

# 操作系统实验报告



**名称：操作系统 Lab1**

学 院：计算机学院 网络空间安全学院

专 业：计算机科学与技术 信息安全

成 员：徐晖宇 付宇 顾知晨 姜奕兵 孙铭 肖阳

课程教师：蒲凌君

时 间：2019 年 10 月

## 小组成员分工

### Lab1-1 启动计算机部分

问题 2，作业 1：徐晖宇（1713666）

作业 2，问题 1：付 宇（1612120）

参与讨论与资料收集：顾知晨（1711323） 姜奕兵（1710218）

孙 铭（1711377） 肖 阳（1610292）

### Lab1-2 内存管理部分

作业 5，问题 4：孙 铭（1711377） 姜奕兵（1710218）

作业 3，问题 3：肖 阳（1610292）

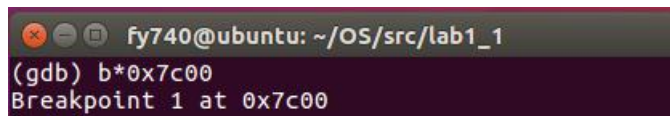
作业 4：顾知晨（1711323）

参与讨论与资料收集：徐晖宇（1713666） 付 宇（1612120）

## 一、问题 1

1) 处理器从哪开始执行 32 位代码？是什么导致了从 16 位到 32 位代码的切换？

利用 `b*0x7c00` 设置断点。如下图：



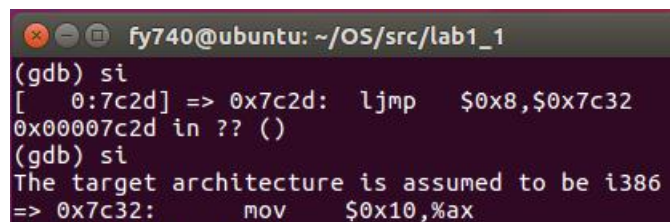
```
fy740@ubuntu: ~/OS/src/lab1_1
(gdb) b*0x7c00
Breakpoint 1 at 0x7c00
```

打开 `.../boot/boot.s` 代码，代码中第一个注释：

`# Start the CPU: switch to 32-bit protected mode, jump into C.`

从该段代码得知，`boot.s` 主要是将处理器从实模式转换到 32 位的保护模式。

接下来我们使用 `si` 命令单步执行，至下图停止 `si` 命令：



```
fy740@ubuntu: ~/OS/src/lab1_1
(gdb) si
[ 0:7c2d] => 0x7c2d:  ljmp  $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c32:  mov  $0x10,%ax
```

我们从上图可知，`The target architecture is assumed to be i386` 这段表明处理器在此开始执行 32 位代码（注：i386 即 intel 80386，其实 i386 通常被用来作为对 intel32 位微处理器的统称）。

找到 `boot.s` 中对应代码部分：

```
53  # Jump to next instruction, but in 32-bit code segment.
54  # Switches processor into 32-bit mode.
55  ljmp  $PROT_MODE_CSEG, $protcseg
56
57  .code32                                # Assemble for 32-bit mode
58  protcseg:
59  # Set up the protected-mode data segment registers
60  movw  $PROT_MODE_DSEG, %ax  # Our data segment selector
```

可见，处理器应该是从 `.code32` 处开始执行 32 位代码。

为了探明 16 位转换至 32 位的原因，我们退回前面研究代码，在 `boot.s` 中得知如下信息：

```

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

```

我们对上述内容非常陌生，查询资料后得知：GDT 全称 Global Descriptor Table，译为全局描述表，是在保护模式下一个重要的数据结构，它是在保护模式所必须的数据结构，也是唯一的。cr0 中包含 6 个预定义标志，第 0 位是保护允许位 PE（protected enable），用于启动保护模式，若 PE 置为 1，则保护模式启动，0 则在实模式下运行。第 1 位是监控协议处理位（monitor coprocessor），它与第 3 位一起决定当 TS=1 时，操作码 WAIT 是否产生一个“协处理器不能使用”的出错信号。第 3 位是任务转换位（task switch），当一个任务转换完成之后，自动将它置为 1，随着 TS=1，就不能使用协处理器。第 2 位是模拟协处理器位 EM（emulate coprocessor），若 EM=1，则不能使用协处理器。第 4 位是微处理器的扩展类型位 ET（processor extension type），其内保存着处理器扩展类型的信息，若 ET=0，则表示系统使用的是 287 协处理器，若 ET=1，则表示系统使用的是 387 浮点协处理器。第 31 位是分页允许位 PG（paging enable），它表示芯片上的分页部件是否允许工作。

在此我们只考虑 PE。如下，CR0\_PE\_ON 在 boot.s 开头处定义为值 1：

```

10      .set CR0_PE_ON, 0x1      # protected mode enable flag

```

至此我们得知，上述代码通过将 cr0 寄存器的 PE 位置为 1 来开启 32 位保护模式（注：如果 PE=0，PG=0，处理器处在实地址模式下；若 PE=1，PG=0，处理器工作在没有开启分页机制的保护模式下；若 PE=0，PG=1，此时由于不在保护模式下不能启用分页机制，因此处理器会产生保护异常；若 PE=1，PG=1，则处理器工作在开启了分页机制的保护模式下）。

2) Boot loader 执行的最后一条指令是什么？Boot loader 加载内核后，内核的第一条指令是什么？

从 boot.asm 中得到以下信息：

```

408      7d6b:  ff 15 18 00 01 00      call  *0x10018

```

从 main.c 中得到以下信息：

```
14 | * * The kernel image must be in ELF format.
33 | #define ELFHDR ((struct Elf *) 0x10000) // scratch space
```

我们可以知道 boot loader 执行的最后一条指令就是

```
7d6b: ff 15 18 00 01 00      call    *0x10018
```

设置断点 b\*7d6b，依次执行 c、si 命令，发现如下情况：

```
7=> 0x7d6b:      call    *0x10018
j
7Breakpoint 2, 0x00007d6b in ?? ()
j(gdb) si
7=> 0x10000c:     movw    $0x1234,0x472
0x0010000c in ?? ()
```

实际跳转地址至 0x10000c。执行 objdump -f kernel 验证，结果同上。我们可以得出结论，内核的加载在 bootmain 函数中完成，加载完成后，内核的第一条指令是 0x10000c: movw \$0x1234,0x472。

```
fy740@ubuntu:~/OS/src/lab1_1/obj/kern$ objdump -f kernel
kernel:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

### 3) 内核的第一条指令在哪？

同上题，内核的第一条指令的地址为 0x10000c。

### 4) Boot loader 是如何知道为了从磁盘获取整个内核所必须读取的扇区数目？它从哪找到这些信息的？

我们从 main.c 中，发现下列加载扇区的代码：

```
50 | // load each program segment (ignores ph flags)
51 | ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
52 | eph = ph + ELFHDR->e_phnum;
53 | for (; ph < eph; ph++)
54 |     // p_pa is the load address of this segment (as well
55 |     // as the physical address)
56 |     readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

其中 ELFHDR 内的某些信息影响着循环次数。接着我们利用 objdump -p kernel 查看内核程序的短信息，如下：



```

fy740@ubuntu:~/05/src/lab1_1/obj/kern$ objdump -p kernel

kernel:      file format elf32-i386

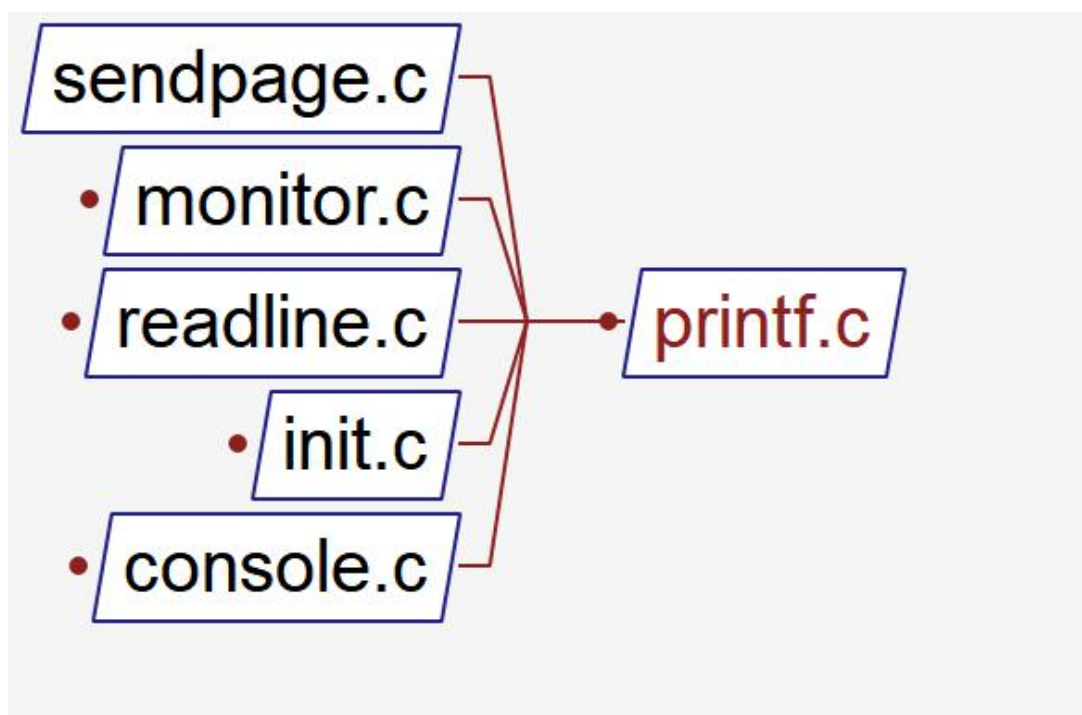
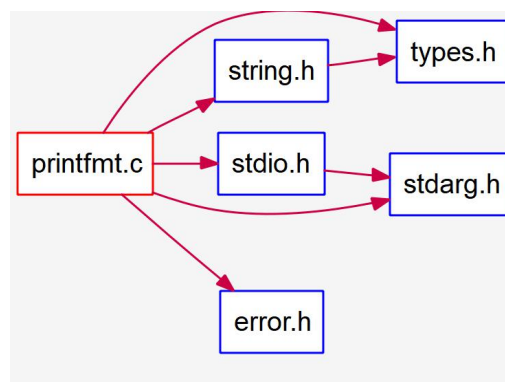
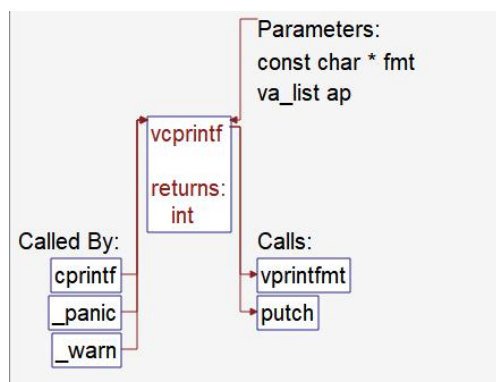
Program Header:
  LOAD off   0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
        filesz 0x0000712f memsz 0x0000712f flags r-x
  LOAD off   0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
        filesz 0x0000a300 memsz 0x0000a944 flags rw-
  STACK off   0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
        filesz 0x00000000 memsz 0x00000000 flags rwx

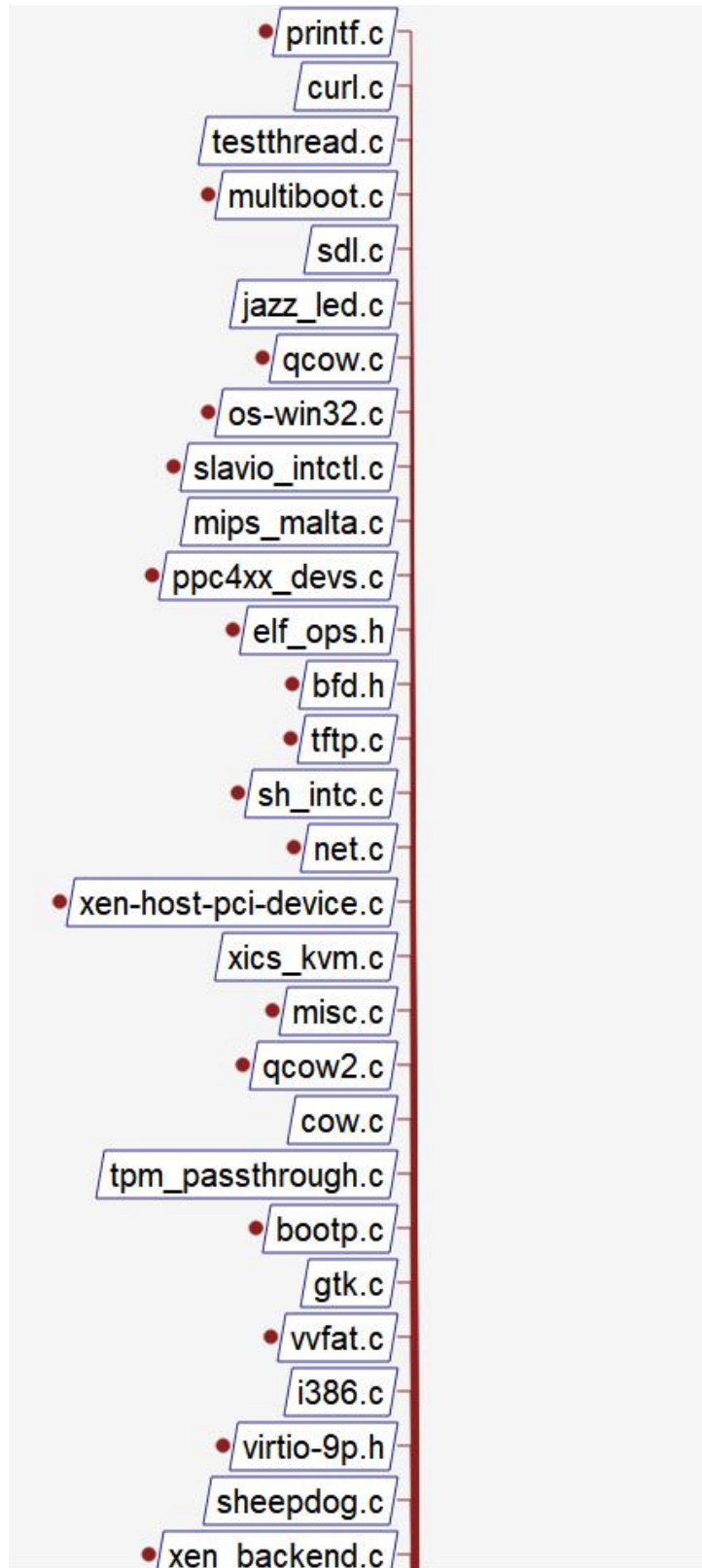
```

至此，我们得出结论，boot loader 通过 program header 中的段数目、每个段的偏移和字节数来知道需要加载的扇区数目。

## 二、问题 2

使用 understand 查看 printf 的父声明关系：





引用关系为 console.c 调用 printf.c, printf.c 调用 printfmt.c  
具体进入 console.c

```
void
cputchar(int c)
{
    cons_putc(c);
}

int
getchar(void)
{
    int c;

    while ((c = cons_getc()) == 0)
        /* do nothing */;
    return c;
}

int
iscons(int fdnum)
{
    // used by readline
    return 1;
}
```

在上面的代码中我发现两点：

1. cputchar 代码的注释中说：这个程序是最高层的 console 的 I/O 控制程序。
2. cputchar 的实现其实是通过调用 cons\_putc 完成的，cons\_putc 程序的功能在它的备注中已经被叙述的很清楚了，即输出一个字符到控制台（计算机的屏幕）。所以我们就知道了 cputchar 的功能也是向屏幕上输出一个字符。



```

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;
    }

    // What is the purpose of this?
    if (crt_pos >= CRT_SIZE) {
        int i;

        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;
    }

    /* move that little blinky thing */
    outb(addr_6845, 14);
    outb(addr_6845 + 1, crt_pos >> 8);
    outb(addr_6845, 15);
    outb(addr_6845 + 1, crt_pos);
}

```

crt\_pos 是当前光标位置, CRT\_SIZE 是屏幕上总共的可以输出的字符数(其值等于行数乘以每行的列数), 这段代码的意思是当屏幕输出满了以后, 将屏幕上的内容都向上移一行, 即将第一行移出屏幕, 同时将最后一行用空格填充, 最后将光标移动到屏幕最后一行的开始处。

### 结论:

printf.c 调用 console.c 提供的接口 cputchar, 将这个函数封装在 putchar, 并将这个封装好的函数作为参数传给 vprintfmt 函数, 用于向屏幕上输出一个字符。

kern/printf.c 提供用户实际需要调用的接口 cprintf

lib/printfmt.c 提供了供 cprintf 函数调用的接口 vprintf, vprintf 的作用是对输出进行格式化, 把不同类型的输出(%s %d %p 等)按不同的方式显示在屏幕上

kern/console.c 提供了供 vprintf 调用的回调函数 cputchar

printfmt.c:

```
// Stripped-down primitive printf-style formatting routines,  
// used in common by printf, sprintf, fprintf, etc.  
// This code is also used by both the kernel and user programs.
```

这一段注释解释了 printfmt 的功能是为 printf 等常见输出定制格式化

```
* The special format %e takes an integer error code  
* and prints a string describing the error.  
* The integer may be positive or negative,  
* so that -E_NO_MEM and E_NO_MEM are equivalent.
```

该段表示 %e 为特殊的控制符, 表示输出错误, 每次输出一个错误声明

printnum 函数递归地打印一个数字串:

```
/  
static void  
printnum(void (*putch)(int, void*), void *putdat,  
          unsigned long long num, unsigned base, int width, int padc)  
{  
    // first recursively print all preceding (more significant) digits  
    if (num >= base) {  
        printnum(putch, putdat, num / base, base, width - 1, padc);  
    } else {  
        // print any needed pad characters before first digit  
        while (--width > 0)  
            putch(padc, putdat);  
    }  
  
    // then print this (the least significant) digit  
    putch("0123456789abcdef"[num % base], putdat);  
}
```

其中各个参数的含义:

void (\*putch)(int, void\*) 这个参数是一个函数指针, 传入的 int 表示要打印的单个字符的值, void\* 表示该字符存在的地址单元的值, 传入该函数指针的目的在于不同的打印方式对应的 putch 不一样, 看到这里的程序段中调用的 putch 是参数指定的。

void \*putdat:表示输入的字符要存放在的地址的指针

unsigned long long num:指需要打印的数字

unsigned base:表示进制

int width:表示输入字符的宽度

int padc:表示填充字符

这里为什么要递归呢，因为多位数（正如程序段中的 num 和 base 的比较）需要一位一位的输出，每一次输出都要对该数字求 base 的商。

getint 函数：

```
static long long
getint(va_list *ap, int lflag)
{
    if (lflag >= 2)
        return va_arg(*ap, long long);
    else if (lflag)
        return va_arg(*ap, long);
    else
        return va_arg(*ap, int);
}
```

该函数指定需要返回的 int 类型，lfag 变量则是专门在输出数字的时候起作用，在我们这个实验中为了简单起见实际上是不支持输出浮点数的，于是 vprintfmt 函数只能够支持输出整形数，当 lflag=0 时，表示将参数当做 int 型的来输出，当 lflag=1 时，表示当做 long 型的来输出，而当 lflag=2 时表示当做 long long 型的来输出。最后 altflag 变量表示当 altflag=1 时函数若输出乱码则用‘?’代替。其中 va\_arg 宏返回可变的参数，第一个形参表示指向参数的指针，第二个参数表示输出的数据类型。

vprintfmt 函数是该文件对外的接口，完成格式化字符的打印

第一个 while 循环：

首先一个一个的输出格式字符串 fmt 中所有‘%’之前的字符，因为它们就是要直接输出的，比如“This is %d test”中的“This is ”。当然如果在把这些字符一个个输出中遇到结束符‘\0’，则结束输出。

剩余的代码都是在处理‘%’符号后面的格式化输出，比如是%d，则按照十进制输出对应参数。另外还有一些其他的特殊字符比如‘%5d’代表显示 5 位，其中

的 5 要特殊处理。

### 三、作业 1

过程参照对应十六进制数的输出的%u 函数段改写，对于八进制数，输出前有一个特殊的/0 需要通过 putchar 打印出来，然后第二步需要获取该数的返回类型，然后设置 base = 8，然后进入 number 处理，number 段处理的方式是调用之前的 printnum 完成格式化输出。

```
case '0':
    // Replace this with your code.
    num = getuint(&p,lflag);
    if ((long long) num < 0) {
        putch('-', putdat);
        num = -(long long) num;
    }
    base = 8;
    goto number;
```

运行结果 (make qemu 对应八进制输出):

```
(process:5283): GLib-WARNING **: 02:48:08.923: ../../../../../../  
om memory allocation vtable not supported  
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.
```

## 四、作业 2

## 作业 2

You can do `mon_backtrace()` entirely in C . You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user. The backtrace function should display a listing of function call frames in the following format:

Waring:

`read_ebp()`(较为底层的函数，返回值为当前的 `ebp` 寄存器的值)

display format

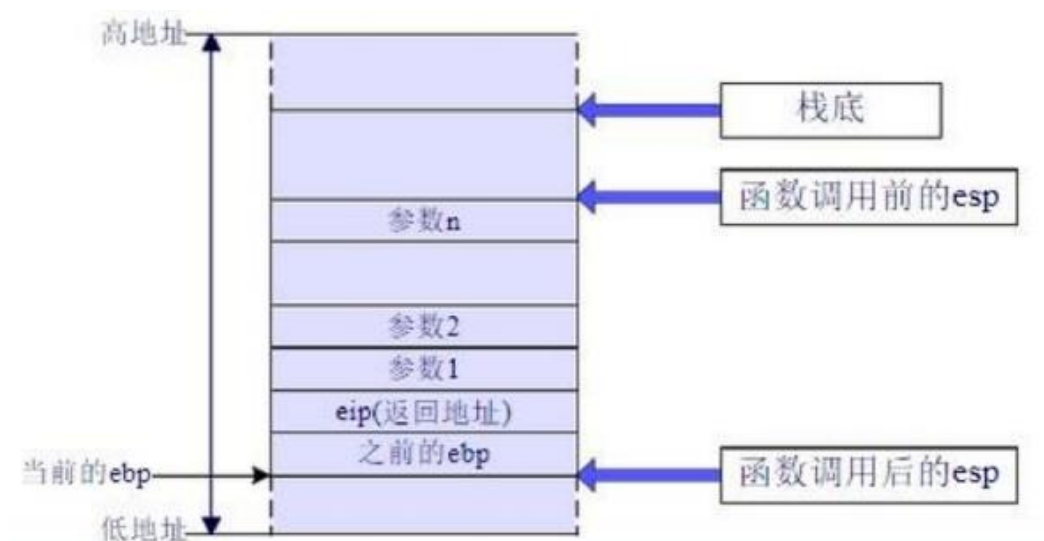
Stack backtrace:

```
ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
...
```

题目要求我们打印调用栈中 `ebp`、`eip` 以及参数的值。

从题中知道通过调用 `read_ebp()`，我们可以得到当前 `ebp` 寄存器的值。（注：`eip` 存储当前执行指令的下一条指令在内存中的偏移地址，`esp` 存储指向栈顶的指针，`ebp` 存储指向当前函数需要使用的参数的指针。）

观察下图：



从文档描述中我们知道：一进入调用函数的时候，第一件事便是将 `ebp` 进栈，然后将当前的 `esp` 的值赋给 `ebp`，而此时 `ebp` 便指向了堆栈中存储 `ebp`、`eip` 和函数参数的地方，所以 `ebp` 通常都是指向当前函数所需要的参数，相当于每个函数都有一个自己的 `ebp`。根据前述信息，我们不难看出 `ebp` 的值实际上是指针值，亦可把它当作数组使用。结合上图，`ebp`，`ebp[1]`，`ebp[2]`，`ebp[3]`，`ebp[4]`，`ebp[5]`，`ebp[6]`分别对应当前 `ebp`、`eip`、参数 1-5 的值。我们采用 `while` 循环来实现打印，



但是存在一个问题，我们并不知道何时中止循环，为了找到跳出循环的临界值，我们查阅了 kern/entry.S 代码，有如下发现：

```
entry.S
# Turn on paging.
movl    %cr0, %eax
orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

我们发现，在内核初始化的时候，ebp 会被置为 0，也就是说在 ebp=0 的时候，循环就应该中止了。因此，我们的代码如下：

```
monitor.c (~OS/src/lab1_1/kern) - gedit
Open  [icon]
monitor.c  x  init.c

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    cprintf("Stack backtrace:\n");

    unsigned int *ebp;

    ebp = (unsigned int*)read_ebp();

    while (ebp!=0)
    {
        cprintf("  ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",
            ebp, ebp[1], ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
        ebp = (unsigned int*)*ebp;
    }

    return 0;
}
```

接着设置断点：

```
(gdb) b kern/monitor.c:4
Breakpoint 1 at 0xf0100678: file kern/monitor.c, line 4.
```

不断执行 c 命令直至结束，得到以下结果：



```
fy740@ubuntu: ~/OS/src/lab1_1
6828 decimal is XXX 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
Stack backtrace:
  ebp f010ff18 eip f010007b args 00000000 00000000 00000000 00000000 f01008ef
  ebp f010ff38 eip f0100068 args 00000000 00000001 f010ff78 00000000 f01008ef
  ebp f010ff58 eip f0100068 args 00000001 00000002 f010ff98 00000000 f01008ef
  ebp f010ff78 eip f0100068 args 00000002 00000003 f010ffb8 00000000 f01008ef
  ebp f010ff98 eip f0100068 args 00000003 00000004 00000000 00000000 00000000
  ebp f010ffb8 eip f0100068 args 00000004 00000005 00000000 00010094 00010094
  ebp f010ffd8 eip f01000d4 args 00000005 00001aac 00000644 00000000 00000000
  ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
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.
```

至此，作业二结束。