Jos Lab 1: Booting a PC
Lab Report

王勇杰 5120379068

2014.10.13

Contexts

1. PC Bootstrap

1.1 Getting Started with x86 assembly

> **Exercise 1.** Read or at least carefully scan the entire PC Assembly Language book, except that you should skip all sections after 1.3.5 in chapter 1, which talk about features of the NASM assembler that do not apply directly to the GNU assembler. You may also skip chapters 5 and 6, and all sections under 7.2, which deal with processor and language features we won't use in JOS.
>
> Also read the section "The Syntax" in Brennan's Guide to Inline Assembly to familiarize yourself with the most important features of GNU assembler syntax.

在 *PC Assembly Language Book* 中主要介绍了汇编语言，其中大部分的内容和我们在 ICS 课程中所学习的内容相同或类似，基本上是对之前 ICS 课程所学知识的一个小总结。

在 *Brennan's Guide to Inline Assembly* 中介绍了 C 代码中内嵌汇编的一些基本语法，并将其与 Intel 的语法进行了比较。

1.2 Simulating the x86

文档简要介绍了一下关于 QEMU 的使用方法。

1.3 The PC's Physical Address Space

文档简要介绍了一下内存的分配形式。

1.4 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 Jos reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

首先 BIOS 在 0xFFFF0 处得到控制权，然后跳转到 0xFE05B 处，接下来开始对一些寄存器和中断开关和一些 IO 接口进行初始化。

2. The Boot Loader

Exercise 3. Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. 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 GNU disassembly in obj/boot/boot.asm and the 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.

Be able to answer the following questions:

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

在 boot/boot.S 文件中，第 44 行至 55 行有以下的代码：

```
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    gdtdesc
49  movl    %cr0, %eax
50  orl     $CR0_PE_ON, %eax
51  movl    %eax, %cr0
52
53  # Jump to next instruction, but in 32-bit code segment.
54  # Switches processor into 32-bit mode.
55  ljmp    $PROT_MODE_CSEG, $protcseg
```

从上述代码中我们可以看到，在 48-51 行的代码将 cr0 的寄存器最低位置为 1，说明系统从实模式切换成了保护模式。同时后面一句 ljmp 的语句，则是在保护模式下运行的，实际将会跳转到下一句语句。

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

系统在切换到实模式后，会对一些段寄存器和堆栈指针进行了初始化，再调用 main.c 中的 bootmain 函数。这一个函数主要做了将内核文件的每一段读入内存，最后跳转到内核入口地址开始执行内核程序。boot loader 执行的最后一句指令可以在 main.c 中看到，所做的操作即为跳转至内核程序第一句。

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

然后使用 objdump 指令查看 kernel 的指令，找到第一句指令在 0x10000c 处。

```
oslab@debian:~/labs/jos-2014-fall$ 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
```

而后通过使用 gdb 增加断点，找到进入 kernel 后的第一句指令，如下表所示。

```
(gdb) b *0x10000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
0x10000c:   movw   $0x1234,0x472
```

可以看到在进入 kernel 后的第一句指令为 movw $0x1234,0x472，这句指令能在 kern/entry.S 文件中找到，具体如下图：

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

- 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?

在 bootmain 函数中，boot loader 首先读取了文件的前 4KB，而这 4KB 中包括了 ELF 文件头和程序表头，通过使用指令 objdump -x obj/kern/kernel 可以看到在 kernel 文件的头信息。

```
oslab@debian:~/labs/jos-2014-fall$ 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

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align
2**12
         filesz 0x0000791e memsz 0x0000791e flags r-x
```

```
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align
2**12
         filesz 0x0000a300 memsz 0x0000a960 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align
2**2
         filesz 0x00000000 memsz 0x00000000 flags rwx

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001b05  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000760  f0101b20  00101b20  00002b20  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003c01  f0102280  00102280  00003280  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001a9d  f0105e81  00105e81  00006e81  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000660  f0112300  00112300  00013300  2**5
                  ALLOC
  6 .comment      0000001c  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
```

通过上面的信息中可以看到只要有这些头信息，通过 File off 就能找到相对应区块在硬盘中的位置，再用 Size 得到区块的大小，就能完整地将整个文件读出到内存中了。

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).

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.

在 pointers.c 中：

1：程序开始时随机分配三个地址空间给 a、b、c，因此第一个输出是随机的地址。

2：由于 c=a，因此现在 a 指针和 c 指针指向同一个地址，*c 修改的就是 a[0]的值。

3：三种不同的修改指针指向元素值的方法。

4：c 指针指向的元素向后推了一个，现在 c 指向 a[1]这个元素。

5：c 的指针向后推了一个 byte，现在 c 指向 a[1]这个元素的第二个 byte。因此在此时对 c 进行修改时将修改 a[1]的第 2、3、4 个 byte 和 a[2]的第 1 个 byte。

6：b 指向 a 指针向后推一个 int 长度的位置，即向后推了 4 个 byte。c 指向 a 指针向后推一个 char 长度的位置，即向后推了 1 个 byte 的长度。

Exercise 5. 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.)

在进入 boot loader 时得到的内存信息如下：

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] 0x7c00: cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:   0x00000000  0x00000000  0x00000000  0x00000000
0x100010:   0x00000000  0x00000000  0x00000000  0x00000000
```

在进入 kernel 时得到的内存信息如下：

```
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(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
```

在 boot loader 阶段，0x100000 处的内容为空，而在进入 kernel 时里面的数值非空，原因为 0x100000 处是内核载入内存的地址，而在进入 boot loader 阶段时内核还未被载入，该地址空间为空，而在进入 kernel 后，boot loader 将 kernel 代码从内存载入到此内存块，因此从 0x100000 开始即为 kernel 的 ELF 文件内容。

2.2 Link vs. Load Address

> **Exercise 6.** 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` afterwards!

我将 boot/Makefrag 文件中-Ttext 后的参数从 0x7C00 改成了 0x7C01，再次 make 并运行后发现系统一直在不断刷新重新运行，猜想这可能是由于在修改这个参数后，导致在执行 ljmp $PROT_MODE_CSEG, $protcseg 语句后，语句跳转到 0x7C00 执行操作，但是实际上的指令在 0x7C01，最后执行了错误的指令而系统无法正常启动。

3. The Kernel

3.1 Using segmentation to work around position dependence

> **Exercise 7.** Use QEMU and GDB to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.
>
> What is the first instruction *after* the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in `kern/entry.S`, trace into it, and see if you were right.

在 kern/entry.S 中有如下的代码：

```
58  # Turn on paging.
59 movl    %cr0, %eax
60 orl $(CR0_PE|CR0_PG|CR0_WP), %eax
61 movl    %eax, %cr0
62
63 # Now paging is enabled, but we're still running at a low EIP
```

这段代码对应的汇编在 obj/kern/kernel.asm 中如下：

```
31      # Turn on paging.
32     movl    %cr0, %eax
33 f010001d:   0f 20 c0                mov    %cr0,%eax
34     orl $(CR0_PE|CR0_PG|CR0_WP), %eax
35 f0100020:   0d 01 00 01 80          or     $0x80010001,%eax
36     movl    %eax, %cr0
37 f0100025:   0f 22 c0                mov    %eax,%cr0
38
39     # Now paging is enabled, but we're still running at a low EIP
```

可以看出这段代码将 CR0_PG 的标志进行了改变，开启了 paging，虚拟地址将会被页表转化为物理地址。通过汇编可以看到其在 0x00100025 的代码被执行后才会开启页表，因此使用 gdb 将断点设置在 0x00100025 处查看开启页表前后对内存访问的区别。

```
Breakpoint 1, 0x00100025 in ?? ()
(gdb) x 0x00100000
0x100000:    add    0x1bad(%eax),%dh
(gdb) x 0xf0100000
0xf0100000: (bad)
(gdb) si
0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x 0x00100000
0x100000:    add    0x1bad(%eax),%dh
(gdb) x 0xf0100000
0xf0100000: add    0x1bad(%eax),%dh
```

从上面运行的结果可以看出，启动页表前 0xf0100000 的物理地址不存在(bad)，在启动页表后，系统将 0xf0100000 的虚拟地址指向了 0x00100000 的实际物理地址上，使得这 2 个地址的内容相同。

如果将 kern/entry.S 中第 61 行的代码进行注释，这样系统将不会开启页表，此时再运行 qemu，会发现如下问题：

```
qemu -hda obj/kern/kernel.img -serial mon:stdio
Could not open '/dev/kqemu' - QEMU acceleration layer not activated:
No such file or directory
```

这说明不使用页表后，系统无法找到实际需运行的代码储存在哪里，导致系统无法正常的启动。

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. Remember the octal number should begin with '0'.

这个练习是要写 8 进制数字的输出。将 lib/printfmt.c 中相关代码修改如下：

```
213   case 'o':
214   // Replace this with your code.
215   // display a number in octal form and the form should begin with '0'
216       putch('0', putdat);
217       num = getuint(&ap, lflag);
218       base = 8;
219       goto number;
220       break;
```

其中 216 行代码表示先在屏幕上打印出一个数字 0，217-219 行则参考了 10 进制输出的代码。

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?

printf.c 中主要是实现了 cprintf 的输出方法，而 console.c 中则提供了与硬件交互的方式和接口，这里向 printf.c 中提供了输出的调用，printf.c 中的 putch 函数调用了 console.c 的 cputchar 函数。

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     }
```

这段代码是处理显示溢出的问题。当显示屏上能显示的部分已经满时，这段代码将会把第一行擦除，将第二行之后的部分向上移一行，将最后一行置空，最后将光标上移。

第 1 行代码为每次输出一个字符后执行的，作用为检测当前输出字符的位置是否大于屏幕可以输出的文字数，如果是，则说明当前显示已满，将会执行后面的操作。

第 3 行即为执行上移一行的操作。它将 crt_buf+CRT_COLS 的内容向前移动 CRT_COLS 个单位，这样就完成了擦除第一行内容的操作。

第 4-5 行将最后一行的显示内容变为空格。

第 6 行将现在输出内容的光标向前移动一行，这样现在又能在最后一行进行输出显示。

3、For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

Trace the execution of the following code step-by-step:

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

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

在调用 cprintf 函数后，fmt 将会指向"x %d, y %x, z %d\n"这个字符串，而 ap 将会指向 x 存放的位置。对每次字符判定时，如果不是%则会直接输出这个字符，如果为%则会根据%符号之后的类型读取 ap 指针内的内容，同时将 ap 指针通过 va_arg 这个宏向后推移所读取的数据类型的长度，这样 ap 就会指向下一个参数的地址。

4、Run the following code.

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

What is the output? Explain how this output is arrived out in the step-by-step manner of the previous exercise.

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

打印出来的结果为：He110 World

因为 57616 的 16 进制数为 e110，其通过之前 16 进制数的输出方式打印出来。16 进制数在 little endian 和 big endian 上不需要进行修改。

0x00646c72 被按照字符串翻译出来，在 little endian 上被翻译为 72(r)，6c(l)，64(d)，00(\0)，而在 big endian 上会在读到第一个 00 就停止，这样输出结果会变为 He110 Wo。如果在 big endian 上要得到原先的结果，那么数字需要改为 0x726c6400。

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?

```
cprintf("x=%d y=%d", 3);
```

y=后面输出的内容不定，内容为内存空间上在参数 3 所存地址的后一地址空间的内容。因为 ap 指针在读取完 3 之后，指向了不存在实际数据的第二个参数地址，此时这个地址内的内容不能确定，打印出来的内容就是这个地址所存的一个随机数字。

6、Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

由于 ap 指针通过 va_start 这个宏确定初始位置，va_arg 这个宏向后推移，因此在修改压栈顺序时，需要修改 va_start 的值，同时修改 va_arg 计算下一参数地址的方向。

Exercise 9. Enhance the cprintf function to allow it print with the %n specifier, you can consult the %n specifier specification of the C99 printf function for your reference by typing "man 3 printf" on the console. In this lab, we will use the **char *** type argument instead of the C99 **int *** argument, that is, "the number of characters written so far is stored into the **signed char** type integer indicated by the char * pointer argument. No argument is converted." You must deal with some special cases properly, because we are in kernel, such as when the argument is a NULL pointer, or when the char integer pointed by the argument has been overflowed. Find and fill in this code fragment.

%n 的功能为统计输出的所有字符，而在代码中已有 putdat 这个变量统计了，因此只需判断特别情况输出特定字符串，其余正常情况使用 putdat 即可。当传入的指针为 null 时，输出 null_error。当 putdat 大于 127（因为%n 使用的是带符号的 char 类型，最大值为 127）时，输出 overflow_error。具体代码位于 lib/printfmt.c 的 261 行。

Exercise 10. Modidy function `printnum()` in `lib/printfmt.c` file to support patten of the form`"%-"` when outputs number. Padding spaces on the right if cprintf's parameter includes patten of the form `"%-"`. For example, when call function:

```
cprintf("test:[%-5d]", 3)
```

, the output result should be

```
"test:[3    ]"
```

(4 spaces after '3'). Before modidy `printnum()`, make sure you know what happened in function `vprintffmt()`.

在完成这个练习时卡了比较长的时间，一开始想强制执行 width 次递归，最后 num 为 0 时输出空格，但是发现这种实现方式会使空格出现在数字前面，不符合要求，因此最后想到使用变量记录输出格式，然后再在输出完数字后再补上空格的方法。具体实现代码如下：

```
1   static void
2   printnum(void (*putch)(int, void*), void *putdat,
3   unsigned long long num, unsigned base, int width, int padc)
4   {
5
6       int flag = padc;
7       if (padc == '-') padc = -1;
8
9       // first recursively print all preceding (more significant) digits
10      if (num >= base) {
11          printnum(putch, putdat, num / base, base, width - 1, padc);
12      } else {
13          // print any needed pad characters before first digit
14          if (padc != -1)
15              while (--width > 0)
16                  putch(padc, putdat);
```

```
17      }
18
19      // then print this (the least significant) digit
20      putch("0123456789abcdef"[num % base], putdat);
21
22      // at last print space if necessary
23      if (flag == '-')
24      {
25          while (num>=base)
26          {
27              num = num / base;
28              width--;
29          }
30          while (--width > 0)
31              putch(' ', putdat);
32      }
33  }
```

实现的思路为：

首先在第 6 行记录 padc，然后在第 7 行对 padc 进行判定，如果为'-'就说明可能需要在最后输出空格，则修改 padc 为其他内容，这样可以保证只有第一层递归会在最后进行输出空格的操作。

在第 14 行增加一次判定，如果是-1 说明为数字后补空格的输出方式，那么此时不会做在数字前填字符的操作。

第 23 行开始做最后的输出空格操作。这句 if 语句只有在所有递归返回至第一层，输出完数字之后才会执行，思想为根据 num 和 width 单独计算需要补充的空格数，然后输出对应数量的空格。

3.3 The Stack

> **Exercise 11.** 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?

系统内核进行栈初始化的代码在 kern/entry.S 中，具体如下：

```
68    relocated:
69
70    # Clear the frame pointer register (EBP)
71    # so that once we get into debugging C code,
72    # stack backtraces will be terminated properly.
73    movl    $0x0,%ebp           # nuke frame pointer
74
75    # Set the stack pointer
76    movl    $(bootstacktop),%esp
```

在这里第 73 行和 76 行将与堆栈相关的寄存器%esp、%ebp 进行了初始化，将%esp 的值置为了 bootstacktop，虽然在后面有对其的定义，但是 bootstacktop 具体是什么值我不是很清楚。

Exercise 12. 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. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

首先在 obj/kern/kernel.asm 中找到相应的代码，然后对代码进行分析

在调用 test_backtrace 后，首先会将原先的%ebp 和%ebx 压入栈（183 行和 185 行），而后将 esp 减去 0x14 作为这次调用的参数内存占用空间（186 行），最后在 198 行执行了 call test_backtrace 的操作，此时执行跳转之前系统会将当前代码的执行进度%eip 压入栈中。

Exercise 13. 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.

这题为输出栈的情况。根据栈的特点，在得到 ebp 后，ebp 所指向的内容即为其返回的地址，在往上依次为第 1、2、3、4、5 个参数，最终代码实现如下。

```
1   int
2   mon_backtrace(int argc, char **argv, struct Trapframe *tf)
3   {
4       // Your code here.
5       uint32_t bp = read_ebp();
6       uint32_t *ip;
7       cprintf("Stack backtrace:\n");
8       while (bp != 0) {
9           ip = (uint32_t*)bp + 1;
10          cprintf("  eip %08x  ebp %08x  args %08x %08x %08x %08x %08x\n",
11              ip[0],bp,ip[1],ip[2],ip[3],ip[4],ip[5]);
12          bp = *((uint32_t*)bp);
13      }
14
15      overflow_me();
16      cprintf("Backtrace success\n");
17      return 0;
18  }
```

Exercise 14. 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

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 ofmon_backtrace to call debuginfo_eip and print a line for each stack frame of the form:

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).

首先通过查看 inc/stab.h 可以看到 stab 的结构，其中 n_strx 存储字符串索引偏移，n_type 表示类型，n_desc 为描述域，可以理解为行号，n_value 表示符号值（不是很理解这个有什么用）。

因此要输出行号只要找到对应的 n_desc。于是在 kern/kdebug.c 中增加有关行号获取的部分，先使用 stab_binsearch 找到 lline 和 rline，如果找到的话设置行号为 stabs 在 lline 的 n_desc 值，否则返回-1 退出。具体代码在 182-186 行中

然后再对 monitor.c 进行修改，增加相应 debug 信息的输出：

```
1  struct Eipdebuginfo debuginfo;
2  debuginfo_eip(ip[0], &debuginfo);
3  cprintf("\t %s:%d: %.*s+%d\n",
4      debuginfo.eip_file, debuginfo.eip_line,
5      debuginfo.eip_fn_namelen, debuginfo.eip_fn_name,
6      ip[0]-debuginfo.eip_fn_addr);
7  bp = *((uint32_t*)bp);
```

最后再在 monitor.c 的 Command 中增加 backtrace 操作。

Exercise 15. Recall the buffer overflow attack in ICS Lab. Modify your start_overflow function to use a technique similar to the buffer overflow to invoke the do_overflow function. **You must use the above cprintf function with the %n specifier you augmented in "Exercise 9" to do this job, or else you won't get the points of this exercise**, and the do_overflow function should return normally.

这个练习要求通过修改 start_overflow 这个函数使其在返回时跳转到 do_overflow，再返回到原始地址。因此只需先读出目前栈上保存的%eip 的数值，将其向下推移 4 个 byte，然后将原来%eip 的储存值改为 do_overflow 的地址即可。具体代码如下：

```
1.  int i;
2.  pret_addr = (char*)read_pretaddr();
3.  *(uint32_t *)(pret_addr + 4) = *(uint32_t *)pret_addr;
4.  for (i=0; i<4; ++i)
5.  {
6.      nstr = ((uint32_t)do_overflow) >> (8*i);
7.      nstr = nstr & 0xff;
8.      memset(str, 'a', nstr);
9.      str[nstr] = '\0';
10.     cprintf("%s%n", str, pret_addr + i);
11. }
```

第二行中将原先的返回地址向后存 4 个 byte。然后在将 do_overflow 的地址存入原先的地址。由于题目要求使用%n 来写，因此只能一个一个 byte 来覆盖。第 6、7 行代码为读出 do_overflow 地址中的每个 byte，而后讲 str 这个字符串长度置为这个 byte 的大小，在通过 cprintf 的%n 来写入相应地址。

Exercise 16. In this exercise, you need to implement a rather easy "time" command. The output of the "time" is the running time (in clocks cycles) of the command. The usage of this command is like this: "time [command]".

K> time kerninfo

Special kernel symbols:

_start f010000c (virt) 0010000c (phys)

etext f0101a75 (virt) 00101a75 (phys)

edata f010f320 (virt) 0010f320 (phys)

end f010f980 (virt) 0010f980 (phys)

Kernel executable memory footprint: 63KB

kerninfo cycles: 23199409

K>

Here, 23199409 is the running time of the program in cycles. As JOS has no support for time system, we could use CPU time stamp counter to measure the time.

最后这个练习是需要完成一个时钟计时，首先用汇编实现获得 CPU 时钟的函数，具体如下

```
1  typedef unsigned long long cycles_t;
2  inline cycles_t rdtsc() {
3      cycles_t result;
4      __asm__ __volatile__ ("rdtsc" : "=A" (result));
5      return result;
6  }
```

这段代码主要参考了 linux 中关于 rdtsc 的宏定义。

接下来模仿之前的指令实现 mon_time 的函数：

```
7   int
8   mon_time(int argc, char **argv, struct Trapframe *tf)
9   {
10      if (argc != 2) {
11          cprintf("Usage: time [command] \n");
12          return 0;
13      }
14      unsigned long long begin;
15      unsigned long long end;
16      begin = rdtsc();
17      runcmd(argv[1], tf);
18      end = rdtsc();
19      cprintf("kerninfo cycles: %llu\n",end - begin);
20      return 0;
21  }
```

在第 10 行先判定后面是否跟着 1 个 command，而后使用 rdtsc 分别获得 command 执行前后的时钟，相减得到具体运行需要的 cycle 数。在第 17 行执行这句 command。