5140379016 周剑宇

> PC bootstrap

作业文档解释的十分清楚：启动后把ROM BIOS加载到内存0x000f0000-0x000fffff，初始的段寄存器和指针寄存器是0xf000和0xfff0，在实模式下对应的第一条指令的物理地址就是0xffff0,第一条跳转到0xfe05b

```
in %al,port #把%al中的值写到端口地址为port的端口中
out port,%al #与上相反
```

可以看到写了这些端口地址0xd,0xda,0xd6,0x70,0x71...再看了链接材料：

```
0000-000f       Slave DMA controller
00C0-00DE       Master DMA controller
0070-0071       NMI Enable / Real Time Clock
```

大概是对DMA做了一些初始化吧，然后关闭NMI，开启时钟

> Boot loader

文档中讲过的略过，直接开始回答问题：

```
7c00 - 7c1d:
#asm的注释中写的十分清楚，关中断，设置段寄存器，开启a20线
7c1e: lgdt    gdtdesc
7c24: movl    %cr0, %eax
7c26: orl     $CR0_PE_ON, %eax
7c2a: movl    %eax, %cr0
#load GDT?
#看了一下gdtdesc，应该是把gdt的地址和大小-1载入到gdt寄存器中
#然后是把cr0修改为1，也就是进入了保护模式
7c32 - 7c3f:
#进入保护模式后又设置了一遍段寄存器
7c40: movl    $start, %esp
#把栈指针设为7c40
#不是很明白，但是看上去很舒服，很有灵性
```

然后就进入了main.c中的bootman：

```
// read 1st page off disk
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
//...
// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
```

功能是把从kernel开始8*512个byte的内容读入到ELFHDR(0x10000)的地方

readseg是通过readsect实现的，从名字上看是读指定扇区位置的内容。

而exercise3的答案是

```
((void (*)(void)) (ELFHDR->e_entry))();
```

就是跳转到entry，即内核入口，boot.asm中：

```
((void (*)(void)) (ELFHDR->e_entry))();
    7d74:   ff 15 18 00 01 00       call   *0x10018
```

回答文档下面的三个问题：

1. ```
   ljmp    $PROT_MODE_CSEG, $protcseg
       7c2d:   ea 32 7c 08 00 66 b8    ljmp   $0xb866,$0x87c32
   ```

2. 最后一条是((void (*)(void)) (ELFHDR->e_entry))();

   跳转后到(*0x10018) , 第一条指令是entry.S中的

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

3. 放在ELF头部信息中，代码很清楚：

   ```
       ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
       eph = ph + ELFHDR->e_phnum;
       for (; ph < eph; ph++)
           // p_pa is the load address of this segment (as well
           // as the physical address)
           readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
   ```

   后续的readseg都是根据ELFHDR中的数据来的。

练习5就不用讲了，关于c的指针的。

练习6：

> Kernel

这是进入kernel后最初的代码：

```
entry:
    movw    $0x1234,0x472           # warm boot
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    movl    %cr0, %eax
    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0
```
```

通过gdb一步步运行，发现在movl　%eax, %cr0指令前，print一个0xf00xxxxxx处的内容，都是0，而在这一步之后，映射到了address-0xf0000000的低位地址处。

也就是说在这一步之后，虚拟地址的映射完成。如果注释掉，之后的“mov　$relocated, %eax ;jmp *%eax”就会出错，因为要jmp的地址是虚拟地址。

然后看到`kern/printf.c`，`lib/printfmt.c`,`kern/console.c`三个文件，我没有去trace他们的调用关系，因为读代码就可以明白了。前两个文件中调用了console.c中的函数，看了下显然console.c实现的就是打印的功能，和i/o相关的，所以不需要认真读。printf.c中的cprintf和vcprintf函数就是我们调用的print函数，而他们是通过printfmt.c中的vprintfmt和自己定义的putch（通过console.c中的cputchar实现）这两个函数实现的，而printfmt.c中实现的是根据输入的内容格式完成最终的打印内容，所以我认为printf.c是系统print功能的接口通过调用lib/printfmt.c得到最终的打印内容，通过kern/console.c实现实际打印到显示屏上的效果。

实习八进制：

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

实现“%+”：

```
// flag to get a '+' precede positive numbers
        case '+'
            pflag = 1;
            goto reswitch;
```

实现“%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";
            // Your code here
            signed char *arg = va_arg(ap ,signed char * );
            if(arg == NULL){
                printfmt(putch,putdat,"%s",null_error);
            }else{
                int i = *(int *)putdat;
                if( i > 127){
                    printfmt(putch,putdat,"%s",overflow_error);
                    *arg = -1;
                }else{
                    *arg = i;
                }
            }
```

靠左行驶：

```
number:
            if(rtpflag == 1){
                printnumRP(putch, putdat, num, base, width, padc , 0);
            }else{
                printnum(putch, putdat, num, base, width, padc);
            }

            break;
//...
static int
printnumRP(void (*putch)(int, void*), void *putdat,
     unsigned long long num, unsigned base, int width, int padc , int or)
{
    int sp = 0;
    // first recursively print all preceding (more significant) digits
    if (num >= base) {
        // first print this (the least significant) digit

        sp = printnumRP(putch, putdat, num / base, base, width - 1, padc ,
or + 1);
    } else {
        // print any needed pad characters after first digit
        sp = width;
    }
    putch("0123456789abcdef"[num % base], putdat);
    if(or == 0){
            while (--sp > 0)
            putch(padc, putdat);
    }
    return sp;
}
```

我没有去改原来的printnum，因为我觉得把所有的判断放在上一层，这里简单的实现最简单的print number功能更为合适。

> Stack

```
    # stack backtraces will be terminated properly.
    movl    $0x0,%ebp           # nuke frame pointer
    # Set the stack pointer
    movl    $(bootstacktop),%esp
    # now to C code
    call    i386_init
```

这是在指针的初始值。

```
    .p2align    PGSHIFT     # force page alignment
    .globl      bootstack
bootstack:
    .space      KSTKSIZE
    .globl      bootstacktop
bootstacktop:
```

在inc/memlayout.h中找到KSTKSIZE的大小，是8个page size的大小：8*4096

```
#define KSTKSIZE    (8*PGSIZE)          // size of a kernel stack
```

所以栈的地址的空间是从bootstacktop开始向下8*PGSIZE大小。

之后的练习基本上就是写写代码了，没有什么可提的。