sysfs分析

**宝德技术研究院 李磊**

# 概述

tmpfs是向用户空间导出内核对象的文件系统,它不仅提供了查看内核内部数据结构的能力,还可以修改这些数据结构.笔者通过学习tmpfs,对接下来的块设备,ext2文件系统的学习做好铺垫.

# 基本数据结构

## sysfs\_dirent

struct sysfs\_dirent {

atomic\_t s\_count;

struct list\_head s\_sibling;

struct list\_head s\_children;

void \* s\_element;

int s\_type;

umode\_t s\_mode;

struct dentry \* s\_dentry;

struct iattr \* s\_iattr;

atomic\_t s\_event;

};

### s\_count

引用计数,表示s\_count被引用的次数.

### s\_sibling

同级目录的链表,链表头是父目录的s\_children.

### s\_children

链表头,用于组织子目录.目录的组织形式可以由图1-1表示



图1-1

VFS中的struct dentry和sysfs\_dirent的关系如图1-2



图1-2

由图1-2可以看出,sysfs的目录项和vfs的dentry是通过dentry中的d\_fsdata连接的.sysfs\_dirent中的s\_element也是void \*型,根据不同的文件类型指向不同的文件属性.

### s\_element

指向具体文件的数据结构的指针.在sysfs中,文件有3种类型:属性文件,二进制属性文件,符号链接和目录.对应的s\_element成员将被转化为attribute, bin\_ attribute, sysfs\_symlink, koject对象

◆attribute

struct attribute {

const char \* name;

struct module \* owner;

mode\_t mode;

};

◆bin\_attribute

struct bin\_attribute {

struct attribute attr;

size\_t size;

void \*private;

ssize\_t (\*read)(struct kobject \*, char \*, loff\_t, size\_t);

ssize\_t (\*write)(struct kobject \*, char \*, loff\_t, size\_t);

int (\*mmap)(struct kobject \*, struct bin\_attribute \*attr,

struct vm\_area\_struct \*vma);

};

◆sysfs\_symlink

struct sysfs\_symlink {

char \* link\_name;

struct kobject \* target\_kobj;

};

## sysfs\_buffer

这个结构体中保存了实际数据在page成员指向的页面中,用户调用read时就可以获取内核中导出来的数据.

struct sysfs\_buffer {

//缓存区中数据的长度

size\_t count;

//数据内部当前位置,用于读取部分数据和定位

loff\_t pos;

//指向1页,用于存储数据

char \* page;

struct sysfs\_ops \* ops;

struct semaphore sem;

//指定缓存区的内容是否需要填充

int needs\_read\_fill;

int event;

};

# 初始化过程分析

在kernel初始化mnt tree的过程中, tmpfs就被初始化.

void \_\_init mnt\_init(unsigned long mempages)

{

...

sysfs\_init();

...

}

由于tmpfs中的目录数据需要经常变更,在初始化的过程中需要对tmpfs的目录项申请一个缓存,后续过程就是挂载.

int \_\_init sysfs\_init(void)

{

...

sysfs\_dir\_cachep = kmem\_cache\_create("sysfs\_dir\_cache",

sizeof(struct sysfs\_dirent),

0, 0, NULL, NULL);

...

}

关于这种缓存,可以通过cat /proc/slabinfo查看.

[root@localhost ~]# cat /proc/slabinfo | grep sysfs

sysfs\_dir\_cache 7458 7584 80 48 1 : tunables 120 60 8 : slabdata 158 158 0

# 挂载过程分析

## 普通文件系统的挂载过程

在文档<<根文件系统挂载过程分析>>中已经分析过一个文件系统是如何挂载的了,这里指针对sysfs部分做详细分析,一个文件系统的挂载大体过程如图3-1所示.



图3-1

一个文件系统的挂载由vfs\_kern\_mount开始,通过alloc\_vfsmnt获取一个初始化过的vfsmnt对象,接着调用type->get\_sb获取super block对象,根目录的inode对象及dentry对象.最后完成一些挂载点的设置及mnt tree的设置.

## sysfs的挂载过程

sysfs的挂载过程区别于其他文件系统的就是获取super block的过程.sysfs获取super block的函数为sysfs\_get\_sb. sysfs\_get\_sb实际上是get\_sb\_single的简单封装. get\_sb\_single的调用链如图2-2.



图3-2

### sget

get\_sb\_single通过下面代码调用sget

s = sget(fs\_type, compare\_single, set\_anon\_super, NULL);

sget的代码实现如下:

1 struct super\_block \*sget(struct file\_system\_type \*type,

2 int (\*test)(struct super\_block \*,void \*),

3 int (\*set)(struct super\_block \*,void \*),

4 void \*data)

5 {

6 struct super\_block \*s = NULL;

7 struct list\_head \*p;

8 int err;

9

10 retry:

11 //获取一个sb

12 spin\_lock(&sb\_lock);

13 //遍历type管理的所有sb

14 if (test) list\_for\_each(p, &type->fs\_supers) {

15 struct super\_block \*old;

16 //获取sb

17 old = list\_entry(p, struct super\_block, s\_instances);

18

19 if (!test(old, data))

20 continue;

21 if (!grab\_super(old))

22 goto retry;

23 if (s)

24 destroy\_super(s);

25 return old;

26 }

27 //分配一个sb

28 if (!s) {

29 spin\_unlock(&sb\_lock);

30 //分配并初始化一个sb

31 s = alloc\_super(type);

32 if (!s)

33 return ERR\_PTR(-ENOMEM);

34 goto retry;

35 }

36

37 err = set(s, data);

38 if (err) {

39 spin\_unlock(&sb\_lock);

40 destroy\_super(s);

41 return ERR\_PTR(err);

42 }

43 s->s\_type = type;

44 strlcpy(s->s\_id, type->name, sizeof(s->s\_id));

45 //super\_blocks是一个链表头,用于管理系统中所有的super block

46 list\_add\_tail(&s->s\_list, &super\_blocks);

47 //把sb头插到file\_system\_type中的sb链表中

48 list\_add(&s->s\_instances, &type->fs\_supers);

49 spin\_unlock(&sb\_lock);

50 get\_filesystem(type);

51 return s;

52 }

由于函数指针test是compare\_single,并不为空,故14行条件判断成立.注意sget具有两个功能:一个是查找super block,第2个是分配super block.这两个功能是通过test参数控制的,当存在test时候,sget先去查找super block,如果test为空,sget就直接分配super block.在当前情况下,这个函数会首先查找super block,但是sysfs在这时候还没有super block, (&type->fs\_supers)->next肯定等于&type->fs\_supers. list\_for\_each中的循环肯定不会进去.在28行条件判断成功后就会为sysfs分配一个super block.

d\_alloc\_root对根目录项进行初始化.

### fill\_super

fill\_super最终会调用sysfs\_fill\_super

1 static int sysfs\_fill\_super(struct super\_block \*sb, void \*data, int silent)

2 {

...

3 //将根目录的权限设为755

4 inode = sysfs\_new\_inode(S\_IFDIR | S\_IRWXU | S\_IRUGO | S\_IXUGO,

5 &sysfs\_root);

6 if (inode) {

7 //在sysfs\_new\_inode也有设置i\_o

8 inode->i\_op = &sysfs\_dir\_inode\_operations;

9 inode->i\_fop = &sysfs\_dir\_operations;

10 /\* directory inodes start off with i\_nlink == 2 (for "." entry) \*/

11 inode->i\_nlink++;

12 }

13 ...

14 root = d\_alloc\_root(inode);

...

15

16 root->d\_fsdata = &sysfs\_root;

17 sb->s\_root = root;

18 return 0;

19 }

这个函数通过sysfs\_new\_inode获取一个inode,并针对sysfs设置inode,比如设置block size为PAGE\_SIZE, 将sysfs\_inode\_operations 赋值给i\_op.流程如图3-3.由于在sysfs\_root中没有指定s\_iattr成员,sysfs\_new\_inode最终会调用set\_default\_inode\_attr.此函数做了一些简单的操作:给inode的i\_mode成员赋值,指定inode的uid,gid均为root.接着给atime/ctime/mtime设置时间.



图3-3

### do\_remount\_sb

在分析函数sget中提到过,sysfs调用sget的时候会首先去查找一个super block, 如果查找失败才会分配.这里就存在一个问题了,获取的super block的mount flag不一定是我们想要的mount flag.这里就需要进行remount操作.

当一个文件系统mount flag为rw,但是我们需要将其mount flag改为ro, 如果在remount过程中在此文件系统中有进程对文件系统中的文件进行write,或者delete,remount操作必须马上返回-EBUSY.这就是调用fs\_may\_remount\_ro的原因.只有fs\_may\_remount\_ro返回成功时,remount才能修改mount flag.

对于sysfs来说,由于没有定义sb->s\_op->remount\_fs,故remount仅仅修改sb->s\_flags.这里要注意的是remount能够修改只有MS\_RDONLY,MS\_SYNCHRONOUS,MS\_MANDLOCK.比较有意思的是,如果用于在系统中键入命令将一个挂载点的权限改为rw:

mount -o remount,rw dir

内核中没有类似MS\_RDWR这样的标志,经过笔者查看busybox代码,发现如下注释:

// Always understood.

/\* "ro" \*/ MS\_RDONLY, // vfs flag

/\* "rw" \*/ ~MS\_RDONLY, // vfs flag

很明显,rw标志的是~MS\_RDONLY.

### simple\_set\_mnt

int simple\_set\_mnt(struct vfsmount \*mnt, struct super\_block \*sb)

{

mnt->mnt\_sb = sb;

mnt->mnt\_root = dget(sb->s\_root);

return 0;

}

这个函数实现非常简单,就是将alloc\_vfsmnt中分配的vfsmount对象的指向super block及指向根目录的指针赋值.

# open过程分析

在文档 -- <<vfs中的路径查询>>中详细分析过,一个文件的open主要调用了函数do\_sys\_open.

long do\_sys\_open(int dfd, const char \_\_user \*filename, int flags, int mode)

{

char \*tmp = getname(filename);

int fd = PTR\_ERR(tmp);

if (!IS\_ERR(tmp)) {

//找到一个未使用的fd

fd = get\_unused\_fd();

if (fd >= 0) {

struct file \*f = do\_filp\_open(dfd, tmp, flags, mode);

if (IS\_ERR(f)) {

put\_unused\_fd(fd);

fd = PTR\_ERR(f);

} else {

fsnotify\_open(f->f\_dentry);

fd\_install(fd, f);

}

}

putname(tmp);

}

return fd;

}

### do\_lookup

函数do\_lookup,这个函数首先在缓存中搜索以nd->dentry为父目录,名称为name的目录项,如果搜索失败,就调用real\_lookup函数搜索.

static int do\_lookup(struct nameidata \*nd, struct qstr \*name,

struct path \*path)

{

...

need\_lookup:

dentry = real\_lookup(nd->dentry, name, nd);

...

}

real\_lookup函数调用具体文件系统的lookup函数

static struct dentry \* real\_lookup(struct dentry \* parent, struct qstr \* name, struct nameidata \*nd)

{

...

if (dentry) {

//调用具体文件系统的lookup函数

result = dir->i\_op->lookup(dir, dentry, nd);

...

}

对于sysfs来说就是sysfs\_lookup.sysfs\_lookup的函数原型是

static struct dentry \* sysfs\_lookup(struct inode \*dir, struct dentry \*dentry,struct nameidata \*nd)

◆dir是父目录的inode

◆dentry是我们需要查询的dentry

◆nd是父母的struct nameidata对象

sysfs\_lookup的主要工作有:

1. 找到与dentry名称相同的sysfs\_dirent对象
2. 为dentry分配一个inode
3. 为dentry分配一个目录操作函数集
4. 为inode分配一个文件操作函数集

经过sysfs\_lookup后, 原本不在hash表中的dentry将被加入hash表. 目标文件在内存中也有了对应的inode及fop.

### open

具体的文件系统的open函数被调用过程如图4-1



图4-1

为了更清楚的看清open函数如何被调用,下面列出\_\_dentry\_open的部分代码.

static struct file \*\_\_dentry\_open(struct dentry \*dentry, struct

vfsmount \*mnt,int flags, struct file \*f,

int (\*open)(struct inode \*, struct file \*))

{

...

f->f\_op = fops\_get(inode->i\_fop);

if (!open && f->f\_op)

//使用inode中的i\_fop

open = f->f\_op->open;

/\*如果是设备文件，需要打开\*/

if (open) {

error = open(inode, f);

if (error)

goto cleanup\_all;

}

...

}

通过上面所示代码, sysfs\_open\_file最终被调用,可以发现sysfs\_open\_file只是函数check\_perm的简单封装.chec\_perm除了一些权限上的检查,最重要的部分就是将sysfs\_buffer对象赋值给file->private\_data.这样做后,sysfs就和file对象衔接起来.

static int check\_perm(struct inode \* inode, struct file \* file)

{

...

buffer = kzalloc(sizeof(struct sysfs\_buffer), GFP\_KERNEL);

if (buffer) {

init\_MUTEX(&buffer->sem);

buffer->needs\_read\_fill = 1;

buffer->ops = ops;

file->private\_data = buffer;

} else

error = -ENOMEM;

goto Done;

...

}

# read过程分析

下面是vfs层中read过程的调用链.最终调用具体文件系统的read函数.



图5-1

对于sysfs来说read函数是sysfs\_read\_file,函数基本过程如图5-2



图5-2

### fill\_read\_buffer

这个函数对sysfs\_buffer对象进行填充,其中page成员调用subsys\_attr\_show继而调用具体subsystem实现的show函数进行填充

### flush\_read\_buffer

这个函数仅仅判断边界,然后将sysfs\_buffer对象中的数据拷贝到用户的buf中.

static int flush\_read\_buffer(struct sysfs\_buffer \* buffer, char

\_\_user \* buf,

size\_t count, loff\_t \* ppos)

{

int error;

if (\*ppos > buffer->count)

return 0;

if (count > (buffer->count - \*ppos))

count = buffer->count - \*ppos;

error = copy\_to\_user(buf,buffer->page + \*ppos,count);

if (!error)

\*ppos += count;

return error ? -EFAULT : count;

}

# write过程分析

write过程与read过程相反,本文略过.

# 目录创建过程分析

sysfs的目录创建函数是sysfs\_create\_dir,它首先获取kobject父节点,然后调用create\_dir创建目录.

int sysfs\_create\_dir(struct kobject \* kobj)

{

...

if (kobj->parent)

parent = kobj->parent->dentry;

else if (sysfs\_mount && sysfs\_mount->mnt\_sb)

//如果kobject没有父节点,就使用sysfs的root

parent = sysfs\_mount->mnt\_sb->s\_root;

else

return -EFAULT;

error=create\_dir(kobj,parent,kobject\_name(kobj),&dentry);

...

}

create\_dir创建目录的过程如图7-1



图7-1

create\_dir首先调用lookup\_one\_len在缓存中搜索目标目录的dentry,如果没有找到则分配一个dentry,接着调用sysfs\_lookup函数填充dentry一些特定于文件系统的成员并返回.

接下来调用sysfs\_dirent\_exist检查父目录是否已经存在于目标目录同名的目录,如果有,则返回一个错误值.

然后调用sysfs\_make\_dirent创建一个目录.如果创建成功,就需要调用sysfs\_create为这个目录分配一个inode.这里要注意的一点是,如果创建了一个子目录,父目录的硬链接数要加1.

# 文件创建

文件创建过程和目录相似,这里不再赘述.

# 符号链接处理

## 符号链接跟踪

vfs中调用follow\_link跟踪sysfs的符号链接时,最终调用了sysfs\_follow\_link.

static void \*sysfs\_follow\_link(struct dentry \*dentry, struct nameidata \*nd)

{

int error = -ENOMEM;

//用一个页来保存路径名

unsigned long page = get\_zeroed\_page(GFP\_KERNEL);

if (page)

//将符号链接的路径名保存到page中

error = sysfs\_getlink(dentry, (char \*) page);

nd\_set\_link(nd, error ? ERR\_PTR(error) : (char \*)page);

return NULL;

}

这个函数首先通过get\_zeroed\_page获取一个空白页,这个空白页用于保存符号链接目标的路径.然后调用sysfs\_getlink将路径名存入在前一步申请的页中.最后调用nd\_set\_link函数将这个含有路径名的页面赋值给nd->saved\_names[nd->depth].

sysfs\_getlink首先分别获取符号链接的父目录及目标文件的kobject,然后调用sysfs\_get\_target\_path获取目标路径.

static int sysfs\_getlink(struct dentry \*dentry, char \* path)

{

...

kobj = sysfs\_get\_kobject(dentry->d\_parent);

if (!kobj)

return -EINVAL;

//获取链接的目标的kobj

target\_kobj = sysfs\_get\_kobject(dentry);

if (!target\_kobj) {

kobject\_put(kobj);

return -EINVAL;

}

...

error = sysfs\_get\_target\_path(kobj, target\_kobj, path);

...

}

sysfs\_get\_target\_path的调用关系图如图,调用object\_depth是为了获取目标文件在sysfs中的深度.这个深度是指处于根目录下多少级目录.我们知道一个kobject在sysfs中就以一个目录的形式出现,而kobject的父节点当然就是这个目录的父目录.通过查找kobject的父节点可以获取目标文件的深度.object\_path\_length用于获取目标文件的路径总长度,目标文件是个目录,但是在路径中最后一个结尾的’/’不要,故用object\_path\_length获取长度后还要减1.由于连接文件首先要通过”../”返回sysfs的根目录,再从根目录下查找目标文件,这里的depth \* 3就是需要”../”的总长度.下面那个for循环用于在目标路径前加上需要的”../”.函数最后通过fill\_object\_path,将目标路径填充.



图8-1

代码如下:

static int sysfs\_get\_target\_path(struct kobject \* kobj, struct kobject \* target,

char \*path)

{

char \* s;

int depth, size;

depth = object\_depth(kobj);

size = object\_path\_length(target) + depth \* 3 - 1;

if (size > PATH\_MAX)

return -ENAMETOOLONG;

pr\_debug("%s: depth = %d, size = %d\n", \_\_FUNCTION\_\_, depth, size);

//在路径前加depth个'../'以返回sysfs的根目录

for (s = path; depth--; s += 3)

strcpy(s,"../");

fill\_object\_path(target, path, size);

pr\_debug("%s: path = '%s'\n", \_\_FUNCTION\_\_, path);

return 0;

}

## 符号链接创建

符号链接的创建由函数sysfs\_create\_link开始，这个函数的调用链比较长，图表示出sysfs\_create\_link的调用过程。



图8-2

int sysfs\_create\_link(struct kobject \* kobj, struct kobject \* target, const char \* name)

{

struct dentry \* dentry = kobj->dentry;

int error = -EEXIST;

BUG\_ON(!kobj || !kobj->dentry || !name);

mutex\_lock(&dentry->d\_inode->i\_mutex);

if (!sysfs\_dirent\_exist(dentry->d\_fsdata, name))

error = sysfs\_add\_link(dentry, name, target);

mutex\_unlock(&dentry->d\_inode->i\_mutex);

return error;

}

函数首先调用了sysfs\_dirent\_exist检查dentry->d\_fsdata中是否已经存名字为name的子成员。如果不存在，就调用sysfs\_add\_link添加符号链接。

在sysfs\_add\_link中，首先申请一个sysfs的符号链接对象，接着给这个对象赋值。最后在函数sysfs\_make\_dirent调用sysfs\_new\_dirent分配一个sysfs的目录项sd.并将这个sd插入目录树。

本文一开始也提到过，sysfs目录项中有个s\_element成员。这个成员根据不同的文件类型，包含不同的数据类型。在当前情况下，这个目录项表示一个符号连接，故 s\_element实际上是sysfs符号链接对象，将在sysfs\_add\_link中申请并且填充完成的符号连接对戏那个赋值给sd->s\_element。这样就完成了sysfs中的符号连接的创建过程。下图是从vfs中查找符号连接的图示，其中虚线部分是在符号连接创建的时候创建的。



图8-3