实验报告

lab1

姓名：朱瑞媛

学号：14307130373

注意事项：

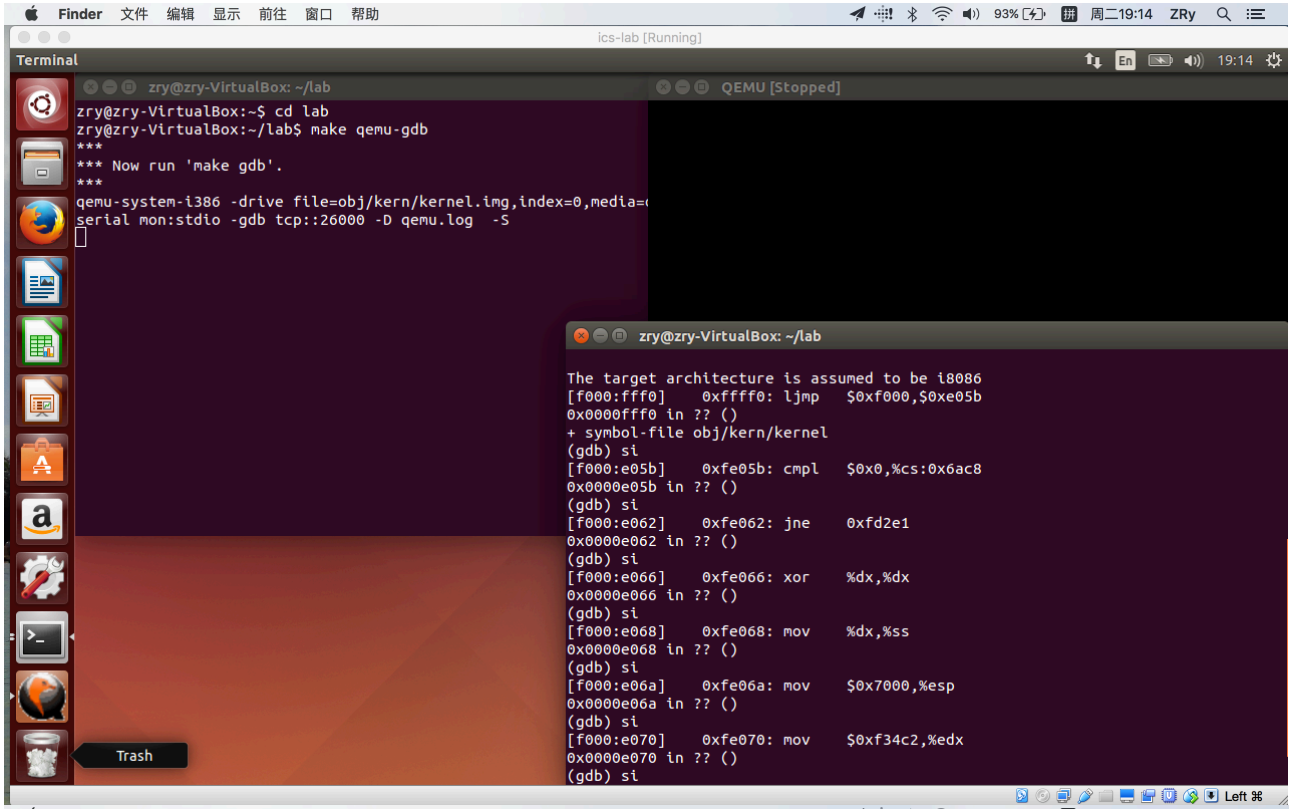
1.请在每个 exercise 之后简要叙述实验原理，详细描述实验过程。
2.请将你认为的关键步骤附上必要的截图。
3.有需要写代码的实验，必须配有代码、注释以及对代码功能的说明。
4.你还可以列举包括但不局限于以下方面：实验过程中碰到的问题、你是如何解决的、实验之后你还留有哪些疑问和感想。

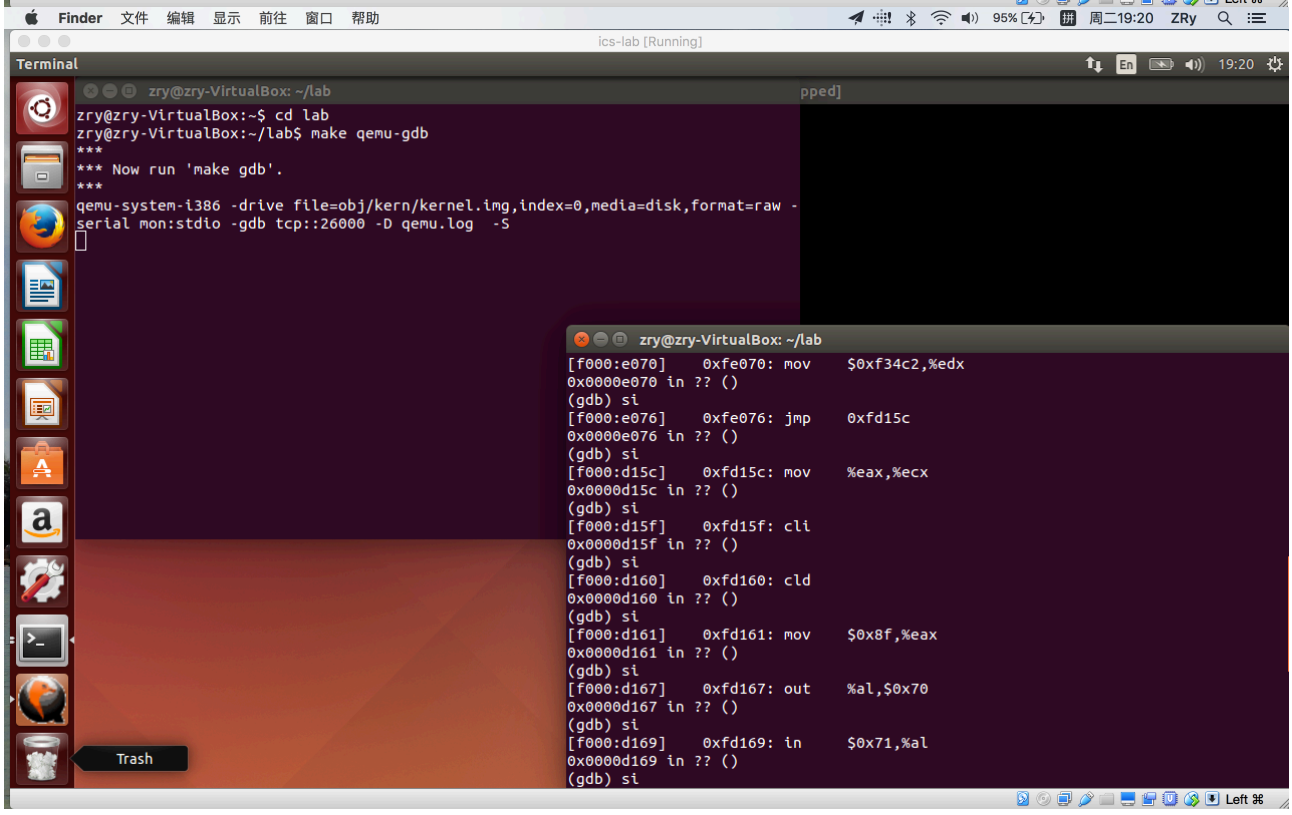
5.请在截止日期前将代码和报告上传到 ftp 的指定目录下，文件名为 os_lab1_学号.zip，该压缩文件中应包含实验报告和代码，其中实验报告格式为 pdf，置于压缩文件的根目录。
6.如果实验附有 question，请在对应 exercise 后作答，这是实验报告评分的重要部分。
7.Challenge 为加分选作题。每个 lab 可能有多个 challenge，我们会根据完成情况以及难度适当加分，这部分的实验过程描述应该比 exercise 更加详细。

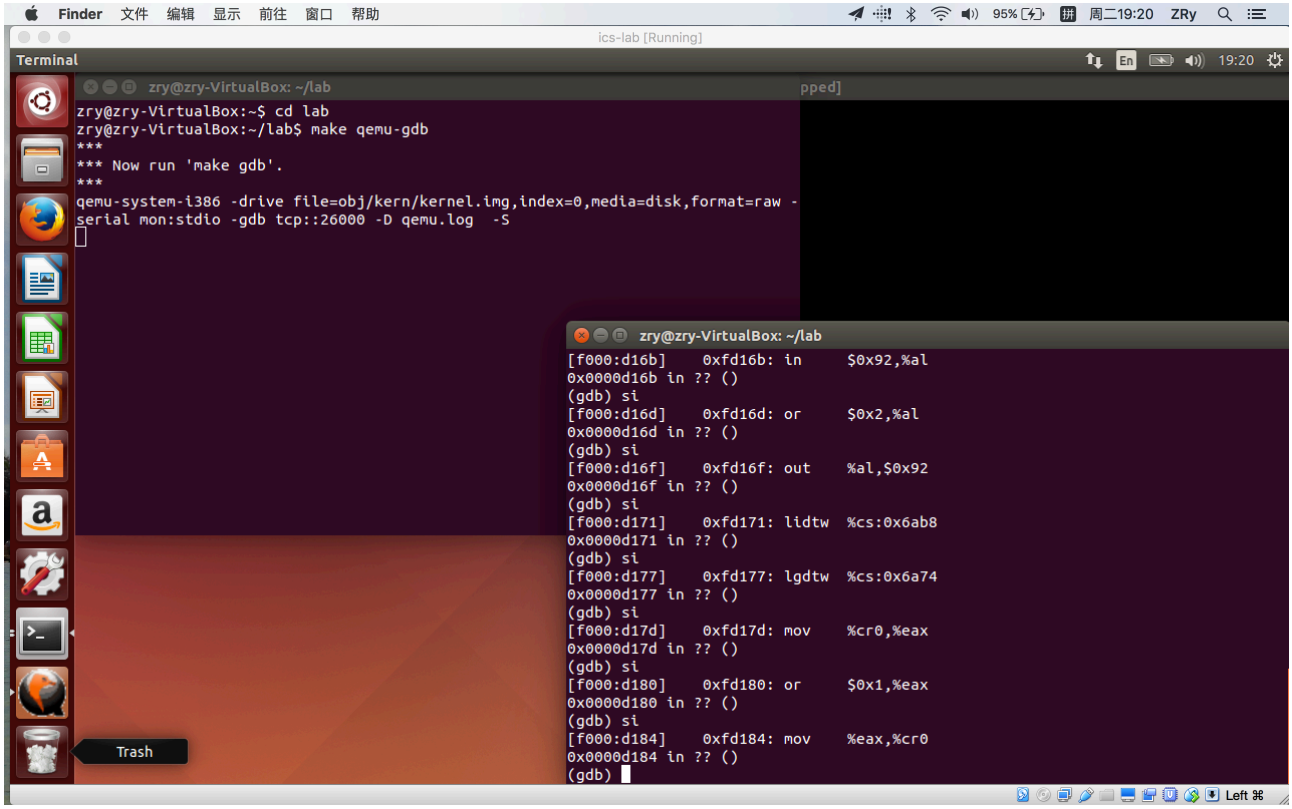
Exercise 1. Familiarize yourself with the assembly language materials available on the 6.828 reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.
We do recommend reading the section "The Syntax" in Brennan's 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.

阅读了 Exercise 中提到的“The Syntax”部分，对于 GNU assembler 中需要的语法有了一定的了解。在之后的过程中 reference page 给了我很大的帮助。

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


Finder 文件 编辑 显示 前往 窗口 帮助
93% 周二19:14 ZRy
ics-lab [Running]
Terminal
19:14
zry@zry-VirtualBox: ~/lab
QEMU [Stopped]
zry@zry-VirtualBox:~$ cd lab
zry@zry-VirtualBox:~/lab$ make qemu-gdb
***
*** Now run 'make gdb'.
***
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=
serial mon:stdio -gdb tcp::26000 -D qemu.log -S
zry@zry-VirtualBox: ~/lab
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) si
[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x6ac8
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xfd2e1
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 $0xf34c2,%edx
0x0000e070 in ?? ()
(gdb) si
Trash
Left ⌘


Finder 文件 编辑 显示 前往 窗口 帮助
95% 周二19:20 ZRy
ics-lab [Running]
Terminal
19:20
zry@zry-VirtualBox: ~/lab
zry@zry-VirtualBox:~$ cd lab
zry@zry-VirtualBox:~/lab$ make qemu-gdb
***
*** Now run 'make gdb'.
***
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log -S
zry@zry-VirtualBox: ~/lab
[f000:e070] 0xfe070: mov $0xf34c2,%edx
0x0000e070 in ?? ()
(gdb) si
[f000:e076] 0xfe076: jmp 0xfd15c
0x0000e076 in ?? ()
(gdb) si
[f000:d15c] 0xfd15c: mov %eax,%ecx
0x0000d15c in ?? ()
(gdb) si
[f000:d15f] 0xfd15f: cli
0x0000d15f in ?? ()
(gdb) si
[f000:d160] 0xfd160: cld
0x0000d160 in ?? ()
(gdb) si
[f000:d161] 0xfd161: mov $0x8f,%eax
0x0000d161 in ?? ()
(gdb) si
[f000:d167] 0xfd167: out %al,$0x70
0x0000d167 in ?? ()
(gdb) si
[f000:d169] 0xfd169: in $0x71,%al
0x0000d169 in ?? ()
(gdb) si
Trash
Left ⌘



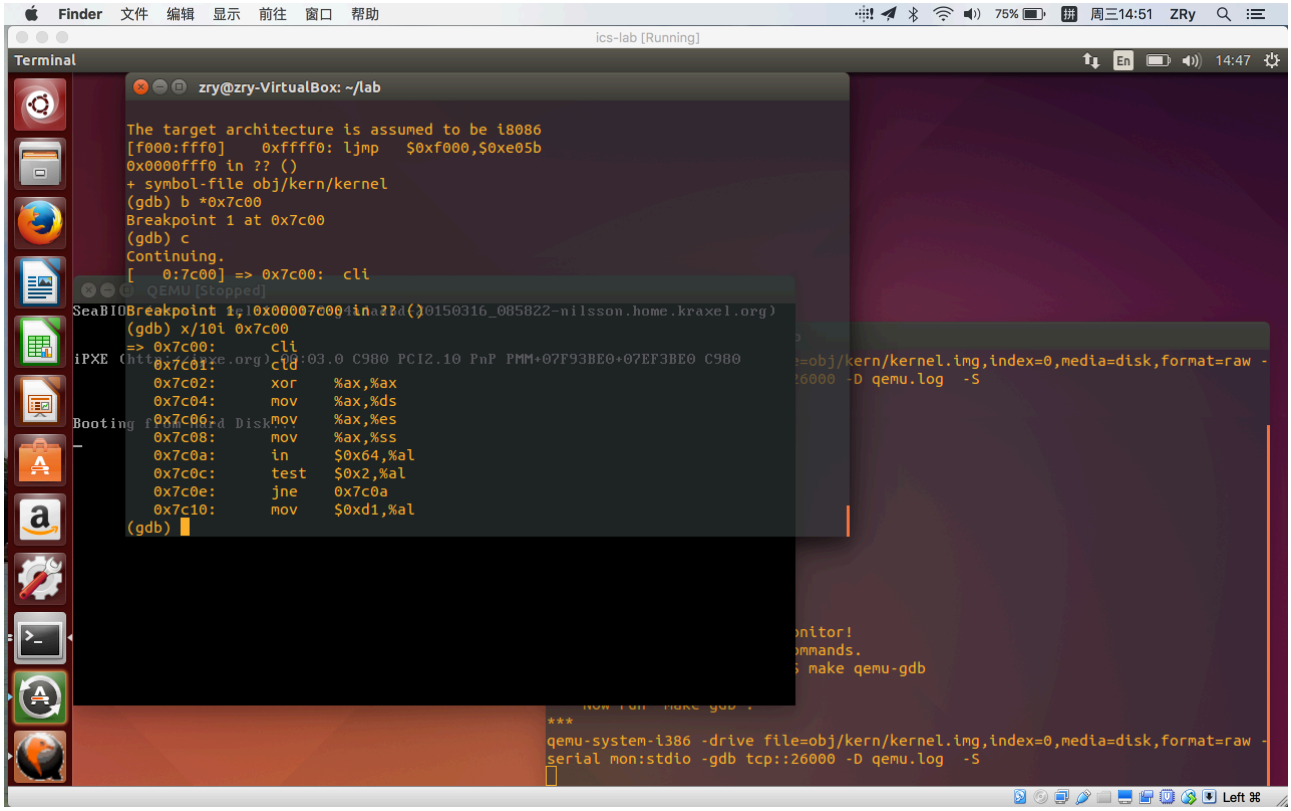
0x0000fff0:
跳转指令，跳转到 0xfe05b 地址处。
0x0000e05b：把 0x0 这个立即数和$cs:0x6ac8 所代表的内存地址处的值比较。其中$cs 就代表 CS 段寄存器的值。
0x0000e062:
如果 ZF 标志位为 0 的时候跳转，即上一条指令 cmpl 的结果不是 0 时跳转，也就是$cs:0x6ac8 地址处的值不是 0x0 时跳转。
0x0000e066:
由地址可见上面的跳转指令并没有跳转。这条指令的功能是把 %dx 寄存器清零。
0x0000e068，0x0000e06a，0x0000e070:
这些指令就是设置一些寄存器的值。
0x0000e076:
跳转至 0x0000e15c。
0x0000d15c:
寄存器赋值。
0x0000d15f:
关闭中断指令。启动时的操作是比较关键的，所以肯定是不能被中断的。这个关中断指令用于关闭那些可以屏蔽的中断。
0x0000d160:
设置方向标识位为 0，表示后续的串操作比如 MOVS 操作，内存地址的变化方向，如果为 0 代表从低地址值变为高地址。
0x0000d161，0x0000d167，0x0000d169:
这三条命令中涉及到两个新的指令 out， in。这两个操作是用于操作 IO 端口 0x70，0x71.CPU 与外部设备通讯时，通常是通过访问，修改设备控制器中的寄存器来实现的。0x70 端口和 0x71 端口是用于控制系统中一个叫做 CMOS 的设备，这个设备是一个低功耗的存储设备，它可以用于在计算机关闭时存储一些信息。这三条指令是用来关闭 NMI 中断的。
0x0000d16b，0x0000d16d，0x0000d16f:
这三条命令是在控制端口 0x92. 这里的这个操作应该是去测试可用内存空间。
0x0000d171:
lidt 指令：加载中断向量表寄存器(IDTR)。这个指令会把从地址 0xf6ab8 起始的后面 6 个字节的数据读入到中断向量表寄存器(IDTR)中。

0x0000d177:
把从 0xf6a74 为起始地址处的 6 个字节的值加载到全局描述符表格寄存器中 GDTR 中。
0x0000d17d，0x0000d180，0x0000d184:
计算机中包含 CR0~CR3 四个控制寄存器，用来控制和确定处理器的操作模式。其中这三个语句的操作明显是要把 CR0 寄存器的最低位(0bit)置 1。CR0 寄存器的 0bit 是 PE 位，启动保护位，当该位被置 1，代表开启了保护模式。推测它在检测是否机器能工作在保护模式下。
综上，作为 PC 启动后运行的第一段程序，BIOS 最重要的功能是把操作系统从磁盘中导入内存，然后再把控制权转交给操作系统。

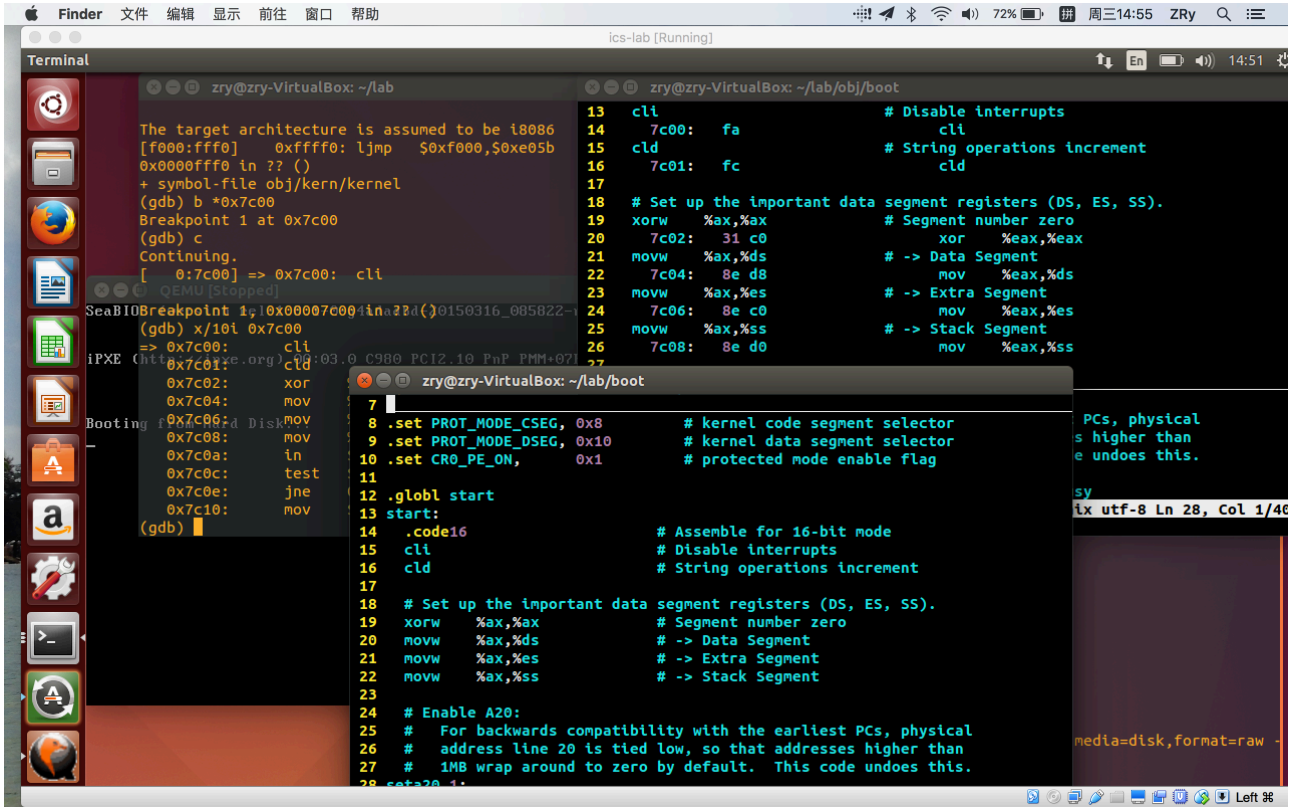
Exercise 3. Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

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.

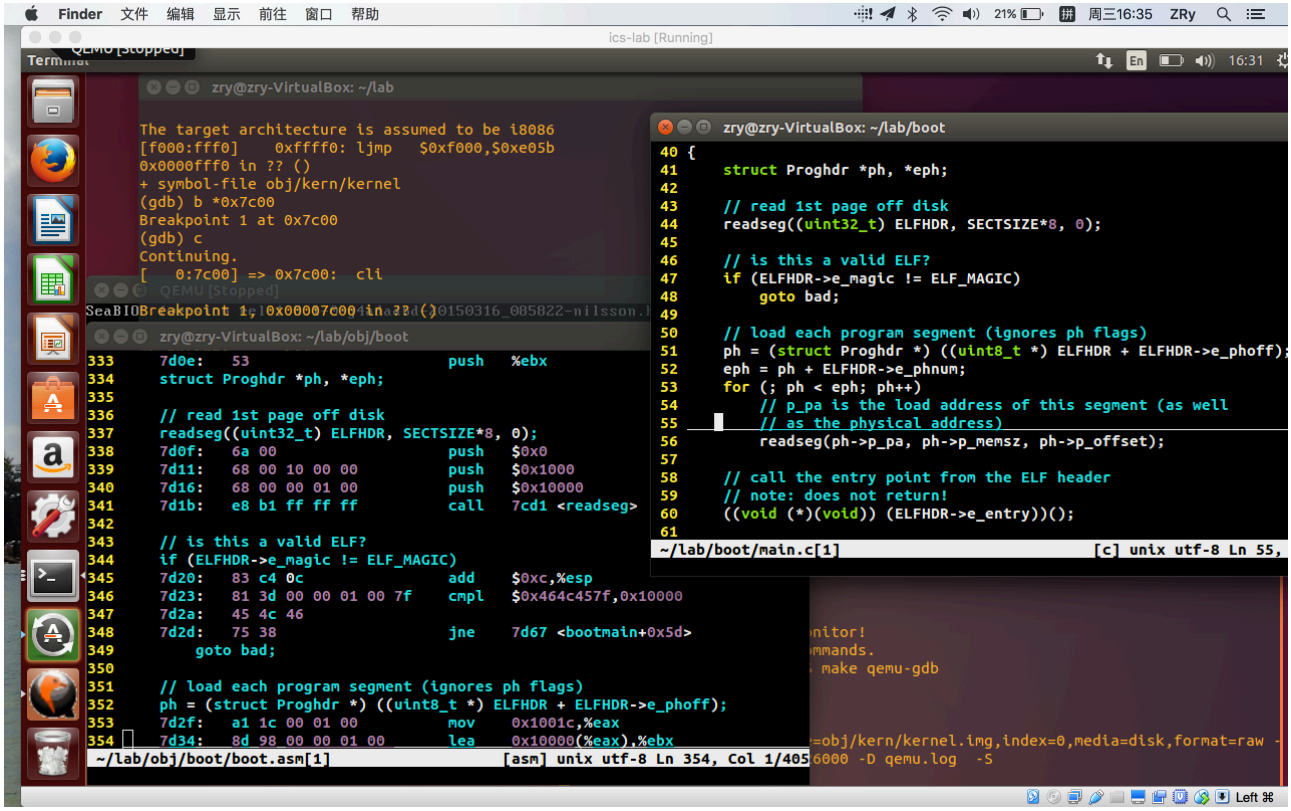


在 0x7c00 处设置 breakpoint，并获取之后的 10 步反汇编指令。

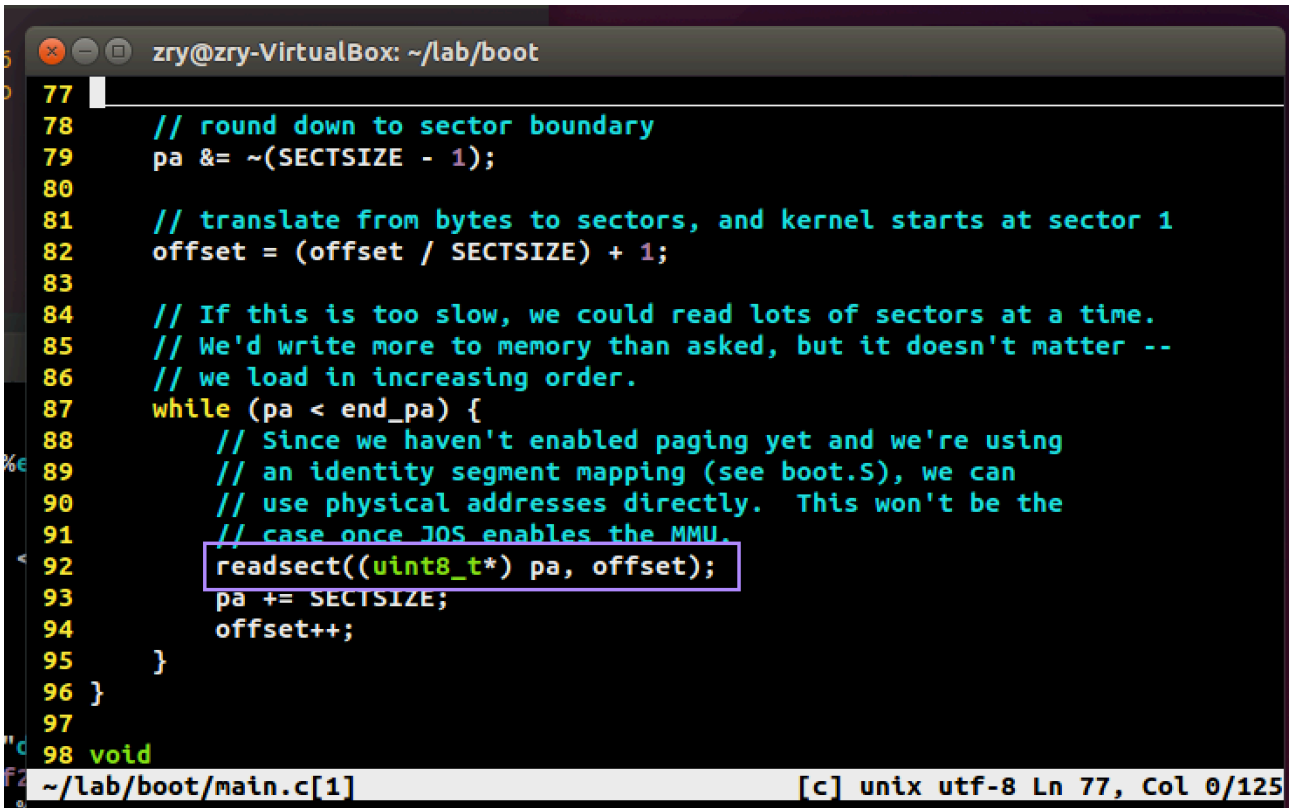


将它和 boot.S 以及 boot.asm 进行对比。

可见这三者在指令上没有区别，在源代码中，我们指定了很多标识符在被汇编成机器代码后都会被转换成真实物理地址。在 boot.asm 中还把这种对应关系列出来了，但是在真实执行时完全是真实物理地址。



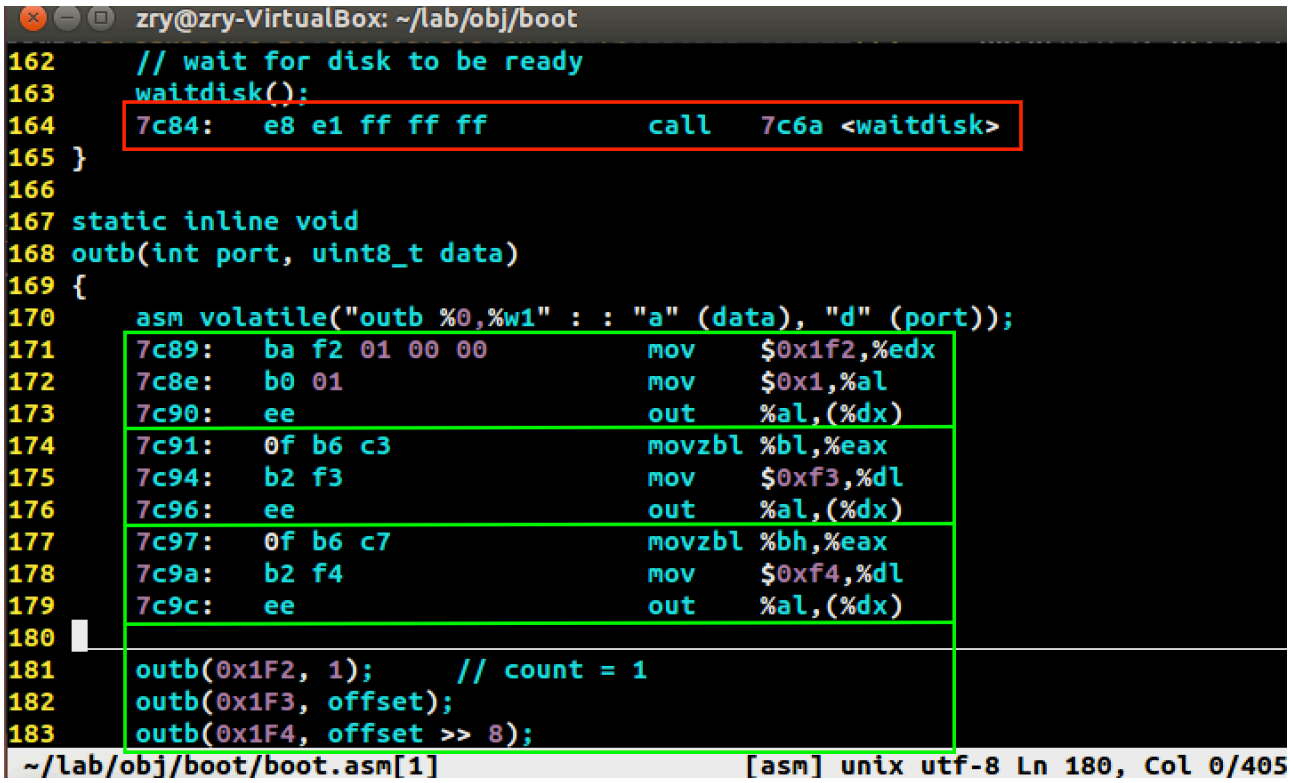
进入了 bootmain 后调用了 readseg 的 c 代码及汇编代码。

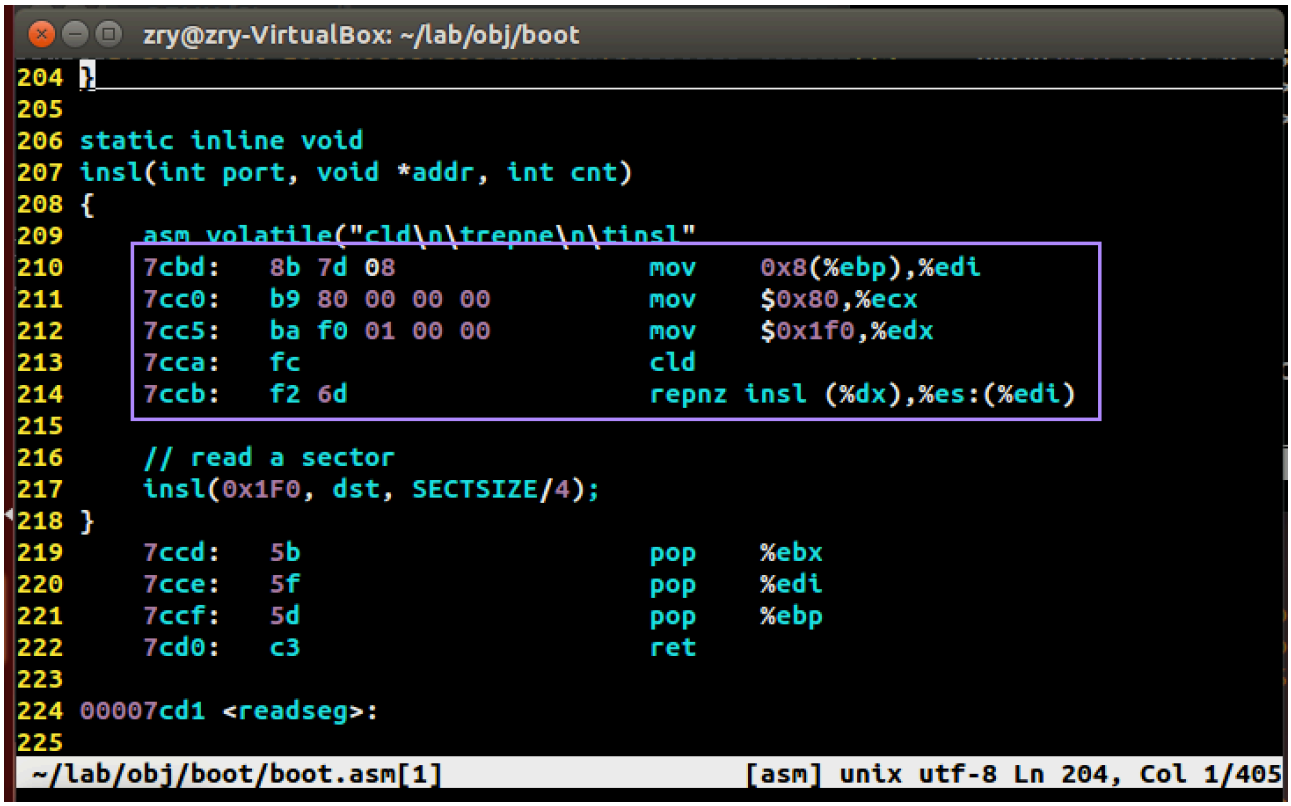


```
zry@zry-VirtualBox: ~/lab/obj/boot
03         // Since we haven't enabled paging yet and we're using
04         // an identity segment mapping (see boot.S), we can
05         // use physical addresses directly.  This won't be the
06         // case once JOS enables the MMU.
07         readsect((uint8_t*) pa, offset);
08     7cf9:   e8 7e ff ff ff          call   7c7c <readsect>
09         pa += SECTSIZE;
10         offset++;
11     7cfe:   58                      pop    %eax
12     7cff:   5a                      pop    %edx
13     7d00:   eb ea                   jmp    7cec <readseg+0x1b>
14     }
15 }
16     7d02:   8d 65 f4                lea    -0xc(%ebp),%esp
17     7d05:   5b                      pop    %ebx
18     7d06:   5e                      pop    %esi
19     7d07:   5f                      pop    %edi
20     7d08:   5d                      pop    %ebp
21     7d09:   c3                      ret
22
23 00007d0a <bootmain>:
24 void readsect(void*, uint32_t);
```

调用 readseg 函数。

```
zry@zry-VirtualBox: ~/lab/boot
104 }
105
106 void
107 readsect(void *dst, uint32_t offset)
108 {
109     // wait for disk to be ready
110     waitdisk();
111
112     outb(0x1F2, 1);     // count = 1
113     outb(0x1F3, offset);
114     outb(0x1F4, offset >> 8);
115     outb(0x1F5, offset >> 16);
116     outb(0x1F6, (offset >> 24) | 0xE0);
117     outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
118
119     // wait for disk to be ready
120     waitdisk();
121
122     // read a sector
123     insl(0x1F0, dst, SECTSIZE/4);
124 }
125
 ~/lab/boot/main.c[1]                          [c] unix utf-8 Ln 122, Col 1/125
```





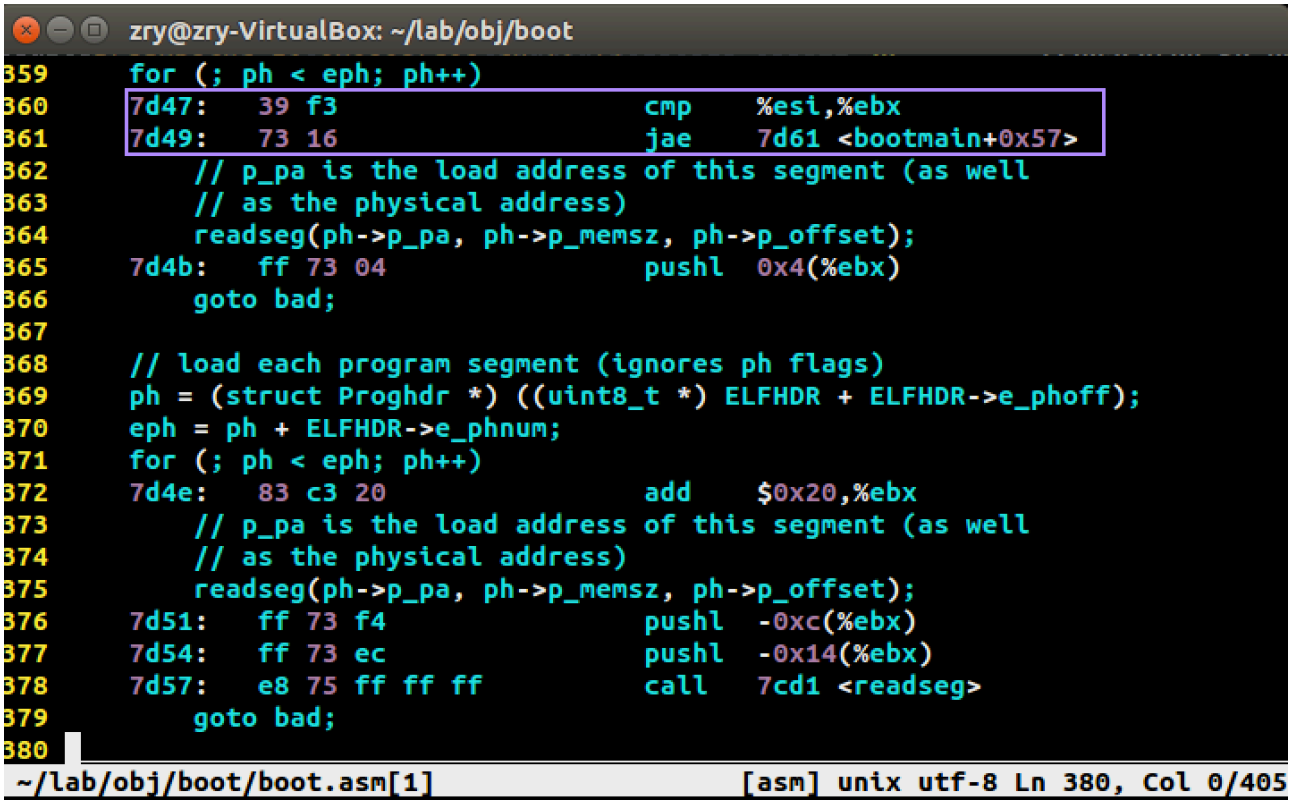
首先是过程调用时的修改栈帧等操作，然后调用 waitdisk()函数。这个函数用于查询当前磁盘的状态是否已经准备好进行操作。如果没有准备好，那么程序就会一直停在这里，直到磁盘准备好。

下一步是调用一系列的 outb 函数。outb 函数有两个参数，第 1 个参数是端口号，第 2 个参数是输入的值。整个函数的功能就是向该端口输出一个字节的数据。其中汇编语言中的每个绿色小框的指令对应一条 outb 函数。

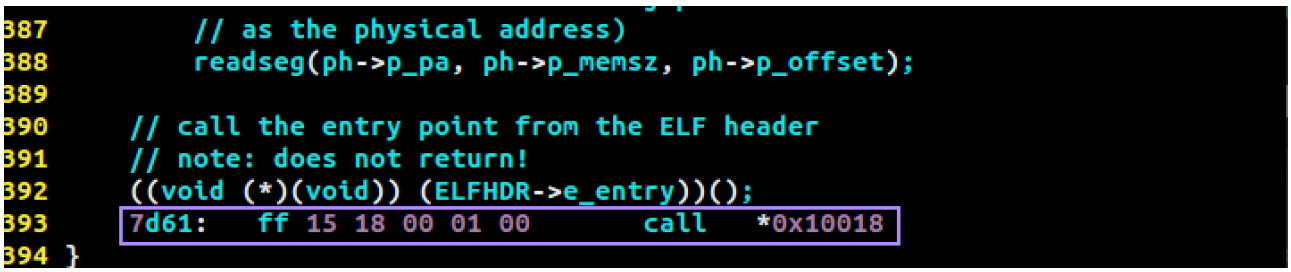
接下来的系统调用 waitdisk 过程来等待磁盘完成读取。waitdisk 退出后，代表数据已经被读取。

最后是函数 insl。首先 0x7cbf 指令会把 readsect 函数的第 1 个参数送入%edi，第 1 个参数是 dst，即这个扇区数据存放的目的起始地址，其中 readseg 是把 pa 送给 readsect 作为第一个参数的。所以当前%edi 中存放的是 pa，0x7cc2 指令会把%ecx 赋值为 0x80，0x7cc7 会把%edx 赋值为 0x1f0。0x7ccc 执行指令 cld，用于清

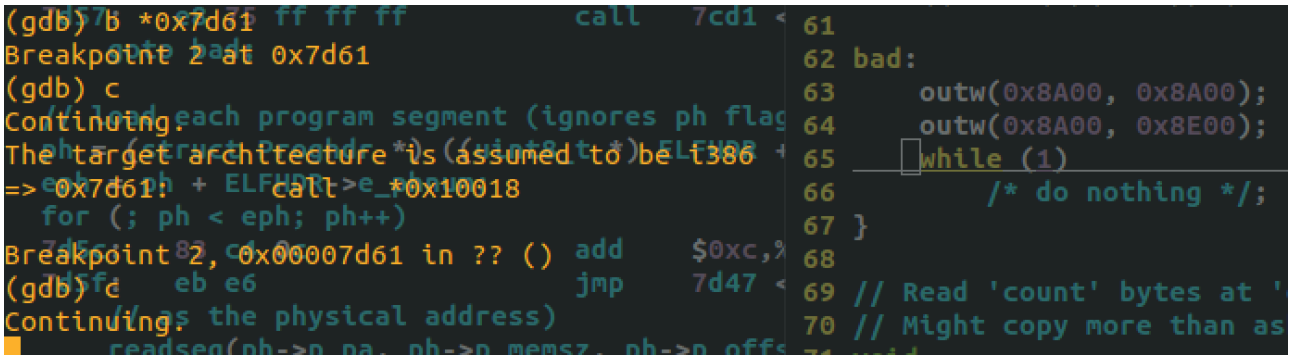
除方向标识，表明一个串操作完成后源操作数和目的操作数的地址加 1。repnz 指令使得 insl 指令被重复调用%exc 中存放的数值次数。



判断循环条件。如果不满足条件跳出循环到 0x7d61。



循环结束后执行语句((void (*)(void)) (ELFHDR->e_entry))();
e_entry 字段指向的是这个文件的执行入口地址。所以这里相当于开始运行这个文件。也就是内核文件。这句话的含义就是把控制权转移给操作系统内核。



添加 breakpoint,继续执行。

At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
当运行完 " ljmp $PROT_MODE_CSEG, $protcseg " 语句后，正式进入 32 位工作模式。根本原因是此时 CPU 工作在保护模式下。

What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
boot loader 执行的最后一条语句是 bootmain 子程序中的最后一条语句 " ((void (*)(void)) (ELFHDR->e_entry))(); "，即跳转到操作系统内核程序的起始指令处。这个第一条指令位于/kern/entry.S 文件中，第一句 movw $0x1234, 0x472

Where is the first instruction of the kernel?
第一条指令位于/kern/entry.S 文件中，第一句 movw $0x1234, 0x472

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?
先关于操作系统一共有多少个段，每个段又有多少个扇区的信息位于操作系统文件中的 Program Header Table 中。这个表中的每个表项分别对应操作系统的一个段。并且每个表项的内容包括这个段的大小，段起始地址偏移等等信息。所以如果我们能够找到这个表，那么就能够通过表项所提供的信息来确定内核占用多少个扇区。关于这个表存放在哪里的信息，则是存放在操作系统内核映像文件的 ELF 头部信息中。

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 (e.g., A tutorial by Ted Jensen that cites K&R heavily), though not as strongly recommended.
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.

```c
#include <stdio.h>
#include <stdlib.h>

void f(void)
{
    int a[4];
    int *b = malloc(16);
    int *c;
    int i;

    printf("1: a = %p, b = %p, c = %p\n", a, b, c);

    c = a;
    for (i = 0; i < 4; i++)
    a[i] = 100 + i;
    c[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]);

    c[1] = 300;
    *(c + 2) = 301;
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
       a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
       a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
    *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]);

    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);
    printf("6: a = %p, b = %p, c = %p\n", a, b, c);
}
```

```
Last login: Wed Oct  5 21:00:10 on ttys000
/Users/Zry/Desktop/Operating\ System/lab1/pointers ; exit;
➜  ~ /Users/Zry/Desktop/Operating\ System/lab1/pointers ; exit;
1: a = 0x7fff5f1b6a90, b = 0x7fef78403180, c = 0x100a49000
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 = 0x7fff5f1b6a90, b = 0x7fff5f1b6a94, c = 0x7fff5f1b6a91

[进程已完成]
```

pointers.c 以及运行结果。

1:打印出 a[]的首地址，*b 的地址，*c 的地址。

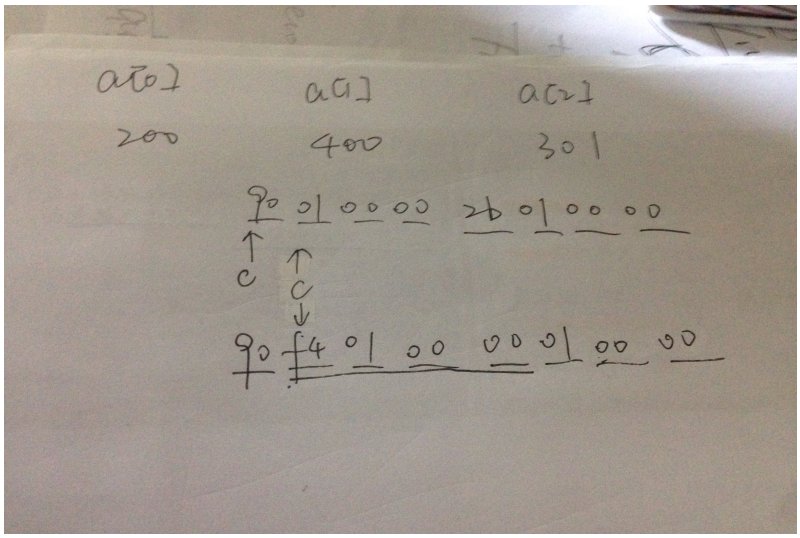
2:c 指向 a 指向的 a[0],将*c 赋值为 200，即 a[0] = *a = 200.其余 a[i] = i+100.

3:a[1] = c[1] = 300.a[2] = *(c+2) = 301.a[3] = 3[c] = 302.

4:c 指向 a[1],a[1] = *c = 400.

5:c 强制类型转换为 char 指针，char 为 1byte,int 为 4byte.所以 c 指针向后移动了 1byte.

即：



6:int 占 4byte，char 占 1byte.所以 b 是 a 后 4byte 的地址，c 是 a 后 1byte 的地址。

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!

```
23      @echo + cc -Os $<
24      $(V)$(CC) -nostdinc $(KERN_CFLAGS) -Os -c -o $(OBJDIR)/boot/main.o boot/
    main.c
25
26  $(OBJDIR)/boot/boot: $(BOOT_OBJS)
27      @echo + ld boot/boot
28      $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
29      $(V)$(OBJDUMP) -S $@.out >$@.asm
30      $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
31      $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
32
```

修改前的 Makefrag. 其中的-Ttext 0x7C00，就是指定链接地址，把它修改为 0x7E00.

```
21
22  $(OBJDIR)/boot/main.o: boot/main.c
23      @echo + cc -Os $<
24      $(V)$(CC) -nostdinc $(KERN_CFLAGS) -Os -c -o $(OBJDIR)/boot/main.o boot/main.c
25
26  $(OBJDIR)/boot/boot: $(BOOT_OBJS)
27      @echo + ld boot/boot
28      $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7E00 -o $@.out $^
29      $(V)$(OBJDUMP) -S $@.out >$@.asm
30      $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
31      $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

```
 9
10  .globl start
11  start:
12    .code16                     # Assemble for 16-bit mode
13    cli                         # Disable interrupts
14      7c00:   fa                      cli
15    cld                         # String operations increment
16      7c01:   fc                      cld
17
18    # Set up the important data segment registers (DS, ES, SS).
19    xorw    %ax,%ax             # Segment number zero
20      7c02:   31 c0                   xor    %eax,%eax
21    movw    %ax,%ds             # -> Data Segment
22      7c04:   8e d8                   mov    %eax,%ds
23    movw    %ax,%es             # -> Extra Segment
24      7c06:   8e c0                   mov    %eax,%es
25    movw    %ax,%ss             # -> Stack Segment
26      7c08:   8e d0                   mov    %eax,%ss
27
```

修改前的 boot.asm. 可执行文件中的链接地址是从 0x7C00 开始.

```
 5 Disassembly of section .text:
 6
 7 00007e00 <start>:
 8 .set CR0_PE_ON,      0x1         # protected mode enable flag
 9
10 .globl start
11 start:
12   .code16                     # Assemble for 16-bit mode
13   cli                         # Disable interrupts
14     7e00:   fa                        cli
15   cld                         # String operations increment
16     7e01:   fc                        cld
17
18   # Set up the important data segment registers (DS, ES, SS).
19   xorw    %ax,%ax             # Segment number zero
20     7e02:   31 c0                     xor    %eax,%eax
21   movw    %ax,%ds             # -> Data Segment
22     7e04:   8e d8                     mov    %eax,%ds
23   movw    %ax,%es             # -> Extra Segment
24     7e06:   8e c0                     mov    %eax,%es
25   movw    %ax,%ss             # -> Stack Segment
26     7e08:   8e d0                     mov    %eax,%ss
27
```

修改后重新 make 得到的 boot.asm, 可执行文件中的链接地址是从 0x7E00 开始.

```
[   0:7c1e] => 0x7c1e:  lgdtw  0x7e64
0x00007c1e in ?? ()
(gdb) x/6xb 0x7e64
0x7e64: 0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/6xb 0x7c64
0x7c64: 0x17    0x00    0x4c    0x7e    0x00    0x00
(gdb)
```

使用 gdb 进行单步执行，直到 0x7c1e 处。当前这条指令读取的内存地址是 0x7e64。打印出后 6byte 的值全部是 0，出现了错误，本来应该读取 0x7c64 处的值。

```
   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7e32
x00007c2d in ?? ()
```

继续运行，直到 0x7c2d 处。

```
Triple fault.  Halting for inspection via QEMU monitor.
```

出现了错误，QEMU 强制 halt 了程序。

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.)

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

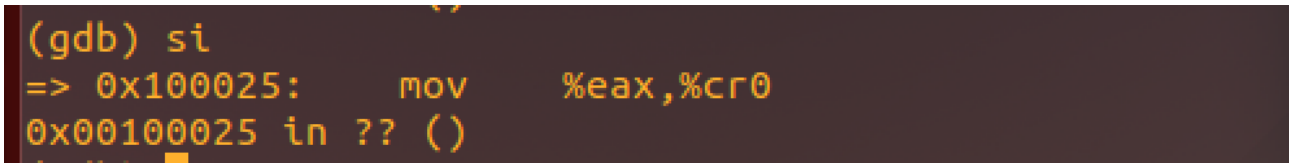
进入 boot loader 前。

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

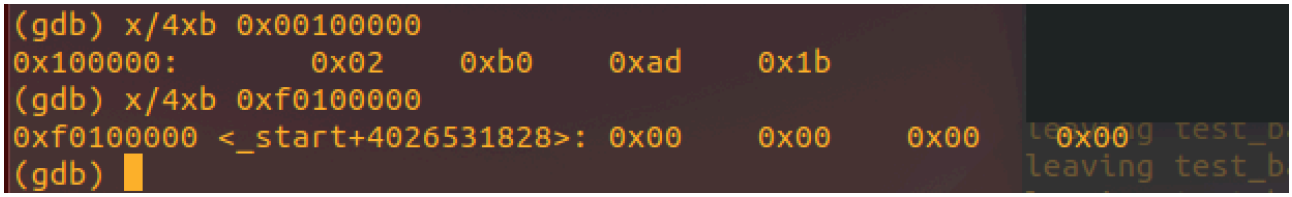
进入 kernel 前。

bootmain 函数在最后会把内核的各个程序段送入到内存地址 0x00100000 处，所以这里现在存放的就是内核的某一个段的内容，由于程序入口地址是 0x0010000C，正好位于这个段中，所以可以推测，这里面存放的应该是指令段，即.text 段的内容。

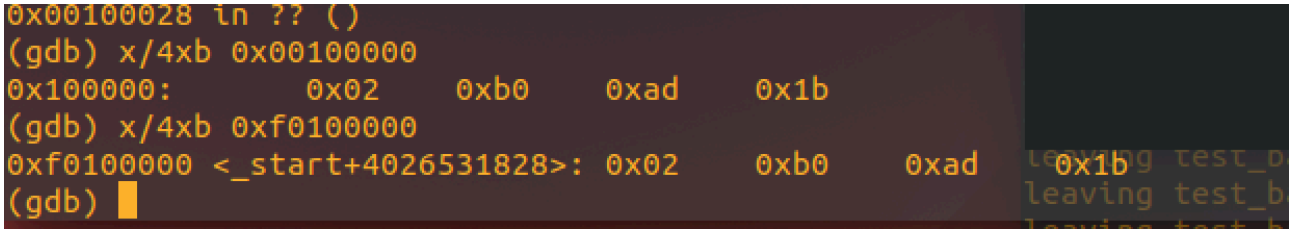
Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the **`stepi`** GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just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



运行到 movl %eax %cr0.

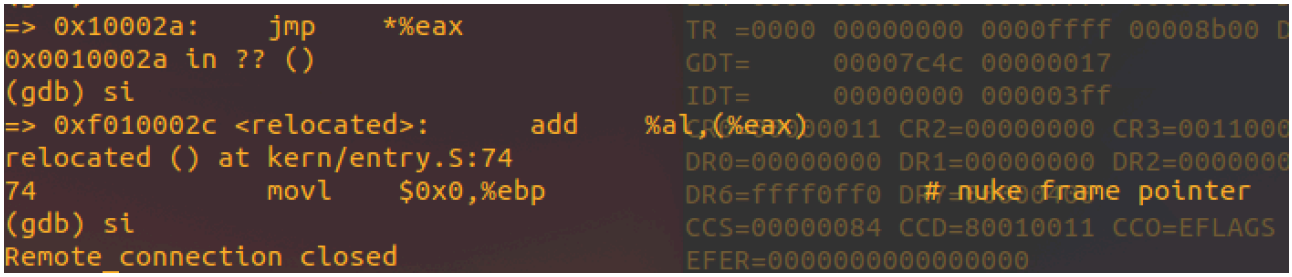


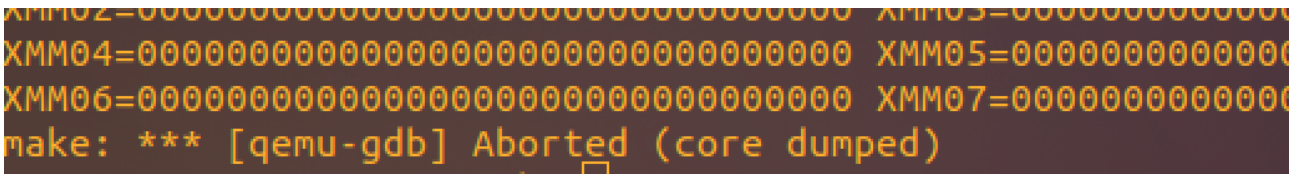
打印出 0x00100000 及 0xf0100000 内存里的内容。



单步运行后打印出 0x00100000 及 0xf0100000 的内容。可以发现这步后 0x00100000 中的内容被读入 0xf0100000 中。

把 entry.S 文件中的%movl %eax, %cr0 这句话注释掉，重新编译内核。
直到 0x10002a 处发生跳转。





于是下一步出现错误。QEMU 强行终止程序。

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

在 lib/printfmt.c 中添加代码。

回答问题：
1. console.c 中大多函数都可以被外部调用。其中 print.c 使用了其中的 cputchar 函数。
2. crt_pos 代表最后一个字符在屏幕上显示的位置。当 if 后的条件成立代表当前页面上将要输出的内容已经超过页面最大容量，所以将页面向上滚动一行。Memmove 执行的就是将 1～n 行复制到 0～n-1 行，for 执行的是将最后一行全部变成’’.
3.

```
7
8       int x = 1, y = 3, z = 4;
9       cprintf("x %d, y %x, z %d\n", x, y, z);
0       //unsigned int i = 0x00646c72;
1       //cprintf("H%x Wo%s", 57616, &i);
```

```
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
x 1, y 3, z 4
K>
```

把这段程序添加到 kern/monitor.c 中。
fmt 指向的就是"x %d, y %x, z %d\n"字符串. ap 指向所有输入参数的集合。
cprint 中调用了 vcprintf 函数，并且把 fmt， ap 作为输入参数传给了 vcprintf。
Vcprintf 中调用了 vprintfmt 子程序。其中调用的 getint 函数调用了 va_arg.
对 va_arg 进行了一次调用后，调用前 ap 中包括 x，y，z 三个参数的内容：1，3，4。调用完成后只剩下 y，z 的内容：3,4.

4.

```
   unsigned int i = 0x00646c72;
   cprintf("H%x Wo%s", 57616, &i);
---  6 lines: while (1) {-----------------------------------------------
```

```
Type 'help' for a list of commands.
He110 WorldK>
```

57616 以十六进制表示即为 e110.i 每一个 byte 对应的字符为 72（r），6c（l），64（d），00（\0）.

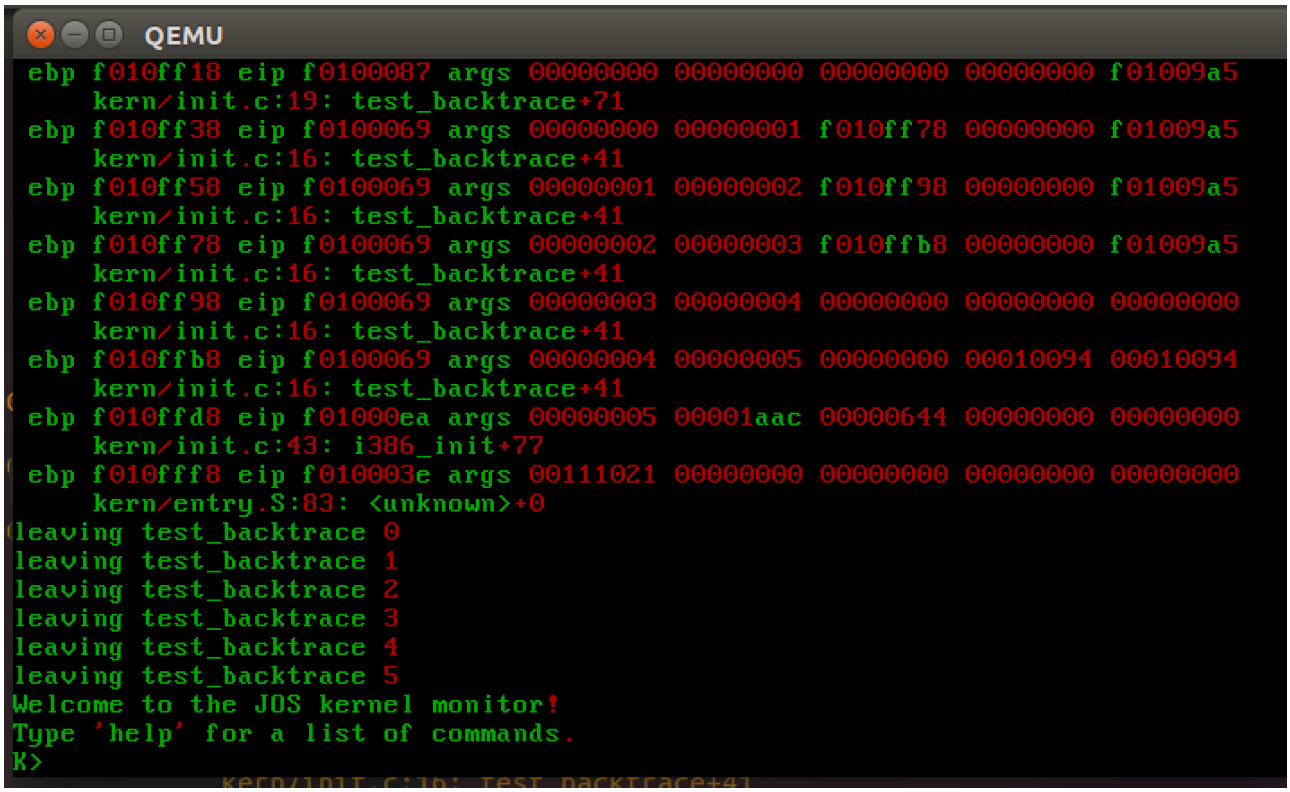
5.

```
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
x = 3 y = -267380452K>
```

输出如图。由于 y 并没有被指定，所以会输出内存中一个不确定的值。

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.

```
52 static void
53 cga_putc(int c)
54 {
55     // if no attribute given, then use black on white
56     if (!(c & ~0xFF))
57     //  c |= 0x0700;
58     {
59         if (c < 58)
70             c |= 0x0400;
71         else
72             c |= 0x0200;
73     }
```



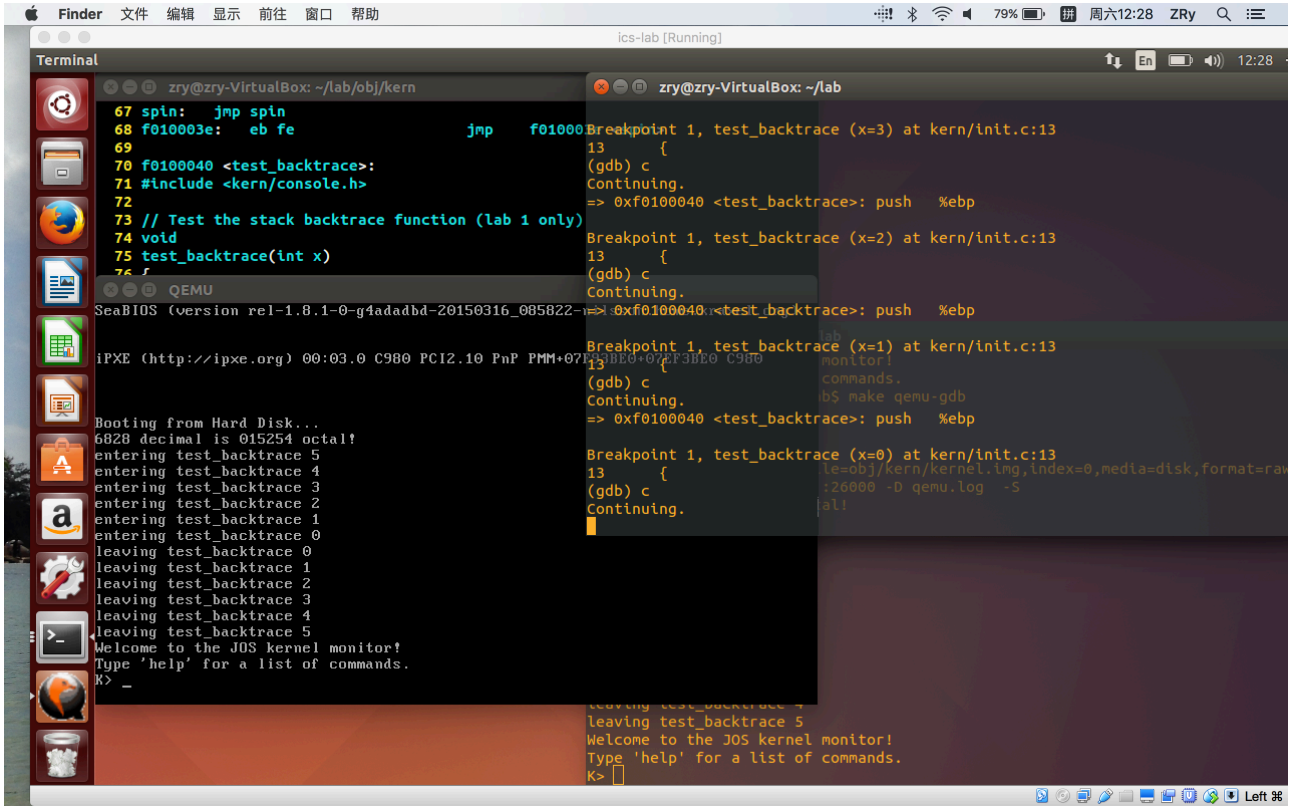
阅读资料可知，字符 c 的 9-16 位控制字体及背景颜色。
参考了颜色对应的二进制字符串的表格。在 kern/console.c 中找到控制字体颜色的函数 cga_putc 进行修改。

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?

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

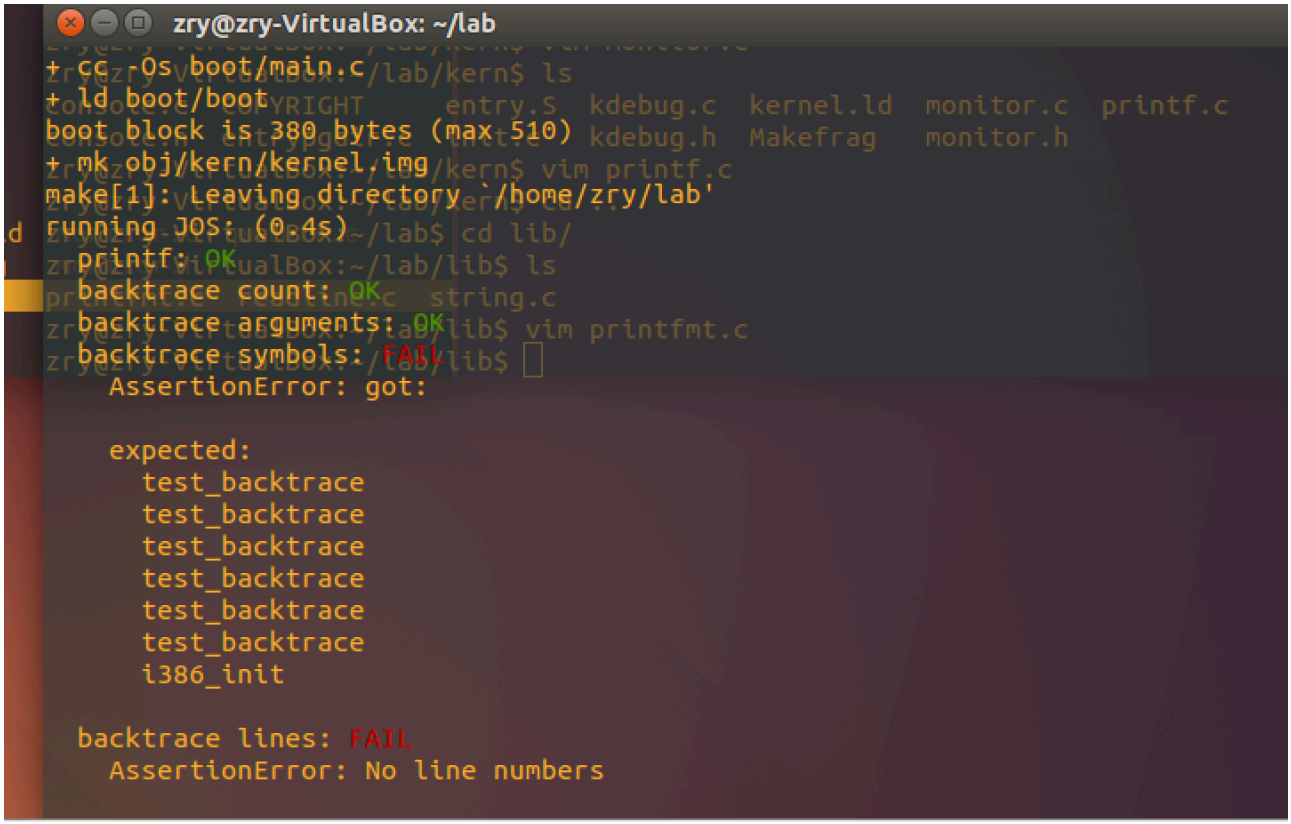
根据注释及前后代码推断是从这条命令开始初始化堆栈的。
Bootstacktop 的值为 0xf0110000，堆栈空间大小为 32kb。这个堆栈实际坐落在内存的 0x0010F000-0x00110000 物理地址空间中。
在 entry.S 中的数据段里面声明一块大小为 32Kb 的空间作为堆栈使用。从而为内核保留了一块空间。
堆栈指针指向最高地址。因为栈向下增长。

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.



根据 obj/kern/kernel.asm, 进入 test_backtrace 函数时的地址为 0xf0100040, 在此处设置 breakpoint. 继续运行可知此函数递归调用 5 次。
对于任意一层调用，它的 esp 和 ebp 的值分别指向的内存单元处存放着上一层程序的 ebp 寄存器的值及内存单元处存放着对下一层子程序调用时传入的参数。每次调用会将当前 ebp 的值压入栈。

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.
If you use `read_ebp()`, note that GCC may generate "optimized" code that calls `read_ebp()` *before* `mon_backtrace()`'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of `mon_backtrace()` and make sure the call to `read_ebp()` is happening after the function prologue.



backtrace 的功能就是要显示当前正在执行的程序的栈帧信息。包括当前的 ebp 寄存器的值，这个寄存器的值代表该子程序的栈帧的最高地址。eip 则指的是这个子程序执行完成之后要返回调用它的子程序时，下一个要执行的指令地址。后面的值就是这个子程序接受的来自调用它的子程序传递给它的输入参数。
按照给定格式打印信息。
完成了 mon_backtrace 函数后打印的结果。详情见代码。

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.

在实际调试中希望在栈帧中打印出更多的内容。这时需要用到 kern/kdebug.c 中的函数。
完成了 debuginfo_eip 函数以及 k>命令行中 backtrace 指令的添加完善。详情见代码。

总结

实验中遇到的问题：
1 刚开始对于如此长的文档而一无所知的我不可谓不恐慌。基本上在 part1 部分我都比较迷茫。不过多读几遍之后再结合所学知识我最终突破了这个障碍。
2 框架代码的阅读量大，弄清各个文件函数之间的关系也是对于我也是个不小的挑战。
3 这个 lab 对于代码水平要求不高，但是其中要求回答的一些问题还是很有难度。
4 在最后的几个 exercise 对于格式都有一些严格的要求。开始并没有注意导致总是报错。
5 虽然完成了这个 lab 的内容和题目但是一些细节问题依旧不明朗。
6 其实最大的挑战还是如何在报告中使用文字来叙述实验过程。。（可能有些过程不知道如何描述就最终没有在报告里面体现 QAQ）

实验感想：
1 虽然在过程中遇到了很多问题，但是对于操作系统的启动，内核的运行有了更进一步的了解，这是这个 lab 非常重要的意义。
2 操作系统是一门很有用的课程。我相信通过这个 lab 和这门课我们能学到非常多有用的基础知识，为以后的课程做了很有必要的奠基。
3 这个 lab 代码量比较少，更多的考察还是对于文档，题目以及框架代码的理解。
4 文档冗长，在实验过程中花费了大量时间筛选 lab 的重点。这份文档并没有很好估计当代大学生的水平（尤其是指针那个 exercise＝ ＝）。
5 想到还有 5 个工作量如此甚至很可能更大的 labs 我感到了来自这个世界的深深恶意＝ ＝
6 无论如何，加油吧！