Lab 1:Booting a PC

Introduction

说明本文主要分为三个部分：

- 熟悉x86汇编语言，Qemu x86模拟器，PC的加电引导过程。
- 阅读在lab源码树boot目录中6.828内核的启动引导代码。
- 深入研究被命名为JOS的6.828内核初始化模板部分的代码，该部分位于kernel目录。

Software Setup

介绍使用git工具下载对应的lab源码。

```
athena% mkdir ~/6.828
athena% cd ~/6.828
athena% add git
athena% git clone https://pdos.csail.mit.edu/6.828/2018/jos.git lab
Cloning into lab...
athena% cd lab
athena%
```

Hand-In Procedure

介绍了如何提交代码，直接使用命令提交修改的代码。

```
make handin
```

Part 1: PC Bootstrap

Getting Started with x86 assembly

提供了部分资料用于学习x86汇编:

```
PC Assembly Language
Brennan's Guide to Inline Assembly
```

> 80386 Programmer's Reference Manual
> IA-32 Intel Architecture Software Developer's Manuals

`Exercise 1:` 熟悉汇编语言资料，不需要通读但是编写x86汇编时需要知道如何进行索引查找。

Simulating the x86

使用make命令编译内核。

> athena% cd lab
> athena% make

使用make qemu 或者 make qemu-nox进行操作系统启动。

The PC's Physical Address Space

熟悉内存地址分布图:

```
+------------------+ <- 0xFFFFFFFF (4GB)
+    32-bit      +
+ memory mapped  +
+    devices     +
+                +
      ...
+                +
+    Unused      +
+                +
+------------------+ <- depends on amount of RAM
+                +
+   Extended     +
+    Memory      +
+------------------+ <- 0x00010000 (1MB)
+   BIOS ROM     +
+------------------+ <- 0x000F0000 (960KB)
+16-bit devices,+
+expansion ROMs +
+------------------+ <- 0x000C0000 (786KB)
+  VGA Display   +
+------------------+ <- 0x000A0000 (640KB)
+                +
+   Low Memory   +
+                +
+------------------+ <- 0x00000000
```

注意：BIOS的地址是从0x00F00000 - 0X00100000的64KB

The ROM BIOS

- PC启动地址是0x000ffff0
- PC启动时 CS=0xf000, IP=0xfff0
- 第一个指令是一个跳转指令，跳转到了 CS=0xf000 和 IP=0xe05b的地址

Quetion 1: 为啥是跳转到0xe05b，这个地址不对齐啊？

物理地址计算公式：**物理地址 = 16 * 段地址 + 偏移地址**

```
Exercise 2: 逐条指令理解BIOS开始时做的工作:

inst[1]: [f000:fff0]    0xffff0:        ljmp   $0x3630,$0xf000e05b
# 跳转到0xfe05b地址，这个0x3630是什么(应该是使用的虚拟机为x86_64的缘故)?
# 段地址不应该是0xf000

inst[2]: [f000:e05b]    0xfe05b:        cmpw   $0xffc8,%cs:(%esi)
# 立即数和寄存器值比较，判断是否为空

inst[3]: [f000:e062]    0xfe062:        jne    0xd241d0a8
# jne 标志Z不等于0，则跳转到0xd241d0a8,cpmw比较之后设置标志Z

inst[4]: [f000:e066]    0xfe066:        xor    %edx,%edx
# 如果没跳转，edx寄存器自己和自己异或，相当于把edx清零了

inst[5]: [f000:e068]    0xfe068:        mov    %edx,%ss
# 赋值0给Stack Segment。ss，sp求物理地址的公式是=ss* 16+ sp
```

Part 2: The Boot Loader

BIOS从能够启动的软盘或者硬盘读取512字节到内存的0x7c00-0x7dff地址，然后使用jmp指令跳转到 CS:IP => 0000:7c00 的地址，至此BIOS将控制权转移给了bootloader。
JOS的bootloader主要位于boot/boot.S和boot/main.c文件，主要做了以下两件事：

1. 将处理器从实模式转换到了32位保护模式，因为只有在32位保护模式下才能程序才能够访问大于1M的内存地址。此时需要了解在保护模式下将分段地址转换成物理地址的方式有什么不同，并且在转换偏移量之后是32位而不是16位。
2. 引导加载程序通过x86的特殊I / O指令直接访问IDE磁盘设备寄存器，从而从硬盘读取内核

> Exercise 3: 熟悉GDB命令，特别是针对OS工作有用的一些深奥的GDB命令。
> 在地址0x7c00设置断点追踪代码运行并于反汇编的boot.asm相比较。从bootmain()函数追踪到readsect()函数。

```
#include <inc/mmu.h>

# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG, 0x8         # kernel code segment selector 内核代码段 .set赋值宏命令
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector 内核数据段
.set CR0_PE_ON,      0x1         # protected mode enable flag   保护模式使能标志位

.globl start
start:
  .code16                     # Assemble for 16-bit mode 汇编成16位模式
  cli                         # Disable interrupts  关中断，启动时不可被打断，相反的命令是sti
  cld                         # String operations increment 地址指针递增,相反指令std

  # 在计算机中，大部分数据存放在主存 中，8086CPU提供了一组处理主存中连续存放的数据串的指令——串操作指令
  # 串操作指令中，源操作数用寄存器SI寻址，默认在数据段DS中，但允许段超越；
  # 目的操作数用寄存器DI寻址，默认在附加段ES中，不允许段超越。
  # 每执行一次串操作指令，作为源地址指针的SI和作为目的地址指
  # 针的DI将自动修改：+/-1（对于字节串）或+/-2（对于字串）。
  # 地址指针是增加还是减少取决于方向标志DF。在系统初始化后或
  # 者执行指令CLD指令后，DF=0，此时地址指针增1或2；在执行指令STD后，DF=1，此时地址指针减1或2。

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero 置零
  movw    %ax,%ds             # -> Data Segment   数据段寄存器
  movw    %ax,%es             # -> Extra Segment  附加段寄存器
  movw    %ax,%ss             # -> Stack Segment  堆栈段寄存器

  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
  inb     $0x64,%al               # Wait for not busy 读取键盘状态寄存器
  testb   $0x2,%al                # 测试第2位，第二位表示缓冲区满，有给8042的数据
  jnz     seta20.1                # 结果不为0则跳转(zf标志位为0)

  movb    $0xd1,%al               # 0xd1 -> port 0x64 开启A20控制的命令
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60 这个是参数
  outb    %al,$0x60
```

```
  # 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.

  # 从实模式转换到保护模式

  lgdt    gdtdesc
  movl    %cr0, %eax            # 取cr0寄存器值到eax寄存器
  orl     $CR0_PE_ON, %eax      # 打开CR0_PE_ON, cr0寄存器的第0位PE为开启保护模式的标志位
  movl    %eax, %cr0            # 再写回cr0寄存器

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg #跳转到代码段


  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp            # start的地址给esp寄存器,相当于栈地址从0x7c00开
  call bootmain

  # If bootmain returns (it shouldn't), loop.
spin:
  jmp spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL                              # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff)     # code seg      代码段
  SEG(STA_W, 0x0, 0xffffffff)           # data seg    数据段

gdtdesc:
  .word   0x17                            # sizeof(gdt) - 1 表长,这里表长为23字节,为啥不是24么？
  .long   gdt                             # address gdt     表基地址
```

完成代码的阅读之后回答以下问题：

1. 处理器从什么时候开始执行32位代码？究竟是什么原因导致从16位模式切换到32位模式？
   答：代码段从ljmp protcseg开始执行32位代码，打开PE标志从16位模式转换到32位模式。

2. 引导加载程序执行的最后一条指令是什么，刚加载的内核第一条指令是什么？
   答：引导程序最后一条指令是((void (*)(void)) (ELFHDR->e_entry))()，内核第一条指令是

```
movw   $0x1234,0x472
```

3. 内核的第一条指令在哪里？
   答：内核第一条指令是在elf的入口点，物理地址是0x10000c。
4. 引导加载程序如何确定必须读取多少个扇区才能从磁盘获取整个内核？在哪里可以找此信息？
   答：