Proc 文件系统的研究与应用①

赵付强[1], 李允俊[1], 宫彦磊[2]

1(延边大学 工学院, 延吉 133002)
2(工业和信息化部软件与集成电路促进中心, 北京 100038)

摘 要: 当前驱动调试过程中, 获取某些寄存器的值大都采用把该值输出到日志文件的方式来查看; 在嵌入式 Linux 应用开发中, 父进程检测子进程的状态往往通过子进程主动向父进程发送消息的方式来获得. 通过分析 proc 文件系统的注册、安装、管理发现该文件系统可以对上述问题进行优化, 提出 proc 文件系统用于驱动程序的调试和监视子进程状态的解决方案, 对加快驱动调试和提高应用程序的稳定性有十分重要的意义.

关键词: Linux; 虚拟文件系统; proc 文件系统; 内核

Research and Application of Proc File System

ZHAO Fu-Qiang[1], LI Yun-Jun[1], GONG Yan-Lei[2]

[1](Deparment of Compute Science & Technology, Yanbian University, Yanji 133002, China)
[2](MIIT Software and Integrated Circuit Promotion Center, Beijing 100038)

Abstract: In the current driver debugging process when access to certain register values, most of us output the value to the log file to see; In embedded Linux application development, parent process detect sub-process's state using the way child process initiatively send message to the parent process. Through analysis the registration, install, management of the proc file system can optimize the above problems, and put forward using proc file system to debugging of the driver and monitoring sub-state which have a great significance to speed up the debug of driver and improve the stability of the application.

Key words: Linux; virtual file system; proc file system; kernel

最初在 Linux 系统中开发 proc 文件系统是为了提供有关系统中进程的信息. 由于这个文件系统非常有用, 所以内核中的很多元素也开始使用它来报告信息, 或启用动态运行时配置. proc 文件系统包含了一些目录(用作组织信息的方式)和虚拟文件. 虚拟文件可以向用户呈现内核中的一些信息, 也可以用作一种从用户空间向内核发送信息的手段.

1 虚拟文件系统(VFS)

虚拟文件系统有时也称作虚拟文件交换(Virual Filesystem Switch ,简称 VFS). VFS 是一个由内核实现的虚拟文件系统, 它是操作系统内核和真实文件系统之间的一个软件层[1].

对具体文件系统来说, VFS 是一个管理者, 而对内核的其它系统来说, VFS 是它们与具体文件系统的一个接口, 整个 Linux 中文件系统的逻辑关系如图 1 所示. 系统中所有文件系统不但依赖 VFS 共存, 而且也依靠 VFS 系统协同工作. 通过虚拟文件系统, 程序可以利用标准的 UNIX 文件系统调用对不同介质上的不同文件系统进行读写操作.

VFS 抽象层之所以能够衔接各种各样的文件系统, 是因为它定义了所有文件系统都支持的基本的、概念上的接口和数据结构. 同时实际文件系统也将自身的诸如"如何打开文件", "目录是什么"等概念在形式上与 VFS 的定义保持一致. 在 VFS 提供的接口中包含向各种物理文件系统转换用的一系列数据结构, 如

① 通讯作者:李允俊, E-mail:yjlee@ybu.edu.cn
收稿时间:2012-06-03;收到修改稿时间:2012-07-18

超级块(superblock)、索引节点(inode)、目录项(dentry)、文件等. 这些数据结构有两个共同点, 一是考虑到对多种具体文件系统的兼容性, 二是它们只存在于内存.

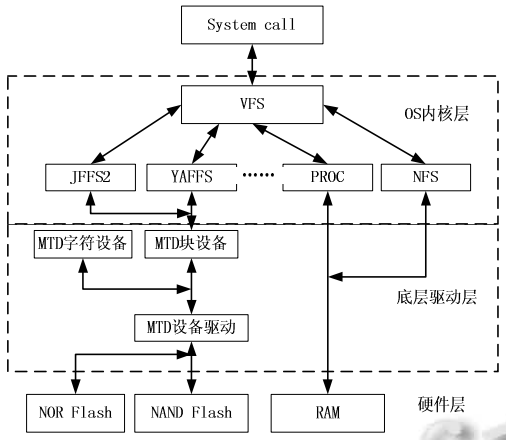

图 1　Linux 中文件系统的逻辑关系示意图

一个具体文件系统要想被 linux 支持, 就必须按照这个接口(file_operation 数据结构)编写自己的操作函数,从而隐藏自己的细节, 使内核子系统及用户程序感觉文件系统都是一样的.

VFS 具有四方面的作用[2].

(1) 对具体文件系统进行抽象, 做到管理方式上的统一;

(2) 接受用户空间文件访问的系统调用,如 read open mkdir 等;

(3) 实现多种文件系统的相互访问;

(4) 接受内核其他子系统的操作请求.

3　proc文件系统分析

3.1 内核中 proc 文件系统的初始化流程

proc 文件系统是一个伪文件系统, 它只存在内存当中, 而不占用外存空间. 它以文件系统的方式为访问系统内核数据的操作提供接口. 用户和应用程序可以通过 proc 得到系统的信息, 并可以改变内核的某些参数.

在使用 proc 之前, 必须首先注册、安装 proc, 在内核内存中创建数据结构来描述文件系统[3]. 但是不同的体系结构拥有不同的 proc 内容, 所以在初始化阶段并不完全创建子目录的内容, 有些文件要等到系统运行时动态创建.

proc 文件系统的初始化是在 Linux 系统启动过程中完成的, 主要内容如下:

①proc_init_inodecache: 为 proc_inode 创建 slab cache[4], 是 proc 文件系统的主要部分, 通常需要快速创建或销毁.

②调用 register_filesystem(&proc_fs_type), 将 proc 文件类型加入到文件类型 file_systems 的单向链表中, 如果发生错误, 则返回.

③vfs_kern_mount: 申请 vfsmount 结构, 并调用到具体的相应文件系统的 mount 函数, 相应的文件系统 mount 返回一个 super_block 的结构, vfs 对其进行处理初步得到 dentry 结构, 最后用 dentry 结构对 vfsmount 结构的成员变量进行初步赋值

④proc_net_init: 创建大量与网络相关的文件, 创建方法与前者相似.

3.2 proc 文件系统注册

每个文件系统都有自己的初始化例程, 它的作用就是在 VFS 中进行注册, 即填写一个 file_system_type 的数据结构, 该地址包含了文件系统的名称和一个指向 VFS 超级块读取例程的地址. 当调用 register_filesystem() 成功注册后, 该结构将被添加到以 file_systems 为表头的链表中[5]. 图 2 显示了当注册成功后 proc 文件系统类型添加到系统注册链表后的示例.

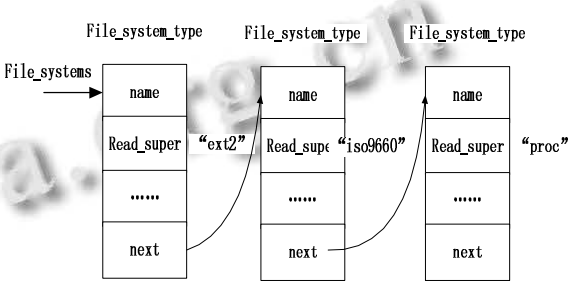

图 2　内核文件系统类型链表示例

3.3 安装 proc 文件系统

一个文件系统只注册是不行的, 必须安装这个文件系统. 需要指定三种信息: 文件系统名称、包含文件系统的物理块设备、文件系统在已有文件系统中的安装点. 安装 proc 文件系统首先需要申请一个 vfsmount 的挂载点并进行一部分初始化, 设备名为 proc, 之后创建内核超级块, 根据 proc 文件系统类型创建 inode 索引节点和根目录项. 最后根据 inode 和 dentry 对超级块和挂载点进行初始化.

3.4 proc 文件的管理

proc 目录是系统模拟出来的一个文件系统, 本身并不存在于磁盘上. 文件表示内核参数的信息, 这些信息分两类, 一类是可都可写的, 这类参数在“/proc/sys”目录下; 另一类是只读的, 就是“/proc/sys”目录之外的其他目录和文件. 当然这只是一种惯例, 实际在其他目录下建立可读写的/proc 文件也是可以的.

4 proc文件系统的应用及实现

4.1 用户空间通过 proc 与内核的信息交互

Linux 内核的整体结构很庞大, 其包含的组件也很多, 把需要的东西包含在内核中有两种方法, 一是把所有的功能都编译进内核, 这会导致内核很大并且修改某些功能时要重新编译内核. 二是使用模块机制, 它本身不编译进内核从而控制了内核大小并且模块一旦被加载就和其他部分一样[6].

本论文采用的是模块机制. 通过内核模块创建的 proc 虚拟文件, 来实现用户空间与内核空间的通信. 内核模块加载时动态创建 proc 目录和虚拟文件, 卸载时卸载目录和文件. 用户通过 echo、cat 命令来验证其正确性.

(1) 在/proc 中添加文件目录和节点

当用户读取/proc/mytest/test 文件时调用的函数.

```
static ssize_t globalmemread(struct file *filp,char __user *buf,size_t size,loff_t *ppos){
……
//proc 文件系统数据拷贝到用户空间缓存.
 len = sprintf(page, "%s\n", mybuf);
  return len;
}
```

当用户写数据到/proc/mytest/test 文件时调用的函数.

```
static ssize_t globalmemwrite(struct file *filp,const char __user *buf,size_t size,loff_t *ppos){
printk(KERN_INFO "write ------\n");
  ……
 if (copy_from_user( mybuf, buf, size )) {
  printk(KERN_INFO "write error\n");
  return -EFAULT;
 }
mybuf[size]=0;
}
```

文件读写操作 globalmemread, globalmemwrite 分别对应到文件操作数据结构 globalmem_fops 的 read, write 字段.

```
static struct file_operations globalmem_fops={
owner:THIS_MODULE,
read: globalmemread,
write:globalmemwrite,
};
```

模块初始化时, 调用 vmalloc 创建内存, 之后在根目录下创建 mytest 目录并在 mytest 目录下创建 test 文件并初始化对应的文件操作.

```
int globalmem_init(void){
our_proc_dir=proc_mkdir(Proc_ENTRY_DIRNAME,NULL);
//创建 root 用户具有读写权限其他用户只有读权限的文件
our_proc_file=create_proc_entry(Proc_ENTRY_FILENAME,0644,proc_dir);
……
our_proc_file->proc_fops=&globalmem_fops;
}
```

模块卸载时把创建的文件目录删除.

```
void globalmem_exit(void){
remove_proc_entry(Proc_ENTRY_FILENAME,proc_dir);
remove_proc_entry(Proc_ENTRY_DIRNAME,NULL);
}
```

(2)测试结果

把源码编译成 linux 内核驱动模块 procmem.ko, 并用 insmod procmem.ko 把模块加载到内核,最后 rmmod procmem 卸载模块.

该设计实现了预期的目的. 该方法完全可以拿到一些复杂设备驱动的调试过程中, 比如 i2c、sd 卡、usb 等. 在调试这些驱动过程中经常需要一些寄存器的值进行分析, 采用该方法可以方便的获取这些信息或设置该信息从而加快驱动的调试.

4.2 进程监视

Linux 系统中, 每个生成的进程都会在 proc 文件系统的根目录生成一个以进程号为名字的目录, 并在其中

会生成一些动态文件描述进程状态和其他信息.

这些信息在内核中的格式为:

seq_printf(m, "%d (%s) %c %d……\n", pid_nr_ns(pid, ns),tcomm, state,ppid,……);

pid_nr_ns(pid, ns)是进程号; tcomm 是应用程序或命令的名字; state 是进程状态; ppid 是父进程 ID……. 嵌入式产品应用环境复杂, 而在其上运行的程序较受到外界影响导致程序的不可控, 有可能需要让设备自行重启.

在实际项目中的具体实现.

```
/*该方法检测某个子进程是否为僵死状态*/
int32_t IsProcessAlive(pid_t pid){
  FILE *fp;/*对应子进程号 pid 的文件指针*/
char_t CStr[64];/*子进程对应的路径*/
  /*从 fp 中读取数据到 CStr,若 fp 中没有数据则关闭 fp*/
    fgets(CStr, sizeof(CStr), fp)
    /*找到第一个右括号, 然后返回指针*/
    pCIdStr = strchr(CStr, ')');
    ……
    pCIdStr += 2;
    switch ((char_t)*pCIdStr) {
      case 'D':        /* 不可中断的睡眠*/
      case 'R':        /* 就绪  */
      case 'S':        /* 睡眠*/
      case 'T':        /* 跟踪或停止*/
          I32IsAlive = TRUE;
          break;
      case 'Z':         /*僵尸进程*/
      default:
          I32IsAlive = FALSE;
          break;
    }
    fclose(fp);
    return I32IsAlive;
}
```

在嵌入式程序开发中, 程序被分为若干子进程, 每个子进程完成自己的具体任务. 主进程创建完子进程后, 启动看门狗. 在这个循环中主进程监视每个子进程的状态, 监视过程是通过子进程进程号对应目录下的 stat 文件中的 state 字段进行判断. 若某个进程出现异常, 主进程应采取重新创建该子进程或系统重启, 从而保证程序的稳定性.

参考文献

1 Maybee S. File system framework. USA: Sun Microsystems Press, 2006.

2 Juan I, Florido S. Journal File Systems. www. Linuxg-azette.com.

3 Cheng GG, Liu XY, Bao KM. Startup process analysis of Linux kernel. Computer Engineering and Design. 2006, 27(9):1528−1529.

4 Huang YJ, Cheng YF. Linux Kernel Slab Memory Buffer Manage, Computer Engineering, 2006,32(24):31−33.

5 Bovet DP, Cesati M. Understanding The Linux Kernel. 3rd Edition.USA: O'Reilly Media. 2007. 328−371.

6 Corbet J, Rublini A, Hartman C. Linux Device Drivers. 3rd Edition. USA:O'Reilly Media. 2006. 21−234.

(上接第 94 页)

2 Yang W, Qang LQ. Summarization of PSO. Chin Eng Sci, 2004,6(5):87−94.

3 王东风,韩璜.基于粒子群优化的混沌系统比例-积分-微分控制.物理学报,2006,55(4):1644−1646.

4 朱童,李小凡,鲁明文.位置加权的改进粒子群算法.计算机工程与应用,2011,47(5):4−6.

5 赵旎,火长跃.基于G语言的液位控制系统研制.河南科技大学学报(自然科学版),2006,(8):46−48.

6 唐进元,黄云飞,磊文利.基于 LabVIEW 的数字 PID 控制器的设计研究.测控技术, 2007,26(3):35−37.