话说看 linux 内核源码真是一件辛苦的事情啊，为了弄清楚操作系统，我从 linux 源码看到 grub 源码，再看到 BIOS，真心是伤不起啊。我研究的 linux 内核版本是 2.6.34.13，它不支持从直接从内核启动，需要一个 bootloader。目前 linux 中使用最广泛的的 bootloader 是 grub，本文就是对 grub2.00（下文中简写为 grub）在 x86 下的分析研究。

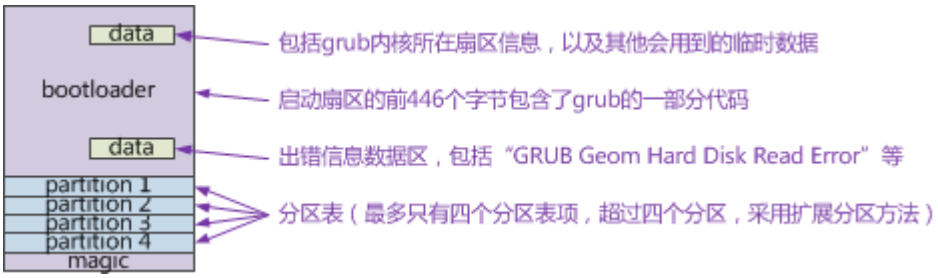
grub 的源码主要分为四个部分，分别为 grub-core/boot/i386/pc/boot.S， grub-core/boot/i386/pc/ diskboot .S，grub-core/boot/i386/pc/ startup_raw .S，以及 grub 的核心代码 。

- boot.S 是 MBR 中的 512 个字节，主要工作是加载 diskboot.S。
- diskboot.S 也是 512 字节的代码，它的作用是加载 startup_raw.S 和 grub 的核心代码。
- startup_raw.S 的作用是解压 grub 的核心代码。
- grub 的核心代码是 grub 真正功能实现的地方。

一般 boot.S 是放在第一个扇区（即 MBR）， diskboot.S 是放在第二个扇区，startup_raw.S 和 grub 核心代码放在接下来的几十个扇区（但是限于 0-63 扇区）。下面按照 grub 的执行流程来研究 grub 的工作原理。

1. grub-core/boot/i386/pc/boot.S

计算机启动后（略过 BIOS 部分）会将硬盘的第一个扇区加载到 0000:7C00 处，这个扇区的布局如下图所示。



此时 IP 寄存器的值为 7C00，第一条指令为跳转指令，跳到 BPB（BIOS Parameter Block）后面的代码处执行。这一部分比较简单，首先初始化段寄存器，然后从启动盘读取第二个扇区到 0000:8000 处。

```
/* boot kernel */
jmp *(kernel_address) /* kernel_address = 0000:8000 */
```

随这这一条指令的执行，boot.S 完成了它的使命，将执行权交给了 diskboot.S。

2. grub-core/boot/i386/pc/diskboot.S

diskboot.S 的工作跟 boot.S 类似，它会根据配置信息从启动盘中读取不多于 62 个扇区的数据。它会将这些数据放在 0000:8200 处的一段 内存中，其中开始部分是 start_raw.S 的代码，后一部分是压缩的 grub 核心代码。

ljmp $0, $(GRUB_BOOT_MACHINE_KERNEL_ADDR + 0x200)

这条跳转指令是 diskboot 的最后一条指令（不考虑意外情况），实际上就是跳转到 0000:8200 处。

3. grub-core/kern/i386/pc/startup_raw.S

startup_raw.S 的部分关键代码如下，首先需要 设置数据段、堆栈段和扩展段寄存器，以及栈指针。

```
/* set up %ds, %ss, and %es */
xorw %ax, %ax
movw %ax, %ds
movw %ax, %ss
movw %ax, %es
/* set up the real mode/BIOS stack */
movl $GRUB_MEMORY_MACHINE_REAL_STACK, %ebp
movl %ebp, %esp
```

进行一些准备工作后开始进入保护模式， real_to_prot 这个函数的代码在 grub-core/kern/i386/realmode .S 中 。

```
/* transition to protected mode */
DATA32 call real_to_prot
```

进入保护模式后就调用 _LzmaDecodeA 解压压缩的核心代码， _LzmaDecodeA 这个函数在 lzma_decode.S 中定义 。紧接着的几条指令的作用是设置参数，然后跳转到核心代码。

```
movl $GRUB_MEMORY_MACHINE_DECOMPRESSION_ADDR, %edi
        ...
        popl %esi
movl LOCAL(boot_dev), %edx
movl $prot_to_real, %edi
movl $real_to_prot, %ecx
movl $LOCAL(realidt), %eax
jmp *%esi
```

解压后的核心最开始放在 0x00100000 处，在上面的指令中，esi 寄存器的值就是核心代码的地址，edx、edi、ecx、eax 分别是启动设备、 从保护模式进入实模式函数的地址、从实模式进入保护模式函数的地址、实模式中断描述符表的地址。

4. grub-core/kern/i386/pc/startup.S

startup.S 首先保存传递过来的参数，具体实现是下面的三条指令。

```
movl %ecx, (LOCAL(real_to_prot_addr) - _start) (%esi)
movl %edi, (LOCAL(prot_to_real_addr) - _start) (%esi)
movl %eax, (EXT_C(grub_realidt) - _start) (%esi)
```

为什么不直接使用这几个变量的地址呢？这是因为 startup.S 当前所在地址为 0x00100000，而代码的目标地址为 0x00008200，所以需要转换一下变量的地址。

现在需要将核心代码移动到目标地址，下面的代码完成了移动的工作。

```
/* copy back the decompressed part (except the modules) */
movl $(_edata - _start), %ecx
movl $(_start), %edi
rep
movsb
movl $LOCAL (cont), %esi
jmp *%esi
LOCAL(cont):
```

ecx 的值是核心代码的大小，由_edata 减_start 得来。edi 的值为_start，也就是目标地址。esi 是核心代码现在所在的地址，在 startup_raw.S 中设置，到现在一直未有变动。

移动完成以后，jmp 指令完成了到目标地址的跳转，这个跳转很巧妙，虽然整个代码移动了位置，但看起来就像没有移动一样。

```
/*
 *  Call the start of main body of C code.
 */
call EXT_C(grub_main)
```

接下来清空 BSS（Block Started by Symbol ），调用 grub_main 进入 grub 的主函数，这个函数在 grub-core/kern/main.c 中 。到这来我们已经来到了 grub 的核心代码部分，这部分主要是 grub 的模块化框架的初始化，各种命令的注册，各种模块的加载等等。

我比较关心的是 grub 在用户选择指定的系统后怎么加载 linux 的，精力有限，其它代码暂时就不去阅读了。当用户选择指定的内核条目后， grub-core/commands/boot.c 中的命令函数 g rub_cmd_boot 开始执行 。

加载 linux 的代码在 grub-core/loader/i386 目录下，grub 支持 16 位、32 位、64 位三种模式启动 linux 内核。16 位模式 下会重新回到实模式，再启动 linux 内核。32 位

和 64 位两种模式下不需要再进入实模式，直接在保护模式启动。

linux.c 文件中的 grub_cmd_linux 函数就是处理 linux 内核加载的 。函数 grub_cmd_linux 的作用是根据命令行参数生成相关配置数据，然后设置一个回调函数，由 g rub_cmd_boot 调用。这样做的好处是减小了代码的耦合度。

g rub_cmd_boot 和 grub_linux_boot 两个函数 将 linux 内核从文件系统中读取到内存中，内核代码分为两部分，一部分是实模式代码，一部分是保护模式代码。实模式代码一般放在 0x00009000 处，保护模式代码一般放在 0x00100000 处，在 32 位和 64 位两种模式启动实际上不需要实模式部分代码 。此外， g rub 为 linux 内核设置了 linux_kernel_params 参数，这些参数在 linux 内核中会用到。

5. grub_relocatorXX_boot

离进入 linux 内核就差最后一步了， grub_linux_boot 最后调用 grub_relocatorXX_boot（.../ lib/.../ relocator.c ） 进入内核。顾名思义，最后所要做的事情是重定位。 grub_cmd_linux 只是将加载了内核的代码 ，这部分代码要能使用还必须对代码的每个 chunk 进行重定位。

重定义这部分代码不是很复杂，需要结合 ELF 文件格式阅读，这里就不多说了。调用 grub_relocator_prepare_relocs 重定位好之后，如下面代码所示，先关闭中断，然后调用 relst() 就进入到了 linux 内核， relst()后面的代码是永远不会执行的 。

```
asm volatile ("cli");
        ((void (*) (void)) relst) ();
        /* Not reached.  */
        return GRUB_ERR_NONE;
```

到这里为止，本文就算是把 grub 启动 linux 简要叙述了一遍，其中还有很多细节这里没有讲，以后有时间再好好看看。

- 分类：Linux 内核
- 字数：1740

notishell

关注此人

- 粉丝: 8 博客数: 7 共码了 15231 字

评论（1）

叮了个咚

1 楼 2013/08/03 21:36

写得非常好，最近也在学习 grub2，有时间一起讨论学习。

关闭 回复

插入： 表情 开源软件

登录后评论

微信分享

顶部