文章编号: 1000-5013(2010)05-0515-06

Linux下Proc文件系统的编程剖析

郭松, 谢维波

(华侨大学 计算机科学与技术学院, 福建 泉州 362021)

摘要: 论述Linux下可加载内核模块和虚拟文件系统的运作原理,比较内核程序运行在微内核和一体化内核两种组织方式下的优缺点.在此基础上,提出Proc文件系统下的编程,包括编写内核模块和Proc文件系统程序的一些过程,并对主要代码进行解析说明.通过一个编程示例,给出Proc文件系统编程的框架,以显示Linux虚拟文件系统的能力.

关键词: Proc文件系统; Linux; 内核模块; 虚拟文件系统

中图分类号: TP 311.1 文献标识码: A

通过对Proc文件系统中文件的读写操作,达到对驱动程序或其他内核程序的控制,实现用户态程序与核心态程序的数据通信.向Proc写入信息,可以使用标准的文件系统提供的机制.文件系统注册的标准机制,是每个文件系统都用自己的函数来处理索引节点和文件操作.通过编写内核模块来实现用户态到核心态通信,通过内核模块在Proc文件系统中创建文件夹或文件,来为用户程序提供接口.基于此,本文介绍编写内核模块和Proc文件系统程序的一些过程,并对主要代码进行解析说明.

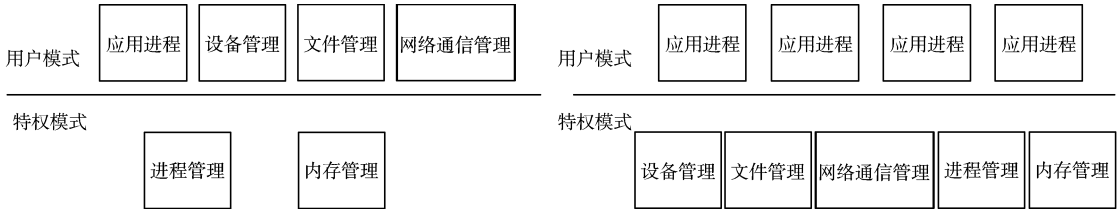
1 可加载内核模块和虚拟文件系统

1.1 可加载内核模块

为了支持构建安全操作系统,现代中央处理器(CPU)一般提供至少两种运行模式:特权模式与用户模式.它们不仅具有不同的优先权等级,还有各自的地址空间,即内核空间和用户空间.所有的现代CPU都具有保护系统软件不受应用程序破坏的功能,即在CPU中实现不同的操作模式或级别[1].

Unix系统使用最高级别和最低级别——内核程序运行在特权模式,在这个级别中可以进行所有的操作;而应用程序运行在用户模式,处理器控制着对硬件的直接访问,以及对内存的非授权访问.

根据各个管理模块是否放在特权模式下,操作系统有微内核和一体化内核两种组织方式[1],如图1所示.传统的Unix及大部分类Unix操作系统基本上都属于一体化内核组织方式,而Mach, Windows NT, Window 2000/XP等基本属于微内核组织方式(图1).



(a) 微内核 (b) 一体化内核

图1 操作系统内核组织方式

Fig. 1 Organization of operating system kernel

收稿日期: 2009-10-23

通信作者: 谢维波((1964-),男,教授,主要从事信号处理与模式识别的研究. E-mail: xwblxf@hqu.edu.cn.

基金项目: 福建省厦门市科技计划项目(3502Z20083047)

一体化内核组织方式的优点是效率高、安全,但其缺点是不容易扩展.若要增加一种新设备驱动或文件系统等功能,则必须对内核重新编译定制,而且占用内存较多.微内核组织方式的优点是可扩展性好,占用内存少,但缺点是效率低,也没有一体化内核安全[1].

在Linux中引入内核模块机制,主要目的是为了充分吸取两者的优点,克服它们的缺点.内核模块是Linux用来高效地利用微内核的理论优点而不会降低系统性能的一种方法.

Linux内核模块本质上是存放在文件系统中的ELF对象文件,没有链接,不能独立运行,但可以动态加载(Insmod)到内核并与内核链接,从而成为内核的一部分,或者从内核解除链接.其基本机理是,Linux内核本身有一张内核符号表(Kernel Symbol Table),包括用来描述系统提供哪些函数服务、系统全局变量及它们的地址.在Linux用一个宏EXPORT_SYMBOL(或EXPORT_SYMBOL_GPL,定义在include/linux/module.h中)导出需要输出的函数及系统全局变量,以使内核各个模块能够互相引用.同理,每个内核目标模块也有一张类似的符号表,其中定义了此模块所提供的功能函数及变量,以及此模块可能要用到的系统内核,或者其他已加载模块中的函数及变量,但此时又不知其实际地址.

为了使内核模块真正成为Linux系统内核的一部分,必须将此内核模块动态装入到内存中,并把相应的各种符号函数解析定位好.然后,内核模块提供的各种函数才能真正被使用.当模块加载入内核时,系统将新加载模块提供的资源和符号加到内核符号表中;而当模块加载时,内核用符号表来解决模块的资源引用问题.Linux允许模块堆栈,即一个模块可请求其他模块为之服务.通过这种通信机制,新加载的模块可以访问已加载模块提供的资源[2].

内核中有一个变量叫做module_list,定义在include/linux/module.h中,是一个全局变量.每当用户将一个模块加载到内核时,这个模块就会被添加到由module_list形成的链表中.必要时,内核就会检索这个链表,找到该模块,然后再使用其提供的函数或变量.

1.2 虚拟文件系统

Linux的最重要特征之一就是支持多种文件系统,因此,它更加灵活并可以和许多其他操作系统共存.文件系统不仅包含着文件中的数据,而且还有文件系统的结构(属性、操作等).每个实际文件系统从操作系统和系统服务中分离出来,它们之间通过一个接口层——虚拟文件系统(VFS)来通讯.虚拟文件系统所隐含的思想是,把表示很多不同种类文件系统的共同信息——通用文件模型放入内核.Linux核心的其他部分及系统中运行程序将看到统一的文件系统.

把通用文件模型看作是面向对象的.在这里对象是一个结构,其中既定义了数据结构,也定义了其上的操作方法.VFS依赖于数据结构来保存其对一个文件系统的一般表示.通用文件模型由超结块、索引节点、目录项和数据块(文件)等对象类型(结构)组成[2].在相应的目录下有对应结构的操作方法的接口结构,如struct inode_operations, struct file_operations,成员几乎都是一些函数指针.

用户程序通过系统调用访问通用的VFS.每个支持的文件系统必须实现一组函数,而这组函数完成VFS所支持的操作.VFS了解它所支持的文件系统和完成每个操作的函数[3].VFS管理文件系统相关的系统调用,并把它们转化成适应文件系统类型的函数,如图2所示[2].

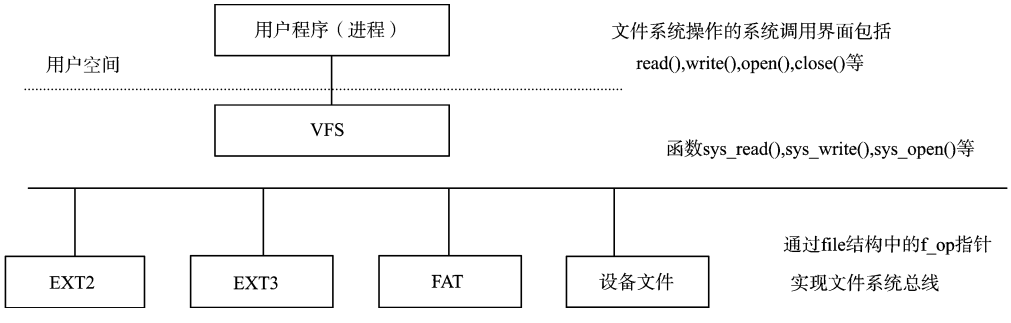


图2 VFS与具体文件系统的关系示意图

Fig.2 Relationship of VFS and concrete file system

具体的文件系统可以设计成可加载模块,在系统需要时进行加载.挂载具体文件系统时,VFS读取它的超级块,得到具体文件系统的拓扑结构,并将这些信息映射到VFS超级块结构中.VFS在系统中

保存着一组已安装文件系统的链表及其 VFS 超块, 而每个 VFS 超块包含一些信息及一个执行特定功能的函数指针.

当进程或者 shell 命令访问目录和文件时, shell 命令及应用程序分解成系统调用, 进入内核空间, 遍历系统的VFS inode, 而 VFS 的 inode 指向了具体文件系统的节点. 通过底层块I/O 函数调用 IDE 接口, 再通过块驱动程序访问块设备, 得到文件数据. 另外, 还有一种特殊的文件系统, 如 Proc 或者 Sysfs, 它们操作内存缓冲区, 在内存缓冲区中存有相关系统管理信息. 所有 Linux 文件系统使用一个通用 buffer cache, 来缓冲来自底层设备的数据以加速对包含此文件系统物理设备的存取[3].

此虚拟文件系统能够管理在任何时刻映像到系统的不同文件系统. 它通过维护一个描述整个虚拟文件系统和实际已安装文件系统的结构来完成这个工作. VFS 对文件系统共有的内核上层及底层部分进行了处理. 上层处理如文件路径的查找、文件的读写操作、从用户空间向下传递到具体文件系统的部分; 在底层进行各种缓存的处理.

2　Proc 文件系统的编程

Proc 文件系统真正显示了 Linux 虚拟文件系统的能力, 但事实上它并不存在. 不管是 Proc 目录, 还是其子目录和文件都不是真正的存在(不保存在硬盘或者其他磁盘中) . Proc 文件系统象一个真正的文件系统一样向虚拟文件系统注册. 然而, 当有对 Proc 中的文件和目录的请求发生时, VFS 系统将从核心中的数据中临时构造这些文件和目录. Proc 文件系统提供给用户一个核心内部工作的可读窗口. Linux 核心模块都在 Proc 文件系统中创建入口.

2.1　内核模块

在 Linux 模块中, 只能调用那些由内核导出的函数, 而不能像其他应用程序那样调用库函数. 因为 Linux 内核函数运行在内核空间, 没被链接到 C 库, 所以在内核空间无法调用 C 库的函数[3]. 内核函数通常定义在内核源代码的 include 目录下, 需要用的大都在 include/ Linux 下. 一个内核模块必须至少有两个函数, 函数 init_module( ) 在该模块加载到内核的时候被调用, 而函数 cleanup_module( ) 在模块被卸载之前调用[3].

2.2　Proc 文件系统程序

实现一个程序, 它是内核维护记录学生名和学号的一张表格. 利用 Proc 文件系统作为与内核通信的手段, 实现对文件的维护和操作. 该程序由两部分组成. 一部分, 作为内核模块在内核运行, 在加载模块时, 初始化表格, 建立相应的 Proc 文件; 在卸载模块时, 清除该表格和 Proc 文件; 在加载后, 则通过 Proc 文件系统接受用户请求, 实现对表格的维护操作. 另一部分, 实现为一个用户态程序, 它从命令行读取用户的命令, 解析用户命令; 通过 Proc 文件系统将命令传递到内核模块, 也可通过 Proc 文件系统从内核模块读取信息[4].

2.2.1　*解决方案*　设计一个用户态程序和一个核心态程序. 用户态程序包括添加记录、删除记录、查看信息; 核心态程序包括实现内存的初始化、管理、卸载等功能. 具体的通信过程: 用户态程序对 Proc 文件系统进行读和写的操作. 当写文件时, 内核获得用户写入的内容, 对写入的内容进行处理和管理; 当用户读文件时, 内核首先把内存中保存的内容写到 Proc 文件中, 用户再读些文件, 得到想要读的内容[4].

2.2.2　*主要代码解析*

```
/* 源代码模块 student.c* /
# include 〈linux/ kernel. h〉     /*  We' re doing kernel work * /
# include 〈linux/ module. h〉     /*  Specifically,  a module * /
# include 〈linux/ Proc_fs. h〉     /*  Necessary because we use Proc fs * /
# include 〈asm/ uaccess. h〉     /*  用户态和核心态通信函数 * /
# define MESSAGE_LENGTH  1000     /* 确定写入和读出的信息长度, 单位是 byte* /
# define N  100     /* 学生总人数* /
# define nLen 10     /* 学生姓名长度* /
# define NUM  8     /* 学号长度* /
```

```
struct student{
        char name[nLen];
        char number[NUM];
}; /* 定义结构体数组,存放学生信息* /
struct student stu[N];
static int c= 0;      /* 用于确定结构体数组显示游标* /
static struct Proc_dir_entry * Our_Proc_File,* Our_Proc_Dir; /* 两个Proc文件系统结构体* /
# define Proc_ENTRY_FILENAME "student"  /* Proc文件系统中的文件* /
# define Proc_ENTRY_DIRNAME "students"  /* Proc文件系统中的文件夹* /
```

2.2.3 主要函数说明 (1) module_input()用于从用户态获取输入数据,并对数据进行处理.

```
static ssize_t module_input(struct file * filp, const char * buff, size_t len, loff_t * off){
/* struct file * filp 由linux/fs.h 定义的* /
/* char * buffer用于存放数据的内存空间* /
/* size_t length 定义 buffer的长度* /
        if(buff[0]= = '-'){ /* 对于输入的操作符进行判断,然后删除数据* /
            if(j= = c) /* 判断被操作数据是否存在 * /
                printk(KERN_INFO "Cann't find the student of the number: %s",number);
            else{ /* 对已经存在的数据,进行覆盖运算* / }
        }
        else { /* 对输入带操作符或不带操作符进行判断和添加数据* /
        }
        for (; i< MESSAGE_LENGTH-1 && i< len && *(buff+ i)! = ''; i+ + )
            get_user(stu[c].name[i- k], buff + i);
            ⋮
        /* 其他具体动作* /
        }
}
```

(2) module_output()用于读出数据表,主要是读取定义的学生信息结构数组.

```
static ssize_t module_output(struct file * filp,char * buffer, size_t length,loff_t * offset){
        ⋮
        /* 将数组空间的数据读出,并使用 sprintf 函数的特性,将数据串接起来* /
        for (i= 0; i< c; i+ + ){
        j+ = sprintf(message+ j,"Name: %s,Number: %s", stu[i].name, stu[i].number);
        }
        /* 按字节顺序读取内存中的数据* /
        for (i= 0; i< length && message[i]; i+ + )
            put_user(message[i], buffer+ i);
        ⋮
        return i; /*  返回读取的比特数 * /
}
```

(3) module_open()用于打开模块.

```
int module_open(struct inode * inode, struct file * file){
        try_module_get(THIS_MODULE);
        return 0;
}
```

(4) module_close( ) 用于关闭模块.

```
int module_close(struct inode * inode, struct file * file){
        module_put(THIS_MODULE);
        return 0; /* success */
}
```

注: struct file_operations File_Ops_4_Our_Proc_File 中对 Proc 的 write, read, open, release 操作字段, 分别与上面 4 个函数对应起来.

```
static struct file_operations File_Ops_4_Our_Proc_File= {
        .read= module_output,
        .write= module_input,
        .open= module_open,
        .release= module_close,
}; /* 定义文件操作结构* /
```

(5) module_permission() 用于定义模块操作权限. 这个函数在进程试图用 /Proc 文件做什么的时候被调用. 文件或程序(可执行文件) 总是与权限相关联, 而权限将被应用于用户或组. 通过权限设置, 可以让让特定的用户或用户组才能访问模块. 此函数被赋予 struct inode_operations Inode_Ops_4_Our_Proc_File 的 permission 字段, 用于进行权限检测.

```
static int module_permission(struct inode * inode, int op, struct nameidata * foo){
/* 允许任何人从模块读但只有 root (UID 为 0) 可以向它写 */
        if (op == 4 || (op == 2 && current-> euid == 0))
                return 0;
        return -EACCES; /* 如果是其他值, 访问被禁止 */
}/* 读写权限判断函数* /
static struct inode_operations Inode_Ops_4_Our_Proc_File= {
        .permission= module_permission,      /* 用于进行权限检测 */
};
```

在 Linux 中对文件系统登记有标准的机制. 既然每个文件系统必须有它自己的处理节点和文件操作(两者的不同在于文件操作处理文件自己, 而节点操作处理对文件的引用, 例如创建对它的连接) 的函数, 所以有一个特殊的结构保存所有这些函数的指针(inode_operations 结构, 包含一个指向 file_operations 结构的指针). 在 Proc 中, 任何时候登记一个新文件都允许特别指定哪个 inode_operations 结构将用于访问它. 这就是使用的机制, inode_operations 结构包含指向 file_operations 结构的指针, 而它又包含指向 module_input 和 module_output 函数的指针[5].

(6) init_module() 用于模块的初始化, 建立相应的 Proc 文件, 并设置权限. 在 init_module 时向内核注册这个结构体, 在 cleanup_module 时注销这个结构体.

```
int init_module(){
Our_Proc_Dir= Proc_mkdir(Proc_ENTRY_DIRNAME, NULL);
/* 在 Proc 目录下创建文件夹* /
Our_Proc_Dir-> owner = THIS_MODULE;
Our_Proc_File= create_Proc_entry( Proc_ENTRY_FILENAME, 0644, Our_Proc_Dir);
/* 创建文件, 设置读写权限为 0644* /
Our_Proc_File-> Proc_iops= &Inode_Ops_4_Our_Proc_File;
Our_Proc_File-> Proc_fops= &File_Ops_4_Our_Proc_File;
⋮
}
```

(7) cleanup_module 用于模块卸载, 清除所有 init_module() 所做的操作, 并返回信息. 实际是工作

是调用 remove_Proc_entry 进行清理工作.

```
void cleanup_module()
{
    remove_Proc_entry(Proc_ENTRY_FILENAME, &Proc_root);
    ⋮
}
module_init(init_module);
module_exit(cleanup_module);/* 在2.6内核中,内核模块必须调用宏module_init与* module_exit() 去注册初始化与退出函数.* /
```

2.2.4 编译及测试 源代码文件为 student.c 在内核2.6版本的编译过程是,先编写一个 makefile 文件,输入 obj-m+ = student.o,保存退出.使用超级管理用户(root),与程序源代码在同一目录下的命令行运行:# make-C/usr/src/linux-kernel SUBDIRS= $ PWD modules.在目录下就会生成想要的.o 或.ko 的模块文件.编译好模块后,使用命令:insmod student.ko 加载此模块.此时,可通过各种模块操作命令对模块信息进行查询. 利用程序定义的操作,通过 shell 解析输入的命令对加载的文件进行操作,并使用 dmesg 查看 printk 输出的内核消息.

3 结束语

利用 Linux 强大易用的可加载内核模块机制,可以灵活地扩充 Linux 内核,广泛利用在 Linux 设备驱动和 netfilter 等方面的开发中.因此,在提高系统的安全性的同时,也可以被攻击者利用而非法入侵系统.它犹如一把双刃剑,在提供强大的灵活性同时有存在着极大的安全隐患,需要用户谨慎使用.

参考文献:

[1] 熊海泉. Linux 模块实现机制剖析[J]. 科技广场,2006(2):7-8.
[2] 吴伟国,李张,任广臣. Linux 内核分析及高级编程[M]. 北京:电子工业出版社,2008:187.
[3] 毛德操,胡希明. Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社,2001:415-430.
[4] 罗宇,褚瑞. 操作系统课程设计[M]. 北京:机械工业出版社,2005:106-120.
[5] CHERAMI. Linux 内核模块编程——将/Proc 作为输入[EB/OL]. [2006-12-16]. http://www.egunao.com/os/linux/18220.html

Analysis in Depth on Linux Proc File System Programming

GUO Song, XIE Wei-bo

(College of Computer Science & Technology, Huaqiao University, Quanzhou 362021, China)

Abstract: The operation principle of Linux loadable kernel module and virtual file system was discussed, and the advantages and disadvantages of the kernel program running in micro kernel and integrated kernel were compared. Base on this foundation, the Proc file system programming is proposed. Which include the programming procedure of kernel module and Proc file system, and to the main code was analyzed. Through a programming demonstration, the framework of Proc file system programming is given to demonstrate the ability of Linux virtual file system.

Keywords: Proc file system; Linux; kernel module; virtual file system

(责任编辑:黄仲一 英文审校:吴逢铁)