Lab #1

陈威宇

Questions after Exercise 3

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

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

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
```

在执行完 boot.S 里的上述指令后，处理器就到了 32 位模式. 是修改%cr0 的最低位为 1 导致的.

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

我们看 boot.asm 的 bootmain 函数的最后一行可以看到，在执行了 boot loader 的最后一条指令 `call *0x10018` 后，就进入了内核，内核的第一条指令是 `movw $0x1234,0x472`.

Where is the first instruction of the kernel?

`0x10000c`.

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?

```
  struct Proghdr *ph, *eph;

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

  // is this a valid ELF?
  if (ELFHDR->e_magic != ELF_MAGIC)
```

```
    goto bad;

  // load each program segment (ignores ph flags)
  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);

  // call the entry point from the ELF header
  // note: does not return!
  ((void (*)(void)) (ELFHDR->e_entry))();
```

上面是 main.c 里的一段代码. 可以看到, 先读了 8 个 sector 到物理地址 `(uint32_t)ELFHDR`, 然后看 `ELFHDR->e_phnum` 来知道要读多少 sector 来获取整个 kernel.

Exercise 5

比方说, 我把 link address 从 `0x7c00` 改成 `0x8c00`, boot loader 运行到 `ljmp $0x8,$0x8c32` 的时候就会出错卡在那了。这是因为, `0x8c32` 处全是 0, 想要正常运行的话, 应该执行 `ljmp $0x8,$0x7c32`.

Exercise 6

`0x00100000` 开始的 16 个字节的值变了, 原来都是 `0x00000000`, 现在变成了非零的值, 原因是, 他们用来存从 disk 上 fetch 过来的 kernel 了.

Exercise 7

在执行 `movl %eax, %cr0` 之前, `0x00100000` 处是 `0x1badb002`, `0xf0100000` 处是 `0x00000000`. 执行之后, `0x00100000` 处是 `0x1badb002`, `0xf0100000` 处是 `0x1badb002`. 这是因为虚拟地址 `0xf0100000` 被映射到了物理地址 `0x00100000`.

```
  # Turn on paging.
  movl  %cr0, %eax
  orl   $(CR0_PE|CR0_PG|CR0_WP), %eax
  movl  %eax, %cr0

  # Now paging is enabled, but we're still running at a low EIP
  # (why is this okay?).  Jump up above KERNBASE before entering
  # C code.
```

```
  mov  $relocated, %eax
  jmp  *%eax
relocated:

  # 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
```

如果没有做映射 (即，把 `movl %eax, %cr0` 删掉)，那么在执行 `jmp *%eax` 的下一条指令时会出错，这是因为，`jmp *%eax` 将 program counter 设为了一个很高的值，又没有做内存映射，之后取指令就出错了.

Exercise 8

```
    case 'o':
      num = getuint(&ap, lflag);
      base = 8;
      goto number;
```

Questions after Exercise 8

1

console.c 给 printf.c 提供了一个接口函数 cputchar, 这是用来在显示屏打印一个字符. printf.c 的 putch 函数调用了它.

2

crt_buff 数组表示整个显示屏的字符，这段代码的作用是，当显示屏满了的时候，把第一行删了，把后面的行都往前移一行，最后一行都填上空格.

3

`fmt` 是指向 format (即 `x %d, y %x, z %d\n` 这种) 的指针，`ap` 是指向栈里的第一个变量的指针 (即，`ap` 指向的位置存的是 `x` 的值, `ap+4` 指向的位置存的是 `y` 的值, `ap+8` 指向的位置存的是 `z` 的值）.

```
vcprintf(fmt = 0xf0101d4e, ap = 0xf010ff04)
cons_putc('x')
cons_putc(' ')
va_arg(*ap, int) // ap = 0xf010ff04 -> ap = 0xf010ff08
cons_putc('1')
cons_putc(',')
```

```
cons_putc(' ')
cons_putc('y')
cons_putc(' ')
va_arg(*ap, unsigned int) // ap = 0xf010ff08 -> ap = 0xf010ff0c
cons_putc('3')
cons_putc(',')
cons_putc(' ')
cons_putc('z')
cons_putc(' ')
va_arg(*ap, int) // ap = 0xf010ff0c -> ap = 0xf010ff10
cons_putc('4')
cons_putc('\n')
```

ap 的变化通过阅读反汇编代码中 va_arg 的片段也可以看出, 下面代码中 lea 指令就是把 ap 加 4.

```
return va_arg(*ap, unsigned int);
mov    0x14(%ebp),%eax
mov    (%eax),%edx
mov    $0x0,%ecx
lea    0x4(%eax),%eax
mov    %eax,0x14(%ebp)

```

4

输出是 He110 World
57616 的 16 进制是 0xe110, 对应 e1 和 10.
0x00646c72 用小端法存，地址从低到高 4 个字节分别是 0x72, 0x6c, 0x64, 0x00, 对应 r, l, d.
如果是大端法, 将 i 改为 0x726c6400 即可, 57616 不用修改.

5

我们没法预测 y= 后面输出的值，事实上，它是栈上的某个数 (3 的后面那个).

6

只要修改 va_start 和 va_arg 即可.
va_start 让 ap 指向第一个参数, va_arg 每次让 ap 减小即可.

Exercise 9

```
relocated:

```

```
3     # Clear the frame pointer register (EBP)
4     # so that once we get into debugging C code,
5     # stack backtraces will be terminated properly.
6     movl  $0x0,%ebp       # nuke frame pointer
7
8     # Set the stack pointer
9     movl  $(bootstacktop),%esp
10
11    # now to C code
12    call  i386_init
13
14    # Should never get here, but in case we do, just spin.
15  spin:  jmp  spin
16
17
18  .data
19  ###################################################################
20  # boot stack
21  ###################################################################
22    .p2align  PGSHIFT     # force page alignment
23    .globl    bootstack
24  bootstack:
25    .space    KSTKSIZE
26    .globl    bootstacktop
27  bootstacktop:
```

上面是 entry.S 中的部分代码, `movl $(bootstacktop),%esp` 初始化了栈指针%esp, 设为了 `$(bootstacktop)`.
栈的位置是在.data 字段的后面一段, 一直到 `$(bootstacktop)` 为止, 大小为 `KSTKSIZE`.
栈指针%esp 是栈空间的最高处, 栈向下生长.

Exercise 10

每次递归调用的时候，有 32 字节被 push 到栈上，也就是 8 个 32 位的数.
他们按照 push 的顺序分别是%ebp(callee saved), %esi(callee saved), %ebx (callee saved), 额外留空的 12 个字节, 传的参数 (下一次进入函数时候的 x), call 的返回地址.

Exercise 11 和 12

利用%ebp，迭代往前求出每一层递归的%ebp 即可. 具体见代码.