Jos Lab 1: Booting a PC

516030910459 邵欣阳

Part 1: PC Bootstrap

Exercise 1:

Nothing to be done

Exercise 2:

Nothing to be done

Part 2: The Boot Loader

Exercise 3:

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

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

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

```
7d71:   ff 15 18 00 01 00       call   *0x10018
0x10000c:       movw   $0x1234,0x472
```

1. 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?

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

- *eph - ph 的值即是内核代码的segment数。*
- *该信息储存于ELFHDR->e_phnum中。*

Exercise 4:

Nothing to be done

Exercise 5:

- 因为第一个断点处，该处内存未初始化，第二个断点处，内核代码已经被加载进该处内存。
- 第二个断点处，0x00100000是内核代码的.text段起始点。

Exercise 6:

从 boot.asm:77，链接错误的版本和正常版本的执行出现分歧。

Part 3: The Kernel

Exercise 7:

注释掉 kern/entry.S:61 `asm orl $(CR0_PE|CR0_PG|CR0_WP), %eax` 会导致实际运行到 kern/entry.S:68 `asm jmp *%eax`

时，机器会彻底崩溃 (gdb) 0x100028: jmp *%eax 0x00100028 in ?? () (gdb) 0xf010002a: (bad) 0xf010002a in ?? () (gdb) The target architecture is assumed to be i8086 [f000:e05b] 0xfe05b: xor %ax,%ax

Exercise 8:

代码见 lib/printfmt.c: void vprintfmt(…)

Exercise 9:

同上

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?
2. *printf.c 为 console.c 提供了 cprintf(…) 函数作为接口。*
3. *cputchar(…) getchar(…) iscons(…)*

2.Explain the following from console.c:

```
1     if (crt_pos >= CRT_SIZE) {
2             int i;
3             memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *sizeof(uint16_t));
4             for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                     crt_buf[i] = 0x0700 | ' ';
6             crt_pos -= CRT_COLS;
7     }
```

屏幕换行

1. 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?
  fmt 指格式化字符串，ap指第一个参数值(x)。
- 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.
  vcprintf->va_arg->cons_putc 我不列！

4.Run the following code. `c unsigned int i = 0x00646c72; cprintf("H%x Wo%s", 57616, &i);` What is the output? Explain how this output is arrived out in the step-by-step manner of the previous exercise.Here's an ASCII table that maps bytes to characters.

He110 World

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

- *需要设为0x726c6400*
- *不需要*
- In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

行为未定义，因为这是标准中的描述

> If va_arg is called when there are no more arguments in ap, the behavior is undefined.

6.Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

va_arg已经替我们解决了这个问题

Exercise 10:

封装putch为my_putch以计数。

Exercise 11:

重写printnum()，使用非递归的方式，维护一个栈结构，以输出左对齐格式。

Exercise 12:

```
    # 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

    # Set the stack pointer
    movl    $(bootstacktop),%esp

    # now to C code
    call    i386_init

    # Should never get here, but in case we do, just spin.
spin:   jmp spin

f010009a:   81 ec 20 01 00 00       sub    $0x120,%esp

    movl    $(bootstacktop),%esp
```

Exercise 13:

%ebp们。每次递归调用时，调用者的栈底指针。

Exercise 14:

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.

    uint32_t ebp = read_ebp();
    while (ebp)
    {
        cprintf("eip %08x ebp %08x args %08x %08x %08x %08x %08x\n", *(uint32_t *)(ebp + 4), ebp,
            *(uint32_t *)(ebp + 8), *(uint32_t *)(ebp + 12), *(uint32_t *)(ebp + 16), *(uint32_t *)(ebp + 20),
            *(uint32_t *)(ebp + 24));
        ebp = *(uint32_t *)ebp;
    }

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

Exercise 15:

```
    stab_binsearch(stabs, &lline, &rline, 0x44, addr);
```

```
    if (lline <= rline)
    {
        info->eip_line = stabs[lline].n_desc;
    }
    else
    {
        return -1;
    }
```

实现搜索行号 `c while (ebp) { cprintf("eip %08x ebp %08x args %08x %08x %08x %08x %08x\n", *(uint32_t *)(ebp + 4), ebp, *(uint32_t *)(ebp + 8), *(uint32_t *)(ebp + 12), *(uint32_t *)(ebp + 16), *(uint32_t *)(ebp + 20), *(uint32_t *)(ebp + 24)); struct Eipdebuginfo info = {0}; debuginfo_eip(*(uint32_t *)(ebp + 4), &info); cprintf(" %s:%d %.*s+%d\n", info.eip_file, info.eip_line, info.eip_fn_namelen, info.eip_fn_name, *(uint32_t *)(ebp + 4) - info.eip_fn_addr); ebp = *(uint32_t *)ebp; }` 调用此函数，格式化打印输出信息。

Exercise 16:

```
void
do_overflow(void)
{
    cprintf("Overflow success\n");
    uint32_t retaddr = read_pretaddr();
    *(uint32_t *)retaddr = 0xf01008f3;
    return;
}

void
start_overflow(void)
{
    uint32_t retaddr = read_pretaddr();
    *(uint32_t *)retaddr = 0xf0100828;
    return;
}
```

在`do_overflow()`函数的末尾，又使用相同的手段跳转回了正常执行的返回地址，否则机器会崩溃。

Exercise 17:

not implemented