vfs中的路径查询

**宝德技术研究院 李磊**

# 总体过程

vfs中的路径查询主要是通过函数path\_lookup完成的, 此函数主要通过文件名获得文件的nameidata结构, 其定义如下

**fs/namei.c**

int fastcall path\_lookup(const char \*name, unsigned int flags,

struct nameidata \*nd)

{

return do\_path\_lookup(AT\_FDCWD, name, flags, nd);

}

path\_lookup继续调用do\_path\_lookup:

**fs/namei.c**

static int fastcall do\_path\_lookup(int dfd, const char \*name,

unsigned int flags, struct nameidata \*nd)

{

int retval = 0;

int fput\_needed;

struct file \*file;

nd->last\_type = LAST\_ROOT; /\* if there are only slashes... \*/

nd->flags = flags;

nd->depth = 0;

//绝对路径

if (\*name=='/') {

...

} else if (dfd == AT\_FDCWD) {

//相对路径

read\_lock(&current->fs->lock);

//在path\_lookup()的调用者完成使用查找结果之前，这个两个对象都不能被释放

nd->mnt = mntget(current->fs->pwdmnt);

nd->dentry = dget(current->fs->pwd);

read\_unlock(&current->fs->lock);

} else {

//path\_lookup不会走到下面来

...

}

//current

current->total\_link\_count = 0;

retval = link\_path\_walk(name, nd);

...

return retval;

}

函数首先判断传入的name是否以’/’开头, 如果是则认为name是一个绝对路径,这里由于从path\_lookup传入的dfd == AT\_FDCWD, 故走相对路径那条分支.增加current->fs->pwdmnt及current->fs->pwd的引用计数后将值分别赋给nd->mnt及nd->dentry, 接着就调用函数link\_path\_walk:

**fs/namei.c**

int fastcall link\_path\_walk(const char \*name, struct nameidata \*nd)

{

struct nameidata save = \*nd;

int result;

/\* make sure the stuff we saved doesn't go away \*/

dget(save.dentry);

mntget(save.mnt);

//从dcache中读取

result = \_\_link\_path\_walk(name, nd);

if (result == -ESTALE) {

//如果报错--过于陈旧, 就把标志置为"force a real lookup"

\*nd = save;

dget(nd->dentry);

mntget(nd->mnt);

nd->flags |= LOOKUP\_REVAL;

result = \_\_link\_path\_walk(name, nd);

}

dput(save.dentry);

mntput(save.mnt);

return result;

}

这个函数最关键的部分就是调用\_\_link\_path\_walk, 这里我们假设name为路径”/tmp/hello/hi/rocklee”.

**fs/namei.c**

static fastcall int \_\_link\_path\_walk(const char \* name, struct nameidata \*nd)

{

struct path next;

struct inode \*inode;

int err;

unsigned int lookup\_flags = nd->flags;

while (\*name=='/')

name++;

if (!\*name)

goto return\_reval;

inode = nd->dentry->d\_inode;

if (nd->depth)

//符号连接相关

lookup\_flags = LOOKUP\_FOLLOW | (nd->flags & LOOKUP\_CONTINUE);

while循环处理掉name中开头的所有’/’,这时name就等于”tmp/hello/hi/rocklee”了,接下来的循环会以name中的’/’为分割符,一次处理文件名.

/\* At this point we know we have a real path component. \*/

for(;;) {

unsigned long hash;

struct qstr this;

unsigned int c;

nd->flags |= LOOKUP\_CONTINUE;

//判断目录是否可执行(进入)

err = exec\_permission\_lite(inode, nd);

if (err == -EAGAIN)

err = vfs\_permission(nd, MAY\_EXEC);

if (err)

break;

this.name = name;

这里的this.name为”tmp/hello/hi/rocklee”

//c仅用于计算hash

c = \*(const unsigned char \*)name;

//返回hash = 0;

hash = init\_name\_hash();

do {

/\* 这个do循环的目的就是计算hash值 \*/

name++;

hash = partial\_name\_hash(c, hash);

c = \*(const unsigned char \*)name;

} while (c && (c != '/'));

this.len = name - (const char \*) this.name;

计算完hash值后name为”/hello/hi/rocklee”, this.len的值就是”tmp”这个字符串的长度3.

this.hash = end\_name\_hash(hash);

/\* remove trailing slashes? \*/

if (!c)

goto last\_component;

//跳过'/'

while (\*++name == '/');

//最后一个字符为'/'

if (!\*name)

goto last\_with\_slashes;

/\*

\* "." and ".." are special - ".." especially so because it has

\* to be able to know about the current root directory and

\* parent relationships.

\*/

if (this.name[0] == '.') switch (this.len) {

default:

break;

case 2:

//如果目录名中含有'..'

if (this.name[1] != '.')

break;

//回到父目录

follow\_dotdot(nd);

inode = nd->dentry->d\_inode;

/\* fallthrough \*/

case 1:

//如果name中的目录名含有'.', 直接跳过

continue;

}

这里比较有意思的地方是, 如果name中包含”.”或”..”时要经过特殊处理,如果是”.”,就直接进行路径中下一部分文件名的解析, 如果是”..”,还需要跳转到父目录.

/\*

\* See if the low-level filesystem might want

\* to use its own hash..

\*/

//如果低级文件系统有一个自定义的d\_hash目录项方法，则调用它来修改

//散列值

if (nd->dentry->d\_op && nd->dentry->d\_op->d\_hash) {

err = nd->dentry->d\_op->d\_hash(nd->dentry, &this);

if (err < 0)

break;

}

/\* This does the actual lookups.. \*/

//next中保存this的path结构

err = do\_lookup(nd, &this, &next);

if (err)

break;

err = -ENOENT;

inode = next.dentry->d\_inode;

if (!inode)

goto out\_dput;

err = -ENOTDIR;

if (!inode->i\_op)

goto out\_dput;

if (inode->i\_op->follow\_link) {

err = do\_follow\_link(&next, nd);

if (err)

goto return\_err;

err = -ENOENT;

inode = nd->dentry->d\_inode;

if (!inode)

break;

err = -ENOTDIR;

if (!inode->i\_op)

break;

} else

path\_to\_nameidata(&next, nd);

上面的一段代码通过do\_lookup找到和this的qstr对应的path结构,将path结构中的目录项对应的inode取出来,检查inode是不是一个符号链接的inode,如果是,调用do\_follow\_link对符号链接进行解析,如果不是,就将nd的mnt和dentry指向next的mnt和dentry.

err = -ENOTDIR;

//检查刚解析的分量是否指向一个目录（next.dentry->d\_inode具有一个

//自定义的lookup方法）

//如果没有，返回一个错误码-ENOTDIR，因为这个分量位于原路径名的中间

if (!inode->i\_op->lookup)

break;

continue;

/\* here ends the main loop \*/

如果inode->i\_op->lookup存在,表示inode代表一个目录.这是因为inode->i\_op->lookup的作用是通过inode和nameidata找出某个目录项.如果不存在,就表示inode不是一个目录,路径中间的分量如果不是目录,就应该报错,退出.

last\_with\_slashes:

lookup\_flags |= LOOKUP\_FOLLOW | LOOKUP\_DIRECTORY;

last\_component:

/\* Clear LOOKUP\_CONTINUE iff it was previously unset \*/

//清除LOOKUP\_CONTINUE标志

nd->flags &= lookup\_flags | ~LOOKUP\_CONTINUE;

if (lookup\_flags & LOOKUP\_PARENT)

goto lookup\_parent;

if (this.name[0] == '.') switch (this.len) {

default:

break;

case 2:

if (this.name[1] != '.')

break;

follow\_dotdot(nd);

inode = nd->dentry->d\_inode;

/\* fallthrough \*/

case 1:

goto return\_reval;

}

if (nd->dentry->d\_op && nd->dentry->d\_op->d\_hash) {

err = nd->dentry->d\_op->d\_hash(nd->dentry, &this);

if (err < 0)

break;

}

err = do\_lookup(nd, &this, &next);

if (err)

break;

inode = next.dentry->d\_inode;

if ((lookup\_flags & LOOKUP\_FOLLOW)

&& inode && inode->i\_op && inode->i\_op->follow\_link) {

err = do\_follow\_link(&next, nd);

if (err)

goto return\_err;

inode = nd->dentry->d\_inode;

} else

//切换到下一分量

path\_to\_nameidata(&next, nd);

err = -ENOENT;

if (!inode)

// 检查nd->dentry->d\_inode是否为NULL, 这发生在没有索引节点与

//目录项对象关联时，

//通常是因为路径名指向一个不存在的文件。在这种情况下，返回一个

//错误码-ENOENT

break;

if (lookup\_flags & LOOKUP\_DIRECTORY) {

err = -ENOTDIR;

if (!inode->i\_op || !inode->i\_op->lookup)

break;

}

goto return\_base;

如果查找路径时,到了最后一个分量(这里暂时不考虑.和..目录及符号链接的情况),获取inode.如果lookup\_flags中LOOKUP\_DIRECTORY置位,表示当前inode是一个目录,是目录就一定有lookup函数,如果没有,就报错并退出.

lookup\_parent:

nd->last = this;

nd->last\_type = LAST\_NORM;

if (this.name[0] != '.')

goto return\_base;

if (this.len == 1)

nd->last\_type = LAST\_DOT;

else if (this.len == 2 && this.name[1] == '.')

nd->last\_type = LAST\_DOTDOT;

else

goto return\_base;

return\_reval:

/\*

\* We bypassed the ordinary revalidation routines.

\* We may need to check the cached dentry for staleness.

\*/

if (nd->dentry && nd->dentry->d\_sb &&

(nd->dentry->d\_sb->s\_type->fs\_flags & FS\_REVAL\_DOT)) {

err = -ESTALE;

/\* Note: we do not d\_invalidate() \*/

if (!nd->dentry->d\_op->d\_revalidate(nd->dentry, nd))

break;

}

return\_base:

return 0;

out\_dput:

dput\_path(&next, nd);

break;

}

path\_release(nd);

return\_err:

return err;

}

函数执行正常的情况下,最后会跳到return\_base,及返回0.若执行符号链接处理过程中出错就会跳转到return\_err,返回一个错误码,其他过程出错,会跳出for循环,释放nd的mnt和dentry,最后返回一个错误码.这是因为在处理符号链接的时候会释放符号链接的nd.整个函数比较长, 有200多行代码, 其大体框图如图1.



图1

以下是具体的流程图



图2

# 检查权限

在\_\_link\_path\_walk中检查权限主要调用了2个函数,这里需要检查的是可执行权限, 首先是exec\_permission\_lite:

**fs/namei.c**

static int exec\_permission\_lite(struct inode \*inode,

struct nameidata \*nd)

{

umode\_t mode = inode->i\_mode;

if (inode->i\_op && inode->i\_op->permission)

return -EAGAIN;

//umode\_t是个unsinged short型, 占2个字节

if (current->fsuid == inode->i\_uid)

mode >>= 6;

else if (in\_group\_p(inode->i\_gid))

mode >>= 3;

//如果有执行权限(进入)

if (mode & MAY\_EXEC)

goto ok;

if ((inode->i\_mode & S\_IXUGO) && capable(CAP\_DAC\_OVERRIDE))

goto ok;

if (S\_ISDIR(inode->i\_mode) && capable(CAP\_DAC\_OVERRIDE))

goto ok;

if (S\_ISDIR(inode->i\_mode) && capable(CAP\_DAC\_READ\_SEARCH))

goto ok;

return -EACCES;

ok:

return security\_inode\_permission(inode, MAY\_EXEC, nd);

}

函数首先获取inode中的imode, imode中可能为如下值及这些值相或后的新值.

Constant Value Description

-- file format --

EXT2\_S\_IFSOCK 0xC000 socket

EXT2\_S\_IFLNK 0xA000 symbolic link

EXT2\_S\_IFREG 0x8000 regular file

EXT2\_S\_IFBLK 0x6000 block device

EXT2\_S\_IFDIR 0x4000 directory

EXT2\_S\_IFCHR 0x2000 character device

EXT2\_S\_IFIFO 0x1000 fifo

-- process execution user/group override --

EXT2\_S\_ISUID 0x0800 Set process User ID

EXT2\_S\_ISGID 0x0400 Set process Group ID

EXT2\_S\_ISVTX 0x0200 sticky bit

-- access rights --EXT2\_S\_IRUSR 0x0100 user read

EXT2\_S\_IWUSR 0x0080 user write

EXT2\_S\_IXUSR 0x0040 user execute

EXT2\_S\_IRGRP 0x0020 group read

EXT2\_S\_IWGRP 0x0010 group write

EXT2\_S\_IXGRP 0x0008 group execute

EXT2\_S\_IROTH 0x0004 others read

EXT2\_S\_IWOTH 0x0002 others write

EXT2\_S\_IXOTH 0x0001 others execute

如果inode中定义了检查权限的函数,就会退出,用vfs\_permission函数继续判断.如果没有定义接着检查:

(1)进程的uid是否和inode的uid相同,如果相同mode右移6位,将mode中关于组和其他的权限全部去掉.

(2)inode的gid是否包含在当前进程所属组的列表中,如果包含,则把other权限去掉.

(3)经过前两部处理,判断是否可执行,如果可执行就返回成功,对一个目录来说就是可以进入的,这样就可以继续解析路径中的子目录了.

如果没有可执行权限,不会马上返回,还需要进行capability的判断,进程如果有CAP\_DAC\_OVERRIDE的能力,对以下的情形,都可以绕过DAC机制,获得所请求的权限.

* 在user, group, other这3部分权限中,如果其中一个或多个有执行权限
* inode表示一个目录
* 读或者写访问,不存在执行访问

CAP\_DAC\_READ\_SEARCH能力允许在读取文件盒搜索目录时,撤销DAC的权限设置,如果一个进程有该能力,对一下情形,可以允许访问:

* 请求读操作
* inode表示的是一个目录,并且没有写访问

另一个检查权限的函数是vfs\_permission:

**fs/namei.c**

int vfs\_permission(struct nameidata \*nd, int mask)

{

return permission(nd->dentry->d\_inode, mask, nd);

}

vfs\_permission是对permissin的封装.

**fs/namei.c**

int permission(struct inode \*inode, int mask, struct nameidata \*nd)

{

umode\_t mode = inode->i\_mode;

int retval, submask;

if (mask & MAY\_WRITE) {

if (IS\_RDONLY(inode) &&

(S\_ISREG(mode) || S\_ISDIR(mode) || S\_ISLNK(mode)))

return -EROFS;

if (IS\_IMMUTABLE(inode))

return -EACCES;

}

if ((mask & MAY\_EXEC) && S\_ISREG(mode) && !(mode & S\_IXUGO))

return -EACCES;

submask = mask & ~MAY\_APPEND;

if (inode->i\_op && inode->i\_op->permission)

retval = inode->i\_op->permission(inode, submask, nd);

else

retval = generic\_permission(inode, submask, NULL);

if (retval)

return retval;

return security\_inode\_permission(inode, mask, nd);

}

permission这个函数不单单只判断是否可进入了,它首先判断调用者需求的权限是否是写入,如果是的,则继续判断inode所处的文件系统是否是只读文件系统,如果是只读文件系统,目录,普通文件及链接就不能写入(设备文件可以).如果文件系统是可写文件系统, 但是文件是不可修改文件,这时候也会返回错误.

如果对一个普通文件要求执行,但是在inode中读出来的权限中user,group,other都没有将可执行权限置位,也会返回错误.

通常的permission例程并不理解MAY\_APPEND,将此位清除.

如果inode中定义了权限检查函数, 就会继续调用inode中的权限检查函数,如果没有,就会调用通用的权限检查函数generic\_permission:

**fs/namei.c**

int generic\_permission(struct inode \*inode, int mask,

int (\*check\_acl)(struct inode \*inode, int mask))

{

umode\_t mode = inode->i\_mode;

if (current->fsuid == inode->i\_uid)

mode >>= 6;

else {

if (IS\_POSIXACL(inode) && (mode & S\_IRWXG) && check\_acl) {

int error = check\_acl(inode, mask);

if (error == -EACCES)

goto check\_capabilities;

else if (error != -EAGAIN)

return error;

}

if (in\_group\_p(inode->i\_gid))

mode >>= 3;

}

if (((mode & mask & (MAY\_READ|MAY\_WRITE|MAY\_EXEC)) == mask))

return 0;

check\_capabilities:

if (!(mask & MAY\_EXEC) ||

(inode->i\_mode & S\_IXUGO) || S\_ISDIR(inode->i\_mode))

if (capable(CAP\_DAC\_OVERRIDE))

return 0;

if (mask == MAY\_READ || (S\_ISDIR(inode->i\_mode) && !(mask &

MAY\_WRITE)))

if (capable(CAP\_DAC\_READ\_SEARCH))

return 0;

return -EACCES;

}

和函数exec\_permission\_lite一样, generic\_permission也会对权限的user和group进行检查, 但是多了对ACL的检查.这里暂时不对ACL进行深入讨论.对DAC检查失败后并不马上返回失败,而是继续检查capability.分析函数exec\_permission\_lite时已经讨论过capability了,这里不再赘述.

## 权限检查的实验

在vfs中的路径查找中需要判断可执行权限, 为了了解这个权限的作用, 特意将此段代码注释掉

  nd->flags |= LOOKUP\_CONTINUE;   
  //err = exec\_permission\_lite(inode, nd);   
  err = 0;   
  if (err == -EAGAIN)   
   err = vfs\_permission(nd, MAY\_EXEC);   
  if (err)

   break;

这样修改后的kernel就成了不安全的kernel, 在/tmp下存在一个文件 /tmp/hello/noctis.c , 修改代码前,rocklee用户对/tmp/hello/目录没有x权限, 执行cat命令后,提示没有权限,获取不到目标文件的inode信息

[rocklee@localhost tmp]$ cat hello/noctis.c

cat: hello/noctis.c: Permission denied

修改后执行cat命令,将文件中的内容显示出来,说明inode读取没有任何问题.

[rocklee@localhost tmp]$ cat hello/noctis.c

hello rocklee

# 对目录.和..的处理

在路径查询中,如果路径中存在.和..就要经过特殊处理, 对.来说,这个目录可以不处理,直接处理下级目录,但对与..来说,就有些复杂了,这需要跳转到父目录,具体函数为follow\_dotdot.

**fs/namei.c**

static \_\_always\_inline void follow\_dotdot(struct nameidata \*nd)

{

while(1) {

struct vfsmount \*parent;

struct dentry \*old = nd->dentry;

if (nd->dentry == current->fs->root &&

nd->mnt == current->fs->rootmnt) {

break;

}

这里读取当前进程信息, 如果当前目录是当前进程的根目录, 就不能跳到父目录.

if (nd->dentry != nd->mnt->mnt\_root) {

nd->dentry = dget(nd->dentry->d\_parent);

dput(old);

break;

}

如果正在解析的目录不是此目录所在vfsmount中的根目录, 将父目录引用计数加1, 当前目录引用计数减1.这意味着,这种情况是可以直接跳转到父目录的.

parent = nd->mnt->mnt\_parent;

if (parent == nd->mnt) {

break;

}

如果最近解析的目录是nd->mnt文件系统的根目录(这点由上个if得出), 并且这个文件系统也没有被安装在其他文件系统之上（nd->mnt等于nd->mnt->mnt\_parent),那么nd->mnt文件系统通常就是命名空间的根文件系统,在这种情况下，再向上追踪是不可能的.

mntget(parent);

nd->dentry = dget(nd->mnt->mnt\_mountpoint);

dput(old);

mntput(nd->mnt);

nd->mnt = parent;

}

follow\_mount(&nd->mnt, &nd->dentry);

}

如果最近解析的目录是nd->mnt文件系统的根目录，而这个文件系统被安装在其他文件系统之上,那么就需要文件系统切换。因此，把nd->dentry置为nd->mnt->mnt\_mountpoint，且把nd->mnt置为nd->mnt->mnt\_parent.然后再对nd进行判断.直到break.还有一种情况是父目录可能还挂载了其他的文件系统,这时就需要用follow\_mount来处理了.

**fs/namespace.c**

static void follow\_mount(struct vfsmount \*\*mnt, struct dentry

\*\*dentry)

{

while (d\_mountpoint(\*dentry)) {

//搜索目录项高速缓存中已安装文件系统的根目录

struct vfsmount \*mounted = lookup\_mnt(\*mnt, \*dentry);

if (!mounted)

break;

dput(\*dentry);

mntput(\*mnt);

\*mnt = mounted;

\*dentry = dget(mounted->mnt\_root);

}

}

如果父目录上的确挂载了其他的文件系统, 找到文件系统的vfsmount结构,给调用者的nd->mnt和nd->dentry赋值.搜索目录项高速缓存中已安装文件通过函数lookup\_mnt完成.

**fs/namespace.c**

struct vfsmount \*lookup\_mnt(struct vfsmount \*mnt, struct dentry

\*dentry)

{

struct vfsmount \*child\_mnt;

spin\_lock(&vfsmount\_lock);

if ((child\_mnt = \_\_lookup\_mnt(mnt, dentry, 1)))

mntget(child\_mnt);

spin\_unlock(&vfsmount\_lock);

return child\_mnt;

}

此函数调用\_\_lookup\_mnt完成搜索.

**fs/namespace.c**

struct vfsmount \*\_\_lookup\_mnt(struct vfsmount \*mnt, struct dentry \*dentry,int dir)

{

struct list\_head \*head = mount\_hashtable + hash(mnt, dentry);

struct list\_head \*tmp = head;

struct vfsmount \*p, \*found = NULL;

//p是当前vfsmount的子vfsmount

for (;;) {

//从头搜索(最先挂载), 还是从尾搜索(最后挂载)

tmp = dir ? tmp->next : tmp->prev;

p = NULL;

if (tmp == head)

break;

p = list\_entry(tmp, struct vfsmount, mnt\_hash);

if (p->mnt\_parent == mnt && p->mnt\_mountpoint == dentry) {

found = p;

break;

}

}

return found;

}

首先找打hash表的表头, vfsmount组成的hash表和dcache中使用的hash表有些不同,vfsmount表头接的是一个内核使用的标准双向循环链表.找到表头后在链表中搜索以mnt为父vfsmount, 以dentry为挂载点的vfsmount.如果找到的话,函数follow\_dotdot中的nd就获得了对应的vfsmount及dentry了.这里为什么链表头是通过mount\_hashtable + hash(mnt, dentry)找出来的?这是因为在挂载的时候mount hashtable就是通过下面这个函数建立的,可以看出hash的backet是通过hash(nd->mnt, nd->dentry)计算得出的.

**fs/namespace.c**

static void attach\_mnt(struct vfsmount \*mnt, struct nameidata \*nd)

{

mnt\_set\_mountpoint(nd->mnt, nd->dentry, mnt);

list\_add\_tail(&mnt->mnt\_hash, mount\_hashtable +

hash(nd->mnt, nd->dentry));

list\_add\_tail(&mnt->mnt\_child, &nd->mnt->mnt\_mounts);

}

# do\_lookup

do\_lookup主要作用是在散列中找出以 nd->dentry 作为父目录项 , 名称为 name 的目录项.将这个目录以path结构返回.

**fs/namei.c**

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

struct path \*path)

{

struct vfsmount \*mnt = nd->mnt;

//首先调用\_\_d\_lookup()在目录项高速缓存中搜索分量的目录项对象

struct dentry \*dentry = \_\_d\_lookup(nd->dentry, name);

if (!dentry)

goto need\_lookup;

if (dentry->d\_op && dentry->d\_op->d\_revalidate)

goto need\_revalidate;

done:

path->mnt = mnt;

path->dentry = dentry;

\_\_follow\_mount(path);

return 0;

//如果在缓存中没有找到合适的目录项, 则在磁盘中找

need\_lookup:

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

if (IS\_ERR(dentry))

goto fail;

goto done;

need\_revalidate:

if (dentry->d\_op->d\_revalidate(dentry, nd))

goto done;

if (d\_invalidate(dentry))

goto done;

dput(dentry);

goto need\_lookup;

fail:

return PTR\_ERR(dentry);

}

函数首先调用\_\_d\_lookup()在目录项高速缓存中搜索分量的目录项对象, 如果没搜索到目录项,就调用real\_lookup在磁盘上寻找.如果找到匹配的数据,这时候还不能保证,必须调用底层文件系统的dentry\_operations中的d\_revalidate函数检查缓存项是否仍然有效.如果有效则调用\_\_follow\_mount检查这个目录项是否是文件系统的挂载点,如果是,还需要切换文件系统, 然后就返回找到的数据.如果缓存项无效,则使在缓存中找到的dentry无效,然后继续在磁盘上搜索.

\_\_d\_lookup首先通过d\_hash找到散列中的链表头,然后在这个链表中找到成员d\_name.hash和hash匹配的dentry, 再通过d\_unhashed确认此dentry在内存的缓存后,返回此dentry.

**fs/dcache.c**

struct dentry \* \_\_d\_lookup(struct dentry \* parent, struct qstr \* name)

{

unsigned int len = name->len;

unsigned int hash = name->hash;

const unsigned char \*str = name->name;

//找到hash值所在的链表头

struct hlist\_head \*head = d\_hash(parent,hash);

struct dentry \*found = NULL;

struct hlist\_node \*node;

struct dentry \*dentry;

//进行rcu之前, 必须rcu\_read\_lock

rcu\_read\_lock();

//dentry为链表中提取出来的成员

hlist\_for\_each\_entry\_rcu(dentry, node, head, d\_hash) {

struct qstr \*qstr;

//寻找hash值和name传进来的hash值一样的子目录

if (dentry->d\_name.hash != hash)

continue;

if (dentry->d\_parent != parent)

continue;

spin\_lock(&dentry->d\_lock);

/\*

\* Recheck the dentry after taking the lock - d\_move may have

\* changed things. Don't bother checking the hash because we're

\* about to compare the whole name anyway.

\*/

if (dentry->d\_parent != parent)

goto next;

/\*

\* It is safe to compare names since d\_move() cannot

\* change the qstr (protected by d\_lock).

\*/

qstr = &dentry->d\_name;

//如果父目录有比较函数

if (parent->d\_op && parent->d\_op->d\_compare) {

if (parent->d\_op->d\_compare(parent, qstr, name))

//一致返回0, 不一致返回非0

goto next;

} else {

if (qstr->len != len)

goto next;

if (memcmp(qstr->name, str, len))

goto next;

}

if (!d\_unhashed(dentry)) {

//如果此dentry在hash表中

atomic\_inc(&dentry->d\_count);

found = dentry;

}

spin\_unlock(&dentry->d\_lock);

break;

next:

spin\_unlock(&dentry->d\_lock);

}

//结束rcu保护

rcu\_read\_unlock();

return found;

}

# 符号链接处理

内核跟踪符号链接时,必须要防止链接循环,比如有3个符号链接,a指向b,b指向c,c又指向a,如图2, 在访问这3个符号链接中的任何一个,如果不做特殊处理,内核就会处于无限循环中,整个系统都会挂死.



图3

Kernel跟踪符号链接通过函数do\_follow\_link完成.

**fs/namei.c**

static inline int do\_follow\_link(struct path \*path, struct

nameidata \*nd)

{

int err = -ELOOP;

if (current->link\_count >= MAX\_NESTED\_LINKS)

goto loop;

if (current->total\_link\_count >= 40)

goto loop;

cond\_resched();

err = security\_inode\_follow\_link(path->dentry, nd);

if (err)

goto loop;

//link\_count表示嵌套的符号连接数, 不能超过8, 超过就放弃搜索

current->link\_count++;

//总的符号连接数, 不能超过40

current->total\_link\_count++;

nd->depth++;

err = \_\_do\_follow\_link(path, nd);

current->link\_count--;

nd->depth--;

return err;

loop:

dput\_path(path, nd);

path\_release(nd);

return err;

}

函数首先检查current->link\_count和current->total\_link\_count,其中current->link\_count记录了嵌套的链接个数, 内核默认不能超过8, current->total\_link\_count记录了总的符号链接个数,不能超过40.这两个count目的是防止内核进入本节开始提到的内核死循环.接着递增link\_count, total\_link\_count, depth.通过\_\_do\_follow\_link获取链接目标文件的inode信息.如果当前的nd中保存的对象不是一个链接,link\_count和depth减1.

\_\_do\_follow\_link进行更进一步的操作.

**fs/namei.c**

static \_\_always\_inline int \_\_do\_follow\_link(struct path \*path,

struct nameidata \*nd)

{

int error;

void \*cookie;

struct dentry \*dentry = path->dentry;

touch\_atime(path->mnt, dentry);

nd\_set\_link(nd, NULL);

if (path->mnt != nd->mnt) {

path\_to\_nameidata(path, nd);

dget(dentry);

}

mntget(path->mnt);

cookie = dentry->d\_inode->i\_op->follow\_link(dentry, nd);

error = PTR\_ERR(cookie);

if (!IS\_ERR(cookie)) {

char \*s = nd\_get\_link(nd);

error = 0;

if (s)

error = \_\_vfs\_follow\_link(nd, s);

if (dentry->d\_inode->i\_op->put\_link)

dentry->d\_inode->i\_op->put\_link(dentry, nd, cookie);

}

dput(dentry);

mntput(path->mnt);

return error;

}

这个函数的工作有:

1. 更新访问时间
2. 在saved\_names中清除当前解析的符号链接路径
3. 如果path和nd不在同一文件系统中,还需改变nd的mnt及dentry
4. 调用与具体文件系统相关的函数来实现follow\_link方法，给它传递的参数为dentry和nd。它读取存放在符号链接索引节点中的路径名，并把这个路径名保存在nd->saved\_names数组的合适项中。
5. 调用\_\_vfs\_follow\_link,参数为nd及符号链接inode中的路径名.

**fs/namei.c**

static \_\_always\_inline int \_\_vfs\_follow\_link(struct nameidata \*nd

, const char \*link)

{

int res = 0;

char \*name;

//如果路径是以'/'开头的,没有必要在内存中保留前一个路径的任何信息

if (\*link == '/') {

path\_release(nd);

//设置从根目录开始

if (!walk\_init\_root(link, nd))

/\* weird \_\_emul\_prefix() stuff did it \*/

goto out;

}

//查找符号链接目标文件

res = link\_path\_walk(link, nd);

out:

if (nd->depth || res || nd->last\_type!=LAST\_NORM)

return res;

/\*

\* If it is an iterative symlinks resolution in open\_namei()

\* we

\* have to copy the last component. And all that crap because

\* of

\* bloody create() on broken symlinks. Furrfu...

\*/

name = \_\_getname();

if (unlikely(!name)) {

path\_release(nd);

return -ENOMEM;

}

strcpy(name, nd->last.name);

nd->last.name = name;

return 0;

fail:

path\_release(nd);

return PTR\_ERR(link);

}

函数首先判断传入的link是否是一个绝对地址,如果是绝对地址,link之前的在内存中保存的路径信息就是无效的,应该被释放.然后将mnt及dentry置成root的.然后将nd和link传给link\_path\_walk,开始对符号链接的目标文件进行查找.

下面是一个处理符号链接的例子,在/tmp/aaa下有3个文件, 分别是rocklee,rockly,noctis.这3个文件的关系如下

[root@localhost aaa]# ll

total 0

-rw-r--r-- 1 root root 0 Jul 7 23:44 noctis

lrwxrwxrwx 1 root root 6 Jul 7 23:45 rocklee -> rockly

lrwxrwxrwx 1 root root 6 Jul 7 23:45 rockly -> noctis

当使用cat读取/tmp/aaa/rocklee时,代码流程如图4.这是一个非常复杂的递归.要注意的是nd的depth表示符号链接嵌套的个数,不会随着目录的改变而改变,nd的saved\_names用于保存符号链接嵌套的路径, saved\_names[0]没有保存文件名, saved\_names[depth]保存的是当前正在解析的文件名



图4