看定义:

```

1 // Debug information about a particular instruction pointer
2 struct Eipdebuginfo {
3     const char *eip_file;    // Source code filename for EIP
4     int eip_line;           // Source code linenumber for EIP
5     const char *eip_fn_name; // Name of function containing EIP
6                             // -Note: not null terminated!
7     int eip_fn_namelen;     // Length of function name
8     uintptr_t eip_fn_addr;  // Address of start of function

```

```

9      int eip_fn_narg;           // Number of function arguments
10 };
11
12 int debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info);

```

然后根据注释查看inc/stab.h中stab的定义:

```

1 // Entries in the STABS table are formatted as follows.
2 struct Stab {
3     uint32_t n_strx;           // index into string table of name
4     uint8_t n_type;           // type of symbol
5     uint8_t n_other;          // misc info (usually empty)
6     uint16_t n_desc;          // description field
7     uintptr_t n_value;        // value of symbol
8 };

```

根据函数debuginfo\_eip注释中的提示, 添加函数stab\_binsearch的调用以搜索行号并完成设定:

```

1 stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
2 if(lline <= rline) {
3     info->eip_line = stabs[rline].n_desc;
4 } else {
5     info->eip_line = -1;
6 }

```

进一步修改mon\_backtrace, 通过debuginfo\_eip获取相关信息:

```

1 int
2 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
3 {
4     // Your code here.
5     int j;
6     struct Eipdebuginfo eipinfo;
7     uint32_t ebp = read_ebp();
8     uint32_t eip = *((uint32_t *)ebp+1);
9     debuginfo_eip(eip, &eipinfo);

```

```

10  cprintf("Stack backtrace:\n");
11  while ((int)ebp != 0)
12  {
13      cprintf("  ebp:0x%08x eip:0x%08x args:", ebp, eip);
14      uint32_t *args = (uint32_t *)ebp + 2;
15      for (j = 0; j < 5; j++) {
16          cprintf("%08x ", args[j]);
17      }
18      cprintf("\n");
19      eip = ((uint32_t *)ebp)[1];
20      ebp = ((uint32_t *)ebp)[0];
21      cprintf("          %s:%d: %.*s+%d\n",
22              eipinfo.eip_file, eipinfo.eip_line, eipinfo.
23                  eip_fn_namelen,
24                  eipinfo.eip_fn_name, eip - eipinfo.eip_fn_addr);
25  }
26  return 0;

```

```

选择Ubuntu 18.04
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
  ebp:0xf010ff18 eip:0xf010006a args:00000000 00000000 00000000 00000000 f010094b
    kern/init.c:18: test_backtrace+42
  ebp:0xf010ff38 eip:0xf010006a args:00000000 00000001 f010ff78 00000000 f010094b
    kern/init.c:18: test_backtrace+79
  ebp:0xf010ff58 eip:0xf010008f args:00000001 00000002 f010ff98 00000000 f010094b
    kern/init.c:18: test_backtrace+79
  ebp:0xf010ff78 eip:0xf010008f args:00000002 00000003 f010ffb8 00000000 f010094b
    kern/init.c:18: test_backtrace+79
  ebp:0xf010ff98 eip:0xf010008f args:00000003 00000004 00000000 00000000 00000000
    kern/init.c:18: test_backtrace+79
  ebp:0xf010ffb8 eip:0xf010008f args:00000004 00000005 00000000 00010094 00010094
    kern/init.c:18: test_backtrace+79
  ebp:0xf010ffd8 eip:0xf010008f args:00000005 00001aac 00000644 00000000 00000000
    kern/init.c:18: test_backtrace+148
  ebp:0xf010fff8 eip:0xf01000d4 args:00111021 00000000 00000000 00000000 00000000
    kern/init.c:18: test_backtrace+-2
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
k>

```

图 7.1: 挑战作业实验结果