JOS Lab1 文档

516030910293　姚子航

注：本文档为操作系统课程 lab1 设计文档，设计文档按照 lab 文档的顺序结构进行组织。

一、BIOS

BIOS 是英文 Basic Input Output System 的缩写，即基本输入输出系统，在现代计算机中，BIOS 通常是一组固化到计算机内主板 ROM 芯片上的一个程序。它保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序，它可从 CMOS 中读写系统设置的具体信息，如加载 Boot loader 时各个设备的优先级以及 BIOS 密码等。在本次 lab 中，BIOS 部分由 qemu 虚拟机提供。

BIOS 启动后，首先进行的工作是加电自检，该阶段检查计算机各个硬件是否工作，第二个部分的工作是初始化，包括设置中断表，寄存器。对一些外部设备进行初始化检测。第三部分工作就是按照用户存储在 BIOS 中的启动引导设置，搜寻软硬盘驱动器及 CDROM 等有效的启动驱动器，读入操作系统 boot loader，然后将控制权交给 boot loader。

在 JOS BIOS 启动时，按照通常规定，第一句代码位于[f000:fff0]处，执行的指令为 ljmp [f000 : e056]，之后执行 BIOS 的逻辑。

二、 Boot loader

由于历史原因，Boot loader 执行的第一条指令在 0x7c00 地址处，这一设置我们可以在 boot / Makefrag 文件中找到相应代码：

```
$(OBJDIR)/boot/boot: $(BOOT_OBJS)
@echo + ld boot/boot
$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
$(V)$(OBJDUMP) -S $@.out >$@.asm
$(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
$(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

可以看到，在 Makefrag 文件中，我们将 boot loader 要执行的启动函数设置为 start，并且地址为 0x7c00，之后使用 objdump 工具生成反汇编代码，接着使用 objcopy 将生成的 boot.out 的.text 节复制出来形成一个单独的文件，最后使用一个 perl 脚本补齐至一个扇区的大小。

Boot loader 一般分为两部分，汇编部分与 C 语言部分，汇编部分执行简单的硬件初始化，C 语言部分负责复制数据，在本次 lab 中也是分成了这样的两个部分，在 Boot.S 中，操作系统从 16 位实模式切换到保护模式，按照一般设计的规格，所有的 x86 CPU 都是在实模式下开机，来确保传统操作系统的兼容性，在实模式下，对一个内存的访问是通过 Segment：offset 的方式进行的，而到了保护模式。内存管理就分为了分段模式与分页模式两种，段模式虽然是必须的，但是 JOS 通过将各个段的 base 设置成 0，弱化了对段的使用，这一设计在 linux 系统中也有体现。由于将要进入 C 代码，将栈顶设置在 0x7c00 处向下增长。

C 代码中，主要执行的就是使用 readsect、readseg 函数将 JOS 内核从磁盘中读取到内存中来，JOS 内核被组织成 ELF 文件的格式，所以在内存中使用 ELF 相关的结构体访问相关信息。bootmain 函数首先读取一个 page 大小的数据，以保证 ELF header 被完整读取进来，之后按照 header 信息依次读取 segment。最后执行 ELF entry 入口函数跳转到内核代码 main 函数。

三、kernel

内核首先执行的代码位于 entry.S 中，这里我们需要注意 load address 与 link address 的区别，一般来说一个 elf 文件的这两个地址是相同的，但是由于这里是内核可执行文件，操作系统执行环境还没有搭建好，还没有开启页保护模式，所以 load address 只能位于低位地址，而由于之后 kernel 要在高位地址运行，所以 link address 是位于高位，这里有一个关键的函数就是 RELOC 函数，因为 link address 是位于高位，所以文件中所有符号的值都是位于高位，但是在开启分页之前，我们只能在低位访问到它们，

所以我们需要将它们的地址减去 kernel base 才能得到它们真正的物理地址。开启分页后 kernel 通过几行汇编设置了页表，这里我们可以看到一级页表是 entry_pgdir。这个页表定义在 entrypgdir.c 中：

```
__attribute__((__aligned__(PGSIZE)))
pde_t entry_pgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE>>PDXSHIFT]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
};
```

我们可以看到它将 VA 的 [0, 4MB) 与[KERNBASE, KERNBASE+4MB)映射到 PA 的 [0, 4MB)，可以看到二级页表为 entry_pgtable 变量，并且物理地址也需要减去 kernel base。二级页表由于 entry 过多，篇幅所限不进行展示。以上页表只是一个临时的页表，操作系统真正要使用的页表会在内存管理，及下一个 lab 中构建。在设置好页表后，kernel 重新设置了 stack 的位置为 bootstacktop，之后调用 i386_init 执行 C 代码。

四、完成格式化输出

monitor.c, printfmt.c, console.c, printf.c 存储了一系列控制台输出的函数，他们的调用关系如下：

vcprintf-> vcprintfmt-> putch-> cputchar-> cons_putc

exercise8 的任务相对简单，按照 16 进制写即可，exercise9 中的符号位，我采用了一个 flag 来进行记录，并且在格式化的时候将 flag 与要输出的数字传进自定义的函数中，并将新的 flag（一般是 0）返回更新 flag。在完成 exercise10“%n”格式化符的时候，我直接使用了 putdat 这一变量来获取已输出字符数量。exercise11 的靠右填充，我也是使用了一个 flag 来记录是否出现%-符号，并将 flag 传入 printnum 函数控制填充左边还是右边。

五、Backtrace & Overflow & time

实现栈回溯，我只需要按照代码与文档注释中提示的方法，调用 read_ebp 函数与 debuginfo_eip 函数，再根据 x86 的 calling convention 即参数压栈顺序等等规则按照文档的格式输出。而每次循环结束，只需要访问 ebp 所指向的地址处的内容，即可得到上一个 stack frame 的 ebp 寄存器，直到 ebp 寄存器为 0，就代表我们的回溯已经结束。

Overflow 的函数相对也并不复杂，先调用所给 read_pretaddr 函数获取返回地址位置，只需要将 overflow 函数地址填进去就可以，按照要求我只能使用%n 符号来填写这个地址，我使用的方法是新建一个长度为 256 的字符数组，一次填写一个字节，在那个要填写的字节值偏移处将数组元素置为 ‘\0’，输出这个字符数组，就能达到我们填写的效果，连续写四次就可以篡改返回地址，注意这里有一个问题要小心，就是我们在 overflow 结束后还要返回之前应该返回的那个地址，这里我采取了直接将合法返回地址向高位移动 4 字节的做法，注意这样是会破坏我们的栈，但是本次 lab 由于被破坏的空间没有再被使用所以没有出问题。

Time 命令行指令的实现中我使用了文档上推荐的 rdtsc 指令结合内联汇编语句实现了一个 rdtsc()帮助函数，剩下的仿照 monitor.c 文件中已有的函数实现就可以达到计算命令执行 cycle 的效果。在下面附上 rdtsc 函数：

```
uint64_t rdtsc()
{
    uint32_t low,high;
```

```
    __asm__ __volatile__
    (
     "rdtsc":"=a"(low),"=d"(high)
    );
    return (uint64_t)high<<32|low;
}
```