X86平台操作系统引导技术研究与实现

吴兆芝
(南京晓庄学院 数学与信息技术学院,江苏 南京 211171)

摘 要:文章介绍X86平台软盘引导操作系统的三种典型介质存储结构,以一个演示实验为例阐述保护模式下操作系统引导启动的常用技术及实现方法,内容包括操作系统的软盘引导、打开A20线、设置GDT及IDT、进入保护模式、安装ELF格式内核模块、内核框架、直接写屏、以及启动运行内核模块等。

关键词:操作系统;引导技术;X86平台;保护模式;ELF格式内核模块
中图分类号:TP316 **文献标识码**:A **文章编号**:1009-7902(2011)06-0094-04

操作系统的引导是操作系统运行的前提,包括两大内容:一是介质引导,将操作系统磁盘镜像从外部存储介质读进内存;二是内核启动,将读入内存的操作系统内核模块安装到指定的地址空间、为操作系统的运行创建必要的工作环境、最后启动操作系统内核等。事实上,无论操作系统的复杂程度如何,其引导启动的基本过程没有本质上的区别。然而,在具体的实现技术上各种操作系统却各具特点、互不相同,如引导介质不同、磁盘镜像的介质存储结构不同、内核运行环境及其创建技术不同等等。

本文以X86平台作为硬件环境,阐述操作系统引导技术的研究与实现。首先介绍X86平台软盘引导操作系统的三种典型存储结构,然后以一个演示实验为例阐述保护模式下操作系统引导启动的常用技术及实现方法,内容包括操作系统的软盘引导、打开A20线、设置GDT及IDT、进入保护模式、安装ELF格式内核模块、内核框架、直接写屏、以及启动运行内核模块等。

1 软盘存储操作系统磁盘镜像的典型结构

操作系统的磁盘镜像通常由三部分组成:引导模块、装载模块及内核模块。

引导模块的作用是被BIOS调用将装载模块和内核模块从软盘读进内存。由于BIOS只能将软盘的引导扇区(第一个扇区)调入内存运行,而一个扇区只有512字节,不足以完成启动操作系统的全部复杂工作,所以引导扇区只能容纳一个简单的引导模块。当装载模块被读进内存之后,引导模块将程序控制权交给装载模块,由其完成后续工作。

装载模块的任务是完成系统内核的启动,对于X86平台而言,内容包括进入保护模式、安装已读入内存的内核模块、启动运行内核模块等。

内核模块是操作系统的核心,随着操作系统的复杂程度不同,其程序结构及实现的功能也各不相同。

一般来说,引导模块及装载模块要求程序尽量简洁高效,所以其目标代码采用无格式二进制结构。内核模块根据操作系统总体结构及内存部局的不同,其目标代码一般有两种:无格式二进制结构和ELF格式二进制结构。

操作系统程序模块在软盘上的存储方式有两类:扇区方式存储和文件方式存储。二者都将引导模块存放在软盘引导扇区,但前者将装载模块和内核模块继续按扇区方式顺序存放在软盘上,而后者则将装载模块和内核模块以文件方式存放,一般采用最简单常用的Fat文件格式。

通常,操作系统文件模块在软盘上的典型存储结构有以下三种:

(1) 以扇区方式存储装载模块和无格式二进制内核模块,Linux[1]、Minix[2]、GeekOS[3]等都是采用

收稿日期:2011-06-30

作者简介:吴兆芝(1954—),女,吉林梅河口人,南京晓庄学院数学与信息技术学院高级实验师,研究方向:计算机应用技术。

这种方式。由于内核模块是无格式二进制代码，所以在内存安装时无需重定位，一般其基地址为0。

（2）以文件方式存储装载模块和ELF格式二进制内核模块，通常用于实验性的小型操作系统。这种方式的软盘已安装了某种文件系统，如Fat文件系统，装载模块和内核模块都以文件方式存放在磁盘上。由于内核模块为ELF格式，所以在引导启动过程中要将内核模块动态安装。使用ELF格式内核模块可以灵活配置内核基址，以满足系统特殊需求。如将内核基址设置为0x1000，则可使内核空间避开内存低端的BIOS数据区，从而使保护模式下工作的内核能够方便地访问BIOS数据。

（3）以扇区方式存储装载模块和ELF格式二进制内核模块，相当于前两种方式的折衷，既简化了软盘读操作，又使用了ELF格式内核文件以便于内核使用BIOS数据区。本文描述的演示实验采用的就是这种方式。

2 引导技术研究与实现

本文以一个演示实验为例，描述操作系统引导技术的研究与实现。演示实验由5个程序文件、1个包含文件及3个Makefile脚本文件构成，其中boot.s是引导模块文件、load.s是装载模块文件、size.s是辅助数据文件、kasm.s是内核框架文件、main.c是内核主函数文件、kasm.inc是汇编程序包含文件。

2.1 软盘引导

这部分功能由引导模块实现。该程序存储在软盘引导扇区，即第一个逻辑扇区。系统加电后，该程序被BIOS读入内存0x7C00处，共512字节。然后系统跳转到0x7C00执行该程序，首先将自身512字节移动到0x90000处，然后跳转到该处继续执行，将装载模块从0x90200处装入、再将内核模块从0x70000处装入。至此，引导模块任务完成，软盘引导结束，程序跳转到0x90200处，执行安装模块任务。

引导模块在读入软盘扇区时，首先显示提示字符串“Loading”表示开始读盘，然后每读一个扇区，将在屏幕上显示一个“.”以提示读盘进度。

读盘操作结束后，不再需要软盘，因此将软盘马达关闭。

由于装载模块和内核模块按软盘扇区顺序依次存放，引导模块在从软盘读取这两个模块时无法直接判定其长度。因此，在引导模块程序中专门设置了两个变量，用来存放上述两个模块的扇区数，在读取软盘过程中根据这两个数据决定读取的扇区数。

这两个变量放在引导扇区尾部结束标志之前，其汇编伪指令定义为：

```
. org0x1fa
load_sects:. word0
kern_sects:. word0
```

显然这两个变量的值应该分别是装载模块和内核模块的扇区数，而不应该为0。如果在源程序中直接设置，则意味着每当装载模块或内核模块经过修改重新汇编或编译链接后，都要重新手工修改这两个变量的定义值，并重新汇编链接引导模块，这无疑不是一个好办法。

本文实验采用修改目标代码的方法，不需修改引导模块源程序，只需将装载模块和内核模块长度的二进制代码值重新写入引导模块目标程序偏移量0x1FA处即可。

为了得到这两个模块目标代码长度的二进值代码，设计了一段编译脚本，用标准的Shell命令检测文件长度并动态生成一个名为size.s的汇编语言源程序。将该程序汇编链接后则得到所需结果。

编译脚本代码如下：

```
$(SIZEBIN): $(LOADBIN) $(KERNELF)
    @(echo -n "LSIZE = (";stat -c%s $(LOADBIN) \
        | tr '\012' ' '; echo " + 511 )/512") > $(SIZEASM);
    @(echo -n" KSIZE = (";stat -c%s $(KERNELF) \
        | tr '\012' ' '; echo " + 511)/512") > > $(SIZEASM);
    @(echo". global_start") > > $(SIZEASM);
    @(echo" _start:. word LSIZE, KSIZE") > > $(SIZEASM);
    (cd boot; make size.bin)
```

其中，stat命令获得由换行符结尾的文件长度字符串，tr命令去掉结尾的换行符。不难看出，该段脚本动态生成的size.s文件代码应为如下结果：

```
LSIZE = (1536 + 511)/512
KSIZE = (7656 + 511)/512
. global_start
_start:. word LSIZE, KSIZE
```

其中1536和7656分别是编译脚本测得的文件长度。

将这个汇编源程序汇编链接后得到目标程序size.bin，其内容为4个字节，分别为0x03，0x00，0x0f，0x00，代表装载模块的扇区数为3、内核模块的扇区数为15。

最后，编译脚本用dd命令修改引导模块目标代码：

```
dd if = size. bin of = boot. bin seek = 506 bs = 1 count = 4 conv = notrunc
```

2.2 打开 A20 线

在机器启动时，A20 线默认是禁止的，安装模块的第一项任务是打开 A20 线，为进入保护模式做准备。开启 A20 地址线有几种方法，如设置 0x92 端口值、设置键盘控制器的端口值或读 0xEE 端口值等。

本文实验程序采用第一种方法实现对 A20 地址线的控制。该方法简单方便，0x92 端口值的 bit 1 用来控制 A20 的打开与关闭：1 为打开、0 为关闭。打开 A20 线只需如下三行汇编代码即可：

```
in $0x92,%al
or $0x2,%al
out%al, $0x92
```

2.3 设置 GDT 及 IDT

在保护模式下，每一个进程都要有自己的 TSS 和 LDT，从而在 GDT 中要有两个对应的表项，一个是进程的 TSS 描述符，一个是进程的 LTD 描述符。

本文实验程序的 GDT 含有 4 个全局描述符，分别为空描述符、代码段描述符、数据段描述符及视频显示区数据段描述符。由于只有 1 个内核进程，所以只需 2 个局部描述符表项，分别为内核进程的 TSS 描述符及 LDT 描述符。本文实验不涉及中断程序，所以只设置了 IDT 空间而不含任何实质性内容。由于 GDT 及 IDT 规模较小，将其移动到 0xA0000 之下，然后用 lgdt 和 lidt 指令分别安装 GDT 及 IDT。

2.4 进入保护模式

至此，进入 32 位保护模式的准备工作已经就绪，下一步工作是加载机器状态字，也称控制寄存器 CR0，其 bit 0 置 1 将导致 CPU 工作在保护模式。主要有两种方法实现，一是使用汇编指令 lmsw（Load Machine Status Word），一是直接对 CR0 编程。本文采用第二种方法，汇编代码如下：

```
mov%cr0,%eax
or $0x1,%eax
mov%eax,%cr0
```

最后，用一条长跳转指令转移到保护模式程序入口。

进入到 32 位保护模式后的第一项任务就是初始化各段寄存器包括堆栈段，然后动态安装 ELF 格式内核模块，最后跳转到内核模块入口启动内核运行。

2.5 安装 ELF 格式内核模块

ELF[4]（Executable and Linkable Format）最初由 Unix 系统实验室作为应用程序二进制接口开发与发行，后来逐步发展成为可执行文件的标准格式。ELF 格式标准包括三种目标文件类型：可重定位（relocatable）目标文件、可执行（executable）目标文件和可共享（sharable）目标文件。本文实验主要实现可执行 ELF 格式内核模块的安装。

ELF 文件由 4 部分组成：ELF 首部（ELF header）、程序首部表（Program header table）、段（Section）及可选的段首部表（Section header table）。

安装 ELF 格式内核的第一步是在 ELF 首部中找到程序首部表数量成员 e_phnum（位于 ELF 首部偏移量 0x2C 处，代表段的数量）和程序首部表位置成员 e_phoff（位于 ELF 首部偏移量 0x1C 处），汇编语言关键代码如下：

```
mov(FILE_BASE+0x2c),%cx
movzwl%cx,%ecx
mov(FILE_BASE+0x1c),%ebp
add $FILE_BASE,%ebp
```

其中寄存器 ECX 存放程序首部表数量成员值，寄存器 EBP 存放程序首部表在当前内存中的线性地址。

第二步是根据程序首部表的数量成员与位置成员，循环读出程序首部表指向的每一个段。

程序首部表中的表项描述一个段信息，其中 p_filsz 成员（位于表项偏移量 0x10 处）代表段的长度、p_vaddr 成员（位于表项偏移量 0x08 处）表示该段程序在进程空间的地址、p_offset 成员（位于表项偏移量 0x04 处）代表该段在 ELF 文件中的偏移位置。有了这三个数据，就可将 ELF 文中的一个段安装到进程空间指定的位置。汇编语言关键代码如下：

```
mov0x10(%ebp),%ecx
mov0x08(%ebp),%edi
mov0x4(%ebp),%esi
add $FILE_BASE,%esi
cld
rep movsb
```

2.6 内核框架

内核框架程序用汇编语言实现，目的是建立内核在内存中的基本程序结构。该程序首先要对所有段寄存器初始化，然后调用内核子程序。关键代码如下：

```
_start:
    mov $SEL_DS,%ax
    mov%ax,%ds
    mov%ax,%es
    mov%ax,%fs
    mov%ax,%gs
    mov%ax,%ss
    mov $stack_top,%esp
    callmain
loop:jmploop
```

```
.fill1024
stack_top:
```

其中,stack_top 是内核栈顶地址,堆栈长度为 1024 字节,由高地址向低地址生长;main 是 C 语言主函数,在 main.c 中实现,其功能是调用直接写屏函数向屏幕输出字符串;最后程序进入无限循环,停滞不前。

2.7 直接写屏

内核在 32 位保护模式下启动之后,可以用 32 位线性地址直接访问位于 0xB8000 地址的屏幕显示缓冲区,实现直接写屏操作。本文实验中的直接写屏操作由 C 语言编程实现,其函数代码如下:

```
void src_str(char *str, int x, int y)
{char *vp=(char *)(0xb8000+y*160+x*2);
for(; *str; *vp++=*str++, *vp++=0x0f);
}
```

该函数有 3 个参数:第一个参数为待显示字符串,第二参数为显示起始位置的列座标,第三个参数为行座标。该函数将待显示字符串在指定的座标位置开始显示,字符显示的属性为亮白(0x0F)。

该函数用一条空循环 for 语句实现判断字符串结束符、向屏幕显示缓冲区存放字符及属性、修改缓冲区指针等多项工作,充分体现出 C 语言程序设计的简洁性。

2.8 启动运行内核模块

本文实验的目的是演示 X86 平台保护方式下操作系统的引导启动,所以其内核没做任何实质性工作,只完成在保护模式下通过直接写屏方式在屏幕上显示字符串。运行结果如图 1 所示。

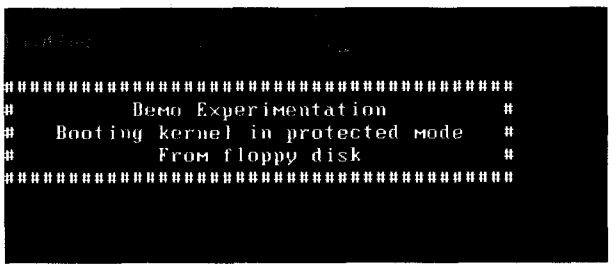

图 1 演示实验运行结果

3 结束语

本文以一个演示实验为例,对 X86 平台操作系统引导技术的研究与实现作了较深入的分析与阐述。

该演示实验程序在 Linux 平台用 GNU 工具链开发,其引导模块和装载模块由 AT&T 语法汇编语言编程,内核模块由 C 语言编程。该程序结构清晰紧凑,源代码只有 350 余行,适用于操作系统课程实验教学,便于学生分析学习。

X86 平台操作系统的引导启动技术涉及到软盘操作、保护模式下的汇编语言及 C 语言程序设计等多方面知识,引导学生加强这方面的研究,对促进操作系统及其相关课程的学习有着重要的实际意义。

参考文献:

[1]C. S. Rodriguez, G. Fischer, S. Smolski, The Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC architectures [M]. Prentice Hall, Inc., 2006.

[2]A. S. Tanenbaum, Operating Systems: Design and Implementation, Third Edition [M]. Prentice Hall, Inc., 2008. Lennart Yseboodt, Michael De Nil, EFSL Embedded Filesystems Library - 0.2 [EB/OL]. http://efsl.be/, 2005.

[3]David Hovemeyer, GeekOS: An Instructional Operating System for Real Hardware [EB/OL]. http://geekos.sourceforge.net/docs/geekos-paper.pdf, 2001.

[4]Tool Interface Standards (TIS) Committee, Executable and Linking Format (ELF) Specification, Version 1.2 [EB/OL]. http://refspecs.freestandards.org/elf/elf.pdf, May, 1995.

(责任编辑:叶 玲)

A Study on and Implementation of Booting Techniques of the Operating System on X86 Platform

WU Zhao-zhi

(School of Mathematics and Information Technology, Nanjing Xiaozhuang University, Nanjing 211171, Jiangsu)

Abstract: The paper introduces three kinds of typical medium storage structure for booting operating system from a floppy disk on the X86 platform, and taking a demo experiment for example, describes the commonly-used techniques and implementation methods of booting and starting operating system in a protected mode, including booting operating system from a floppy disk, turning on A20 line, setting GDT and IDT, jumping into the protected mode, installing the ELF formatted kernel module, kernel frame, direct screen writing, and starting the kernel module, etc.

Key words: operating system; booting techniques; X86 platform; protected mode; ELF formatted kernel module