Lab 1: Booting a PC

▪名词解释：

▪保护模式**(Protected Mode):**是一种和 80286 系列及之后的 x86 兼容 CPU 操作模式。保护模式有一些新的特色,设计用来增强多功能和系统稳定度,比如内存保护、

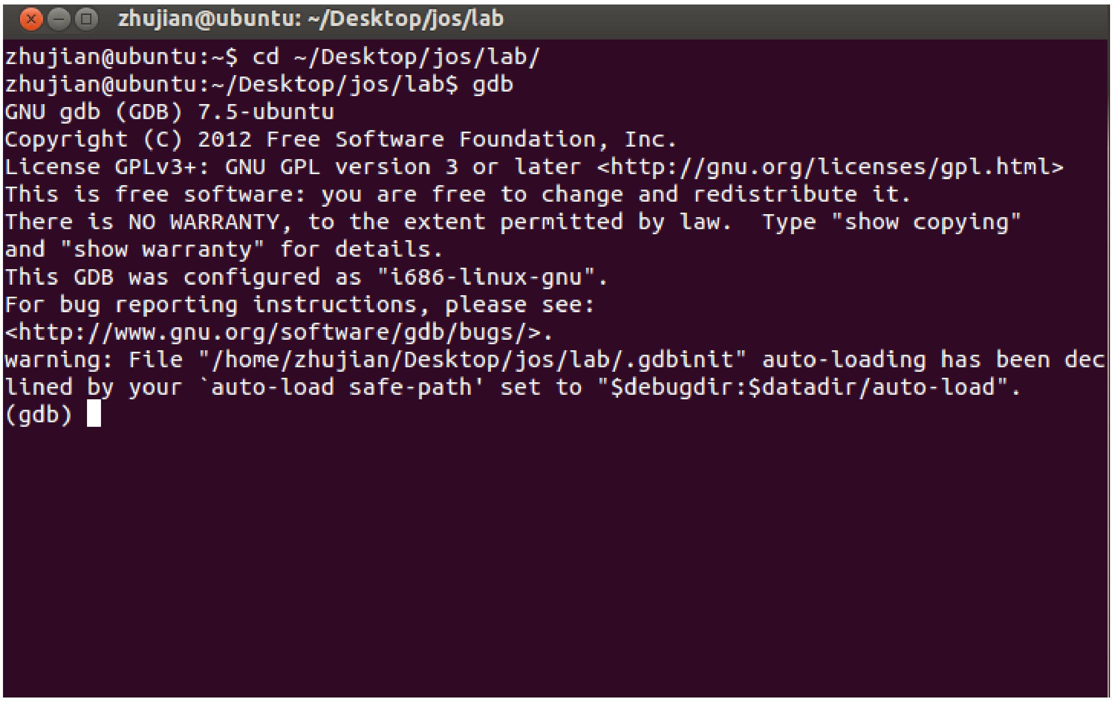
分页、系统以及硬件支持的虚拟内存。

▪**ELF**：ELF = Executable and Linkable Format，可执行连接格式，是 UNIX 系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的。扩展名为 elf。

▪环境配置：

▪使用 **ubuntu-12.10-desktop-i386**

在执行到这一步时，遇到这个问题



warning:File"/home/zhujian/Desktop/jos/lab/.gdbinit"auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".

原因显然是其不能自动加载当前目录下的.gdbinit 文件，解决方法很简单，想想 gdb 的 source 命令，明白了即可。

还是先运行 gdb，gdb 给出上面的提示后，运行一个 gdb 命令"source /home/huang/sdk/.gdbinit"即可。

```
(gdb) source /home/zhujian/Desktop/jos/lab/.gdbinit
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) 
```

Exercise 3

Be able to answer the following questions:

> **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 0x7c00, 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.

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

mode?

```
00007c00 <start>:
.set CR0_PE_ON,      0x1         # protected mode enable flag
```

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

loaded?

```
zhujian@ubuntu:~$ cd ~/Desktop/jos/lab/
zhujian@ubuntu:~/Desktop/jos/lab$ objdump -x obj/kern/kernel

obj/kern/kernel:     文件格式 elf32-i386
obj/kern/kernel
体系结构：i386，  标志 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x0010000c

程序头：
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x000075ad memsz 0x000075ad 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**2
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

Last: Ox7d84 call *%eax ： 调用 elf->entry，开始执行 kernel

First: Ox10000c movw $0x1234, 0x472 ： # warm boot.（可以通过查看 ELF->entry 验证）

```
程序头：
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x000075ad memsz 0x000075ad 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**2
         filesz 0x00000000 memsz 0x00000000 flags rwx

节：
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001a7f  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000704  f0101a80  00101a80  00002a80  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003b35  f0102184  00102184  00003184  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000018f4  f0105cb9  00105cb9  00006cb9  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000644  f0112300  00112300  00013300  2**5
                  ALLOC
  6 .comment      0000002a  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
```

--Where is the first instruction of the kernel?

```
起始地址 0x0010000c
```

Kernel
第一条指令在 0x10000c

```
.globl entry
entry:
        movw    $0x1234,0x472                   # warm boot
f0100000:       02 b0 ad 1b 00 00       add     0x1bad(%eax),%dh
f0100006:       00 00                   add     %al,(%eax)
f0100008:       fe 4f 52                decb    0x52(%edi)
f010000b:       e4 66                   in      $0x66,%al

f010000c <entry>:
f010000c:       66 c7 05 72 04 00 00    movw    $0x1234,0x472
-     kernel.asm      1% L10    (Assembler)
```

--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?

通过 ELF 文件。

Elf->magic: ELF 文件的标识

Elf->phoff: 指定了第一个 section 的位置。

Elf->phnum: 指定了 section 的个数。

Elf->entry: 指定了二进制文件中程序的入口地址。

每个 section 用一个 proghdrs（program headers）描述。

Proghdrs->va: 指定了 section 应该加载到的虚拟地址。

Proghdrs->offset: 指定了 section 相对“ELF header 开始”处的偏移。

Proghdrs->filesz: 指定了 section 在二进制文件中的大小。

Proghdrs->memsz: 指定了 memory 中要为 section 分配的内存大小。

Exercise 4

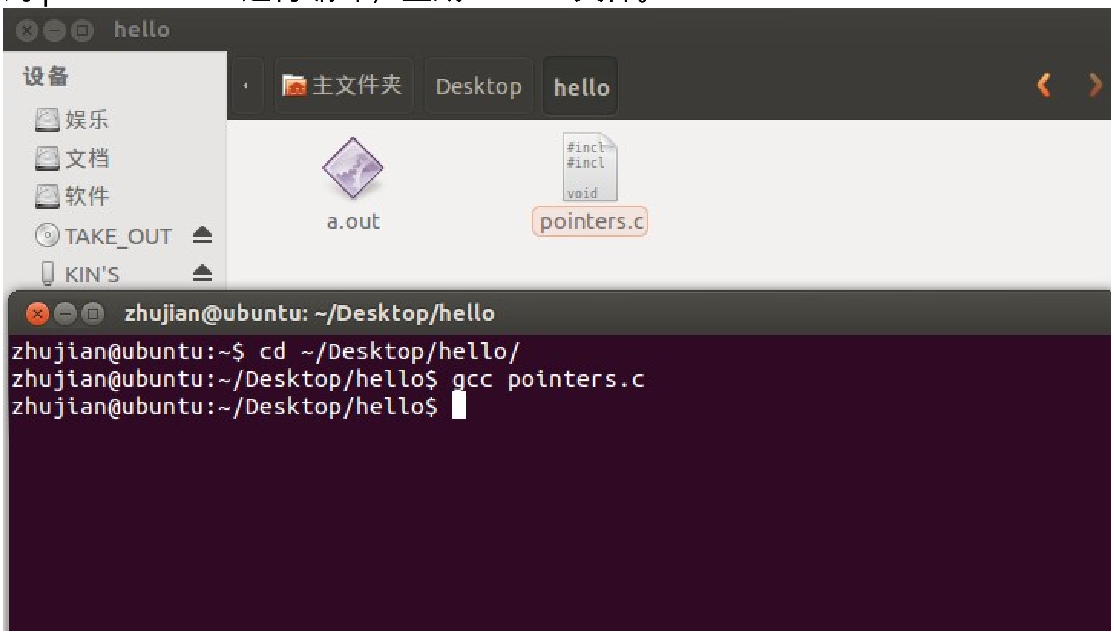
Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 7 copies.

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.

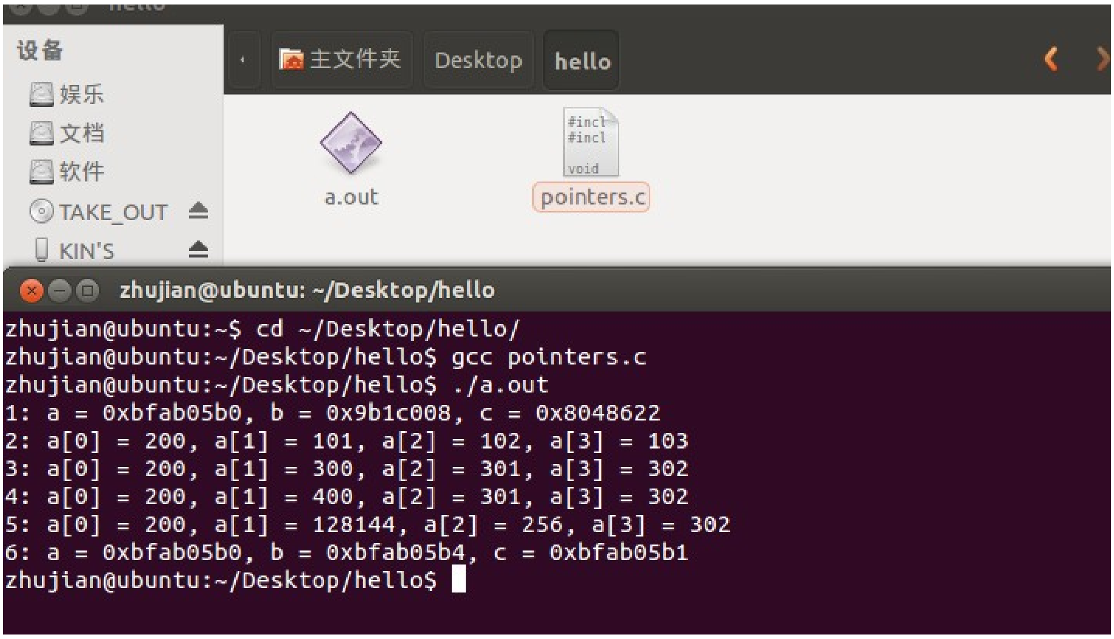
There are other references on pointers in C, though not as strongly recommended. A tutorial by Ted Jensen that cites K&R heavily is available in the course readings.

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.

pointers.c 的代码以及实现功能如下：
对 pointers.c 进行编译，生成 a.out 文件。



之后运行 a.out 文件。



根据运行结果分析代码并添加注释如下：

pointers.c

```c
#include <stdio.h>
#include <stdlib.h>
void
f(void)
{
    int a[4];//创建一个数组
    int *b = malloc(16);//申请16bytes字节的存储空间
    int *c;
    int i;
    printf("1: a = %p, b = %p, c = %p\n", a, b, c);
    //%p在C语言中表示输出一个指针，因为分配空间是随机的，所以输出的地址也是随机的。
    c = a;//*c的值就等于c所指向的内存地址中存储的值，也就是a[0]。
    for (i = 0; i < 4; i++)
        a[i] = 100 + i;//循环赋值a[0]=100,a[1]=101,a[2]=102,a[3]=103
    c[0] = 200;//c[0]=a[0]=200,c[0]此时和a[0]使用相同的内存，a[0]=200
    printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);
    //输出a[0]=200,a[1]=101,a[2]=102,a[3]=103
    c[1] = 300;//c[1]=a[1]
    *(c + 2) = 301;//*(c+2)=a[2]
    3[c] = 302;//3[c]=a[3]
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);
//输出a[0]=200,a[1]=300,a[2]=301,a[3]=302
    c = c + 1;//指针指向a[1]
    *c = 400;//*c=a[1]
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);
//输出a[0]=200,a[1]=400,a[2]=301,a[3]=302
    c = (int *) ((char *) c + 1);//其中c修改的是一个int从第九位开始到第32位,然后将后面一个数的低8位覆盖。
    *c = 500;
    printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);
//输出a[0]=200,a[1]=128144,a[2]=256,a[3]=302
    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);//强制类型转换
    printf("6: a = %p, b = %p, c = %p\n", a, b, c);
}
//a的地址不变，b和c都发生了变化。
int
main(int ac, char **av)
{
    f();
    return 0;
}
```

Exercise 5

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!

找到 the boot loader‘s link address
将 0x7c00 修改为 0x7c01

```
$(OBJDIR)/boot/boot: $(BOOT_OBJS)
	@echo + ld boot/boot
	$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C01 -o $@.out $^
```

重新操作运行，得到如下结果：

```
zhujian@ubuntu:~$ cd ~/Desktop/jos/lab/
zhujian@ubuntu:~/Desktop/jos/lab$ make qemu-gdb
***
*** Now run 'gdb'.
***
qemu -hda obj/kern/kernel.img -serial mon:stdio -g
VNC server running on `127.0.0.1:5900'
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
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>
```

```
GNU gdb (GDB) 7.5-ubuntu
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
warning: File "/home/zhujian/Desktop/jos/lab/.gdbinit" auto-loading has been dec
lined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
(gdb) source /home/zhujian/Desktop/jos/lab/.gdbinit
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) c
Continuing.
```

结果竟然没有报错。。。
又修改了几个值依然没有报错。。。此处有疑问？？？

Exercise 6

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.)

在 bios 进入 Boot Loader 的地址 0x7c00 处和 Boot Loader 进入 kernel 的地址 0x10000c 处打断点。查看内存的变化。
注释：你可以使用 examine 命令（简写是 x）来查看内存地址中的值。

x	从某个位置开始打印存储单元的内容，全部当成字节来看，而不区分哪个字节属于哪个变量

```
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x10000c
0x10000c:       0x00000000      0x00000000      0x00000000      0x00000000
0x10001c:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x001000c
Breakpoint 2 at 0x1000c
(gdb) b *0x10000c
Breakpoint 3 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 3, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb) x/8x 0x0010000
0x10000:        0x464c457f      0x00010101      0x00000000      0x00000000
0x10010:        0x00030002      0x00000001      0x0010000c      0x00000034
(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) 
```

中间打错了断点，断点 1 和断点 **3** 是正确的。可一看到内存发生了变化。
Boot loader 将 kernel
载入到了
0x100000 以及后面的地址上。

Exercise 7

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.

我在 entry.s 中找到 movl%eax, %cr0

```
    # Load the physical address of entry_pgdir into cr3.  entry_pgdir
    # is defined in entrypgdir.c.
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    # Turn on paging.
    movl    %cr0, %eax
    orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0

    # Now paging is enabled, but we're still running at a low EIP
    # (why is this okay?).  Jump up above KERNBASE before entering
    # C code.
    mov     $relocated, %eax
    jmp     *%eax
:ated:
```

boot loader 在进行初始化数据的时候自己定义了 GDT,切换到内核运行后,内核在载入初期马上重新定义了自己的 GDT,然后替换掉了原有的 GDT.

GDT:全局描述符表

　　主要存放操作系统和各任务公用的描述符，如公用的数据和代码段描述符、各任务的 TSS 描述符和 LDT 描述符。

Exercise 8

参照 printfmt.c 中 16 进制的写法重新编写 8 进制代码

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.

截图如下：

将 printfmt.c 中的该部分代码

```c
// (unsigned) octal
case 'o':
        // Replace this with your code.
        putch('X', putdat);
        putch('X', putdat);
        putch('X', putdat);
        break;
```

替换为

```c
// (unsigned) octal
case 'o':
        // Replace this with your code.
        num = getuint(&ap,lflag);
        base = 8;

        goto number;
```

之后进行验证

```
hujian@ubuntu: ~/Desktop/jos/lab
GDB) 7.5-ubuntu
 (C) 2012 Free Software Foundat
PLv3+: GNU GPL version 3 or lat
ree software: you are free to c
NO WARRANTY, to the extent perm
 warranty" for details.
was configured as "i686-linux-g
eporting instructions, please s
ww.gnu.org/software/gdb/bugs/>.
File "/home/zhujian/Desktop/jos
your `auto-load safe-path' set
rce /home/zhujian/Desktop/jos/l
remote localhost:26000
A handler for the OS ABI "GNU/L

Attempting to continue with the

t architecture is assumed to be
0]    0xffff0: ljmp   $0xf000,$
0 in ?? ()
file obj/kern/kernel

g.
```

```
zhujian@ubuntu: ~/Desktop/jos/lab
+ cc lib/printfmt.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
***
*** Now run 'gdb'.
***
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log  -S
VNC server running on `127.0.0.1:5900'
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.
K>
```

右边 15254 处原来是 XXX，证明修改成功。

Be able to answer the following questions:

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`?

kern/console.c 主要提供一些与硬件直接进行交互的接口以便其他程序进

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

行输入输出的调用。

我们可以看到在 printf.c 中调用了这个函数

```
printf.c ×   console.c ×
// Simple implementation of cprintf console output for the kernel,
// based on printfmt() and the kernel console's cputchar().

#include <inc/types.h>
#include <inc/stdio.h>
#include <inc/stdarg.h>


static void
putch(int ch, int *cnt)
{
	cputchar(ch);
	*cnt++;
}
```

2. Explain the following from `console.c`:

```
if (crt_pos >= CRT_SIZE) {
        int i;
        memcpy(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;
}
```

这段代码是用来检验打印后是否满屏，如果满屏就上移一行，空出一行，就是打字满屏之后按下回车效果一样。

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 cprintf()中，fmt 指向的是格式字符串,在上例中即"x %d，y %x，z %d \n"，

而 ap 指向的是不定参数表的第一个参数地址,在上例中即 x。

4. Run the following code.

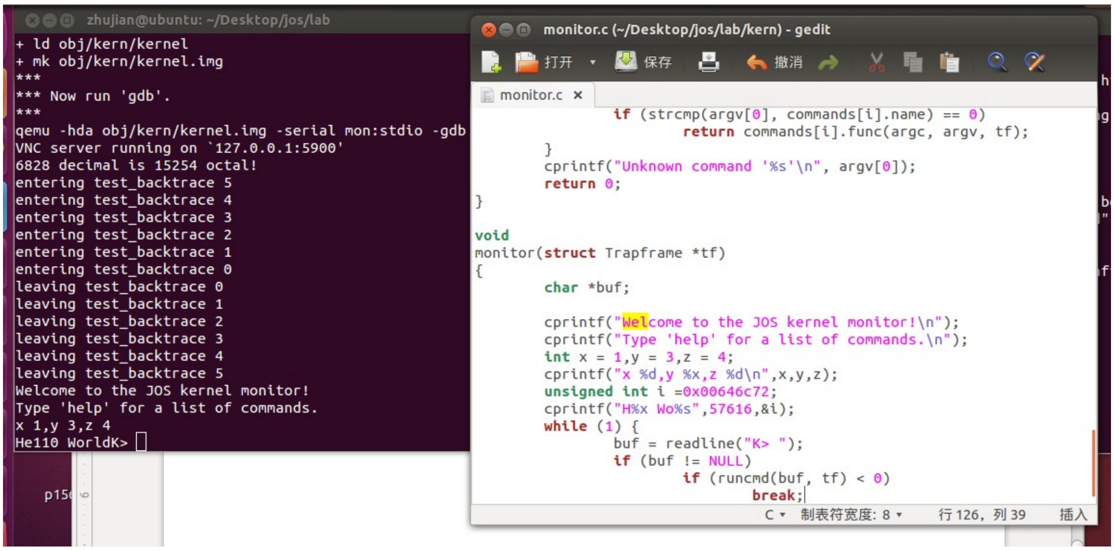
```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

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?

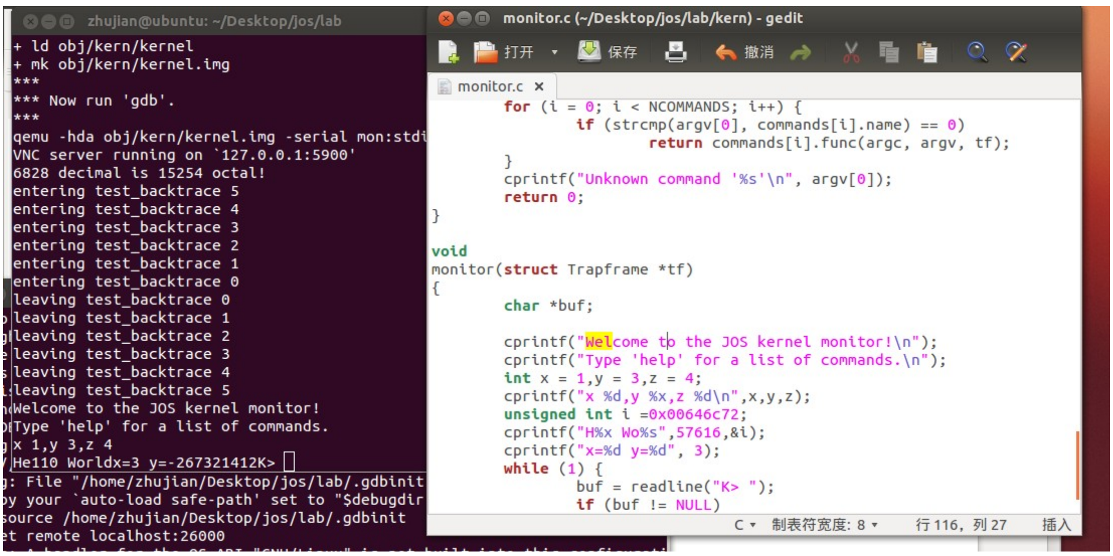
Here's a description of little- and big-endian and a more whimsical description.

3 和 4 一起打印出来。



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);
```



打印出 y=-267321412,y 值是随机的。

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?

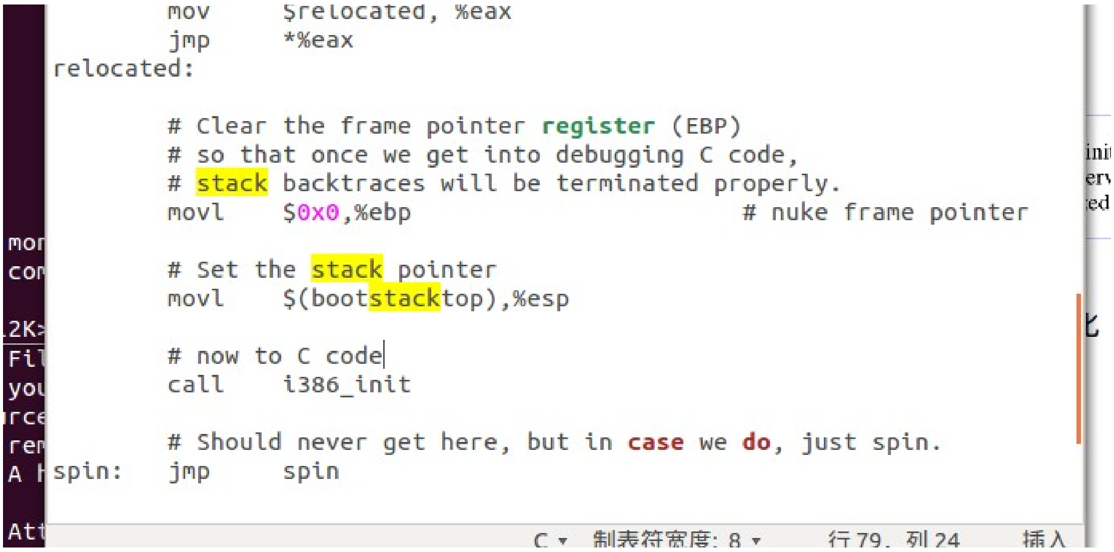
> *Challenge* Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference 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.

没做

Exercise 9

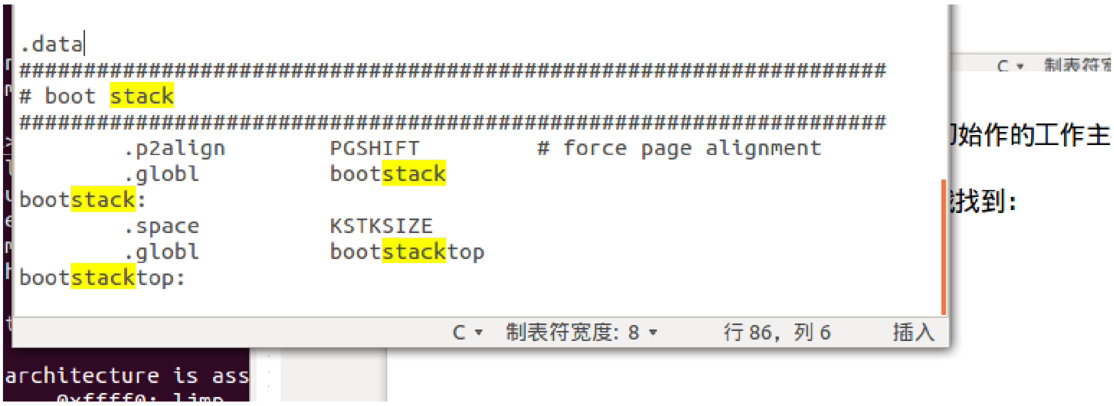
> **Exercise 9.** Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

寻找 kernel 在那里进行栈的初始化



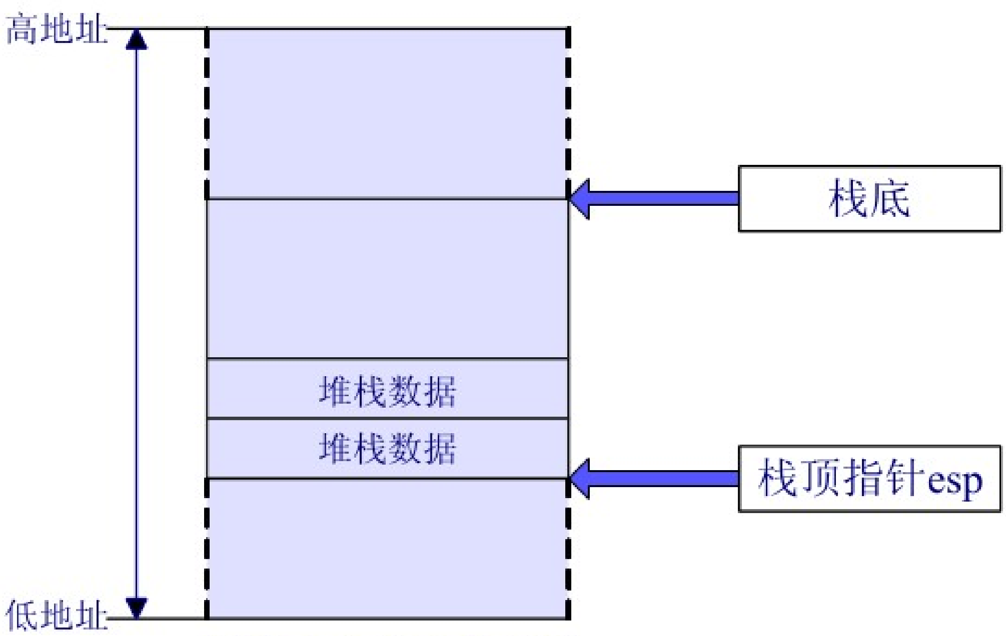
最后在这里找到。可以发现内核初始作的工作主要是将寄存器%ebp 初始为空。

之后我们看到 bootstacktop，查找找到：



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


我们可以了解到在刚进入内核的时候程序定义了一个暂时的堆栈,这个堆栈的大小为 32k,而且刚开始的时候堆栈为空,栈顶指针 esp 是指向栈底。
另外找到堆栈在内存中的示意图如下:



堆栈在内存中的状况

Exercise 10

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.

在 kernel.asm 找到如下内容：

```
// Test the stack backtrace function (lab 1 only)
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 c0 1a 10 f0    movl   $0xf0101ac0,(%esp)
f0100055:       e8 34 09 00 00          call   f010098e <cprintf>
        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 18 07 00 00          call   f010079f <mon_backtrace>
        cprintf("leaving test_backtrace %d\n", x);
f0100087:       89 5c 24 04             mov    %ebx,0x4(%esp)
f010008b:       c7 04 24 dc 1a 10 f0    movl   $0xf0101adc,(%esp)
f0100092:       e8 f7 08 00 00          call   f010098e <cprintf>
}
f0100097:       83 c4 14                add    $0x14,%esp
f010009a:       5b                      pop    %ebx
f010009b:       5d                      pop    %ebp
f010009c:       c3                      ret

f010009d <i386_init>:
```

分析以上内容，可知依次将栈底指针 ebp（4bytes），基底寄存器 ebx（4bytes）入栈，然后栈顶指针 esp 向低地址移动 0x14 个空间(20 bytes），最后将%eip 入栈（4bytes）

。

综上，4+4+20+4=32bytes 的内容入栈。

Exercise 11

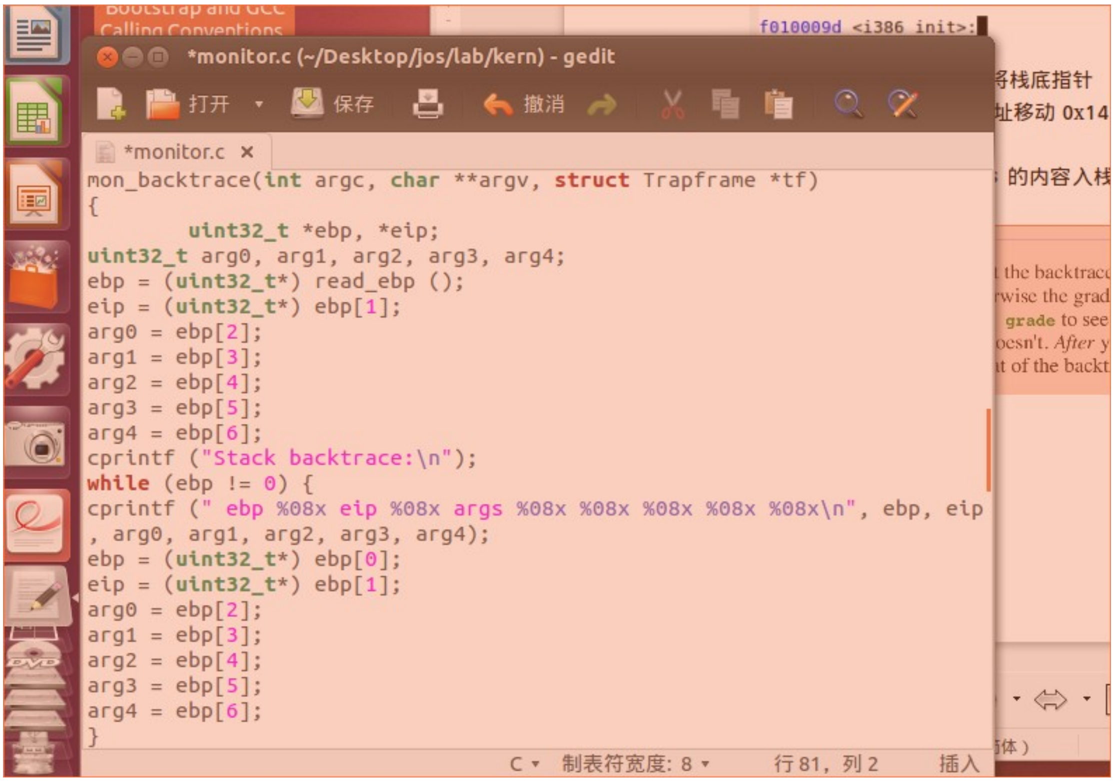
Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. *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.

由上个练习，我们知道了以下几点：

1. 栈中数据从高到低的顺序 ArgN ，ArgN －1 ，．．．，Arg0
2. %eip,函数结束后要返回继续执行的地址

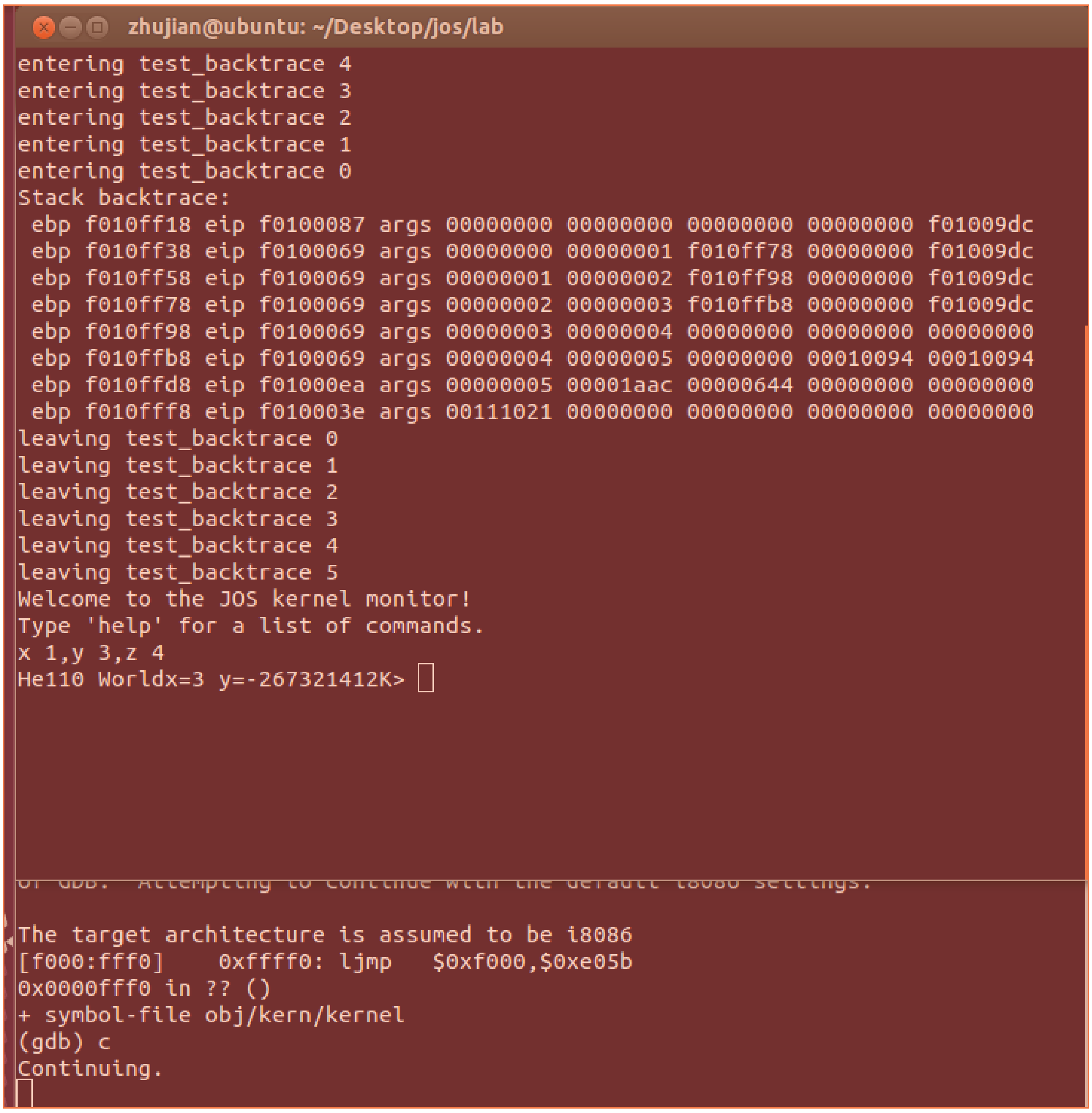
3. %ebp,调用本函数的过程所在的栈指针

因此填充代码到 kern/monitor.c 中



```
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        uint32_t *ebp, *eip;
uint32_t arg0, arg1, arg2, arg3, arg4;
ebp = (uint32_t*) read_ebp ();
eip = (uint32_t*) ebp[1];
arg0 = ebp[2];
arg1 = ebp[3];
arg2 = ebp[4];
arg3 = ebp[5];
arg4 = ebp[6];
cprintf ("Stack backtrace:\n");
while (ebp != 0) {
cprintf (" ebp %08x eip %08x args %08x %08x %08x %08x %08x\n", ebp, eip
, arg0, arg1, arg2, arg3, arg4);
ebp = (uint32_t*) ebp[0];
eip = (uint32_t*) ebp[1];
arg0 = ebp[2];
arg1 = ebp[3];
arg2 = ebp[4];
arg3 = ebp[5];
arg4 = ebp[6];
}
```


运行结果如下



```
zhujian@ubuntu: ~/Desktop/jos/lab
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 f01009dc
 ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f01009dc
 ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f01009dc
 ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f01009dc
 ebp f010ff98 eip f0100069 args 00000003 00000004 00000000 00000000 00000000
 ebp f010ffb8 eip f0100069 args 00000004 00000005 00000000 00010094 00010094
 ebp f010ffd8 eip f01000ea 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.
x 1,y 3,z 4
He110 Worldx=3 y=-267321412K>
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) c
Continuing.
```

Exercise 12

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`).

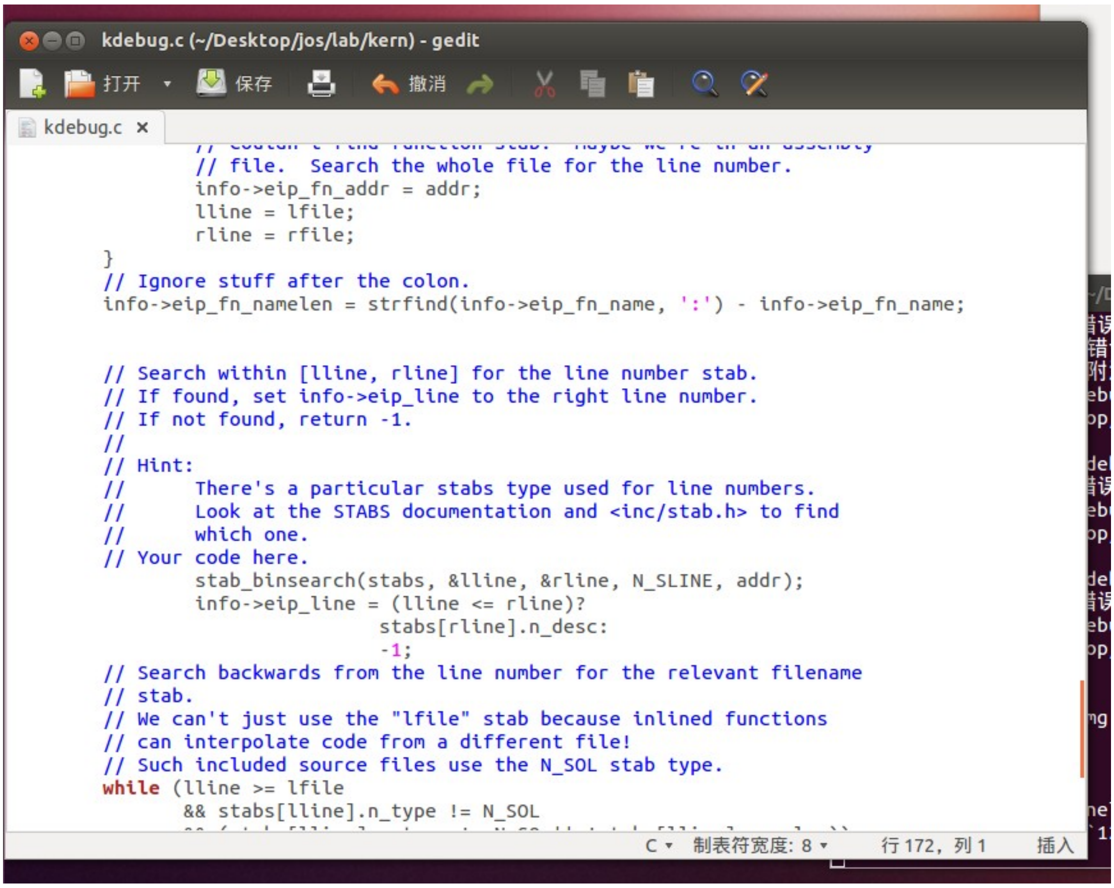
Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

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.

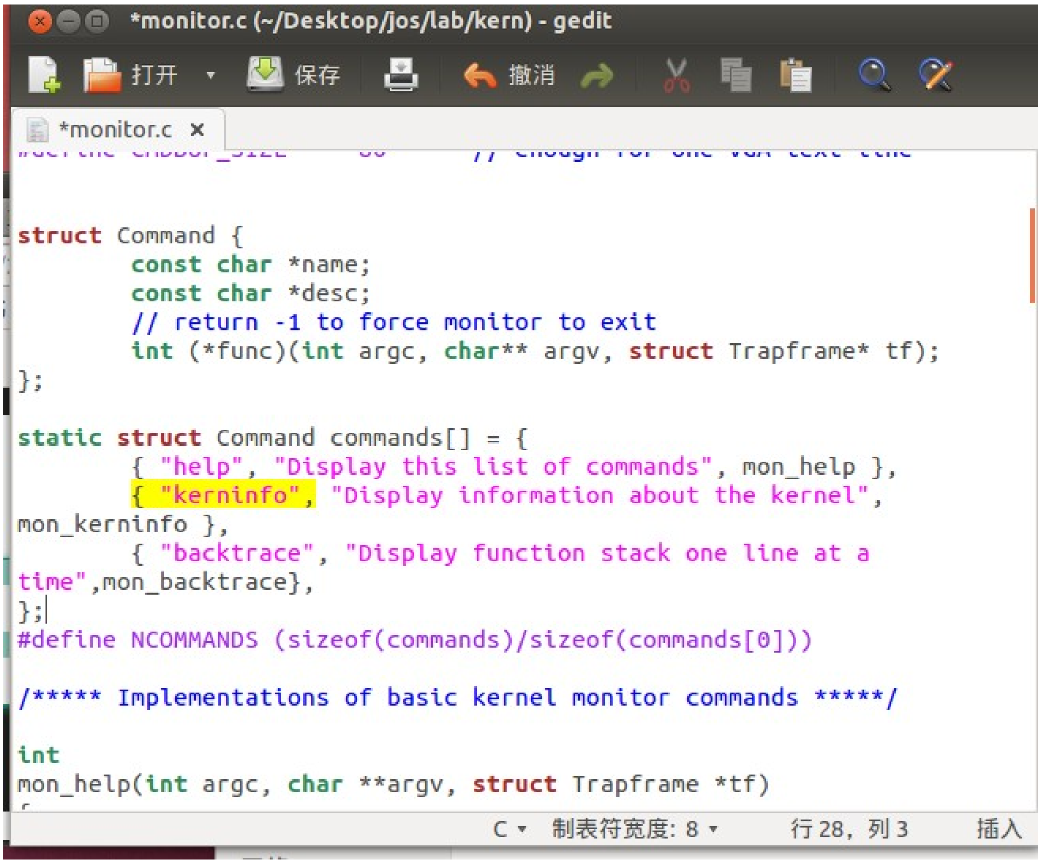
You may find that some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from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

根据联系要求分析后再次位置添加如下代码

在 monitor.c 加入 backrace 命令：

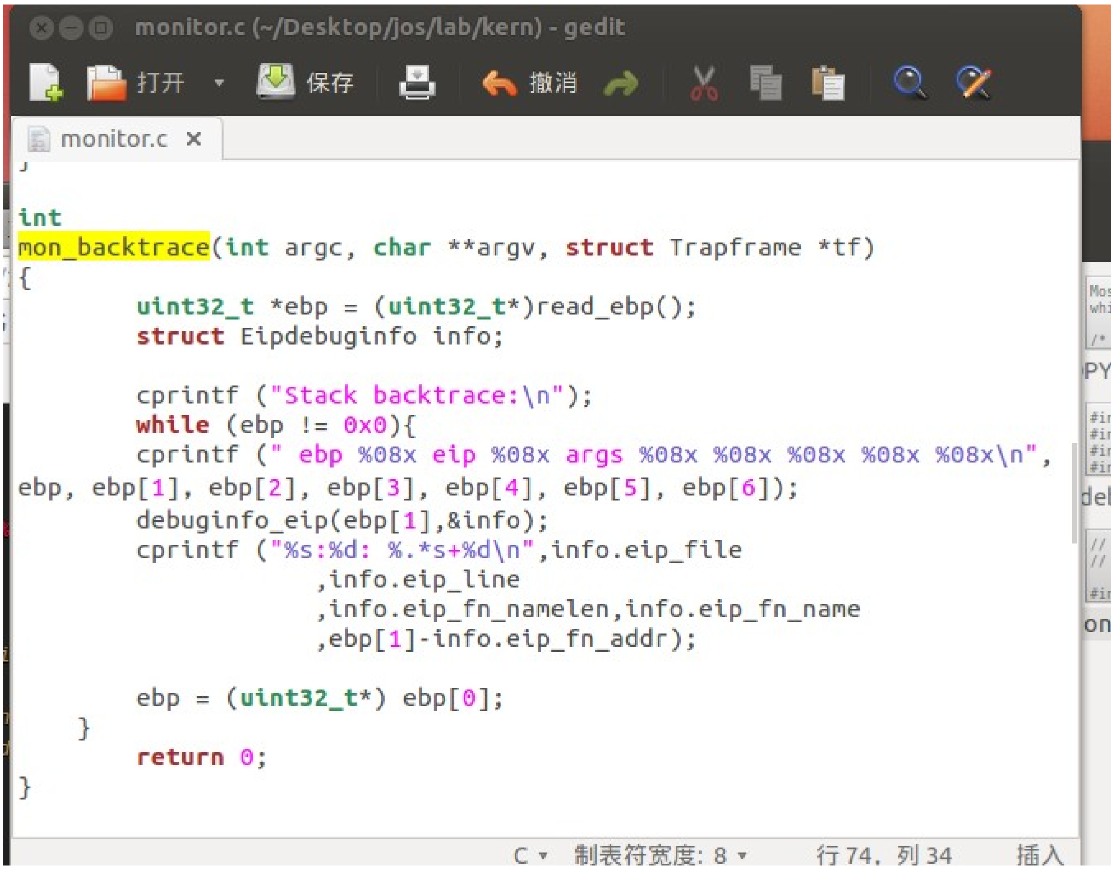



```c
		// file.  Search the whole file for the line number.
		info->eip_fn_addr = addr;
		lline = lfile;
		rline = rfile;
	}
	// Ignore stuff after the colon.
	info->eip_fn_namelen = strfind(info->eip_fn_name, ':') - info->eip_fn_name;


	// Search within [lline, rline] for the line number stab.
	// If found, set info->eip_line to the right line number.
	// If not found, return -1.
	//
	// Hint:
	//	There's a particular stabs type used for line numbers.
	//	Look at the STABS documentation and <inc/stab.h> to find
	//	which one.
	// Your code here.
		stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
		info->eip_line = (lline <= rline)?
				stabs[rline].n_desc:
				-1;
	// Search backwards from the line number for the relevant filename
	// stab.
	// We can't just use the "lfile" stab because inlined functions
	// can interpolate code from a different file!
	// Such included source files use the N_SOL stab type.
	while (lline >= lfile
	       && stabs[lline].n_type != N_SOL
```



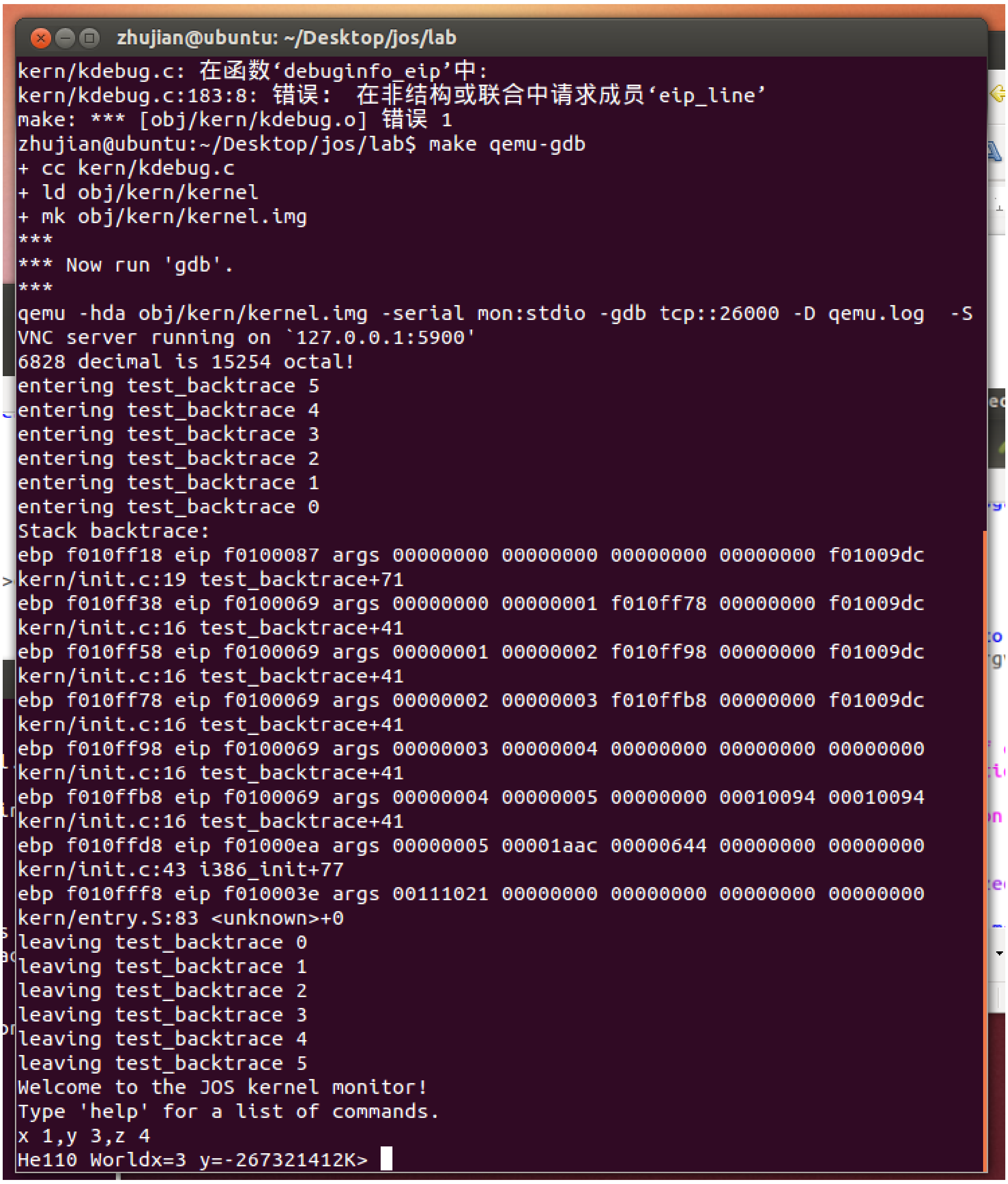

```c
struct Command {
	const char *name;
	const char *desc;
	// return -1 to force monitor to exit
	int (*func)(int argc, char** argv, struct Trapframe* tf);
};

static struct Command commands[] = {
	{ "help", "Display this list of commands", mon_help },
	{ "kerninfo", "Display information about the kernel",
mon_kerninfo },
	{ "backtrace", "Display function stack one line at a
time",mon_backtrace},
};
#define NCOMMANDS (sizeof(commands)/sizeof(commands[0]))

/***** Implementations of basic kernel monitor commands *****/

int
mon_help(int argc, char **argv, struct Trapframe *tf)
```

利用 debuginfo_eip 实现 mon_backtrace:



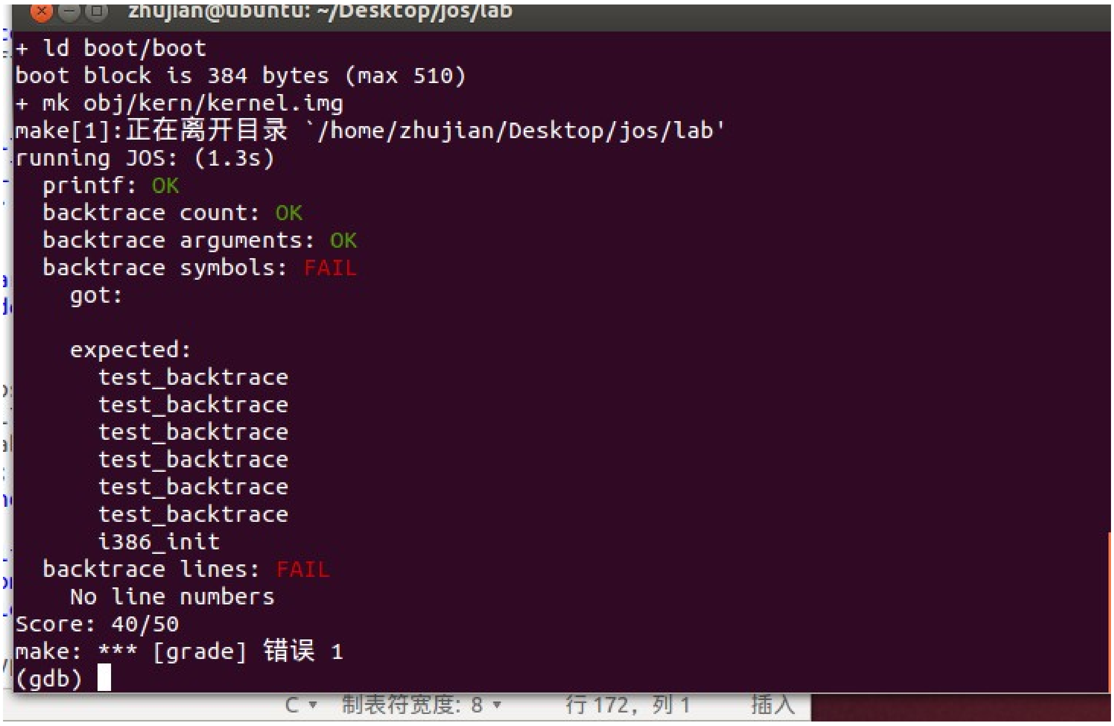

```c
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
	uint32_t *ebp = (uint32_t*)read_ebp();
	struct Eipdebuginfo info;

	cprintf ("Stack backtrace:\n");
	while (ebp != 0x0){
	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]);
	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[0];
    }
	return 0;
}
```

结果如下：



```
zhujian@ubuntu: ~/Desktop/jos/lab
kern/kdebug.c: 在函数‘debuginfo_eip’中:
kern/kdebug.c:183:8: 错误： 在非结构或联合中请求成员‘eip_line’
make: *** [obj/kern/kdebug.o] 错误 1
zhujian@ubuntu:~/Desktop/jos/lab$ make qemu-gdb
+ cc kern/kdebug.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
***
*** Now run 'gdb'.
***
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log  -S
VNC server running on `127.0.0.1:5900'
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 f01009dc
kern/init.c:19 test_backtrace+71
ebp f010ff38 eip f0100069 args 00000000 00000001 f010ff78 00000000 f01009dc
kern/init.c:16 test_backtrace+41
ebp f010ff58 eip f0100069 args 00000001 00000002 f010ff98 00000000 f01009dc
kern/init.c:16 test_backtrace+41
ebp f010ff78 eip f0100069 args 00000002 00000003 f010ffb8 00000000 f01009dc
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 i386_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.
x 1,y 3,z 4
He110 Worldx=3 y=-267321412K>
```

make grade 之后


zhujian@ubuntu: ~/Desktop/jos/lab
+ ld boot/boot
boot block is 384 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]:正在离开目录 `/home/zhujian/Desktop/jos/lab'
running JOS: (1.3s)
printf: OK
backtrace count: OK
backtrace arguments: OK
backtrace symbols: FAIL
got:
expected:
test_backtrace
test_backtrace
test_backtrace
test_backtrace
test_backtrace
test_backtrace
i386_init
backtrace lines: FAIL
No line numbers
Score: 40/50
make: *** [grade] 错误 1
(gdb)
C ▾ 制表符宽度: 8 ▾ 行 172，列 1 插入