JOS1实验报告

王正1110379017

2012 年 7 月 4 日

1 准备工作

首先需要安装qemu，在ubuntu环境下安装非常简单，只需要apt-get install一下即可。可以发现发现一共安装了下列新软件包：bridge-utils cpu-checker libaio1 msr-tools qemu qemu-common qemu-kvm seabios vgabios。随后将jos的工程clone下来。进入该目录，进行编译，形成image文件使得qemu可以装载。之后运行qemu。具体命令如下：

```
sudo apt-get install qemu
git clone http://pdos.csail.mit.edu/6.828/2011/jos.git
    lab
cd lab && make
make qemu
```

在qemu弹出的窗口中运行help发现该窗口的命令只有两种：help和kerninfo.所以需要GDB对jos实验进行调试，调试的方法是打开两个终端，均进入lab目录。在一个终端运行：make qemu-gdb,另外一个运行：gdb.

我们就可以在运行gdb程序的终端内进行调试。常用的命令为si，ni和c。si是汇编中逐条执行指令（单步跟踪进入，ni为单步跟踪，c是继续运行到断点处。b *0x7c00 在地址0x7C00设置断点。

2 启动和装载

> 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.

这个练习的主要目的是学习GNU汇编语言，自己主要注意的是AT&T和Intel汇编之间的区别。Jos实验所使用的是GNU，也就所AT&T格式的汇编，和intel非常相似，最需要注意的是GNU在mov等指令中总是把左边作为source，右边作为destination。寄存器也多了%符号。其他的汇编只是将会在后面的实验中边做边学。

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. No need to figure out all the details - just the general idea of what the BIOS is doing first.

在实验中，在gdb中使用了set logging on来记录每步调试的结果，存储在gdb.txt中。由于该汇编代码很多，所以只看了前面一部分代码。比如如下最开始的一部分：

```
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b;从最开始启动的位置 $0xf000跳转到$0xe05b
[f000:e05b]    0xfe05b: jmp    0xfc7ba;跳转到 0xfc7ba
[f000:c7ba]    0xfc7ba: mov    %cr0,%eax;
[f000:c7bd]    0xfc7bd: and    $0x9fffffff,%eax;
[f000:c7c3]    0xfc7c3: mov    %eax,%cr0;将cr0的2，3位设为0，CD,NW cache 的工作方式，0,0表示正常
[f000:c7c6]    0xfc7c6: cli ;关中断
[f000:c7c7]    0xfc7c7: cld ;
[f000:c7c8]    0xfc7c8: mov    $0x8f,%eax;
[f000:c7ce]    0xfc7ce: out    %al,$0x70;0x70是CMOS索引端口(只写)
```

Bios的主要作用是检测和初始化各种设备和装入操作系统的引导分区，引导分区会被装入到0x7c00到0x7ff之间的内存中。也就是/boot/中的部分内容会在0x7c00开始执行。

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.

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?

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

Where is the first instruction of the kernel?

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?

下面是针对这四个问题的回答：

1. 开始32位模式的地方在设置保护模式的位置之后，保护模式的设置是cr0的最后一位PE位，如果是1就意味着保护模式。boot.s的这几行就在设置保护模式，其中在ljmp之后彻底进入32位模式：

```
# Switch from real to protected mode, using a bootstrap GDT
 # and segment translation that makes virtual addresses
 # identical to their physical addresses, so that the
 # effective memory map does not change during the switch.
 lgdt    gdtdesc
 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
```

在0x7c00设置断点，随后执行c，就看到之后执行的是boot.S中start:块的内容。经过GDB查看实际运行时的代码为：

```
[   0:7c23] => 0x7c23: mov    %cr0,%eax
[   0:7c26] => 0x7c26: or     $0x1,%eax
[   0:7c2a] => 0x7c2a: mov    %eax,%cr0
[   0:7c2d] => 0x7c2d: ljmp   $0x8,$0x7c32
The target architecture is assumed to be i386
=> 0x7c32: mov    $0x10,%ax
```

最后一句话为i386，显然进入了保护模式，实模式时，拿Bios启动时的例子做比较，这里是i8086：

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
```

2. boot loader执行的最后一步指令是装载内核到内存之后的进入内核的执行点。在main.c函数中：

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

   下面通过寻找obj/boot/boot.asm来查看相应的反汇编代码。首先0x7c00的boot.s执行完之后调用00007d0b ¡bootmain¿，进入bootmain中查看，最后调用入口的一句为：

   ```
   The target architecture is assumed to be i386
   => 0x7d63: call   *0x10018
   ```

   随后执行

   ```
   => 0x10000c: movw   $0x1234,0x472
   ```

   查询kernel中的entry.S，得到装载的第一句为：

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

3. 与2中相同，kernel执行的第一句为movw $0x1234,0x472

4. 通过读取kernel的elf文件头来获取这些信息。ELF文件是什么，第四个练习后的一段都在介绍该部分。

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.

该程序pointers的输出结果为：

```
1: a = 0xbf97b044, b = 0x949a008, c = 0x3a7324
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 = 0xbf97b044, b = 0xbf97b048, c = 0xbf97b045
```

这里主要说了c语言中的指针和数组之间的关系和相互表示。第一行打出了abc三个变量的地址，2-4行通过指针来改变数组元素的值，第5行由于int和char的长度分别为四个字节和1个字节，且x86为小端法。开始时c指向a[1]，随后向内存增大方向增加一个字节，指向a[1]中的第二个字节，赋值为500，导致a[1]变为0x1F490，也就是128144，同时影响到a[2]的最小那个字节，使最小的字节变为0，导致a[2]为0x100，也就是256。

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!

这个练习5主要考虑到装载地址和链接地址之间的区别，装载地址是程序自己真正存放在内存中的地址，而链接地址是程序假设自己存放在内存中的地址，也就是虚地址。Bios会将Boot Loader装入到0x7c00位置，所以这个位置是有可能出错的第一个位置。所以无论链接地址怎么改变，装载地址都是在0x7c00处，下面就通过改变boot/Makefrag的0x7C00成其他的地址来改变链接地址，比如改成0x7c08。重新编译，在0x7c00处设断点。断点后继续运行si，可以发现系统在这一步崩溃：

```
[   0:7c35] => 0x7c35: ljmp   $0x8,$0x7c3a
0x00007c2d in ?? ()
```

这一步是一句跳转语句，最有可能的是boot.s中的下面的代码：

```
  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
```

查看反汇编文件obj/boot/boot.asm为：

```
# Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
    7c35:       ea 3a 7c 08 00 66 b8    ljmp   $0xb866,$0x87c3a
00007c3a <protcseg>:
```

由于链接地址往后移动了8位，而实际上protcseg的地址应该为0x7c32而不是0x7c3a，系统在错误的情况下跳转后发现此时的代码错误，不是

```
movw    $PROT_MODE_DSEG , %ax
```

所以会发生错误。正确的代码反汇编后为：

```
  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
    7c2d:       ea 32 7c 08 00 66 b8    ljmp   $0xb866,$0x87c32
00007c32 <protcseg>:
```

> 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.)

首先在0x7c00设置断点，随后执行x/8x 0x00100000，此时结果为：

```
(gdb) x/8x 0x100000
0x100000: 0x00000000 0x00000000 0x00000000 0x00000000
0x100010: 0x00000000 0x00000000 0x00000000 0x00000000
```

随后在0x10000C设置断点，随后执行x/8x 0x00100000，此时结果为：

```
(gdb) x/8x 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x0000b812 0x220f0011 0xc0200fd8
```

再查看反编译文件，obj/kern/kernel.asm

```
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
```

说明了boot loader的主要功能是将内核装入0x100000开始的内存。

3 内核

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.

首先查看obj/kern/kernel.asm查看该指令的位置：

```
movl    %eax, %cr0
f0100025:       0f 22 c0                mov    %eax,%cr0
```

随后在0x100025处设置断点，并且在执行到断点处和往后执行一步处分别查看0x00100000和0xf0100000存储的值，如下：

```
The target architecture is assumed to be i386
=> 0x100025: mov    %eax,%cr0
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
```

显然在执行0x100025:

```
mov    %eax,%cr0
```

之后，0x00100000和0xf0100000存储的值相同。由于在该指令之前就已经设置了内存映射表$entry_pgdir$，该步打开了PG位，使得分页是允许的。所以两个内存映射的值相同。第二个问题，选用的方法是注释掉

```
#       movl    $(RELOC(entry_pgdir)), %eax
#       movl    %eax, %cr3
```

这就使得页表的建立不成功，导致只能使用原先的映射。随后使用gdb调试

```
(gdb) si
=> 0x10001d: mov    %eax,%cr0
0x0010001d in ?? ()
(gdb) si
Cannot access memory at address 0x7bec
```

也就是运行到

```
 mov     $relocated, %eax
```

发生了寻址错误。所以这是第一条出现错误的指令。

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. 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? 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;
}
```

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. 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. 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);
```

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?

修改八进制打印，将lib/printfmt.c中的case 'o'的内容改为：

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

1. console.c 主要将底层的接口抽象，形成接口使得上层代码能够直接进行输入输出。console.c输出的函数为下面代码，意思是输出一个字符到屏幕。

   ```
   // 'High'-level console I/O.  Used by readline and cprintf.
   void
   cputchar(int c)
   {
           cons_putc(c);
   ```

```
}
```

printf.c使用的方法为：

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

2. 查看console.h得知:

```
#define CRT_ROWS        25
#define CRT_COLS        80
#define CRT_SIZE        (CRT_ROWS * CRT_COLS)
```

下面是代码的解释:

```
if (crt_pos >= CRT_SIZE) {
//光标位置（输出的字符）是否满屏幕
                int i;
                memmove(crt_buf, crt_buf + CRT_COLS,
                (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
//将所有行网上移动一行
                for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
                        crt_buf[i] = 0x0700 | ' ';
//最下面一行设置为空
                crt_pos -= CRT_COLS;
//输出的字符数减去一行数目（光标放在最后行首）
        }
```

3. fmt指向字符串”x %d, y %x, z %d
n”，开始时ap指向后面的第一个参数。Cprintf调用vcprintf, vcprintf调
用vprintfmt. 在0xf0100903 ¡vcprintf¿设置断点，查看每次调用时ap的值。
在kern/init.c中加入该句。

```
vcprintf (fmt=0xf01019d2 "x %d, y %x, z %d\n", ap=0xf010ffe4 "\001")
而va_arg在inc/stdarg.h处定义
#define va_arg(ap, type) __builtin_va_arg(ap, type)
由于这里都是int所以是这种情况
```

```
case 'd':
                        num = getint(&ap, lflag);
```

而getinit就会对ap执行va_arg操作

```
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);
}
```

查看汇编知道这三种方式分别对应的代码为:

```
f010102c:       8d 50 08                lea    0x8(%eax),%edx
f0101040:       8d 50 04                lea    0x4(%eax),%edx
f0101052:       8d 50 04                lea    0x4(%eax),%edx
```

在

```
f010104f:       8b 45 14                mov    0x14(%ebp),%eax和
f01010f8:       0f be 55 d8             movsbl -0x28(%ebp),%edx
```

设置断点

得到:

```
Breakpoint 1, vprintfmt (putch=0xf01008f0 <putch>, putdat=0xf010ffac,
fmt=0xf01019d2 "x %d, y %x, z %d\n", ap=0xf010ffe4 "\001")
Breakpoint 1, vprintfmt (putch=0xf01008f0 <putch>, putdat=0xf010ffac,
fmt=0xf01019d2 "x %d, y %x, z %d\n", ap=0xf010ffe8 "\003")
    at lib/printfmt.c:227
Breakpoint 1, vprintfmt (putch=0xf01008f0 <putch>, putdat=0xf010ffac,
fmt=0xf01019d2 "x %d, y %x, z %d\n", ap=0xf010ffec "\004")
```

查看内存知道0xf010ffe4: 0x00000001 0x00000003 0x00000004 0x00000000

所以va_arg是取esp+4的元素，随后ap再加4，如果是longlong则加8。

4. 该句加入init.c中，打印的结果是He110 World 57616用16进制表示是e110，0x00646c72在x86机器上，也就是小端法的机器上表示的是从高到底为0x00,0x64,0x6c,0x72，读取字符串时先从低地址读取就是0x72,0x6c,0x64,0x00,也就是r,l,d,
0.大端法机器上i的值需要变成0x726c6400，而57616无需更改。

5. 该句还是加入到init.c中，打印的结果是x=3 y=1632 因为上文说了va_arg是往高地址增加4位，所以y的值应该是x大一个单元的位置里面的内容。继续在

   ```
   return va_arg(*ap, int);
   f010103f:       8b 45 14                mov    0x14(%ebp),%eax
   ```

   设置断点。

   ```
   Breakpoint 1, vprintfmt (putch=0xf01008e0 <putch>, putdat=0xf010ffac,
   fmt=0xf01019d2 "x=%d y=%d", ap=0xf010ffe4 "\003")
   Breakpoint 1, vprintfmt (putch=0xf01008e0 <putch>, putdat=0xf010ffac,
   fmt=0xf01019d2 "x=%d y=%d", ap=0xf010ffe8 "‘\006")
   ```

   显然ap的值在x之后增大了4，而这个位置的值是未知的。

6. va_arg的具体实现如3所示，由于栈是由高地址往低地址增长，所以参数压栈的顺序是从右往左，这点从上面的xyz的输出也能知道。改变顺序之后，也就是从左向右压栈，这就意味着va_arg需要重写。使得寻找下一个参数是减去4或者8而不是加上。

> 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 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.

控制颜色的语句在console.c中的

```
static void
cga_putc(int c)
        // if no attribute given, then use black on white
        if (!(c & ~0xFF))
                c |= 0x0700;//白底黑字
```

整个打印的调用过程为：

```
cprintf->vcprintf->vprintfmt->putch->cputchar->cons_putc->cga_putc(lpt_putc)
```

所属的文件为：

3.printfmt.c

```
1.2.4.printf.c
5.6.7.console.c


static void
putch(int ch, int *cnt)
{
cputchar(ch);
*cnt++;
}
static void
cons_putc(int c)
{
serial_putc(c);
lpt_putc(c);
cga_putc(c);
}
```

这里c一共16位，高八位表示颜色，低八位表示ASCII码。高八位中前四位表示背景色，后四位表示字体颜色，分别为从高到底I,R,G,B。可以用ANSI CODE的形式来表示颜色。所以这个练习可以通过改c的16-9位来实现。需要修改的是printfmt.c和console.c中的cga_putc方法。ANSI CODE的表示方法的一个例子为
033[0;32;40m 这里
033为转义符号，[为开始，m为结束。0，32，40等定义如下

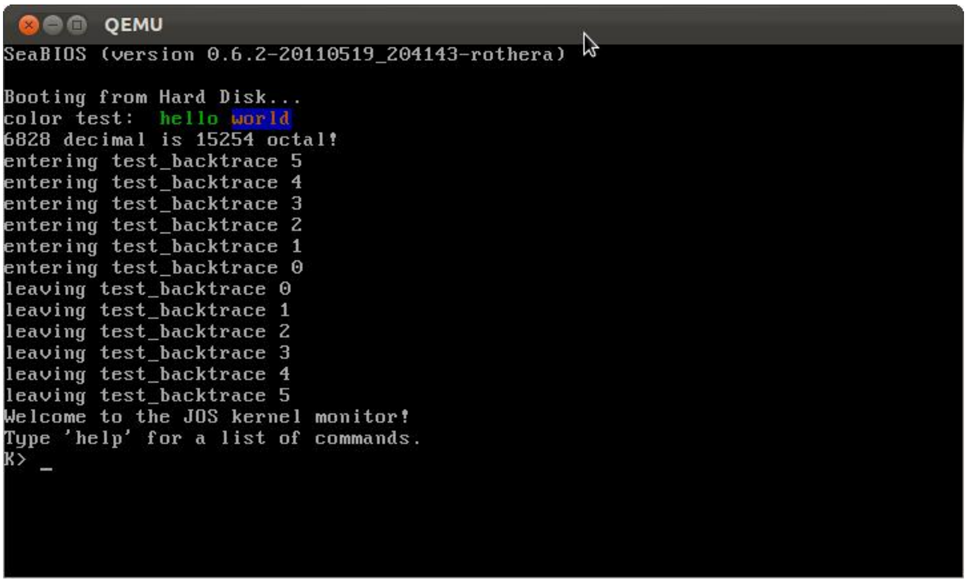
```
Text attributes
       0    All attributes off
       1    Bold on
       4    Underscore (on monochrome display adapter only)
       5    Blink on
       7    Reverse video on
       8    Concealed on

    Foreground colors
       30    Black
       31    Red
       32    Green
       33    Yellow
       34    Blue
       35    Magenta
       36    Cyan
       37    White

    Background colors
       40    Black
       41    Red
       42    Green
       43    Yellow
       44    Blue
       45    Magenta
       46    Cyan
       47    White
```

由于改函数的参数十分繁琐，所以修改的方法是在printfmt.c加入全局变量char_color，并且在console.c外部引用。由于ansi code和cag_putc并不是完全一致，所以需要进行映射。在console.c中加入

```
extern int char_color;

static void
cga_putc(int c)
{
// if no attribute given, then use black on white
c = c | (char_color<<8);
```

在printfmt中的vprintfmt加入

```
if (ch ==’\033’){//if this is Escape character  and this if is char color set.
int temp_color_no=0;
ch = *(unsigned char *) fmt++;
while (ch != ’m’)
{
ch = *(unsigned char *) fmt++;
if ( ch !=’;’ && ch!=’m’)
temp_color_no=temp_color_no*10+ch-’0’;
else if ( ch==’;’ || ch==’m’){
if ( temp_color_no >=30 && temp_color_no<40){// Foreground colors
char_color = (char_color&0xf0) + (temp_color_no-30);
}
```



```
SeaBIOS (version 0.6.2-20110519_204143-rothera)

Booting from Hard Disk...
color test:  hello world
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> _
```

```
if ( temp_color_no >=40 && temp_color_no<50){// Background colors
char_color = (char_color&0x0f) + ((temp_color_no-40)<<4);
}
cprintf("0x%o ,",char_color);
temp_color_no=0;
}
}
if (ch == 'm')
ch = *(unsigned char *) fmt++;

}
```

思想就是通过发现转义序列中的数字来改变char_color的值，从而影响到输出的颜色。打印的代码为，放在init.c中。

```
cprintf("color test: \033[0;32;40m hello \033[0;36;41mworld\033[0;37;40m\n");
```

结果图所示。

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?

内核栈的初始化在entry.S中:

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl $0x0,%ebp # nuke frame pointer
# Set the stack pointer
movl $(bootstacktop),%esp
```

意思就是ebp为0，esp为bootstacktop。Bootstacktop的定义为:

```
bootstack:
.space KSTKSIZE
.globl bootstacktop
bootstacktop:
```

启动栈有两部分构成，栈空间为KSTKSIZE，栈底就为bootstacktop。KSTKSIZE在inc/memlayout.h 中定义为32k。memlayout也规定了栈底的地址为

```
VPT,KSTACKTOP--> +------------------------------+ 0xefc00000
```

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function inobj/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.

反汇编代码中test_backtrace为

```
f0100040 <test_backtrace>:
#include <kern/console.h>

// 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 40 1a 10 f0  movl   $0xf0101a40,(%esp)
f0100055: e8 cc 08 00 00        call   f0100926 <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 ff 06 00 00        call   f0100786 <mon_backtrace>
cprintf("leaving test_backtrace %d\n", x);
f0100087: 89 5c 24 04           mov    %ebx,0x4(%esp)
f010008b: c7 04 24 5c 1a 10 f0  movl   $0xf0101a5c,(%esp)
f0100092: e8 8f 08 00 00        call   f0100926 <cprintf>
}
f0100097: 83 c4 14              add    $0x14,%esp
f010009a: 5b                    pop    %ebx
f010009b: 5d                    pop    %ebp
f010009c: c3                    ret
```

首先%ebp压栈，然后%ebx被保存压栈，随后%esp减去0x14，最后调用时call f0100926 ¡cprintf¿，%eip入栈，所以一共有8+20+4=32字节。经过调试发现，该函数从5到0依次递归。

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.

填充monitor.c中的mon_backtrace 需要完成的工作是对0-5进行回溯，打印ebp,eip和参数。read_ebp 这个函数，这个函数的功能是:利用一些较低层的指令将当前ebp寄存器中的值读出来，函数的返回值便是ebp 的值。代码如下:

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
// Your code here.
uint32_t ebp,eip,arg[5];
ebp = read_ebp();
eip = *((uint32_t*)ebp+1);
arg[0] = *((uint32_t*)ebp+2);
arg[1] = *((uint32_t*)ebp+3);
arg[2] = *((uint32_t*)ebp+4);
arg[3] = *((uint32_t*)ebp+5);
arg[4] = *((uint32_t*)ebp+6);

cprintf("Stack backtrace:\n");

while(ebp != 0x00)
{
cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",
ebp,eip,arg[0],arg[1],arg[2],arg[3],arg[4]);
ebp = *(uint32_t *)ebp;
eip = *((uint32_t*)ebp+1);
arg[0] = *((uint32_t*)ebp+2);
arg[1] = *((uint32_t*)ebp+3);
arg[2] = *((uint32_t*)ebp+4);
arg[3] = *((uint32_t*)ebp+5);
arg[4] = *((uint32_t*)ebp+6);
}
return 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 debuginf_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 ofmonitor). 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("You may find that the 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).

首先可以查看下命令objdump -G obj/kern/kernel显示到是什么，这里显示到有很多个元素，主要一共使用到的有三个n_type，SO，FUN和SLINE，n_value当前表项对应的符号值，n_desc 行号。一个例子如下：

```
0      SO     0      0      f0100000 1      {standard input}
1      SOL    0      0      f010000c 18     kern/entry.S
14     SO     0      2      f0100040 31     kern/entrypgdir.c
67     SO     0      0      f0100040 0
68     SO     0      2      f0100040 2775   kern/init.c
144    SO     0      0      f01001a8 0
145    SO     0      2      f01001b0 3058   kern/console.c

96     FUN    0      0      f0100040 2859   test_backtrace:F(0,18)
106    FUN    0      0      f010009d 2900   i386_init:F(0,18)
114    FUN    0      0      f0100104 2918   _panic:F(0,18)
130    FUN    0      0      f0100163 3014   _warn:F(0,18)
174    FUN    0      0      f01001b0 3100   delay:f(0,18)
180    FUN    0      0      f01001be 3114   serial_proc_data:f(0,1)
192    FUN    0      0      f01001da 3138   cons_intr:f(0,18)
205    FUN    0      0      f0100217 3193   cons_putc:f(0,18)
275    FUN    0      0      f01003fb 3229   kbd_proc_data:f(0,1)
314    FUN    0      0      f01004fb 3275   serial_intr:F(0,18)
319    FUN    0      0      f0100516 3295   kbd_intr:F(0,18)
323    FUN    0      0      f0100528 3312   cons_getc:F(0,1)
336    FUN    0      0      f010056b 3329   cons_init:F(0,18)
377    FUN    0      0      f0100661 3376   cputchar:F(0,18)
382    FUN    0      0      f0100671 3393   getchar:F(0,1)
389    FUN    0      0      f0100682 3408   iscons:F(0,1)

346    SLINE  0      153    0000004d 0
348    SLINE  0      98     00000053 0
350    SLINE  0      154    0000005b 0
352    SLINE  0      48     0000005e 0
354    SLINE  0      154    00000061 0
356    SLINE  0      98     00000067 0
357    SLINE  0      48     0000006f 0
```

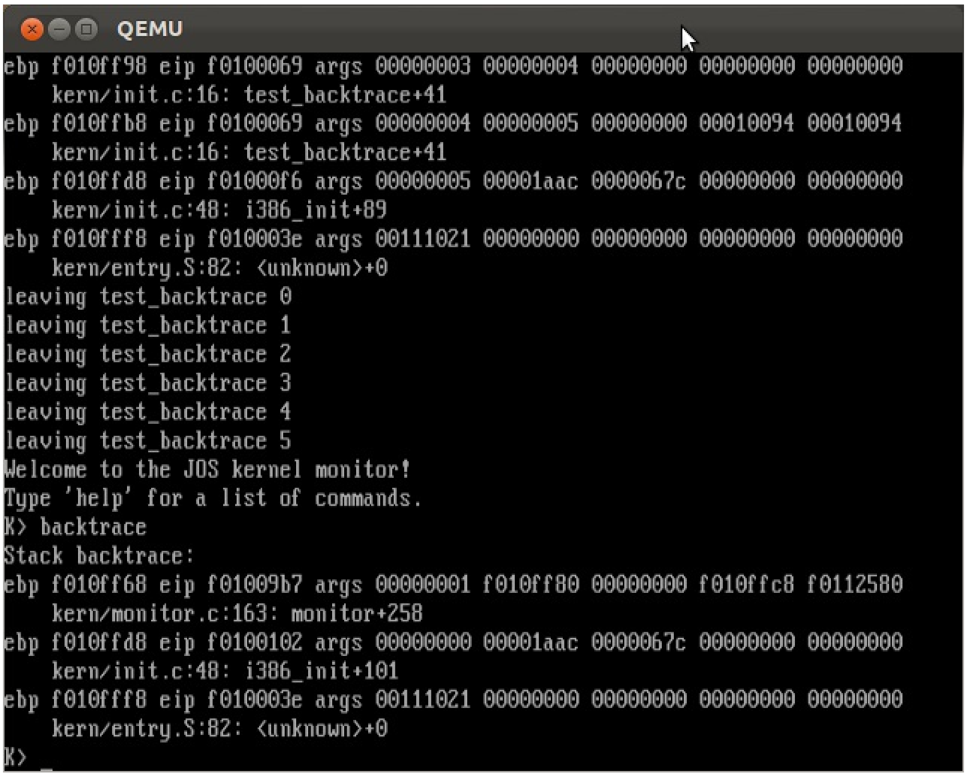
```
359    SLINE  0      158    00000072 0
360    SLINE  0      156    00000078 0
361    SLINE  0      159    0000007d 0
363    SLINE  0      98     00000084 0
364    SLINE  0      48     000000c6 0
366    SLINE  0      101    000000c9 0
368    SLINE  0      48     000000d8 0
370    SLINE  0      449    000000de 0
371    SLINE  0      450    000000e2 0
372    SLINE  0      451    000000ee 0
379    SLINE  0      458    00000000 0
```

显然可以利用SO寻找所属文件，FUN寻找所属到函数，SLINE寻找行数。在debuginfo_eip中实现这些功能，基本思路就是寻找和eip值最接近的较小的n_value值，这种寻找是二分查找实现的。所以填充ebuginfo_eip很简单：

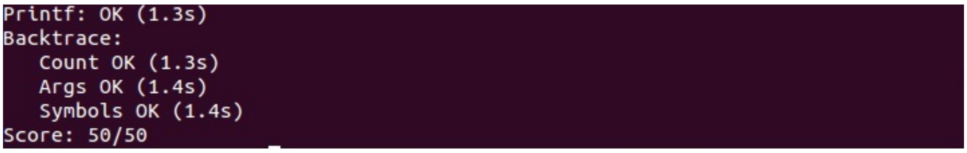
```
// Your code here.
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);

if (lline <= rline) {
info->eip_line = stabs[lline].n_desc;
}
```

由于这些值都返回在info结构体中，所以在monitor中加入如下功能：

```
struct Eipdebuginfo info;
info.eip_file = "<unknown>";
info.eip_line = 0;
info.eip_fn_name = "<unknown>";
info.eip_fn_namelen = 9;
info.eip_fn_addr = eip;
info.eip_fn_narg = 0;
char display_eip_fn_name[30];
debuginfo_eip(eip,&info);
int i;
for ( i=0;(info.eip_fn_name[i]!=':')&&( i<30); i++)
display_eip_fn_name[i]=info.eip_fn_name[i];
display_eip_fn_name[i]='\0';
cprintf("    %s:%d: %s+%d\n",info.eip_file,info.eip_line,
display_eip_fn_name,(eip-info.eip_fn_addr));
```



```
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 f01000f6 args 00000005 00001aac 0000067c 00000000 00000000
    kern/init.c:48: i386_init+89
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
    kern/entry.S:82: <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 f01009b7 args 00000001 f010ff80 00000000 f010ffc8 f0112580
    kern/monitor.c:163: monitor+258
ebp f010ffd8 eip f0100102 args 00000000 00001aac 0000067c 00000000 00000000
    kern/init.c:48: i386_init+101
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
    kern/entry.S:82: <unknown>+0
K> _
```



```
Printf: OK (1.3s)
Backtrace:
   Count OK (1.3s)
   Args OK (1.4s)
   Symbols OK (1.4s)
Score: 50/50
```

由于最后到加号后面是返回指令相对与函数开始时的差，所以可以eip-info.eip_fn_addr实现。另外i386_init:F(0,18) 与要求的i386_init:不一样所以去掉冒号后面到内容。

最后加入新的命令：

```
{ "backtrce", "Display the backtarce of the stack", mon_backtrace},
```

至此实验一完成。