

JOS 实习 Lab1 报告

崔治丞 1100012836

内容:

一. JOS 实习 Lab1

a) PC Bootstrap

- i. Getting started with x86 assembly
- ii. Simulating the x86 .
- iii. The ROM BIOS

b) The Boot Loader

- i. Loading the Kernel

c) The Kernel

- i. Using virtual memory to work around position dependence
- ii. Formatted Printing to the Console
- iii. The Stack

1. PC Bootstrap

1.1 Getting started with x86 assembly

Exercise 1. Familiarize yourself with the assembly language materials available on [the 6.828 reference page](#). You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

We do recommend reading the section "The Syntax" in [Brennan's Guide to Inline Assembly](#). It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

JOS 中使用 AT&T 汇编，在 ICS 课上就学过了，所以还是比较熟悉的。但内联汇编只在操作系统原理课上见到过，并没有深入学习，所以在阅读这篇文章时比较费力，需要查阅其他资料，才能够比较好的掌握内联汇编的使用。

1.2 Simulating the x86

在搭建 JOS 环境时，需要从 mit 网站上下载 qemu，由于下载时间较长，我误以为无法下载，所以就用 `sudo apt-get install qemu` 安装了 linux 自己的 qemu。但由于 mit 的 qemu 是加过补丁的，才可以使用 gdb 进行调试，而 linux 自己的 qemu 则无法使用 gdb。为此耗费了很长时间，最后在 google 搜索时看到一篇博客中提到了这个事情，才得以解决。

1.3 The ROM BIOS

Exercise 2. Use GDB's `si` (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at [Phil Storrs I/O Ports Description](#), as well as other materials on the [6.828 reference materials page](#). No need to figure out all the details - just the general idea of what the BIOS is doing first.

BIOS 开始运行时，首先建立中断描述符表，初始化并检查各种硬件，然后找到一个可加载的硬盘，从中读取 bootloader 引导加载操作系统内核。

I/O 手册好长...

2. The Boot Loader

Exercise 3. Take a look at the [lab tools guide](#), especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address `0xc00`, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

Question 1:

At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

Answer 1:

在 boot/boot.S 中有如下代码。boot.S 是 BOIS 在进入 bootloader 后运行的文件。

```
# 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
```

这段代码将 cr0 寄存器的值由 0 改为 1，使处理器从实模式进入到保护模式中。

Question 2:

What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

Answer 2:

bootloader 最后一段代码在 boot/main.c 中，从 ELF 头文件中读取内核入口点 (e_entry)

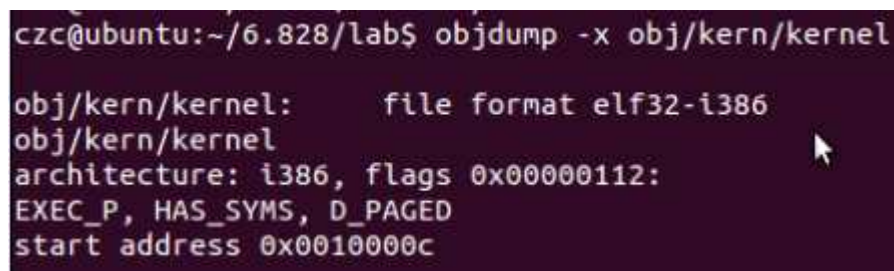
```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
```

Question 3:

Where is the first instruction of the kernel?

Answer 3:

用 objdump -x obj/kern/kernel 命令查询得知 ELF 的入口地址为 0x10000c，但是在反汇编得到的文件中，其地址为 0xf010000c，这是因为后者是虚拟地址，而前者是物理地址，JOS 在加载内核之前完成了虚拟地址和物理地址的映射，所以利用 objdump 命令查到的地址没有高位的 f。



```
czc@ubuntu:~/6.828/lab$ objdump -x obj/kern/kernel
obj/kern/kernel:      file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

再用 gdb 在 0x10000c 设置断点，得到的指令为：

0x10000c: movew \$0x1234, 0x472

这条指令也可以在 kern/entry.S 中看到。



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

Question 4:

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

Answer 4:

在 boot/main.c 中的代码使 bootloader 将 kernel 读入。其中有如下的代码：

```
// 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);
```

从上面的代码中可以看到 ELFHDR->e_phnum, p_memsz 决定了需要读多少节。通过查阅资料知：ELFHDR->e_phnum 指的是程序头部个数，p_memsz 标识了该段在内存中的长度。

2.1 Loading the Kernel

Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an [Amazon Link](#)) or find one of [MIT's 7 copies](#).

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for [pointers.c](#), run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C, though not as strongly recommended. [A tutorial by Ted Jensen](#) that cites K&R heavily is available in the course readings.

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

运行结果：

```
czc@ubuntu:~$ ./pointers
1: a = 0xbf8ccf94, b = 0x8318008, c = 0xb757d225
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 = 0xbf8ccf94, b = 0xbf8ccf98, c = 0xbf8ccf95
```

第一行：分别输出了三个变量的地址

第二行：先将 a[0] 的地址赋给 c，把 a[0] 到 a[3] 分别赋值为 100 到 103，最后将 c 指向的地址的值（也就是 a[0]）改为 200

第三行：由于 c 和 a[0] 的地址相同，前三步操作等同与对 a[1]、a[2]、a[3] 操作

第四行：c 为 int 型指针，c=c+1 使 c 指向 a[1] 的地址

第五行：通过强制类型转换，使 c 指向的地址变为 a[1] 的地址加一个 byte，再转换回 int 型指针，所以会修改 a[1]、a[2] 的值

第六行：b 的值为 a[1] 的地址，c 的值为 a[0] 的地址加 1。

Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would “break” or otherwise do the wrong thing if you were to get the boot loader’s link address wrong. Then change the link address in boot/Makefrag to something wrong, run `make clean`, recompile the lab with `make`, and trace into the boot loader again to see what happens. Don’t forget to change the link address back and `make clean` again afterward!

在 boot/boot.S 中可以看到如下代码：

```
# 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
```

最下面的 `ljmp` 指令就是第一个会 break 的指令。

通过修改 boot/Makefrag 中的链接地址可以产生错误 Triple Fault

Exercise 6. We can examine memory using GDB’s `x` command. The [GDB manual](#) has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints *N* words of memory at *ADDR*. (Note that both ‘x’s in the command are lowercase.) *Warning:* The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the ‘w’ in `xorw`, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

BOIS 进入 bootloader 是在 0x7c00 处，所以第一个断点就设置在 0x7c00 处，而 bootloader 进入 kernel 是在 0x1000c 处，所以第二个断点就在该地址。

查询结果为：

```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:    0x00000000    0x00000000    0x00000000    0x00000000
0x100010:    0x00000000    0x00000000    0x00000000    0x00000000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw    $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:    0x1badb002    0x00000000    0xe4524ffe    0x7205c766
0x100010:    0x34000004    0x0000b812    0x220f0011    0xc0200fd8
(gdb) x/8i 0x100000
0x100000:    add    0x1bad(%eax),%dh
0x100006:    add    %al,(%eax)
0x100008:    decb   0x52(%edi)
0x10000b:    in     $0x66,%al
0x10000d:    movl   $0xb81234,0x472
0x100017:    add    %dl,(%ecx)
0x100019:    add    %cl,(%edi)
0x10001b:    and    %al,%bl
(gdb) █
```

因为内核最后要加载到 0x100000 处，当到达第一个断点时，bootloader 并没有将内核加载，所以在 0x100000 处没有任何代码，全都为 0。当到达第二个断点时，内核已经被加载到 0x100000 处了，所以此时查询到的就是 kernel 的内容了。

3. The Kernel

3.1 Using virtual memory to work around position dependence

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in `kern/entry.S`, trace into it, and see if you were right.

`movl %eax, %cr0` 这条指令在 0x100025 地址处，所以在该处设断点。在该条指令前后，题目所给两个地址的值分别为：

```
(gdb) x/8w 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x0000b812 0x220f0011 0xc0200fd8
(gdb) x/8w 0xf0100000
0xf0100000: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0xf0100010: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
(gdb) si
=> 0x100028: mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/8w 0xf0100000
0xf0100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0xf0100010: 0x34000004 0x0000b812 0x220f0011 0xc0200fd8
(gdb) x/8w 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x0000b812 0x220f0011 0xc0200fd8
(gdb)
```

JOS 会将虚拟地址 [0, 4MB] 和 [0xf0000000, 0xf0000000 + 4MB)映射到物理地址 [0, 4MB)。所以在断点指令执行完，即上述地址映射完成后，两处地址的内容就一样了。

将 `movl %eax, %cr0` 注释掉后：

```
=> 0x10002a: jmp     *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c: (bad)
74          movl    $0x0,%ebp                # nuke frame pointer
(gdb)
```

在 `jmp *%eax` 后产生了错误，JOS 无法继续执行，因为映射机制没有开启，并且没有 0xf0100002 的高地址。

3.2 Formatted Printing to the Console

Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form `"%o"`. Find and fill in this code fragment.

在 `lib/printf.c` 中实现八进制打印，参照 10 进制打印的代码，将代码改为：

```

case 'o':
    // Replace this with your code.
    //putch('X', putdat);
    //putch('X', putdat);
    //putch('X', putdat);
    //break;
    num = getint(&ap, 1flag);
    base = 8;
    goto number;

```

Question 1:

Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

Answer 1:

console.c 主要作用是与硬件交互，处理显示器端口的 IO 请求。console.c 中的 cputchar 函数是到 printf.c 的接口函数，printf.c 中调用的 vprintfmt 函数（printfmt.c 中定义的）中会使用 cputchar 函数的输出。

Question 2:

Explain the following from console.c:

```

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

```

Answer 2:

这段代码的作用是当屏幕已经显示满了的时候，通过对缓存的操作，继而显示接下来的内容。crt_buf 是屏幕显示内容的缓存，当前位置（crt_pos）大于等于屏幕能够显示的大小（CRT_SIZE）时，先将缓存中的内容进行移位，将下一列的内容移进对应大小的 buf 中。

Question 3:

```

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

```

1. In the call to cprintf(), to what does fmt point? To what does ap point?
2. List (in order of execution) each call to cons_putc, va_arg, and vprintf. For cons_putc, list its argument as well. For va_arg, list what ap points to before and after the call. For vprintf list the values of its two arguments.

Answer 3:

1. fmt=0xf0101cbe 指向字符串"x %d, y %x, z %d\n"的地址，ap=0xf010ff74 指向第一个参数 x 的地址。
2. vprintf (fmt=0xf0101cbe "x %d, y %x, z %d\n", ap=0xf010ff74 "\001")
cons_putc (c=120)

```

cons_putc (c=32)
getint va_arg : ap=0xf010ff74 (&x) => 0xf010ff78(&y)
cons_putc (c=49)
cons_putc (c=44)
cons_putc (c=32)
cons_putc (c=121)
cons_putc (c=32)
getuint va_arg : ap=0xf010ff78 (&y) => 0xf010ff7c(&z)
cons_putc (c=51)
cons_putc (c=44)
cons_putc (c=32)
cons_putc (c=122)
cons_putc (c=32)
getint va_arg : ap=0xf010ff7c (&y) => 0xf010ff80
cons_putc (c=52)
cons_putc (c=10)

```

Question 4:

```

unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);

```

What is the output?

Answer 4:

```

He110 World
57616 = e110
0x00646c72 小端存储 = 0x72 0x6c 0x63 0x00 = rld\0
大端 i = 0x726c6400

```

Question 5:

In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

```

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

```

Answer 5:

```

x=3 y=-267321412

```

输出 x=3 后 ap 按照 x 的类型增加对应的值，这里 ap=ap + 4;
但后面缺少参数，函数就将新 ap 指向的地址的值取出来作为输出。

Challenge

Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret [ANSI escape sequences](#) embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on [the 6.828 reference page](#) and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

通过查阅源代码，在 console.c 文件中的 cga_putc(c) 中有注释：
// if no attribute given, then use black on white

所以猜测修改 `c` 的值可以更改其属性。再查阅相关的资料，确认修改其高 8 位可以改变字符显示的 background color 和 foreground color。

这两种颜色都由 4 bit 来控制，从高到低分别为高亮位，R 位，G 位，B 位（对应 RGB 颜色表示法），一共 8 个 bit。

所以在 `monitor.c` 中增加 `setcolor` 函数，并定义 `tempcolor` 记录用户设定的颜色。具体代码如下：

```
int
mon_setcolor(int argc, char **argv, struct Trapframe *tf)
{
    if(argc != 3)
    {
        cprintf("    Choose the color you like.\n");
        cprintf("    The color board:\n");
        cprintf("    Background Color|Foreground Color\n");
        cprintf("    I | R | G | B | | I | R | G | B\n");
        cprintf("    Your input should be: setcolor [8-bit binary number] [a string you want to output]\n");
        cprintf("    Example:setcolor 10000001 Example\n");
        return 0;
    }
    int tempcolor = 0;
    int i=0;
    for(i = 0; i < 8; i++)
    {
        tempcolor = tempcolor * 2 + argv[1][i] - '0';
    }
    tempcolor = tempcolor << 8;
    int c;
    for(i = 0; argv[2][i] != '\0'; i++)
    {
        c = (int)argv[2][i];
        c = c | tempcolor;
        cprintf("%c", c);
    }
    cprintf("\n");
    return 0;
}
```

显示如下：

```
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Display the backtrace
setcolor - Show the string in set color, use help without arguments
K> setcolor 11000101 hello
hello
K> setcolor 01010101 hello
K> setcolor 10000101 a
a
K> 10010101 a
Unknown command '10010101'
K> setcolor 100010101 a
a
K> setcolor 10100110 a
a
K>
```

首先检查用户输入是否符合设定，若不符合就输出帮助信息。若符合设定，则用 `tempcolor` 记录用户输入的 8-bit 颜色信息，将用户希望输出的字符串按字符进行输出。调用 `cprintf` 函数前，进行高 8 位的修改，使其颜色属性改变，从而最终在屏幕上显示出不同颜色的字符。

3.3 The Stack

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which “end” of this reserved area is the stack pointer initialized to point to?

在 entry.S 中可以看到如下代码:

```
# Set the stack pointer
movl    $(bootstacktop),%esp
```

通过反汇编得到的 kernel.asm 中可以看到:

```
# Set the stack pointer
movl    $(bootstacktop),%esp
f0100034:    bc 00 00 11 f0        mov     $0xf0110000,%esp
```

即栈开始的地址为:0xf0110000 同样再 entry.S 中可以看到分布:

```
.data
#####
# boot stack
#####
        .p2align    PGSHIFT    # force page alignment
        .globl      bootstack
bootstack:|
        .space      KSTKSIZE
        .globl      bootstacktop
bootstacktop:
```

从上面的信息可以看出, JOS 为栈预留出了 KSTKSIZE 大小的空间。

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the [tools](#) page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

从 kernel.asm 中可以找到 test_backtrack 的代码, 如下:

```

void test_backtrace(int x)
{
f0100040:      55                push    %ebp
f0100041:      89 e5             mov     %esp,%ebp
f0100043:      53                push    %ebx
f0100044:      83 ec 14          sub     $0x14,%esp
f0100047:      8b 5d 08          mov     0x8(%ebp),%ebx
                cprintf("entering test_backtrace %d\n", x);
f010004a:      89 5c 24 04       mov     %ebx,0x4(%esp)
f010004e:      c7 04 24 a0 19 10 f0 movl    $0xf01019a0,(%esp)
f0100055:      e8 cc 08 00 00    call   f0100926 <cstdio>
                if (x > 0)
f010005a:      85 db             test    %ebx,%ebx
f010005c:      7e 0d             jle     f010006b <test_backtrace+0x2b>
                test_backtrace(x-1);
f010005e:      8d 43 ff          lea     -0x1(%ebx),%eax
f0100061:      89 04 24          mov     %eax,(%esp)
f0100064:      e8 d7 ff ff ff    call   f0100040 <test_backtrace>
f0100069:      eb 1c             jmp     f0100087 <test_backtrace+0x47>
                else
                mon_backtrace(0, 0, 0);
f010006b:      c7 44 24 08 00 00 00 movl    $0x0,0x8(%esp)
f0100072:      00
f0100073:      c7 44 24 04 00 00 00 movl    $0x0,0x4(%esp)
f010007a:      00
f010007b:      c7 04 24 00 00 00 00 movl    $0x0,(%esp)
f0100082:      e8 08 07 00 00    call   f010078f <mon_backtrace>
                cprintf("leaving test_backtrace %d\n", x);
f0100087:      89 5c 24 04       mov     %ebx,0x4(%esp)
f010008b:      c7 04 24 bc 19 10 f0 movl    $0xf01019bc,(%esp)
f0100092:      e8 8f 08 00 00    call   f0100926 <cstdio>
}
f0100097:      83 c4 14          add     $0x14,%esp
f010009a:      5b                pop     %ebx
f010009b:      5d                pop     %ebp
f010009c:      c3                ret

```

中间可以看到有几条指令为：

```

push    %ebp
push    %ebx
sub     $0x14,%esp
0x14 = 20 = 5 * 4 bytes

```

从 ICS 课程中我们知道，每次函数调用，会先将返回地址压入栈中。再由上面的 3 条指令可知，系统会将旧的 ebp 和 esp 压入栈中，最后将 esp-20，为其他变量申请空间，所以 test_backtrace 函数一共会将 5 + 1 + 1 + 1 = 8 个 32-bit 字压入栈中。

这点通过 gdb 在 test_backtrace 函数的地址处（0xf0100040）设置断点，再查询存储器内容可以验证。如图：

```

Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13 {
(gdb) i r
eax                0x0          0
ecx                0x3d4        980
edx                0x3d5        981
ebx                0x10094       65684
esp                0xf010ffdc    0xf010ffdc
ebp                0xf010fff8    0xf010fff8
esi                0x10094       65684
edi                0x0          0
eip                0xf0100040    0xf0100040 <test_backtrace>
eflags             0x82         [ SF ]
cs                 0x8          8
ss                 0x10         16
ds                 0x10         16
es                 0x10         16
fs                 0x10         16
gs                 0x10         16

```

```

Breakpoint 1, test_backtrace (x=4) at kern/init.c:13
13 {
(gdb) i r
eax                0x4          4
ecx                0x3d4        980
edx                0x3d5        981
ebx                0x5          5
esp                0xf010ffbc    0xf010ffbc
ebp                0xf010ffd8    0xf010ffd8
esi                0x10094       65684
edi                0x0          0
eip                0xf0100040    0xf0100040 <test_backtrace>
eflags             0x6          [ PF ]
cs                 0x8          8
ss                 0x10         16
ds                 0x10         16
es                 0x10         16
fs                 0x10         16
gs                 0x10         16

```

两次调用该函数时，ebp 相差 $0x20 = 32 = 8 * 4$ bytes

Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. After you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

只需要将 monitor.c 文件中的 `mon_backtrace` 函数以及前面的 `command` 数组补全即可。

在 ICS 课程中学过，通过函数获得的 `ebp` 的值，再通过 `*(ebp)` 就可以获得上一层递归的旧的 `ebp` 的地址。又由 `entry.S` 中的代码：

```
movl    $0x0,%ebp          # nuke frame pointer
```

可以知道最开始时，`ebp` 被赋值为 0。所以 `while` 循环的结束条件就是 `ebp == 0`。

Exercise 12. Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- run `i386-jos-elf-objdump -h obj/kern/kernel`
- run `i386-jos-elf-objdump -G obj/kern/kernel`
- run `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, and look at `init.s`.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Add a backtrace command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
ebp f010ff78 eip f01008ae args 00000001 f010ff8c 00000000 f0110580 00000000
    kern/monitor.c:143: monitor+106
ebp f010ffd8 eip f0100193 args 00000000 00001aac 00000660 00000000 00000000
    kern/init.c:49: i386_init+59
ebp f010fff8 eip f010003d args 00000000 00000000 0000ffff 10cf9a00 0000ffff
    kern/entry.S:70: <unknown>+0
K>
```

Each line gives the file name and line within that file of the stack frame's eip, followed by the name of the function and the offset of the eip from the first instruction of the function (e.g., `monitor+106` means the return eip is 106 bytes past the beginning of `monitor`).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: printf format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. `printf("%.s", length, string)` prints at most length characters of string. Take a look at the printf man page to find out why this works.

You may find that some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUmakefile`, the backtraces may make more sense (but your kernel will run more slowly).

首先在 `kernel.ld` 中找到 `__STAB_BEGIN` 和 `__STAB_END` 的定义。

```
/* Include debugging information in kernel memory */
.stab : {
    PROVIDE(__STAB_BEGIN__ = .);
    *(.stab);
    PROVIDE(__STAB_END__ = .);
    BYTE(0) /* Force the linker to allocate space
             for this section */
}
```

为了实现完整的 backtrace, 就需要在 `kdebug.c` 文件中补充 `eip_line` 的赋值。代码如下:

```
// Your code here.
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
if (lline <= rline)
    info->eip_line = stabs[lline].n_desc;
else
    return -1;
```

其中 `n_desc` 的含义还不是很清楚, 但在调试中, 发现这个值可以用来表示行数, 所以这里就将这个值赋给 `eip_line` 了。

之后就是继续修改 `monitor.c` 中的 `mon_bakctrace` 函数, 代码如下:

```

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    uint32_t* ebp = (uint32_t*) read_ebp();
    cprintf("Stack backtrace:\n");
    while (ebp != 0) {
        cprintf("  ebp %08x  eip %08x", ebp, *(ebp+1));
        cprintf("  args %08x %08x %08x %08x\n", *(ebp+2), *(ebp+3), *(ebp+4), *(ebp+5), *(ebp+6));
        struct Elpdebuginfo info;
        debuginfo_eip(*(ebp+1), &info);
        cprintf("    %s:%d: %.*s+%d\n", info.eip_file, info.eip_line, info.eip_fn_namelen,
            info.eip_fn_name, *(ebp + 1) - info.eip_fn_addr);

        ebp = (uint32_t*) *ebp;
    }
    return 0;
}

```

结果如下:

```

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
Stack backtrace:
  ebp f010ff18 eip f0100087 args 00000000 00000000 00000000 00000000 f0100980
    kern/init.c:19 test_backtrace+71
  ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f0100980
    kern/init.c:16 test_backtrace+41
  ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f0100980
    kern/init.c:16 test_backtrace+41
  ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f0100980
    kern/init.c:16 test_backtrace+41
  ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
    kern/init.c:16 test_backtrace+41
  ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
    kern/init.c:16 test_backtrace+41
  ebp f010ffd8 eip f01000ea args 00000005 00001aac 00000644 00000000 00000000
    kern/init.c:43 l386_init+77
  ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
    kern/entry.S:83 <unknown>+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> backtrace
Stack backtrace:
  ebp f010ff68 eip f010094b args 00000001 f010ff80 00000000 f010ffc8 f0112540
    kern/monitor.c:140 monitor+258
  ebp f010ffd8 eip f01000f6 args 00000000 00001aac 00000644 00000000 00000000
    kern/init.c:43 l386_init+89
  ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
    kern/entry.S:83 <unknown>+0
K>

```

通过 make grade:

```

running JOS: (1.8s)
printf: OK
backtrace count: OK
backtrace arguments: OK
backtrace symbols: OK
backtrace lines: OK
Score: 50/50
czc@ubuntu:~/6.828/lab$

```