**1）Exercise 3.**

**GDB disassemble出的mov指令没有严格满足16-bit的操作数操作；GDB将jnz处理成jne，这两条指令只是有不同的助记符，是完全相同的指令**

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

**orl $CR0\_PE\_ON, %eax**

**7c26: 66 83 c8 01 or $0x1,%ax**

**设置了flags位**

**# 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**

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

**bootmain的代码对应**

**00007d19 <bootmain>:**

**{**

**7d19: 55 push %ebp**

**7d1a: 89 e5 mov %esp,%ebp**

**7d1c: 56 push %esi**

**7d1d: 53 push %ebx**

**readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0);**

**7d1e: 52 push %edx**

**7d1f: 6a 00 push $0x0**

**7d21: 68 00 10 00 00 push $0x1000**

**7d26: 68 00 00 01 00 push $0x10000**

**7d2b: e8 aa ff ff ff call 7cda <readseg>**

**if (ELFHDR->e\_magic != ELF\_MAGIC)**

**7d30: 83 c4 10 add $0x10,%esp**

**7d33: 81 3d 00 00 01 00 7f cmpl $0x464c457f,0x10000**

**7d3a: 45 4c 46**

**7d3d: 75 38 jne 7d77 <bootmain+0x5e>**

**ph = (struct Proghdr \*) ((uint8\_t \*) ELFHDR + ELFHDR->e\_phoff);**

**7d3f: a1 1c 00 01 00 mov 0x1001c,%eax**

**eph = ph + ELFHDR->e\_phnum;**

**7d44: 0f b7 35 2c 00 01 00 movzwl 0x1002c,%esi**

**ph = (struct Proghdr \*) ((uint8\_t \*) ELFHDR + ELFHDR->e\_phoff);**

**7d4b: 8d 98 00 00 01 00 lea 0x10000(%eax),%ebx**

**eph = ph + ELFHDR->e\_phnum;**

**7d51: c1 e6 05 shl $0x5,%esi**

**7d54: 01 de add %ebx,%esi**

**for (; ph < eph; ph++)**

**7d56: 39 f3 cmp %esi,%ebx**

**7d58: 73 17 jae 7d71 <bootmain+0x58>**

**readseg(ph->p\_pa, ph->p\_memsz, ph->p\_offset);**

**7d5a: 50 push %eax**

**for (; ph < eph; ph++)**

**7d5b: 83 c3 20 add $0x20,%ebx**

**readseg(ph->p\_pa, ph->p\_memsz, ph->p\_offset);**

**7d5e: ff 73 e4 pushl -0x1c(%ebx)**

**7d61: ff 73 f4 pushl -0xc(%ebx)**

**7d64: ff 73 ec pushl -0x14(%ebx)**

**7d67: e8 6e ff ff ff call 7cda <readseg>**

**for (; ph < eph; ph++)**

**7d6c: 83 c4 10 add $0x10,%esp**

**7d6f: eb e5 jmp 7d56 <bootmain+0x3d>**

**((void (\*)(void)) (ELFHDR->e\_entry))();**

**7d71: ff 15 18 00 01 00 call \*0x10018**

**}**

**last instruction of boot loader executed：((void (\*)(void)) (ELFHDR->e\_entry))();**

**first instruction of the kernel it just loaded：movw $0x1234,0x472**

**3)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?**

**2）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.)**

**BIOS enters the boot loader时：是0**

**boot loader enters the kernel时：0x1badb002 0x00000000 0xe4524ffe 0x7205c766**

**3）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!**

**改成0x7d00很奇怪，boot.out的start地址变成0x7d00，但是，断点打在那里不会命中，反而还是在0x7c00命中；执行到0x7c1e都没问题，下一条本来是0x7c21，这次却变成0x7c23，再下一条0x7c26又正常了，0x7c2d后一条会跳到0xf000e05b，由于没正确开启保护模式，地址变成0xe05b，之后就乱套了；**

**正常配置为7c00时，会跳到0x7c32，正常执行下去**

**4)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.**

**是 movl %cr0, %eax**

**orl $(CR0\_PE|CR0\_PG|CR0\_WP), %eax**

**movl %eax, %cr0**

**进到relocated后，会变成高位地址**

**tips:%n包括打印出的格式化后的字符**

**5)Exercise 12. 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?**

**init.S中**

**movl $(bootstacktop),%esp**

**# now to C code**

**call i386\_init**

**初始化了kernel的stack**

**bootstack:**

**.space KSTKSIZE**

**.globl bootstacktop**

**bootstacktop:**

**6)Exercise 13. 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?**

**ebp，esi, %ebx, %esi, %eax**

**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?**

**printf中的putch函数会调console.c的cputchar，putch将要显示的字符打印在屏幕上**

1. **Explain the following from console.c:**

**换行，屏幕上显示的字符满了之后，整体上移一行，留出最下面的一行为空白**

1. **1 if (crt\_pos >= CRT\_SIZE) {**
2. **2 int i;**
3. **3 memmove(crt\_buf, crt\_buf + CRT\_COLS, (CRT\_SIZE - CRT\_COLS) \* sizeof(uint16\_t));**
4. **4 for (i = CRT\_SIZE - CRT\_COLS; i < CRT\_SIZE; i++)**
5. **5 crt\_buf[i] = 0x0700 | ' ';**
6. **6 crt\_pos -= CRT\_COLS;**
7. **7 }**
8. **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?**

**格式化字符串；参数列表**

* + **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.**

**cprintf vcprintf vcprintffmt putch cons\_putc … getint va\_arg …**