Lab 1

516030910101 罗宇辰

Exercise 1 \

Exercise 2 \

Exercise 3

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

```
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

将`CR0`的`PE`位置为1

```
 ljmp    $PROT_MODE_CSEG, $protcseg
    7c2d:    ea                       .byte 0xea
    7c2e:    32 7c 08 00              xor    0x0(%eax,%ecx,1),%bh
```

通过 `ljmp` 操作把 `%cs` 寄存器置为 `$PROT_MODE_CSEG` ，即 *Protected Mode* ，标志着processor切换到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?

```
((void (*)(void)) (ELFHDR->e_entry))(); # last instruction
    7d71:    ff 15 18 00 01 00        call   *0x10018
```

```
0x10000c:    movw    $0x1234,0x472    # first instruction
```

- 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?

首先，把disk的第一个page中的内容以ELF文件的格式读取出来，这个ELF结构中就记录了启动程序的program segment信息：

```
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
```

然后，根据ELF中的`e_phoff`值确定Program header table在ELF文件中的偏移量，根据`e_phnum`值确定Program header table的条目数量：

```
    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
```

最后，遍历每一个Program header条目，根据条目中记录的segment大小和偏移量将program从disk中读取出来：

```
    for (; ph < eph; ph++)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

Exercise 4

```
# point.c output
1: a = 00DDF830, b = 013050C0, c = 00DDF830
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 = 00DDF830, b = 00DDF834, c = 00DDF831
```

Exercise 5

```
Breakpoint 1, 0x00007c00 in ?? () # the BIOS enters the boot loader
(gdb) x/8x 0x100000
0x100000:   0x00000000  0x00000000  0x00000000  0x00000000
0x100010:   0x00000000  0x00000000  0x00000000  0x00000000

Breakpoint 2, 0x00007d71 in ?? () # the boot loader enters the kernel
(gdb) x/8x 0x100000
0x100000:   0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010:   0x34000004  0x3000b812  0x220f0011  0xc0200fd8
```

从kernel的ELF的记录中

```
shell$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001aef  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, COD
```

可以看到，从地址0x100000开始是`.text` section，所以在kernel被load之后，地址0x100000存放的是executable instructions

Exercise 6

boot loader 的link address为**0x7C00**时：

```
# ljmp    $PROT_MODE_CSEG, $protcseg
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c32
The target architecture is assumed to be i386
=> 0x7c32:  mov    $0x10,%ax    # long jump之后成功切换到32-bit mode
```

将link address 改为**0X7C10**之后：

```
# ljmp    $PROT_MODE_CSEG, $protcseg
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c42
[f000:e05b]    0xfe05b: cmpw   $0x48,%cs:(%esi)     # 依然是16-bit mode
```

Exercise 7

```
    movl    %cr0, %eax
    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0          # 将cr0的PG置为1，标志着paging开始生效
```

在注释了`movl %eax, %cr0`之后，

```
    movl    %cr0, %eax
    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
    # movl  %eax, %cr0
    mov $relocated, %eax
    jmp *%eax

    relocated:
    movl    $0x0,%ebp           # FAIL
```

当运行到`movl $0x0,%ebp`时，QEMU报错：

```
Booting from Hard Disk..qemu-system-i386: Trying to execute code outside RAM orc
```

这是因为这段代码位于virtual address 0xf010002f处：

```
=> 0xf010002f <relocated>:  mov    $0x0,%ebp
```

当paging生效时，virtual address会被mapping为对应physical address，就能正常执行代码；当paging无效时，会直接将0xf010002f理解为RAM中的地址，显然这个地址超过了RAM的地址范围，所以会报错

Exercise 8

```
    // (unsigned) octal
        case 'o':
            num = getuint(&ap, lflag);
            base = 8;
            putch('0', putdat);
            goto number;
```

Exercise 9

```
    // flag to precede the result with a plus or minus sign
        case '+':
            posi = 1;    // int posi is initiated as 0
            goto reswitch;
    ...
        number:
            if(posi)
                putch('+', putdat); //print sign
            printnum(putch, putdat, num, base, width, padc);
            break;
```

1. Explain the interface between `printf.c` and `console.c`.

`console.c` export function *cputchar*, and `printf.c` use it in funtion *putch* to print a character on the console

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

```
    if (crt_pos >= CRT_SIZE) {// 如果当前的输入光标已经超过屏幕显示的范围
        int i;
        //将当前的buf的第二行及之后的内容copy到第一行的位置（相当于屏幕内容向下滚动了一行）
        memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
        //将新的空出来的一行填充为黑色
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
            crt_buf[i] = 0x0700 | ' ';
        //将当前光标的位置减去CRT_COLS个单元（相当于光标上一一行，之前超过显示范围的光标现在位于屏幕最下方的起点处）
        crt_pos -= CRT_COLS;
    }
```

3.

```
cprintf (fmt=0xf0101c57 "x %d, y %x, z %d\n")
    vcprintf (fmt=0xf0101c57 "x %d, y %x, z %d\n", ap=0xf0110e94 "\001")
        cons_putc (c=120)   //x
```

```
        cons_putc (c=32)    // 
        cons_putc (c=49)    //1
        cons_putc (c=44)    //,
        cons_putc (c=32)    // 
        cons_putc (c=121)   //y
        cons_putc (c=32)    // 
        cons_putc (c=51)    //3
        cons_putc (c=44)    //,
        cons_putc (c=32)    // 
        cons_putc (c=122)   //z
        cons_putc (c=32)    // 
        cons_putc (c=52)    //4
        cons_putc (c=10)    //\n
```

4.

```
cprintf (fmt=0xf0101c69 "H%x Wo%s")
    vcprintf (fmt=0xf0101c69 "H%x Wo%s", ap=0xf0110e94 <incomplete sequence \341>)
        cons_putc (c=72)    //H
        cons_putc (c=101)   //e
        cons_putc (c=49)    //1
        cons_putc (c=49)    //1
        cons_putc (c=48)    //0
        cons_putc (c=32)    // 
        cons_putc (c=87)    //W
        cons_putc (c=111)   //o
        cons_putc (c=114)   //r
        cons_putc (c=108)   //l
        cons_putc (c=100)   //d
```

5.

```
cprintf("x=%d y=%d", 3);
// output: x=3 y=-267317588
```

因为是依次从stack中读取参数，虽然没有y的参数，但是会把栈中原来的内容当作int输出

Exercise 10

```
    case 'n': {

        const char *null_error = "\nerror! writing through NULL pointer! (%n argument)\n";
        const char *overflow_error = "\nwarning! The value %n argument pointed to has been
overflowed!\n";

        // 首先将%n对应的参数以signed char指针的形式读取出来
        signed* ptr = (signed*) va_arg(ap, void *);

        if(!ptr){
            // 如果这个指针是NULL，报错
```

```
            printfmt(putch, putdat, "%s", null_error);
        }
        else {
            // 指针非NULL，则将当前的输入字符数量以signed char形式（1 byte）读到指针所指向的地址
            *(signed char*)ptr = *(signed char*)putdat;
            if(*(int*)putdat > 0x7F){
                // 如果输入字符数量超过0x7F（signed char能表示的最大值），报错
                printfmt(putch, putdat, "%s", overflow_error);
            }
            break;
        }
    }
```

Exercise 11

```
    //  padc变量为‘-’时在右边补空格
    if(padc == '-'){
        padc = ' ';
        //  先输出内容
        printnum(putch,putdat,num,base,0,padc);
        //  补充空格
        while (--width > 0)
            putch(padc, putdat);
        return;
    }
```

Exercise 12

```
# entry.S
relocated:
    movl    $0x0,%ebp           # nuke frame pointer
    # Set the stack pointer
    movl    $(bootstacktop),%esp
    # now to C code
    call    i386_init
```

从`entry.S`中的这段代码可以看到， stack pointer初始时指向 *bootstacktop* 这个预定义的位置

Exercise 13/14/15

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    cprintf("Stack backtrace:\n");
```

```
    uint32_t ebp = read_ebp();  // 读取%ebp寄存器中的值，%ebp指向当前帧，（%ebp）指向上一帧

    while(ebp!=0){  // %ebp为0时函数遍历到最外层，backtrace到达终点
        uint32_t eip = *(int*)(ebp+4);  // 读取(4(%ebp))，即%eip的值
        cprintf("  eip %08x  ebp %08x  args %08x %08x %08x %08x %08x\n",
                eip, ebp,
                *(int*)(ebp+8),*(int*)(ebp+12),*(int*)(ebp+16),*(int*)(ebp+20),*(int*)
(ebp+24));
        struct Eipdebuginfo info;
        if(debuginfo_eip(eip,&info)>=0){    // 查找%eip对应函数的相关信息
            cprintf("         %s:%d %.*s+%d\n",
            info.eip_file, info.eip_line,
            info.eip_fn_namelen, info.eip_fn_name, eip-info.eip_fn_addr);
        }

        ebp = *(int*)ebp;   // 更新ebp，下一轮循环将输出外一层函数的相关信息
    }

    overflow_me();
        cprintf("Backtrace success\n");
        cprintf("debug\n");
    return 0;
}
```

Exercise 16

```
void
start_overflow(void)
{
    char str[256] = {};
    int nstr = 0;
    char *pret_addr;

    // Your code here.
    pret_addr = (char*)read_pretaddr(); // 读取eip所在的地址

    // 原本的ret addr: 0xF0100A24(overflow_me)
    cprintf("old rip: %lx\n", *(uint32_t*)pret_addr);
    cprintf("%45d%n\n",nstr, pret_addr);    // 更改 0x24 -> 0x2d
    cprintf("%9d%n\n",nstr, pret_addr+1);   // 更改 0x0A -> 0x09
    // 新的ret addr: 0XF010092d(do_overflow)
    cprintf("new rip: %lx\n", *(uint32_t*)pret_addr);

    //  在8(%ebp)处填入原本的ret addr，这样do_overflow才能正常return
    cprintf("%36d%n\n",nstr, pret_addr+4);  // 填入0x24
    cprintf("%10d%n\n", nstr, pret_addr+5); // 填入0x0A
    cprintf("%16d%n\n",nstr, pret_addr+6);  // 填入0x10
    cprintf("%240d%n\n",nstr, pret_addr+7); // 填入0xF0

    /*
```

```
        | BLANK |                           | overflow_me addr |
        | overflow_me addr |                | do_overflow addr |
        | last ebp |                        | last ebp |

          old stack      ------------>        new stack
    */
}
```