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 10 月 21 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 成员与分工

小组共有6名成员，具体分工如下表2.1

姓名	学号	分工	贡献比列
罗宸	1711364	挑战作业、文档修正与排版	25%
朱勋建	1711418	作业二代码与文档	20%
赵审铎	1711414	问题二文档编写	18%
朱志成	1711419	作业一代码与文档	16%
罗思唯	1711275	参与问题一文档编写	13%
吴昆默	1711279	练习一文档编写	8%

表 2.1: 小组成员信息与贡献

3 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。

4 问题一 Boot Loader

4.1

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    gdtdesc
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。

4.2

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$call \quad *0x10018$，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

```
.globl entry
entry:
    movw $0x1234,0x472              # warm boot
```

4.3

内核执行的第一条指令在哪？

调用objdump -f obj/kern/kernel，得到

```
$ objdump -f obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
.........
```

由此可知内核的第一条指令的地址为0x0010000c。

4.4

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：

```
// read 1st page off disk
readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

// is this a valid ELF?
if (ELFHDR->e_magic != ELF_MAGIC)
    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。

5 问题二 Kernel

5.1

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。

5.2

```
1 if (crt_pos >= CRT_SIZE) {
2     int i;
3     memmove(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;
}
```

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_CLOS=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。

6 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：

```
int
vcprintf(const char *fmt, va_list ap)
{
    int cnt = 0;
    vprintfmt((void*)putch, &cnt, fmt, ap);
    return cnt;
}
```

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：

```
// output a character to the console
static void
cons_putc(int c)
{
```

```
    serial_putc(c);
    lpt_putc(c);
    cga_putc(c);
}
```

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：

```
static void
serial_putc(int c)
{
    int i;
    for (i = 0;
         !(inb(COM1 + COM_LSR) & COM_LSR_TXRDY) && i < 12800;
         i++)
        delay();
    outb(COM1 + COM_TX, c);
}
```

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：

```
static void
lpt_putc(int c)
{
    int i;
    for (i = 0; !(inb(0x378+1) & 0x80) && i < 12800; i++)
        delay();
    outb(0x378+0, c);
```

```
    outb(0x378+2, 0x08|0x04|0x01);
    outb(0x378+2, 0x08);
}
```

它的作用是将字符给并口设备。

最后分析cga_putc函数：

```
static void
cga_putc(int c)
{
    // if no attribute given, then use black on white
    if (!(c & ~0xFF))
        c |= 0x0700;
    switch (c & 0xff) {
    case '\b':
        if (crt_pos > 0) {
            crt_pos--;
            crt_buf[crt_pos] = (c & ~0xff) | ' ';
        }
        break;
    case '\n':
        crt_pos += CRT_COLS;
        /* fallthru */
    case '\r':
        crt_pos -= (crt_pos % CRT_COLS);
        break;
    case '\t':
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        break;
    default:
        crt_buf[crt_pos++] = c;      /* write the character */
        break;
    }
```

```
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 -
                CRT_COLS) * sizeof(uint16_t));
36          for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
37              crt_buf[i] = 0x0700 | ' ';
38          crt_pos -= CRT_COLS;
39      }
40      /* move that little blinky thing */
41      outb(addr_6845, 14);
42      outb(addr_6845 + 1, crt_pos >> 8);
43      outb(addr_6845, 15);
44      outb(addr_6845 + 1, crt_pos);
45  }
```

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 'o':
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：

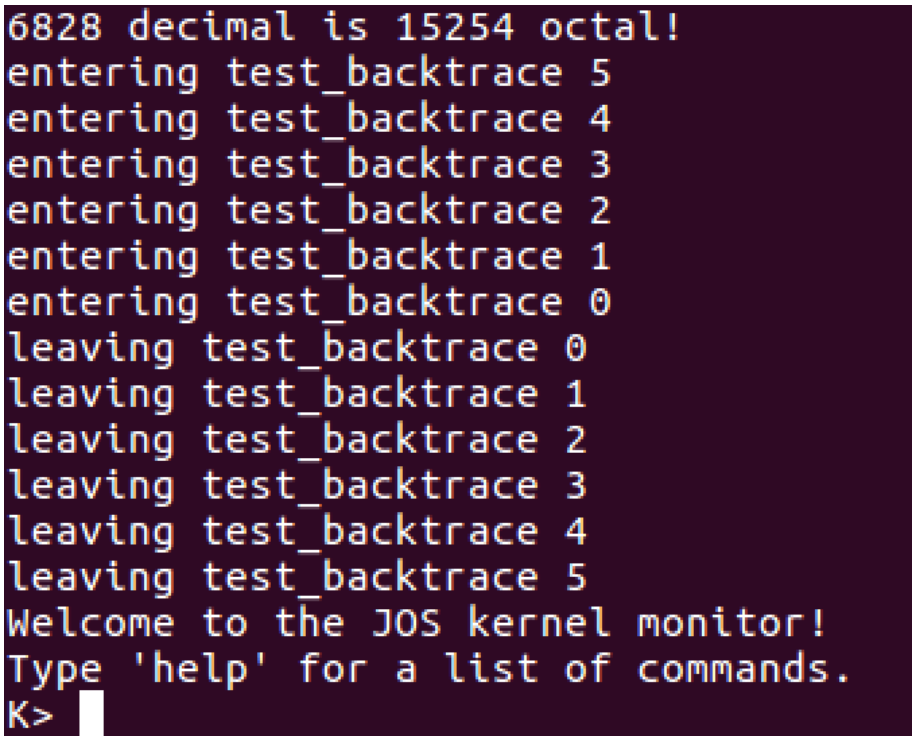

图 6.1: printf("%o")

可见第一行完成了十进制数6828转八进制。

7 作业二

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

7.1 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

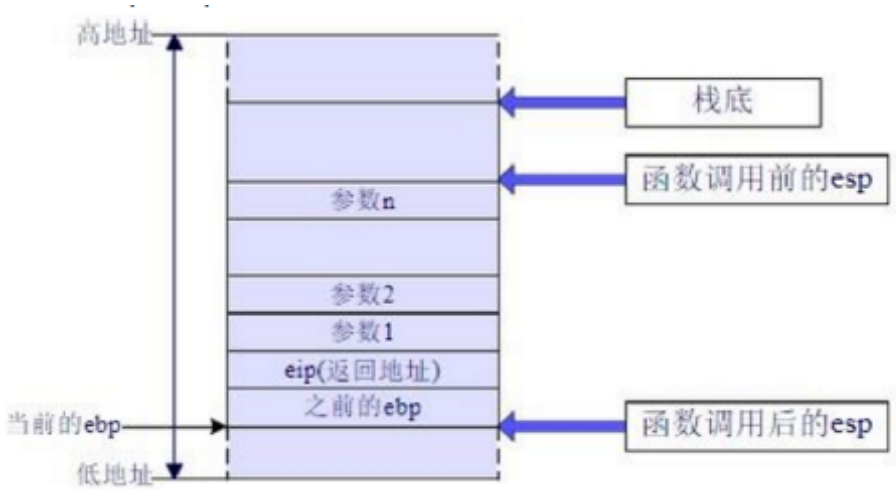

图 7.1: 栈

7.2 涉及函数

mon_backtrace函数的原型在kern/monitor.c中

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    return 0;
}
```

调用mon_backtrace的函数test_backtrace在kern/init.c中

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
    cprintf("entering test_backtrace %d\n", x);
    if (x > 0)
        test_backtrace(x-1);
    else
        mon_backtrace(0, 0, 0);
    cprintf("leaving test_backtrace %d\n", x);
}
```

kern/entry.s中提供的停止信息如下

```
relocated:
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
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

```
static __inline uint32_t
read_ebp(void)
{
    uint32_t ebp;
    __asm __volatile("movl %%ebp,%0" : "=r" (ebp));
    return ebp;
}
```

则read_edp的返回值类型为uint32_t，对应可确定edp变量类型。

7.3 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：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    int i;
    cprintf("Stack backtrace:\n");
    uint32_t ebp = read_ebp();
    while ((int)ebp != 0)
    {
        cprintf("  ebp:0x%08x eip:0x%08x args:%08x %08x %08x %08
            x %08x", ebp,
        *((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 0000000
0
  ebp:0xf010ffb8 eip:0xf0100069 args:00000004 00000005 00000000 00010094 0001009
4
  ebp:0xf010ffd8 eip:0xf01000ea args:00000005 00001aac 00000644 00000000 0000000
0
  ebp:0xf010fff8 eip:0xf010003e args:00111021 00000000 00000000 00000000 0000000
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 7.2: 作业二结果

8 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
```

```
    int eip_fn_narg;        // Number of function arguments
};

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：

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

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：

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

进一步修改mon_backtrace，通过debuginfo_eip获取相关信息：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    int j;
    struct Eipdebuginfo eipinfo;
    uint32_t ebp = read_ebp();
    uint32_t eip = *((uint32_t *)ebp+1);
    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.
                eip_fn_namelen,
23          eipinfo.eip_fn_name, eip - eipinfo.eip_fn_addr);
24      }
25      return 0;
26  }
```

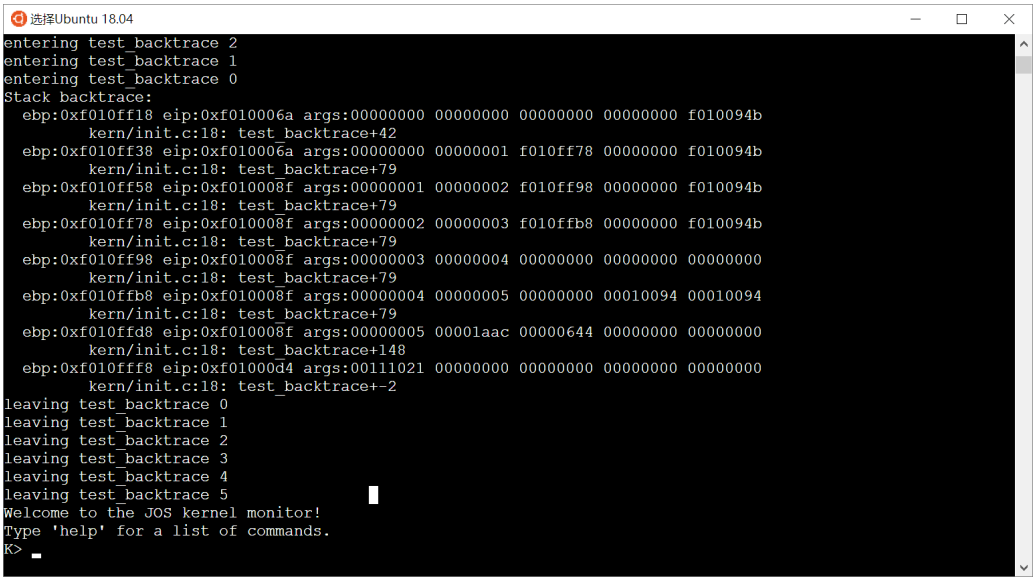

图 8.1: 挑战作业实验结果