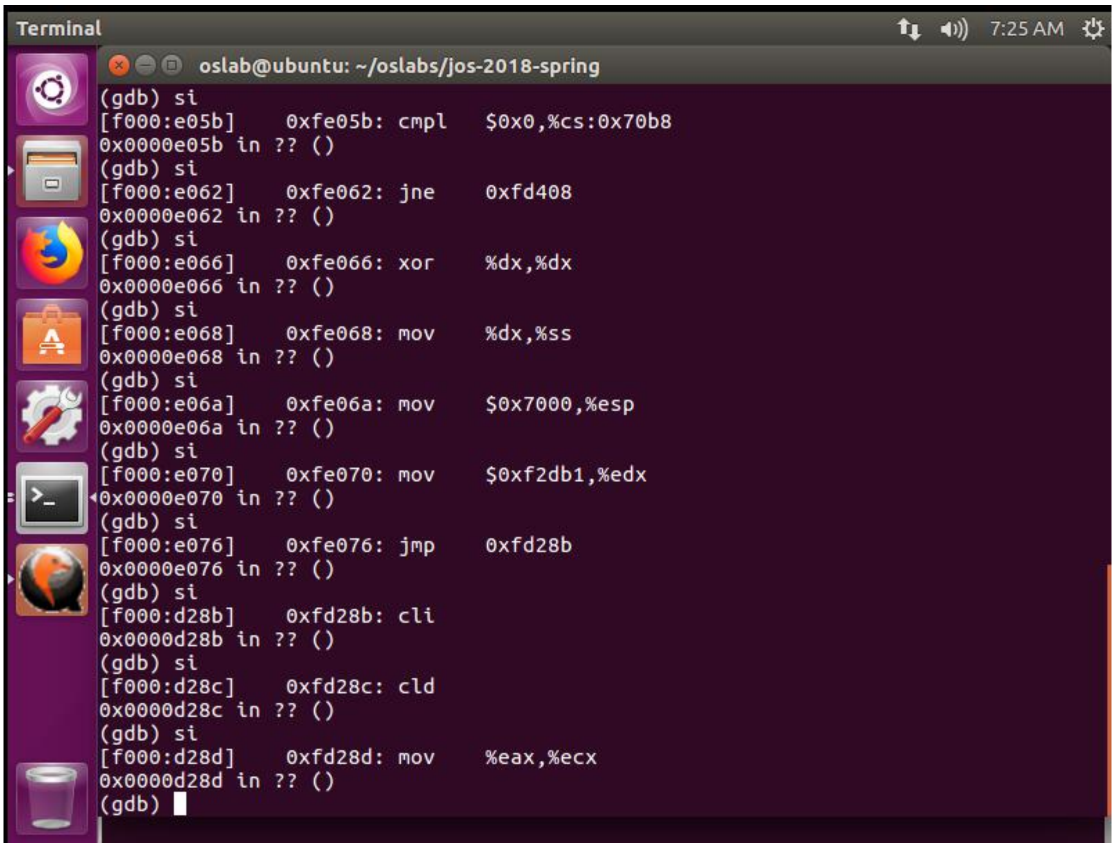
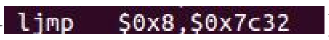
Part 1:
这部分讲的是机子从通电到加载 bootloader 的过程。

启动过程:
计算机刚通电时,设置 CS=0xf000,IP=0xfff0 (CS,IP) 根据实模式 physical address = 16 * segment + offset = 16 * cs + ip 所以机子起始执行的物理地址为 0xffff0 这是 BIOS 的起始地址,之后就开始执行 BIOS 代码,检查 VGA,加载中断向量表,中断服务程序,BIOS 数据区等。



```
(gdb) si
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x70b8
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xfd408
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %dx,%dx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %dx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a]    0xfe06a: mov    $0x7000,%esp
0x0000e06a in ?? ()
(gdb) si
[f000:e070]    0xfe070: mov    $0xf2db1,%edx
0x0000e070 in ?? ()
(gdb) si
[f000:e076]    0xfe076: jmp    0xfd28b
0x0000e076 in ?? ()
(gdb) si
[f000:d28b]    0xfd28b: cli
0x0000d28b in ?? ()
(gdb) si
[f000:d28c]    0xfd28c: cld
0x0000d28c in ?? ()
(gdb) si
[f000:d28d]    0xfd28d: mov    %eax,%ecx
0x0000d28d in ?? ()
(gdb) 
```

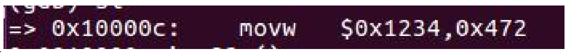
Part 2:
这部分就是加载 bootloader 了。
当 BIOS 执行完自检等操作之后,计算机硬件体系结构的设计与 BIOS 联手操作,会让 CPU 接收到一个 int 0x19 中断,CPU 接收到这个中断之后,会立即在中断向量表中找到 int 0x19 这个中断向量。此时,CPU 指向 int 0x19 这个中断向量所对应的中断服务程序的入口地址,这个中断服务程序的作用就是把软盘(硬盘)的第一个扇区中的程序(bootsect)加载到内存中的指定位置(0x7c00 -- 0x7dff)。接下来就开始执行这部分的代码。
我们的 lab 对应的是先执行 boot.s 再执行 main.c
我们来看看 boot.s 做了些什么,从代码我们可以看到 boot.s 首先还是在实模式下执行,它先关闭了中断,打开了代码自增,设置段寄存器,打开 A20(注:什么是 A20?我上网查了一下,如果它为零,则比特 20 及以上地址都被清除,打开 A20 就是为了实现兼容性),接下来就是转到保护模式,转到保护模式之后,地址就变成虚地址,需要在 GDT 表里面查对应的物理地址。然后设置保护模式下的段寄存器,栈指针,boot.s 的最后一步是 call bootmain,就是调用执行 main.c 了。

接着我们看看 main.c 做了些什么操作。
内核是以 elf 的格式存储的，所以加载内核实际上就是加载 elf 的每个 program seg
首先 readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);读取 elf 文件的头，接着一个 for 循环加载 elf 文件的每个段，最后((void (*)(void)) (ELFHDR->e_entry))();从 elf 的入口开始执行。我读了一下 readsec（）和 readsect（），这两个函数具体执行了，把内核从磁盘读到内存上面的操作。
回答问题：
At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
在 boot.s 里面 movl %eax , %cr0（下一个指令

```
ljmp    $0x8,$0x7c32
```

） 被执行之后开始执行 32 模式。
What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
我看 boot.asm 里面 main 函数最后一条指令是

```
// call the entry point from the ELF header
// note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
    7d6b:    ff 15 18 00 01 00        call    *0x10018
```

我 b *0x7d6b 然后 continue 下一条指令是

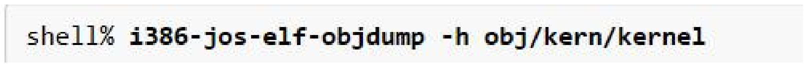
```
=> 0x10000c:    movw   $0x1234,0x472
```

这就是内核执行的第一条指令。

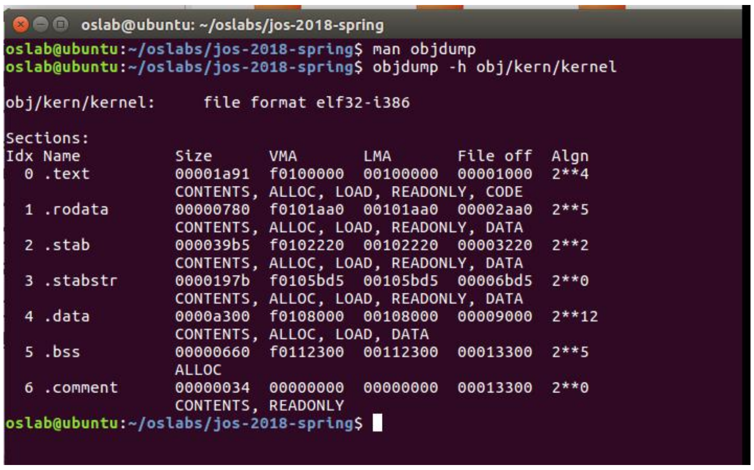
How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

```
// read 1st page off disk
readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
```

ELFHDR->e_phnum program 的总数，也就是要读的扇区。

加载内核：
这里首先介绍了 elf 文件格式。ELF 文件由 4 部分组成，分别是 ELF 头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。每个段也有自己的格式。
下面这条指令可以显示段的 name，size 和 linkaddress。


```
shell% i386-jos-elf-objdump -h obj/kern/kernel
```



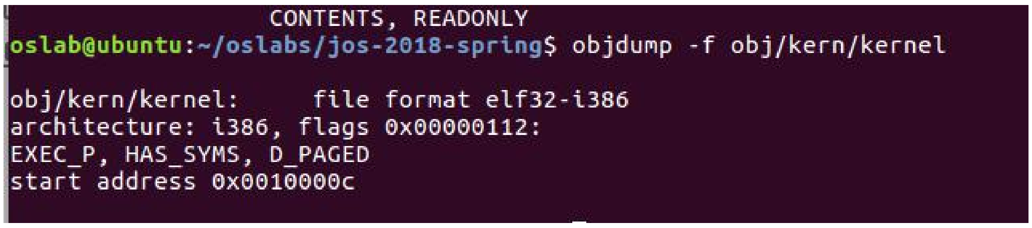
```
oslab@ubuntu: ~/oslabs/jos-2018-spring
oslab@ubuntu:~/oslabs/jos-2018-spring$ man objdump
oslab@ubuntu:~/oslabs/jos-2018-spring$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001a91  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000780  f0101aa0  00101aa0  00002aa0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000039b5  f0102220  00102220  00003220  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      0000197b  f0105bd5  00105bd5  00006bd5  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      00000034  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
oslab@ubuntu:~/oslabs/jos-2018-spring$ 
```

下面这条指令可以显示 e_entry()即 elf 文件的入口地址。

```
shell% i386-jos-elf-objdump -f obj/kern/kernel
```

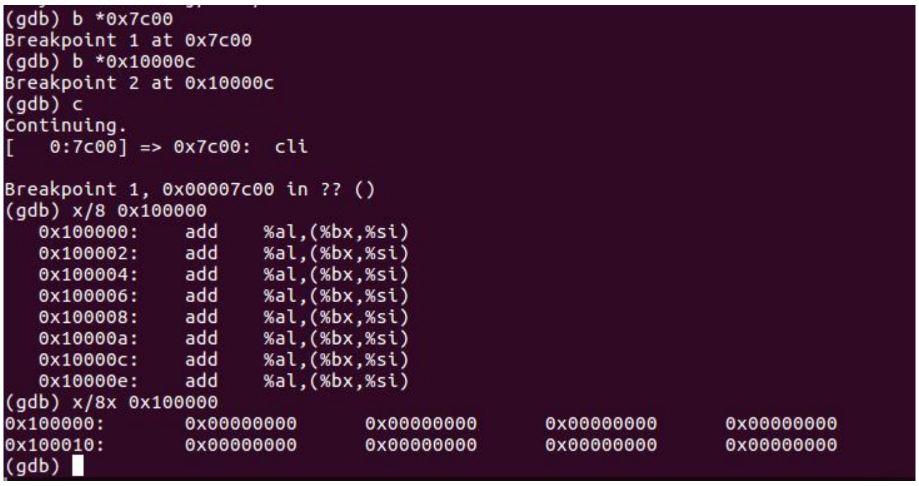


```
                  CONTENTS, READONLY
oslab@ubuntu:~/oslabs/jos-2018-spring$ objdump -f obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

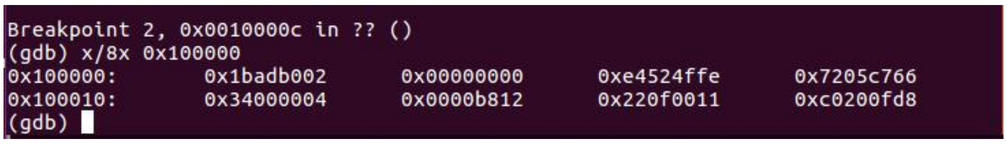
Exercise 5. Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

我在 bootloader 起始地址 0x7c00 跟 kernel 起始地址 0x10000c 处下断点，然后查看 0x100000 后面 8geword 的值，发现刚开始执行 bootloader 时，0x100000 后面为空，等到 bootloader 执行结束，开始执行内核时，0x100000 后面有了值。这说明 bootloader 加载了内核。



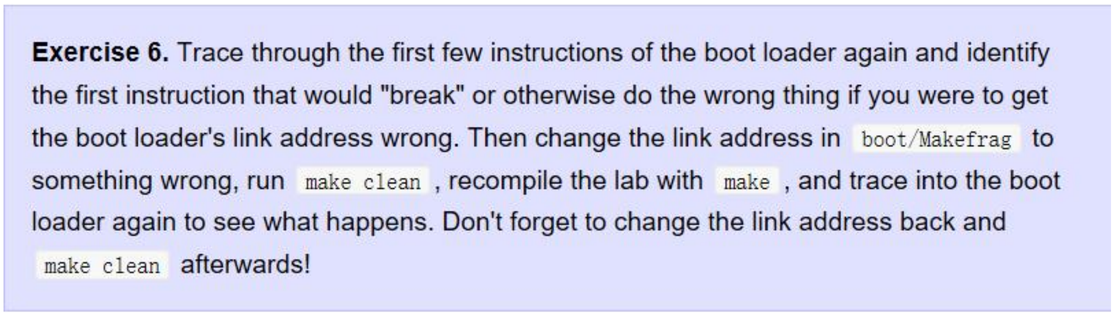
```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8 0x100000
   0x100000:    add    %al,(%bx,%si)
   0x100002:    add    %al,(%bx,%si)
   0x100004:    add    %al,(%bx,%si)
   0x100006:    add    %al,(%bx,%si)
   0x100008:    add    %al,(%bx,%si)
   0x10000a:    add    %al,(%bx,%si)
   0x10000c:    add    %al,(%bx,%si)
   0x10000e:    add    %al,(%bx,%si)
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) 
```



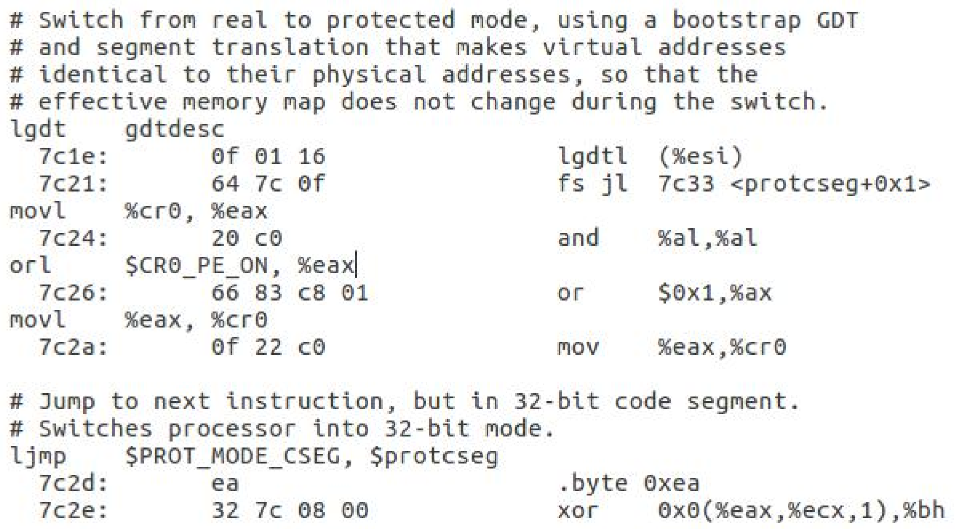
```
Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb) 
```

链接与加载地址：
链接地址与加载地址不一样，一个是虚地址，一个是物理地址，这两者之间差了一个kernelbase。



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!

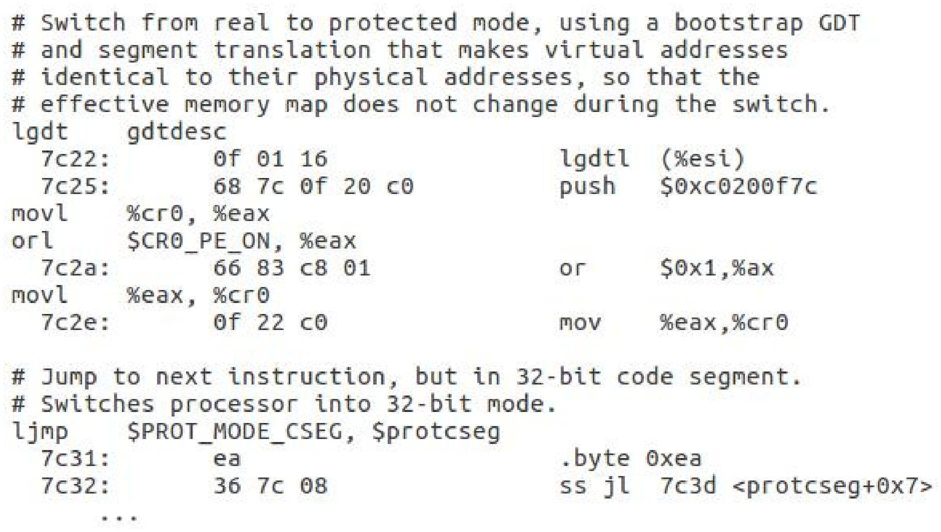
前几条指令地址错误的话肯定就不是赋值指令，肯定是 jmp，call，或者调用摸个具体地址的值。
我改了$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C04 -o $@.out $^
相较于之前的$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
然后我对比前后两个版本的 boot.asm
我发现在指令 ljmp $PROT_MODE_CSEG, $protcseg 执行之前，前后两个版本的指令都一样，只是位置往后挪了 4，但是 ljmp $PROT_MODE_CSEG, $protcseg 执行之后，指令就变了
之前是



```
# 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
    7c1e:       0f 01 16                lgdtl  (%esi)
    7c21:       64 7c 0f                fs jl  7c33 <protcseg+0x1>
movl    %cr0, %eax
    7c24:       20 c0                   and    %al,%al
orl     $CR0_PE_ON, %eax
    7c26:       66 83 c8 01             or     $0x1,%ax
movl    %eax, %cr0
    7c2a:       0f 22 c0                mov    %eax,%cr0

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
    7c2d:       ea                      .byte 0xea
    7c2e:       32 7c 08 00             xor    0x0(%eax,%ecx,1),%bh
```

之后是



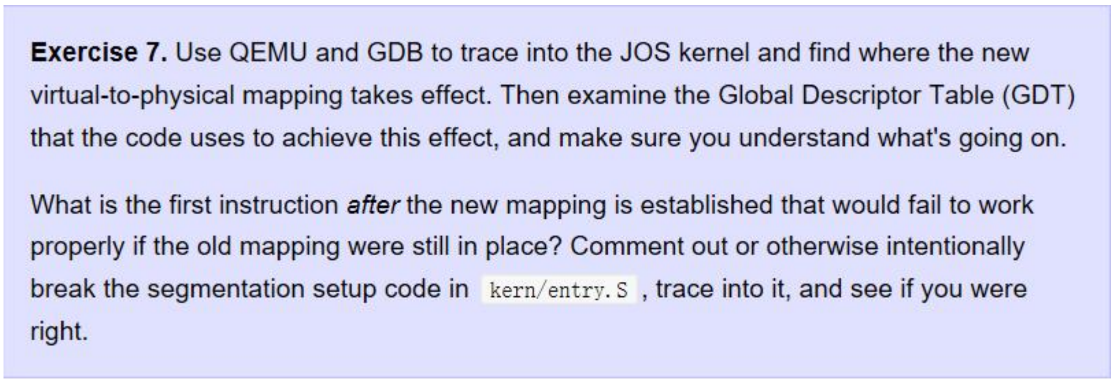
```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
lgdt    gdtdesc
  7c22:       0f 01 16                lgdtl  (%esi)
  7c25:       68 7c 0f 20 c0          push   $0xc0200f7c
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
  7c2a:       66 83 c8 01             or     $0x1,%ax
movl    %eax, %cr0
  7c2e:       0f 22 c0                mov    %eax,%cr0

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
  7c31:       ea                      .byte 0xea
  7c32:       36 7c 08                ss jl  7c3d <protcseg+0x7>
      ...
```

在 设置CR0_PG位以前 都是把地址访问当做物理地址对待,一旦该位被设置,地址访问将被**硬件**视为虚拟地址,需要进行转换。当目标代码不包含以这种方式对链接地址进行编码的绝对地址时，我们说该代码是位置无关的：无论它在何处加载，它都会正确运行。

Part 3:内核

这里首先将整个PC的物理地址空间的底部256MB（从0x00000000到0x0fffffff）分别映射到虚拟地址0xf0000000到0xffffffff。

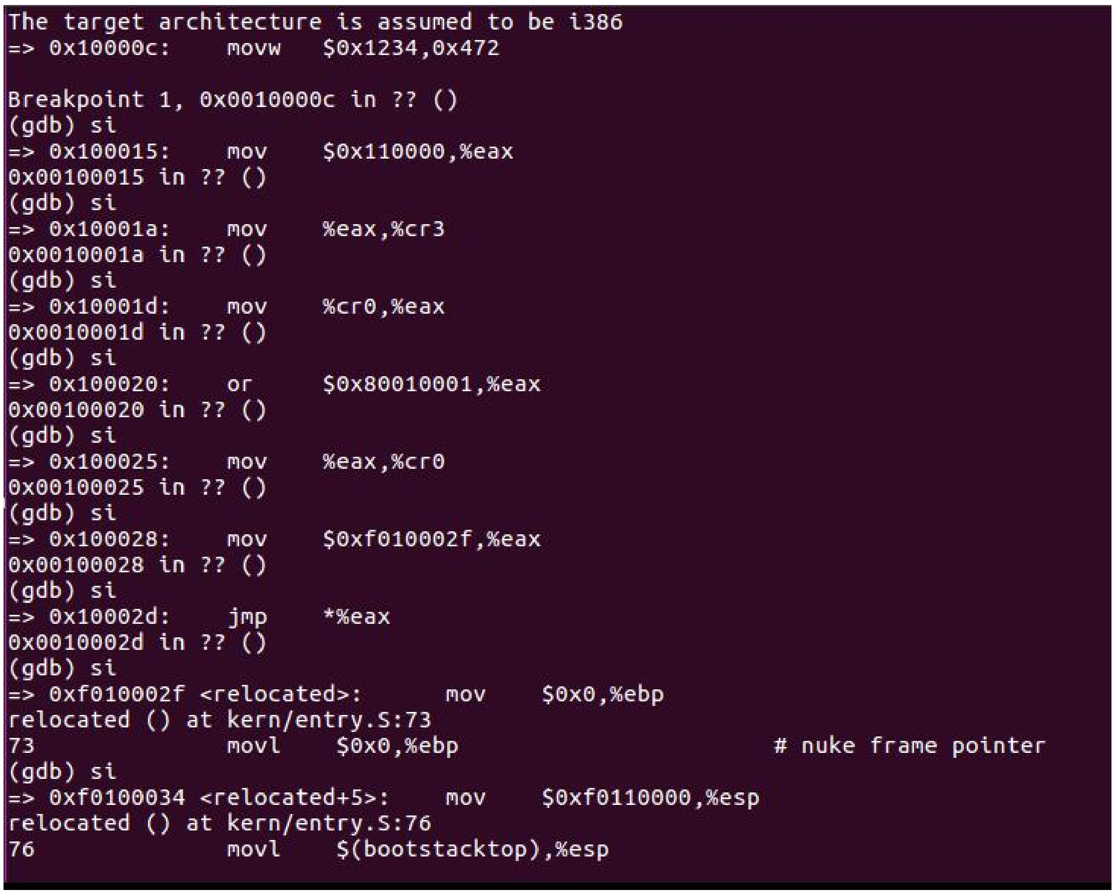


Exercise 7. Use QEMU and GDB to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.

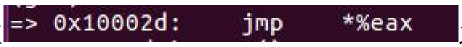
What is the first instruction *after* the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in `kern/entry.S`, trace into it, and see if you were right.

这里GDT被用来将虚拟地址转换为物理地址。

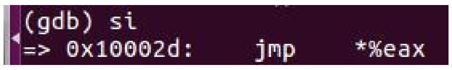
我在0x10000c断点，然后si，如下


```
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 1, 0x0010000c in ?? ()
(gdb) si
=> 0x100015:    mov    $0x110000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
(gdb) si
=> 0xf010002f <relocated>:      mov    $0x0,%ebp
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
=> 0xf0100034 <relocated+5>:    mov    $0xf0110000,%esp
relocated () at kern/entry.S:76
76              movl    $(bootstacktop),%esp
```

可以看到在
`=> 0x10002d:    jmp    *%eax`
之后，地址就改变了

显示地址改变 relocated 在 kern/entry.S:73
entry 里面的显示内核的 link 地址为 ~（KERNBASE+ 1Meg）这是个 virtual address，bootloader 实际上 load 的地址是 1Meg，也就是原来的 virtual address - KERNBASE

哪条指令会出错：
```
(gdb) si
=> 0x10002d:    jmp    *%eax
```
，这个跳转到虚地址的指令会出错。

格式化打印到控制台：
这里讲述了如何底层的实现 printf()函数。
我们先看看 prinf.c, printfmt.c, console.c 这三个函数。
首先看 printf.c 它是基于 printfmt.c 和 console.c 的，相对来说还不是最底层。printf.c 里面三个函数 putch，vcprintf,cprintf。
首先解释一下一些参数。

fmt 即格式化字符串

ap 除了字符串以外传给 printf 的参数

p 为%s 或%e 输出作为临时的 char *

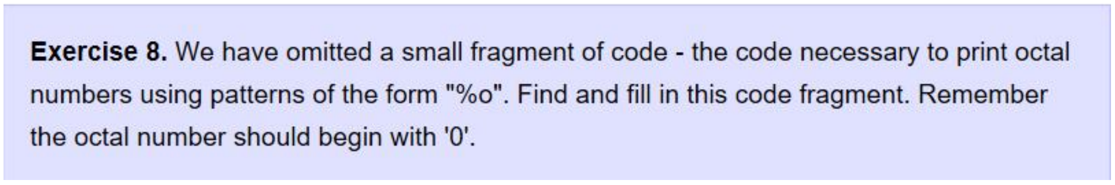
ch 当前从 fmt 取的字符

err %e 错误号

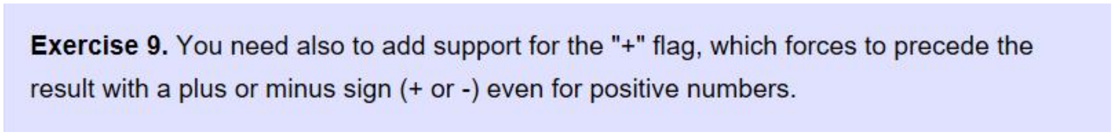
num 为数字输出的值

base 为数字输出的基/进制
lflag 为数字输出 long 或 long long 的标识
width 小数的宽度标识
precision 小数或字符串长度的具体值
altflag 为%#,表示输出字符串时是否需要把非 Printable characters 转换成? 和标准的 printf 的#不一样
padc 表示用来填补宽度所用的字符

putch(int ch, int *cnt)的作用是在屏幕上输出字符“ch”，然后光标向后一位。
cprintf(const char *fmt, ...)调用 vcprintf(const char *fmt, va_list ap)执行 print 的操作。
因为 print 的字符串里面可能会有其他的要求，例如“%d, %s”之类的，所以 vcprintf 调用了 vprintfmt 来处理。
我们接下来看 printfmt.c 函数。
printfmt.c 里面我关注的就是 vprintfmt(void (*putch)(int, void*), void *putdat, const char *fmt, va_list ap)函数。

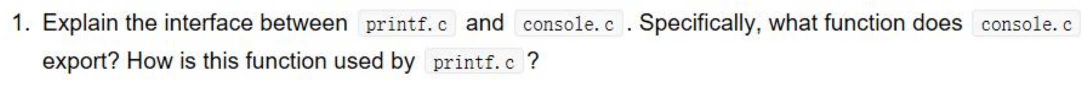

Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment. Remember the octal number should begin with '0'.

输出八进制。
这里我模仿了 case ‘u’的写法，getuint(&ap, lflag)的作用是获取“%o”的对应值。
为什么这个函数能实现八进制输出呢？关键在于 printnum(void (*putch)(int, void*), void *putdat,unsigned long long num, unsigned base, int width, int padc) 这个函数，不停地 num/base，然后循环调用自身。

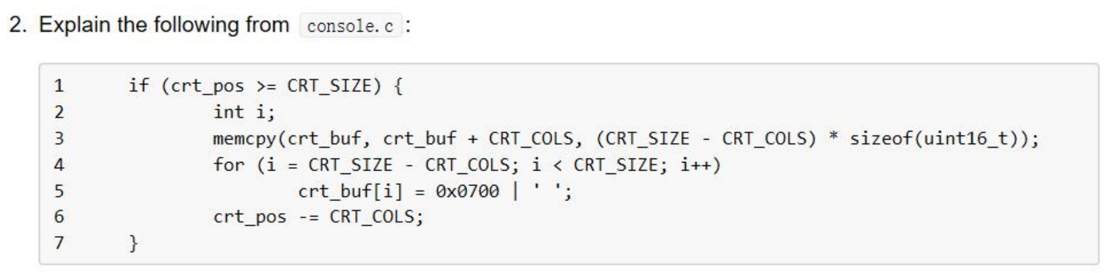

Exercise 9. You need also to add support for the "+" flag, which forces to precede the result with a plus or minus sign (+ or -) even for positive numbers.

这里我就直接来了一个判断。

回答问题：

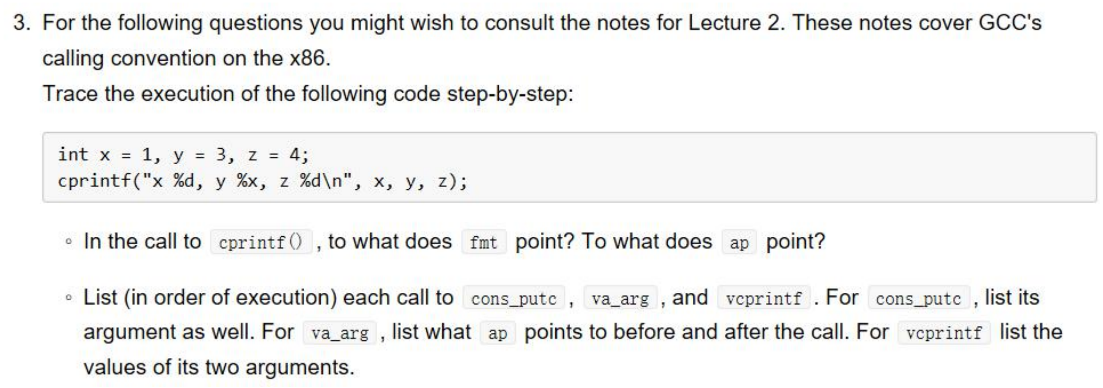

1. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?

console.c 最后有个注释“// `High'-level console I/O. Used by readline and cprintf.”这之后留下了三个接口， cputchar， getchar， iscons。
Printf.c 里面 putch 函数使用了 cputchar。


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;
    }
```

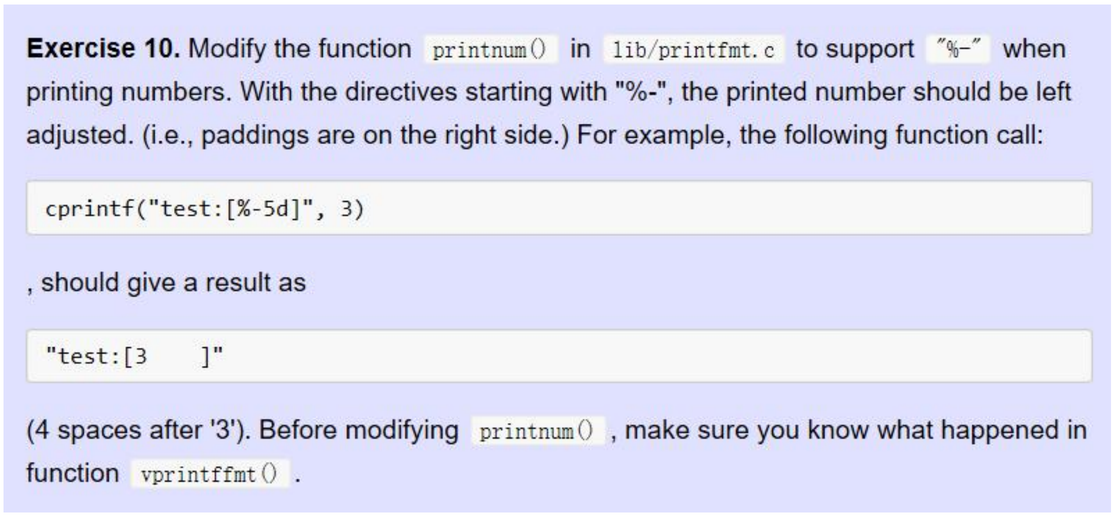
这部分在函数 cga_putc 里面，cag_putc 这个函数顾名思义是在设备上显示字符串。所以根据代码推断，只要大于 CRT_SIZE 就操作，而且最后有个 crt_pos -= CRT_COLS,我们看之前的代码 case '\n': crt_pos += CRT_COLS;所以我推断上述代码的目的是，当显示内容超出一定宽度，就换行。


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.

fmt 指要输出的字符串， ap 指后面的参数 xyz。


Exercise 10. Modify the function `printnum()` in `lib/printfmt.c` to support `"%-"` when printing numbers. With the directives starting with "%-", the printed number should be left adjusted. (i.e., paddings are on the right side.) For example, the following function call:

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

, should give a result as

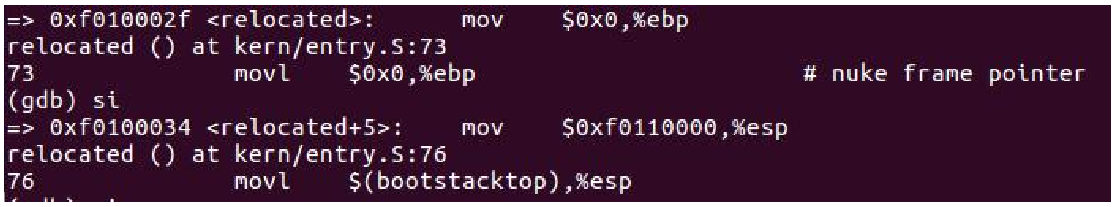
```
"test:[3    ]"
```

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

这个目的就是把空格从左边移到右边。修改 printnum 的关键就是确定什么时候可以开始打印空格，因为 printnum 执行过程中会不停的调用自身，而输出空格只能输出一次。这里因为 printnum 的 num 参数会不停变化，所以我记录下 num 的初始值，当 num 为初始值时，打印空格，保证了空格只打印一次。

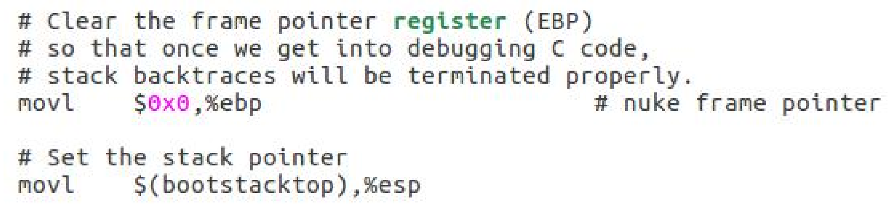
Exercise 11. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

初始化栈：



```
=> 0xf010002f <relocated>:      mov    $0x0,%ebp
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
=> 0xf0100034 <relocated+5>:    mov    $0xf0110000,%esp
relocated () at kern/entry.S:76
76              movl    $(bootstacktop),%esp
```

在 entry.s 的

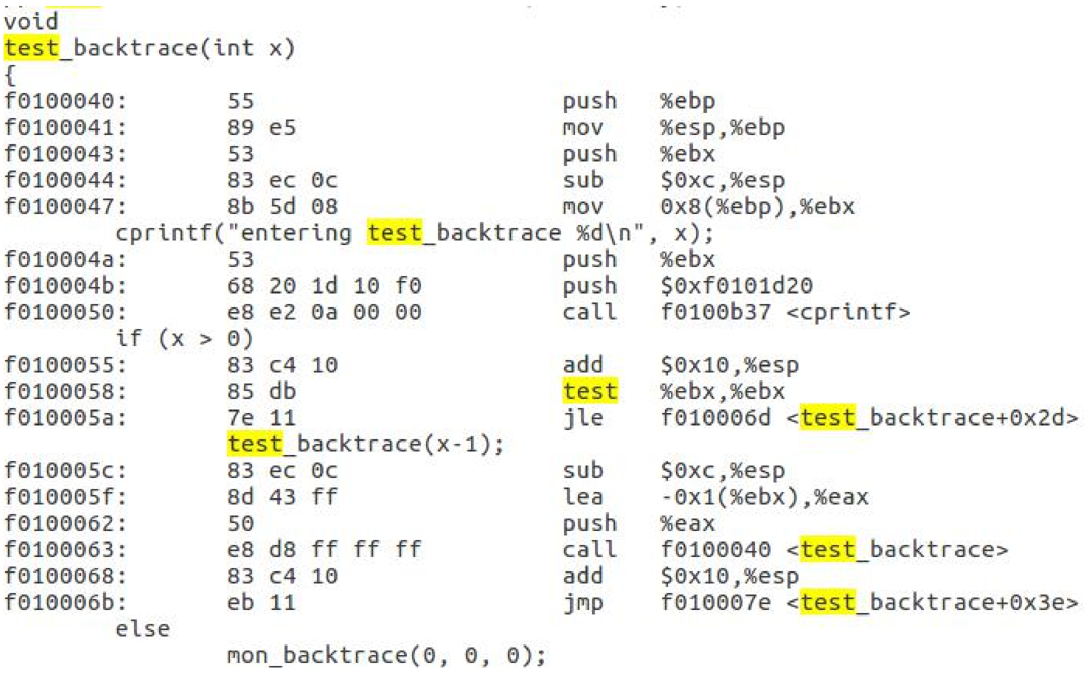


```
 # Clear the frame pointer register (EBP)
 # so that once we get into debugging C code,
 # stack backtraces will be terminated properly.
 movl    $0x0,%ebp                      # nuke frame pointer

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

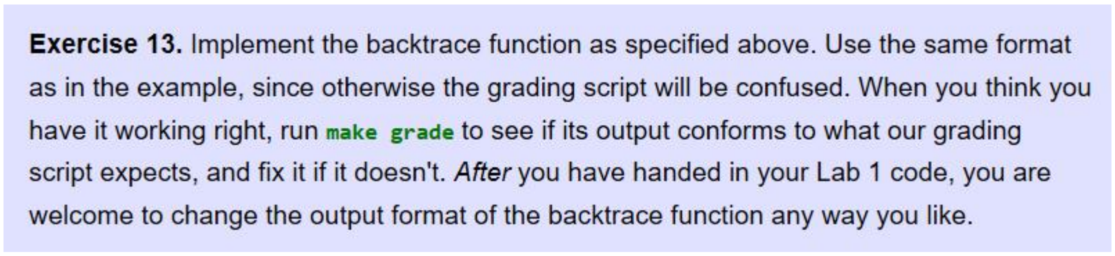
栈的位置在 0xf0110000 指向地址较大的那一端。

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?

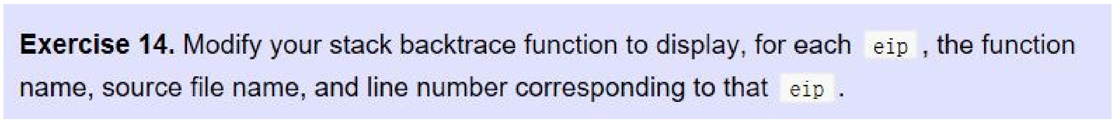


```
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 0c                sub    $0xc,%esp
f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
        cprintf("entering test_backtrace %d\n", x);
f010004a:       53                      push   %ebx
f010004b:       68 20 1d 10 f0          push   $0xf0101d20
f0100050:       e8 e2 0a 00 00          call   f0100b37 <cprintf>
        if (x > 0)
f0100055:       83 c4 10                add    $0x10,%esp
f0100058:       85 db                   test   %ebx,%ebx
f010005a:       7e 11                   jle    f010006d <test_backtrace+0x2d>
                test_backtrace(x-1);
f010005c:       83 ec 0c                sub    $0xc,%esp
f010005f:       8d 43 ff                lea    -0x1(%ebx),%eax
f0100062:       50                      push   %eax
f0100063:       e8 d8 ff ff ff          call   f0100040 <test_backtrace>
f0100068:       83 c4 10                add    $0x10,%esp
f010006b:       eb 11                   jmp    f010007e <test_backtrace+0x3e>
        else
                mon_backtrace(0, 0, 0);
```

4+4+12+4+4+4=32


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.

首先我们看 x86.h 里面的 read_ebp()函数。read_ebp 函数返回一个 ebp 的指向栈的指针，eip 是函数的返回地址指针，所以 eip 在 ebp 之前，ebp 后面五个就是五个参数的位置

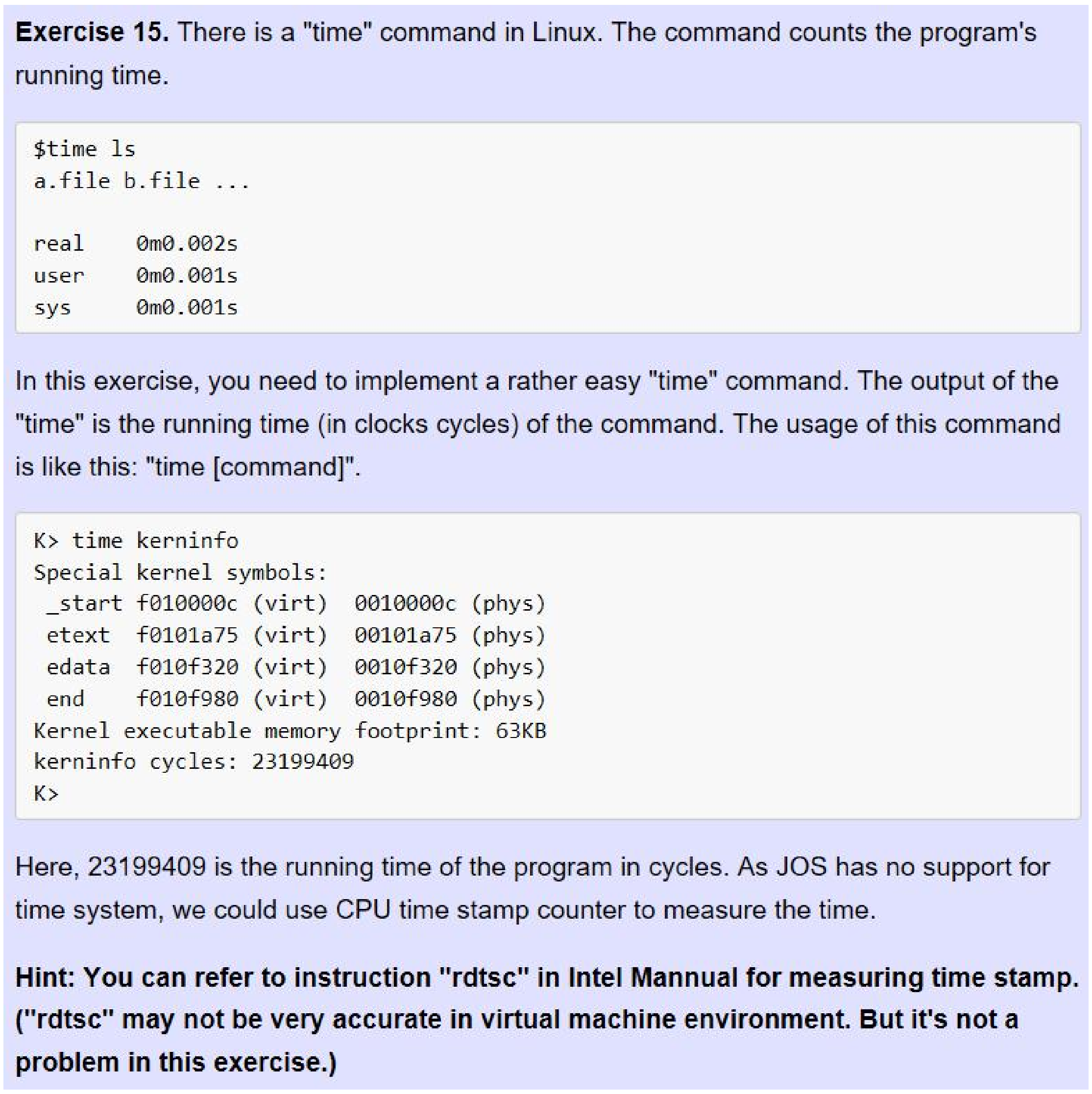

Exercise 14. Modify your stack backtrace function to display, for each `eip`, the function name, source file name, and line number corresponding to that `eip`.

在 kdebug.c 里面加上搜索行数的代码，最主要的是 info->eip_line = stabs[lline].后面应该接什么，我看了 stab.h

```
uint32_t n_strx;      // index into string table of name
uint8_t n_type;              // type of symbol
uint8_t n_other;             // misc info (usually empty)
uint16_t n_desc;             // description field
uintptr_t n_value;   // value of symbol
```

除了 desc 其他都出现过了，所以我猜就是这个。

之后就是在 mon_backtrace 里面加入 debuginfo。



Exercise 15. There is a "time" command in Linux. The command counts the program's running time.

```
$time ls
a.file b.file ...

real    0m0.002s
user    0m0.001s
sys     0m0.001s
```

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.

Hint: You can refer to instruction "rdtsc" in Intel Mannual for measuring time stamp. ("rdtsc" may not be very accurate in virtual machine environment. But it's not a problem in this exercise.)

加入 mon_time 函数，只要查一下 rdtsc 的用法，前后减一下就可以。