OS_lab1

最近编辑过的 2011年3月14日

系统启动过程：

1. 开机以后，计算机将CS=0xF000, IP=0xFFF0, 物理地址为0xFFFF0，这个位置是BIOS ROM所在位置(0xF0000~0xFFFFF)，所以机器一开始控制权交给BIOS
2. BIOS接到控制权以后，肯定跳转，因为0xFFFF0到0xFFFFF就1Byte信息，这个无法做任何事，所以跳到前面位置开始运行
3. BIOS做的工作主要是初始化一些关键的寄存器和中断开关，做完后，BIOS从IDE硬盘第一个扇区512Byte读入boot loader到内存的0x7c00到0x7dff，然后设置CS=0x0000, IP=0x7c00，控制权交给boot loader
4. boot loader的任务是将处理器从实模式切换到保护模式，然后将内核kernel从硬盘上读取到内存中，然后切换到内核开始启动操作系统
5. kernel放置在硬盘上，在JOS中是存在在了紧接在boot loader的第二个扇区里，如果想修改位置的话，需要修改产生硬盘镜像的kern/Makefrag文件，看到第73行：

```
68 # How to build the kernel disk image
69 $(OBJDIR)/kern/kernel.img: $(OBJDIR)/kern/kernel $(OBJDIR)/boot/boot
70     @echo + mk $@
71     $(V)dd if=/dev/zero of=$(OBJDIR)/kern/kernel.img~ count=10000 2>/dev/null
72     $(V)dd if=$(OBJDIR)/boot/boot of=$(OBJDIR)/kern/kernel.img~ conv=notrunc 2>/dev/null
73     $(V)dd if=$(OBJDIR)/kern/kernel of=$(OBJDIR)/kern/kernel.img~ seek=1 conv=notrunc 2>/dev/null
74     $(V)mv $(OBJDIR)/kern/kernel.img~ $(OBJDIR)/kern/kernel.img
75
```

seek = 1表示是硬盘上第二个扇区，72行将boot即是将boot loader放入第一扇区。如果想改变kernel在硬盘上的位置，那么boot loader在载入内核的时候也需要做相应的修改，从对应的硬盘扇区中读取内核文件

关于内核kernel的ELF文件

1. 内核可能很大，我们不知道会占多少个扇区，但是内容区(.text, .rodata, .stab, .data ...etc)被两种方式组织起来，第一种就是分节(.text就是一节, .rodata就是一节)，另一种就是分段，按照程序头中的程序头表分开，一个段可能包含多个节。但是一般我们都以段的方式读入整个ELF文件，因为ELF文件格式是：ELF文件头，程序头表，文件内容，节头表这样顺序排列的，我们一般只需要读入前两个部分：ELF文件头和程序头表后，就知道整个kernel需要读取多大的空间了。
2. 内核载入分为两部，根据上面一条，我们不知道kernel的具体大小，所以第一步我们将ELF文件的文件头和程序头表读入内存的0x10000地址（后接4个0，按照memory layout应该是内存最前面的一块：low memory），作为临时空间，然后根据从中得到的ELF段信息，这时才将硬盘上的ELF文件解压到内存的0x100000（后接5个0，是内存空间里BIOS ROM之后的空间）作为内核的放置地址。

所以我们可以看到boot/main.c中45行，ELFHDR的地址设置为0x10000，这一步就是读取部分的ELF文件，为释放整个内核作准备

```
43     // read 1st page off disk
44     readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
45
```

设置为0x10000的原因为这个地址在low memory段，读取4KB还没有超出这个memory段， 但是为什么4KB就能读取完整个ELF文件头和程序头表，这个我还没搞懂，如果极限情况下内核文件巨大的话，程序头表中段数目应该会非常多才对。

而后面从50行开始，才是解压整个ELF内核

```
50     // load each program segment (ignores ph flags)
51     ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
52     eph = ph + ELFHDR->e_phnum;
53     for (; ph < eph; ph++)
54         readseg(ph->p_va, ph->p_memsz, ph->p_offset);
```

这里读出的ELFHDR->e_phoff, ph->p_va都是链接地址0xf010000x（5个0）开头的，这个与其内核最终实际加载到的内存位置是对应的。（当然访问之前需要和0xFFFFFF做与操作，这就是一个简单的手动地址转换）

3. 为什么kernel在objdump得到 入口地址是0xf010000c？是人为规定的么？老板告诉我在kern/kernel.ld文件里，第10行开始就是指定这个内核最终链接以后载入的地址是哪里

```
 8 SECTIONS
 9 {
10     /* Load the kernel at this address: "." means the current address */
11     . = 0xF0100000;
```

相应的入口地址则通过其在ELF文件内的相对地址和这个载入地址共同算出来为0xf010000c。这里就涉及到load address和link address的问题，lab1的文档中有相应的说明。

4. ELF文件的具体信息可以由objdump命令完成，执行objdump -x obj/kern/kernel可以得到入口地址、 程序头表 以及节头表的详细信息，如下输出所示：

```
zhangchi@zhangchi-laptop:~/Course/OSPractise/lab/obj/kern$ objdump -x kernel

kernel:     file format elf32-i386
kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xf010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0xf0100000 align 2**12
         filesz 0x000072e7 memsz 0x000072e7 flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0xf0108000 align 2**12
         filesz 0x00008320 memsz 0x00008980 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rwx

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000019f5  f0100000  f0100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000006dc  f0101a00  f0101a00  00002a00  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000038e9  f01020dc  f01020dc  000030dc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001922  f01059c5  f01059c5  000069c5  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00008320  f0108000  f0108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000660  f0110320  f0110320  00011320  2**5
                  ALLOC
  6 .comment      00000023  00000000  00000000  00011320  2**0
                  CONTENTS, READONLY
```

关于打印：

1. 为什么在kern/console.c中cga_print的第177行，打印完\n之后不break?

```
169     switch (c & 0xff) {
170     case '\b':
171         if (crt_pos > 0) {
172             crt_pos--;
173             crt_buf[crt_pos] = (c & ~0xff) | ' ';
174         }
175         break;
176     case '\n':
177         crt_pos += CRT_COLS;
178         /* fallthru */
179     case '\r':
```

答：是为了换行以后光标马上跳到当行的首位，江主席威武

2. 在lib/vprintfmt.c中vprintfmt函数89行定义段, lflag代表打印整数的类型，0为int, 1为long, 2为long long, 其中int和long有区别么。

```
 83 void
 84 vprintfmt(void (*putch)(int, void*), void *putdat, const char *fmt, va_list ap)
 85 {
 86     register const char *p;
 87     register int ch, err;
 88     unsigned long long num;
 89     int base, lflag, width, precision, altflag;
 90     char padc;
```

答：貌似JOS是可以干到64位机器上的

关于内核

1. kernel中所有的入口和链接地址都是0xf01000x，高位为f，实际上物理内存没那么高，代码都是放在内存0x100000开始的内存段，中间的地址转换怎么做的？bochs的可以查看GDT，可以看到Code segment和Data Segment的起始位置都是0x10000000，所以和f一加上进位就抹掉了。qemu何如查看GDT?

好像木有。。。可以查看info registers看gdt头地址，然后看内存xp/Nx paddr看具体内容，如果有bochs的info gdt命令就好了

2. 内核在进行初始化的时候kern/entry.S中，会进行新的GDT读取工作，原来的GDT无论是代码段和数据段起始地址都是0，现在新的变成了-KERNBASE, KERNBASE=0xF0000000，和链接地址0xf10000c一相加，得到的地址就是真正在内存中的地址了。

3. 内核初始化部分kern/init.c中i386_init的第30行，为什么要初始化所有global data？ 因为bss代表的段是C语言中声明的全局变量，C语言要求这些变量在初始时全部为0，boot loader在将内核载入内存时并没有做bss段相应的初始化工作，所以在运行内核之前，一定要保证这些变量的初始化正确设置为0

```
22 void
23 i386_init(void)
24 {
25     extern char edata[], end[];
26
27     // Before doing anything else, complete the ELF loading process.
28     // Clear the uninitialized global data (BSS) section of our program.
29     // This ensures that all static/global variables start out zero.
30     memset(edata, 0, end - edata);
```