Lab 1: Booting a PC———Part1&2

December 24, 2019

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 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。

2 问题一 Boot Loader

2.1

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

```
# 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    gdtdesc
  7c1e:   0f 01 16                lgdtl  (%esi)
  7c21:   64 7c 0f                fs jl  7c33 <protcseg+0x1>
movl    %cr0, %eax
  7c24:   20 c0                   and    %al,%al
orl     $CR0_PE_ON, %eax
  7c26:   66 83 c8 01             or     $0x1,%ax
movl    %eax, %cr0
  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。

2.2

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

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

2.3

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。

2.4

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;

// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++)
    // p_pa is the load address of this segment (as well
    // as the physical address)
    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。

3 问题二 Kernel

3.1

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。

3.2

```
1 if (crt_pos >= CRT_SIZE) {
2     int i;
3     memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS)
          * sizeof(uint16_t));
```

```
4       for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5           crt_buf[i] = 0x0700 | ' ';
6       crt_pos -= CRT_COLS;
7   }
```

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。

4 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   vcprintf(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：

```
// 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;
```

```
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 -
                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：

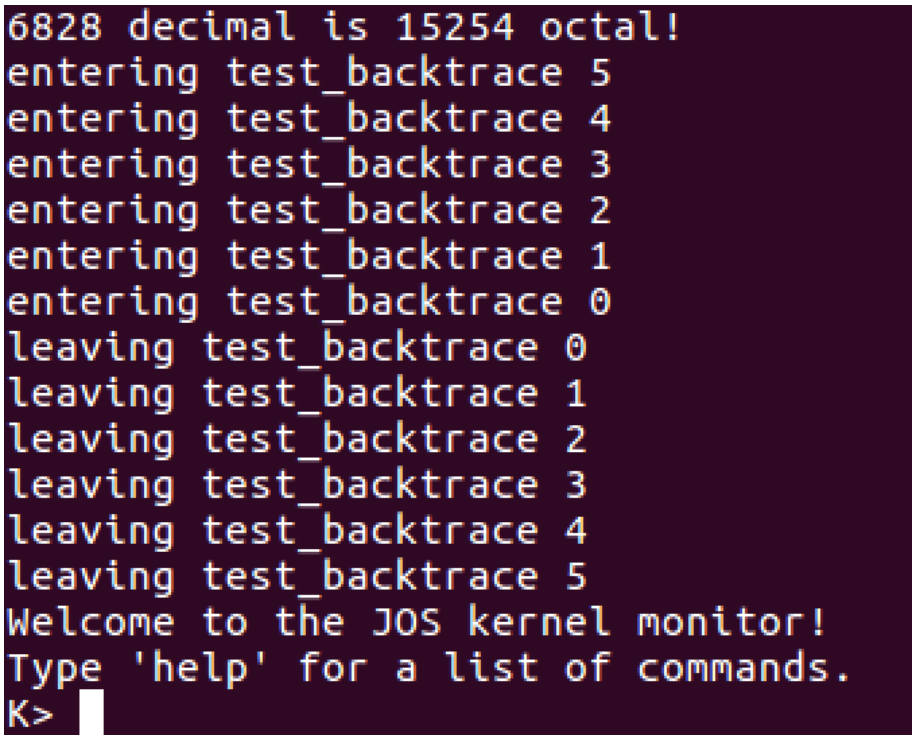

图 4.1: printf("%o")

可见第一行完成了十进制数 6828 转八进制。

5 作业二

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

5.1 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

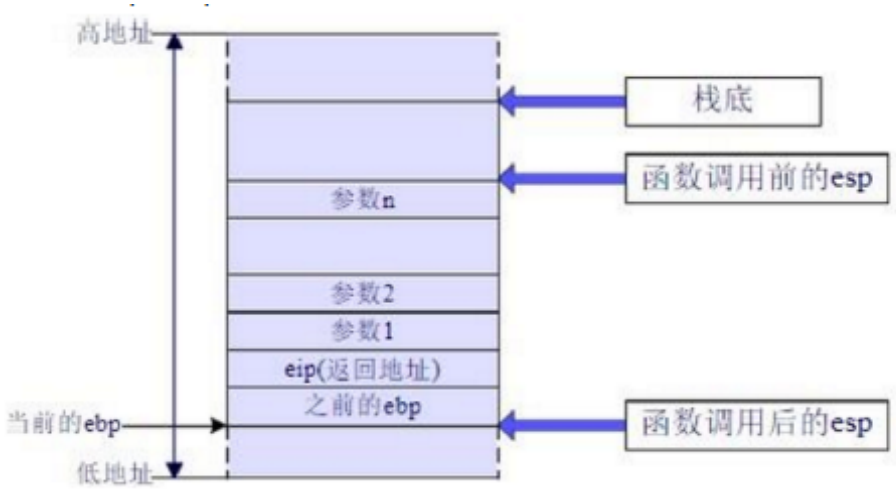

图 5.1: 栈

5.2 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

```
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。

5.3 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;
```

```
5       cprintf("Stack backtrace:\n");
6       uint32_t ebp = read_ebp();
7       while ((int)ebp != 0)
8       {
9           cprintf("  ebp:0x%08x eip:0x%08x args:%08x %08x %08x
                %08x %08x", ebp,
10          *((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 5.2: 作业二结果

6 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：

```
// Debug information about a particular instruction pointer
struct Eipdebuginfo {
    const char *eip_file;   // Source code filename for EIP
    int eip_line;           // Source code linenumber for EIP
    const char *eip_fn_name;// Name of function containing EIP
                            // -Note: not null terminated!
    int eip_fn_namelen;     // Length of function name
    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);
    cprintf("Stack backtrace:\n");
    while ((int)ebp != 0)
    {
        cprintf("  ebp:0x%08x eip:0x%08x args:", ebp, eip);
        uint32_t *args = (uint32_t *)ebp + 2;
        for (j = 0; j < 5; j ++) {
            cprintf("%08x ", args[j]);
        }
        cprintf("\n");
        eip = ((uint32_t *)ebp)[1];
        ebp = ((uint32_t *)ebp)[0];
        cprintf("         %s:%d: %.*s+%d\n",
        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  }
```

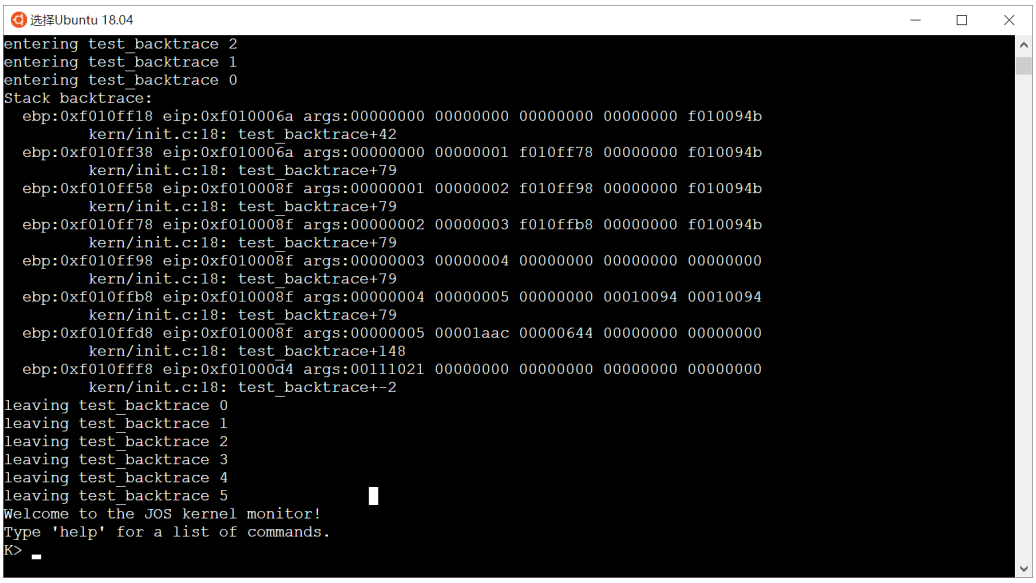

图 6.1: 挑战作业实验结果