**Linux文VFS件系统分析**

注：本文是基于linux2.6.18内核代码进行分析的，新版本内核有多处改动，其中我所知道的一个是对进程namespace的改动，新版的内核中又增加了一个数据结构nsproxy其中的mnt\_namespace应该和本文中所述的namespace一致。另一个变化是vfsmount结构，新版内核将其一部分字段抽出形成一个新的结构体mount，作为vfsmount的一个字段。还有一个巨大变化是，在2.6.38内核中path\_lookup函数被完全重写，所以在新版内核中完全找不到该函数了。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | 版本 | 修改内容简介 | 修改日期 | 修改人 |
| 1 | V1.0 | 基本整理完毕 | 2016-03-02 | 李小龙 |
| 2 | V1.1 | 添加了路径查找流程图以及目录 | 2016-03-07 |  |

[一、 注册文件系统 2](#_Toc2071886800)

[1、 注册流程和数据结构 2](#_Toc159310525)

[2、 内核源代码摘录 2](#_Toc20237980)

[二、 安装文件系统 3](#_Toc991350683)

[1、 数据结构 3](#_Toc1590546415)

[2、 安装文件系统要进行的操作 3](#_Toc1500094482)

[3、 数据结构关系图 4](#_Toc1192536539)

[4、 内核代码摘录 6](#_Toc1578214676)

[5、 ext2文件系统的超级块创建 16](#_Toc110525186)

[三、 根文件系统的安装 18](#_Toc1941186500)

[1、 第一阶段，安装特殊文件系统rootfs 19](#_Toc177176116)

[2、 挂载实际跟文件系统 23](#_Toc1759194291)

[四、 卸载文件系统 29](#_Toc600650294)

[1、 从数据结构推测卸载操作要做的工作 29](#_Toc767432551)

[2、 内核代码摘录 29](#_Toc1578130258)

[五、 路径名查找 32](#_Toc1376194648)

[1、 基础知识 32](#_Toc883648388)

[2、 代码流程图 33](#_Toc695496318)

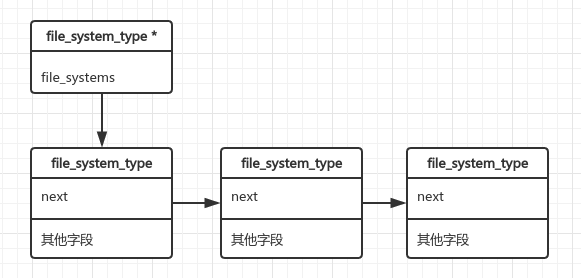
[3、 内核代码摘录 34](#_Toc2084631755)

[4、 从磁盘查找的过程中建立目录项-ext2 49](#_Toc332829219)

1. **注册文件系统**

所有能被Linux系统识别的文件系统都要被注册到一个文件系统类型的单向链表中，file\_systems是链表指针，主要用到的数据结构是file\_system\_type，该结构中next自动指向链表中下一个文件系统类型。

1. 注册流程和数据结构



1. 内核源代码摘录

List\_head<file\_system\_type>fs\_supers:指向文件系统的实例的super\_block对象。因为是双向链表表头所以通过INI\_LIST\_HEAD初始化指向自身。所有文件系统类型都保存在一个file\_systems指向的链表中，file\_system\_type\*<file\_system\_type>next指向链表的下一个元素。

67 int register\_filesystem(struct file\_system\_type \* fs)

68 {

69 int res = 0;

70 struct file\_system\_type \*\* p;

71

72 if (!fs)

73 return -EINVAL;

74 if (fs->next)

75 return -EBUSY;

76 INIT\_LIST\_HEAD(&fs->fs\_supers);

77 write\_lock(&file\_systems\_lock);

78 p = find\_filesystem(fs->name);

79 if (\*p)

80 res = -EBUSY;

81 else

82 \*p = fs;

83 write\_unlock(&file\_systems\_lock);

84 return res;

85 }

下面函数实现查找给定名字的文件系统，如果该文件系统已存在返回指向该文件系统指针的指针，如果不存在则注册文件系统（此时p指向file\_systems链表的尾端），即将fs加入到链表中。（上段代码中还有一个小问题，为什么要返回指针的指针？因为要改变指针指向的对象，而不是改变指针所指对象的值）

45 static struct file\_system\_type \*\*find\_filesystem(const char \*name)

46 {

47 struct file\_system\_type \*\*p;

48 for (p=&file\_systems; \*p; p=&(\*p)->next)

49 if (strcmp((\*p)->name,name) == 0)

50 break;

51 return p;

52 }

1. **安装文件系统**

从mount命令的参数我们知道挂载一个文件系统需要挂载点目录，设备文件名，可选的文件系统类型参数，了解磁盘和文件系统的人知道，每个文件系统为了管理磁盘空间，都会有一个超级块结构。所以安装文件系统就是创建一个一些数据结构（包括超级块），并修改与其相关的数据结构。

1. 数据结构

super\_blocks：是一个list\_head类型的全局变量，是一个双向链表的表头。用于链接所有创 建的超级块数据结构；

vfsmount：保存已安装文件系统的信息，例如挂载点目录项，以及安装在那个文件系统下；

mount\_hashtable：哈希数组，每个元素保存具有相同hash值的文件系统vfsmount结构的链

表头；

namespace：属于进程，进程通过namespace保存其文件系统树。

nameidata：是一个临时的数据结构，在挂载文件系统时用于保存挂载点的相关信息。

1. 安装文件系统要进行的操作

①、创建与具体文件系统类型相对应的超级快super\_block，系统中有一个全局变量super\_blocks用于保存所有的超级快的双向链表的表头，所以要把安装的文件系统的super\_block加入该链表。（super\_block的s\_list字段）

②、创建vfsmout结构，每个安装的文件系统都有一个vfsmount数据结构，应在mount\_hashtable散列表中加入该vfsmoun，vfsmount的mnt\_hash指向相邻的元素。

用于指明该文件系统的父文件系统（mnt\_parent)

该文件系统的安装点dentry(mnt\_mountpoint)

该文件系统的根目录dentry(mnt\_root)

包含的子文件系统（mnt\_mounts)

该文件系统的超级快（mnt\_sb)

该文件系统的设备名称（mnt\_dev)

③、每一个进程都有一个namespace，用于说明该进程所拥有的文件系统，组成一个双向循环链表，namespace中list存放表头，每个vfsmount的mnt\_list指向前后的成员。

④、每个类型的文件系统都有一个file\_system\_type结构，该结构中保存了所有安装的该文件系统类型的超级块链表头（fs\_supers)，super\_block中的s\_instance字段用于指向链表的相邻元素。

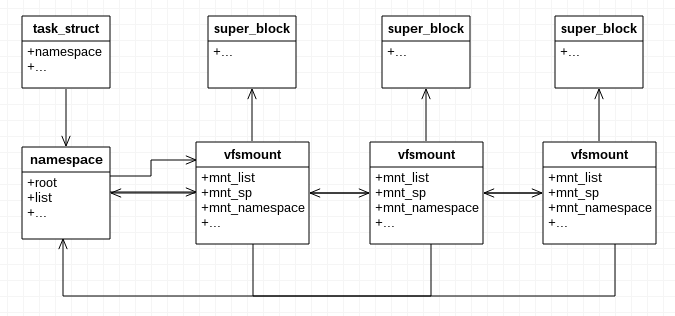
所以安装一个文件系统就是修改或者初始化上面这些数据结构

1. 数据结构关系图

①、task\_struct、namespace、vfsmount和super\_block结构。

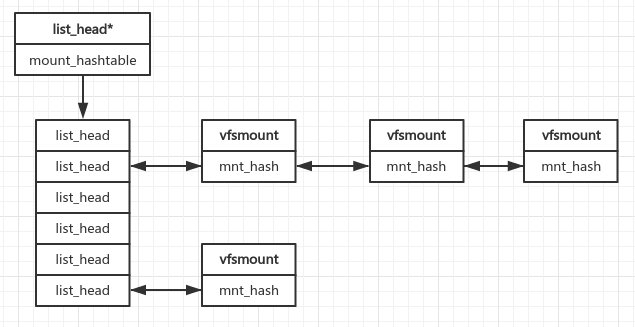
注：新版内核对namespace和vfsmount更改较大，主要是为了实现操作系统级的虚拟化技术，不过只要看懂了下面这些原理，弄清楚新内核中的数据结构关系也是很容易的，只是对一些数据结构的字段进行拆分和重组。

进程描述符的task\_struct字段指向namespace结构，namespace的root字段指向根目录已安装文件系统的根vfsmount结构，list字段做为双向链表的表头，将其下安装的文件系统vfsmount结构串联起来，mnt\_list指向相邻元素，mnt\_sp指向超级块结构。



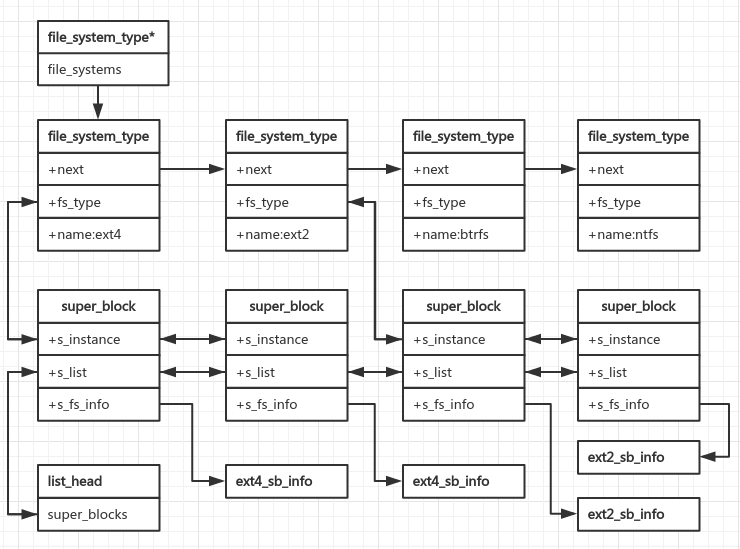
②、mount\_hashtable和vfsmount结构

所有的vfsmout结构都存储在一个mount\_hashtable指向的散列表中，散列表中存放一个双向链表的头指针，处于同一个链表中的vfsmount结构具有相同的散列值，mnt\_hash字段指向相邻元素。



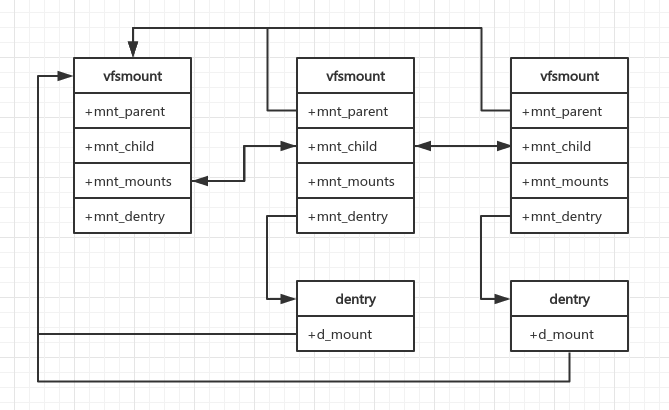
③、file\_system\_type、super\_block、super\_blocks、file\_systems和具体文件系统超级块

同一种文件系统的超级块通过s\_instance字段链接成一个双向链表，file\_system\_type的fs\_type字段是双向链表的头。所有的超级块通过s\_list连接成一个双向链表，表头保存在super\_blocks全局变量中。s\_fs\_info字段指向具体文件系统的超级块信息。



④、子文件系统vfsmount和父文件系统vfsmout以及挂载点目录项dentry

所有挂载在一个文件系统下的vfsmount结构组成一个双向链表，父文件系统的mnt\_mounts字段是链表头，子文件系统的mnt\_parent指向父文件系统vfsmount结构。mnt\_dentry字段指向父文件系统下的目录项dentry，父文件系统下的dentry都有一个字段d\_mount指向其vfsmount结构。



1. 内核代码摘录

把参数从用户空间拷贝到内核空间，然后调用do\_mount()进行安装操作。

1549 asmlinkage long sys\_mount(char \_\_user \* dev\_name, char \_\_user \* dir\_name,

1550 char \_\_user \* type, unsigned long flags,

1551 void \_\_user \* data)

1552 {

1553 int retval;

1554 unsigned long data\_page;

1555 unsigned long type\_page;

1556 unsigned long dev\_page;

1557 char \*dir\_page;

1558

1559 retval = copy\_mount\_options(type, &type\_page);

1560 if (retval < 0)

1561 return retval;

1562

1563 dir\_page = getname(dir\_name);

1564 retval = PTR\_ERR(dir\_page);

1565 if (IS\_ERR(dir\_page))

1566 goto out1;

1567

1568 retval = copy\_mount\_options(dev\_name, &dev\_page);

1569 if (retval < 0)

1570 goto out2;

1571

1572 retval = copy\_mount\_options(data, &data\_page);

1573 if (retval < 0)

1574 goto out3;

1575

1576 lock\_kernel();

1577 retval = do\_mount((char \*)dev\_page, dir\_page, (char \*)type\_page,

1578 flags, (void \*)data\_page);

1579 unlock\_kernel();

1580 free\_page(data\_page);

1581

1582 out3:

1583 free\_page(dev\_page);

1584 out2:

1585 putname(dir\_page);

1586 out1:

1587 free\_page(type\_page);

1588 return retval;

1589 }

①、对路径名和设备名做一些检查，例如:不能为空字符串，必须已’\0’结束符结尾;

②、对标志flags做相应的处理，例如如果没有设备则清除flags中相对应的位;

③、调用path\_lookup()获取挂载点的相关信息，存放在nameidata数据结构nd中;

struct nameidata {

struct dentry \*dentry;

struct vfsmount \*mnt;

struct qstr last;

unsigned int flags;

int last\_type;

unsigned depth;

char \*saved\_names[MAX\_NESTED\_LINKS + 1];

/\* Intent data \*/

union {

struct open\_intent open;

} intent;

};

从上述定义中可以看出nameidata数据结构保存挂载点的目录项，vfsmount描述符等文件系统相关信息。

④、根据flag进行具体操作。do\_new\_mount()进行挂载；

⑤、因为查找结果已经使用完毕，调用path\_release()减小目录项和vfsmount的引用计数；

1370 /\*

1371 \* Flags is a 32-bit value that allows up to 31 non-fs dependent flags to

1372 \* be given to the mount() call (ie: read-only, no-dev, no-suid etc).

1373 \*

1374 \* data is a (void \*) that can point to any structure up to

1375 \* PAGE\_SIZE-1 bytes, which can contain arbitrary fs-dependent

1376 \* information (or be NULL).

1377 \*

1378 \* Pre-0.97 versions of mount() didn't have a flags word.

1379 \* When the flags word was introduced its top half was required

1380 \* to have the magic value 0xC0ED, and this remained so until 2.4.0-test9.

1381 \* Therefore, if this magic number is present, it carries no information

1382 \* and must be discarded.

1383 \*/

1384 long do\_mount(char \*dev\_name, char \*dir\_name, char \*type\_page,

1385 unsigned long flags, void \*data\_page)

1386 {

1387 struct nameidata nd;

1388 int retval = 0;

1389 int mnt\_flags = 0;

1390

1391 /\* Discard magic \*/

1392 if ((flags & MS\_MGC\_MSK) == MS\_MGC\_VAL)

1393 flags &= ~MS\_MGC\_MSK;

1394

1395 /\* Basic sanity checks \*/

1396

1397 if (!dir\_name || !\*dir\_name || !memchr(dir\_name, 0, PAGE\_SIZE))

1398 return -EINVAL;

1399 if (dev\_name && !memchr(dev\_name, 0, PAGE\_SIZE))

1400 return -EINVAL;

1401

1402 if (data\_page)

1403 ((char \*)data\_page)[PAGE\_SIZE - 1] = 0;

1404

1405 /\* Separate the per-mountpoint flags \*/

1406 if (flags & MS\_NOSUID)

1407 mnt\_flags |= MNT\_NOSUID;

1408 if (flags & MS\_NODEV)

1409 mnt\_flags |= MNT\_NODEV;

1410 if (flags & MS\_NOEXEC)

1411 mnt\_flags |= MNT\_NOEXEC;

1412 if (flags & MS\_NOATIME)

1413 mnt\_flags |= MNT\_NOATIME;

1414 if (flags & MS\_NODIRATIME)

1415 mnt\_flags |= MNT\_NODIRATIME;

1416

1417 flags &= ~(MS\_NOSUID | MS\_NOEXEC | MS\_NODEV | MS\_ACTIVE |

1418 MS\_NOATIME | MS\_NODIRATIME);

1419

1420 /\* ... and get the mountpoint \*/

1421 retval = path\_lookup(dir\_name, LOOKUP\_FOLLOW, &nd);

1422 if (retval)

1423 return retval;

1424

1425 retval = security\_sb\_mount(dev\_name, &nd, type\_page, flags, data\_page);

1426 if (retval)

1427 goto dput\_out;

1428

1429 if (flags & MS\_REMOUNT)

1430 retval = do\_remount(&nd, flags & ~MS\_REMOUNT, mnt\_flags,

1431 data\_page);

1432 else if (flags & MS\_BIND)

1433 retval = do\_loopback(&nd, dev\_name, flags & MS\_REC);

1434 else if (flags & (MS\_SHARED | MS\_PRIVATE | MS\_SLAVE | MS\_UNBINDABLE))

1435 retval = do\_change\_type(&nd, flags);

1436 else if (flags & MS\_MOVE)

1437 retval = do\_move\_mount(&nd, dev\_name);

1438 else

1439 retval = do\_new\_mount(&nd, type\_page, flags, mnt\_flags,

1440 dev\_name, data\_page);

1441 dput\_out:

1442 path\_release(&nd);

1443 return retval;

1444 }

下面的所有代码都是为了完成第2小结的操作：安装一个文件系统就是修改或者初始化那些数据结构

①、检查字符串是否以”\0”结尾；

②、进行进程权限验证；

③、调用do\_kern\_mount()函数创建并初始化一个vfsmount数据结构。

④、调用do\_add\_mount()修改相关的数据结构。

1065 /\*

1066 \* create a new mount for userspace and request it to be added into the

1067 \* namespace's tree

1068 \*/

1069 static int do\_new\_mount(struct nameidata \*nd, char \*type, int flags,

1070 int mnt\_flags, char \*name, void \*data)

1071 {

1072 struct vfsmount \*mnt;

1073

1074 if (!type || !memchr(type, 0, PAGE\_SIZE))

1075 return -EINVAL;

1076

1077 /\* we need capabilities... \*/

1078 if (!capable(CAP\_SYS\_ADMIN))

1079 return -EPERM;

1080

1081 mnt = do\_kern\_mount(type, flags, name, data);

1082 if (IS\_ERR(mnt))

1083 return PTR\_ERR(mnt);

1084

1085 return do\_add\_mount(mnt, nd, mnt\_flags, NULL);

1086 }

①、调用get\_fs\_type()获取名字为fstype的文件系统类型结构指针file\_system\_type。

②、调用vfs\_kern\_mount()创建一个vfsmount结构，如果需要还要创建super\_block。

869 struct vfsmount \*

870 do\_kern\_mount(const char \*fstype, int flags, const char \*name, void \*data)

871 {

872 struct file\_system\_type \*type = get\_fs\_type(fstype);

873 struct vfsmount \*mnt;

874 if (!type)

875 return ERR\_PTR(-ENODEV);

876 mnt = vfs\_kern\_mount(type, flags, name, data);

877 put\_filesystem(type);

878 return mnt;

879 }

①、调用alloc\_secdata()动态创建一个vfsmount结构。

②、调用get\_sb()获取或者创建一个超级块，其和底层文件系统密切相关。

struct vfsmount \*

vfs\_kern\_mount(struct file\_system\_type \*type, int flags, const char \*name, void \*data)

{

struct vfsmount \*mnt;

char \*secdata = NULL;

int error;

if (!type)

return ERR\_PTR(-ENODEV);

error = -ENOMEM;

mnt = alloc\_vfsmnt(name);

if (!mnt)

goto out;

if (data) {

secdata = alloc\_secdata();

if (!secdata)

goto out\_mnt;

error = security\_sb\_copy\_data(type, data, secdata);

if (error)

goto out\_free\_secdata;

}

error = type->get\_sb(type, flags, name, data, mnt);

if (error < 0)

goto out\_free\_secdata;

error = security\_sb\_kern\_mount(mnt->mnt\_sb, secdata);

if (error)

goto out\_sb;

mnt->mnt\_mountpoint = mnt->mnt\_root;

mnt->mnt\_parent = mnt;

up\_write(&mnt->mnt\_sb->s\_umount);

free\_secdata(secdata);

return mnt;

out\_sb:

dput(mnt->mnt\_root);

up\_write(&mnt->mnt\_sb->s\_umount);

deactivate\_super(mnt->mnt\_sb);

out\_free\_secdata:

free\_secdata(secdata);

out\_mnt:

free\_vfsmnt(mnt);

out:

return ERR\_PTR(error);

}

do\_add\_mount修改与vfsmout结构相关的数据结构。调用graft\_tree进行具体操作。

1088 /\*

1089 \* add a mount into a namespace's mount tree

1090 \* - provide the option of adding the new mount to an expiration list

1091 \*/

1092 int do\_add\_mount(struct vfsmount \*newmnt, struct nameidata \*nd,

1093 int mnt\_flags, struct list\_head \*fslist)

1094 {

1095 int err;

1096

1097 down\_write(&namespace\_sem);

1098 /\* Something was mounted here while we slept \*/

1099 while (d\_mountpoint(nd->dentry) && follow\_down(&nd->mnt, &nd->dentry))

1100 ;

1101 err = -EINVAL;

1102 if (!check\_mnt(nd->mnt))

1103 goto unlock;

1104

1105 /\* Refuse the same filesystem on the same mount point \*/

1106 err = -EBUSY;

1107 if (nd->mnt->mnt\_sb == newmnt->mnt\_sb &&

1108 nd->mnt->mnt\_root == nd->dentry)

1109 goto unlock;

1110

1111 err = -EINVAL;

1112 if (S\_ISLNK(newmnt->mnt\_root->d\_inode->i\_mode))

1113 goto unlock;

1114

1115 newmnt->mnt\_flags = mnt\_flags;

1116 if ((err = graft\_tree(newmnt, nd)))

1117 goto unlock;

1118

1119 if (fslist) {

1120 /\* add to the specified expiration list \*/

1121 spin\_lock(&vfsmount\_lock);

1122 list\_add\_tail(&newmnt->mnt\_expire, fslist);

1123 spin\_unlock(&vfsmount\_lock);

1124 }

1125 up\_write(&namespace\_sem);

1126 return 0;

1127

1128 unlock:

1129 up\_write(&namespace\_sem);

1130 mntput(newmnt);

1131 return err;

1132 }

graft\_tree又调用attach\_recursive\_mount()函数。

855 static int graft\_tree(struct vfsmount \*mnt, struct nameidata \*nd)

856 {

857 int err;

858 if (mnt->mnt\_sb->s\_flags & MS\_NOUSER)

859 return -EINVAL;

860

861 if (S\_ISDIR(nd->dentry->d\_inode->i\_mode) !=

862 S\_ISDIR(mnt->mnt\_root->d\_inode->i\_mode))

863 return -ENOTDIR;

864

865 err = -ENOENT;

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

867 if (IS\_DEADDIR(nd->dentry->d\_inode))

868 goto out\_unlock;

869

870 err = security\_sb\_check\_sb(mnt, nd);

871 if (err)

872 goto out\_unlock;

873

874 err = -ENOENT;

875 if (IS\_ROOT(nd->dentry) || !d\_unhashed(nd->dentry))

876 err = attach\_recursive\_mnt(mnt, nd, NULL);

877 out\_unlock:

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

879 if (!err)

880 security\_sb\_post\_addmount(mnt, nd);

881 return err;

882 }

①、通过mnt\_set\_mountpoint()将子vfsmount中的mnt\_parent指向父vfsmount，将子vfsmount的mnt\_mountpoint指向位于父文件系统中的挂载点dentry；

②、通过commit\_tree()将子文件系统添加到内核的文件系统哈希表中，并将子文件系统添加到父文件系统对应的子文件系统链表中；

/\*

...

816 \* if the source mount is a tree, the operations explained above is

817 \* applied to each mount in the tree.

818 \* Must be called without spinlocks held, since this function can sleep

819 \* in allocations.

820 \*/

821 static int attach\_recursive\_mnt(struct vfsmount \*source\_mnt,

822 struct nameidata \*nd, struct nameidata \*parent\_nd)

823 {

824 LIST\_HEAD(tree\_list);

825 struct vfsmount \*dest\_mnt = nd->mnt;

826 struct dentry \*dest\_dentry = nd->dentry;

827 struct vfsmount \*child, \*p;

828

829 if (propagate\_mnt(dest\_mnt, dest\_dentry, source\_mnt, &tree\_list))

830 return -EINVAL;

831

832 if (IS\_MNT\_SHARED(dest\_mnt)) {

833 for (p = source\_mnt; p; p = next\_mnt(p, source\_mnt))

834 set\_mnt\_shared(p);

835 }

836

837 spin\_lock(&vfsmount\_lock);

838 if (parent\_nd) {

839 detach\_mnt(source\_mnt, parent\_nd);

840 attach\_mnt(source\_mnt, nd);

841 touch\_namespace(current->namespace);

842 } else {

843 mnt\_set\_mountpoint(dest\_mnt, dest\_dentry, source\_mnt);

844 commit\_tree(source\_mnt);

845 }

846

847 list\_for\_each\_entry\_safe(child, p, &tree\_list, mnt\_hash) {

848 list\_del\_init(&child->mnt\_hash);

849 commit\_tree(child);

850 }

851 spin\_unlock(&vfsmount\_lock);

852 return 0;

853 }

详解见下面的叙述。

174 void mnt\_set\_mountpoint(struct vfsmount \*mnt, struct dentry \*dentry,

175 struct vfsmount \*child\_mnt)

176 {

177 child\_mnt->mnt\_parent = mntget(mnt);

178 child\_mnt->mnt\_mountpoint = dget(dentry);

179 dentry->d\_mounted++;

180 }

190 /\*

191 \* the caller must hold vfsmount\_lock

192 \*/

193 static void commit\_tree(struct vfsmount \*mnt)

194 {

195 struct vfsmount \*parent = mnt->mnt\_parent;

196 struct vfsmount \*m;

197 LIST\_HEAD(head);

198 struct namespace \*n = parent->mnt\_namespace;

199

200 BUG\_ON(parent == mnt);

201

202 list\_add\_tail(&head, &mnt->mnt\_list);

203 list\_for\_each\_entry(m, &head, mnt\_list)

204 m->mnt\_namespace = n;

205 list\_splice(&head, n->list.prev);

206

207 list\_add\_tail(&mnt->m内核代码摘录nt\_hash, mount\_hashtable +

208 hash(parent, mnt->mnt\_mountpoint));

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

210 touch\_namespace(n);

211 }

上述mnt\_set\_mountpoint()代码详解

①、将当前文件系统的名字空间设置为父名字空间，父vfsmount通过当前vfsmount中的mnt\_parent获取；将每个mnt\_list所在的vfsmount的namespace指向父vfsmount的命名空间，再将其连接到父名字空间的双向链表中。

202 list\_add\_tail(&head, &mnt->mnt\_list);

203 list\_for\_each\_entry(m, &head, mnt\_list)

204 m->mnt\_namespace = n;

205 list\_splice(&head, n->list.prev);

②、将当前vfsmount加入到对应哈希值的冲突链表当中,哈希值通过hash()计算。其中，mnt\_hash作为链表元素。

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

208 hash(parent, mnt->mnt\_mountpoint));

③、将当前vfsmount加入到父vfsmount对应的子文件系统链表mnt\_mounts中。其中，mnt\_child作为链表元素。最后一句应该是唤醒一个因等待namespace->sem的信号量阻塞的进程。

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

210 touch\_namespace(n);

从整个挂载的处理流程上看，挂载的本质就是将源文件系统的vfsmount结构连接到目的文件系统对应的vfsmount结构中，即具体涉及到两个vfsmount中字段的指向问题。两个vfsmount具体父子等级关系，这也对应着内核中目录树的父子等级关系。

【参考】1、<http://blogread.cn/it/article/7046>

在使用完nd对象之后再释放查找结果的dentry和mnt指向的对象，因为结果都引用了原有的对象，在引用是增加对象的引用计数值即d\_count和mnt\_count，所以使用完结果后要减小引用计数，如果引用计数等于0则可以释放该对象了。

347 void path\_release(struct nameidata \*nd)

348 {

349 dput(nd->dentry);

350 mntput(nd->mnt);

351 }

1. ext2文件系统的超级块创建

file\_system\_type数据结构：

1310 struct file\_system\_type {

1311 const char \*name;

1312 int fs\_flags;

1313 int (\*get\_sb) (struct file\_system\_type \*, int,

1314 const char \*, void \*, struct vfsmount \*);

1315 void (\*kill\_sb) (struct super\_block \*);

1316 struct module \*owner;

1317 struct file\_system\_type \* next;

1318 struct list\_head fs\_supers;

1319 struct lock\_class\_key s\_lock\_key;

1320 struct lock\_class\_key s\_umount\_key;

1321 };

ext2文件系统的file\_system\_type结构初始值。

27 static struct file\_system\_type efs\_fs\_type = {

28 .owner = THIS\_MODULE,

29 .name = "efs",

30 .get\_sb = efs\_get\_sb,

31 .kill\_sb = kill\_block\_super,

32 .fs\_flags = FS\_REQUIRES\_DEV,

33 };

通过get\_sb读取给定文件系统的超级块。在ext2文件系统中，该函数指针指向efs\_get\_sb()函数，该函数接收efs\_fill\_super()函数做为参数，从ext2磁盘分区读取文件系统的超级块。

21 static int efs\_get\_sb(struct file\_system\_type \*fs\_type,

22 int flags, const char \*dev\_name, void \*data, struct vfsmount \*mnt)

23 {

24 return get\_sb\_bdev(fs\_type, flags, dev\_name, data, efs\_fill\_super, mnt);

25 }

①、调用open\_bdev\_excl()打开设备文件名为dev\_name的块设备

②、调用sget()搜索是否已经有该设备对应的超级块对象，如果存在返回其地址，否则分配并初始化一个新的超级块对象，并将其插入到文件系统链表和超级块全局链表中。

③、拷贝设置超级块的s\_flags字段和其他一些值；

④、调用依赖文件系统的fill\_super函数从设备中读取超级块信息填充到super\_block;

⑤、返回超级块对象的地址；

686 int get\_sb\_bdev(struct file\_system\_type \*fs\_type,

687 int flags, const char \*dev\_name, void \*data,

688 int (\*fill\_super)(struct super\_block \*, void \*, int),

689 struct vfsmount \*mnt)

690 {

691 struct block\_device \*bdev;

692 struct super\_block \*s;

693 int error = 0;

694

695 bdev = open\_bdev\_excl(dev\_name, flags, fs\_type);

696 if (IS\_ERR(bdev))

697 return PTR\_ERR(bdev);

698

699 /\*

700 \* once the super is inserted into the list by sget, s\_umount

701 \* will protect the lockfs code from trying to start a snapshot

702 \* while we are mounting

703 \*/

704 mutex\_lock(&bdev->bd\_mount\_mutex);

705 s = sget(fs\_type, test\_bdev\_super, set\_bdev\_super, bdev);

706 mutex\_unlock(&bdev->bd\_mount\_mutex);

707 if (IS\_ERR(s))

708 goto error\_s;

709

710 if (s->s\_root) {

711 if ((flags ^ s->s\_flags) & MS\_RDONLY) {

712 up\_write(&s->s\_umount);

713 deactivate\_super(s);

714 error = -EBUSY;

715 goto error\_bdev;

716 }

717

718 close\_bdev\_excl(bdev);

719 } else {

720 char b[BDEVNAME\_SIZE];

721

722 s->s\_flags = flags;

723 strlcpy(s->s\_id, bdevname(bdev, b), sizeof(s->s\_id));

724 sb\_set\_blocksize(s, block\_size(bdev));

725 error = fill\_super(s, data, flags & MS\_SILENT ? 1 : 0);

726 if (error) {

727 up\_write(&s->s\_umount);

728 deactivate\_super(s);

729 goto error;

730 }

731

732 s->s\_flags |= MS\_ACTIVE;

733 bdev\_uevent(bdev, KOBJ\_MOUNT);

734 }

735

736 return simple\_set\_mnt(mnt, s);

737

738 error\_s:

739 error = PTR\_ERR(s);

740 error\_bdev:

741 close\_bdev\_excl(bdev);

742 error:

743 return error;

744 }

【参考】2、深入理解linux内核。

1. **根文件系统的安装**

Linux文件系统的安装工作的第一阶段是由init\_rootfs和init\_mount\_tree函数完成的。而内核的初始化主要是由start\_kernel()函数完成的，所有start\_kernel函数经过若干次系统调用最终调用init\_mount\_tree()函数进行根文件系统的安装工作。

1. 第一阶段，安装特殊文件系统rootfs

下面几个函数反应了从start\_kernel到init\_mount\_tree函数的调用过程。

456 asmlinkage void \_\_init start\_kernel(void)

457 {

/\*其他一些初始化工作

...

\*/

570 vfs\_caches\_init(num\_physpages);

/\*其他一些初始化工作

...

\*/

587 rest\_init();

588 }

1754 void \_\_init vfs\_caches\_init(unsigned long mempages)

1755 {

1756 unsigned long reserve;

1757

1758 /\* Base hash sizes on available memory, with a reserve equal to

1759 150% of current kernel size \*/

1760

1761 reserve = min((mempages - nr\_free\_pages()) \* 3/2, mempages - 1);

1762 mempages -= reserve;

1763

1764 names\_cachep = kmem\_cache\_create("names\_cache", PATH\_MAX, 0,

1765 SLAB\_HWCACHE\_ALIGN|SLAB\_PANIC, NULL, NULL);

1766

1767 filp\_cachep = kmem\_cache\_create("filp", sizeof(struct file), 0,

1768 SLAB\_HWCACHE\_ALIGN|SLAB\_PANIC, NULL, NULL);

1769

1770 dcache\_init(mempages);

1771 inode\_init(mempages);

1772 files\_init(mempages);

1773 mnt\_init(mempages);

1774 bdev\_cache\_init();

1775 chrdev\_init();

1776 }

1819 void \_\_init mnt\_init(unsigned long mempages)

1820 {

1821 struct list\_head \*d;

1822 unsigned int nr\_hash;

1823 int i;

1824

1825 init\_rwsem(&namespace\_sem);

1826

1827 mnt\_cache = kmem\_cache\_create("mnt\_cache", sizeof(struct vfsmount),

1828 0, SLAB\_HWCACHE\_ALIGN | SLAB\_PANIC, NULL, NULL);

1829

1830 mount\_hashtable = (struct list\_head \*)\_\_get\_free\_page(GFP\_ATOMIC);

1831

1832 if (!mount\_hashtable)

1833 panic("Failed to allocate mount hash table\n");

1834

1835 /\*

1836 \* Find the power-of-two list-heads that can fit into the allocation..

1837 \* We don't guarantee that "sizeof(struct list\_head)" is necessarily

1838 \* a power-of-two.

1839 \*/

1840 nr\_hash = PAGE\_SIZE / sizeof(struct list\_head);

1841 hash\_bits = 0;

1842 do {

1843 hash\_bits++;

1844 } while ((nr\_hash >> hash\_bits) != 0);

1845 hash\_bits--;

1846

1847 /\*

1848 \* Re-calculate the actual number of entries and the mask

1849 \* from the number of bits we can fit.

1850 \*/

1851 nr\_hash = 1UL << hash\_bits;

1852 hash\_mask = nr\_hash - 1;

1853

1854 printk("Mount-cache hash table entries: %d\n", nr\_hash);

1855

1856 /\* And initialize the newly allocated array \*/

1857 d = mount\_hashtable;

1858 i = nr\_hash;

1859 do {

1860 INIT\_LIST\_HEAD(d);

1861 d++;

1862 i--;

1863 } while (i);

1864 sysfs\_init();

1865 subsystem\_register(&fs\_subsys);

1866 init\_rootfs();

1867 init\_mount\_tree();

1868 }

init\_\_mount\_tree()函数完成挂载rootfs特殊文件系统。

①、调用do\_kern\_mount()函数创建文件系统类型名（file\_system\_type->name）为rootfs的特殊文件系统的vfsmount结构和super\_block结构。

②、创建super\_block结构最终调用type->get\_sb()函数，该函数又调用rootfs\_get\_sb()函数，接而又调用get\_sb\_nodev()函数，该函数又调用sget()函数创建一个特殊文件系统的超级块（通过回调函数set\_anon\_super。然后调用ramfs\_fill\_super()回调函数对超级块进行填充。

③、创建并设置namespace，mnt\_namespace指向它

④、系统中每个进程的namespace字段设置为namespace对象的地址；

⑤、进程0的根目录和当前工作目录设置为根文件系统。

1787 static void \_\_init init\_mount\_tree(void)

1788 {

1789 struct vfsmount \*mnt;

1790 struct namespace \*namespace;

1791 struct task\_struct \*g, \*p;

1792

1793 mnt = do\_kern\_mount("rootfs", 0, "rootfs", NULL);

1794 if (IS\_ERR(mnt))

1795 panic("Can't create rootfs");

1796 namespace = kmalloc(sizeof(\*namespace), GFP\_KERNEL);

1797 if (!namespace)

1798 panic("Can't allocate initial namespace");

1799 atomic\_set(&namespace->count, 1);

1800 INIT\_LIST\_HEAD(&namespace->list);

1801 init\_waitqueue\_head(&namespace->poll);

1802 namespace->event = 0;

1803 list\_add(&mnt->mnt\_list, &namespace->list);

1804 namespace->root = mnt;

1805 mnt->mnt\_namespace = namespace;

1806

1807 init\_task.namespace = namespace;

1808 read\_lock(&tasklist\_lock);

1809 do\_each\_thread(g, p) {

1810 get\_namespace(namespace);

1811 p->namespace = namespace;

1812 } while\_each\_thread(g, p);

1813 read\_unlock(&tasklist\_lock);

1814

1815 set\_fs\_pwd(current->fs, namespace->root, namespace->root->mnt\_root);

1816 set\_fs\_root(current->fs, namespace->root, namespace->root->mnt\_root);

1817 }

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

165 {

166 struct inode \* inode;

167 struct dentry \* root;

168

169 sb->s\_maxbytes = MAX\_LFS\_FILESIZE;

170 sb->s\_blocksize = PAGE\_CACHE\_SIZE;

171 sb->s\_blocksize\_bits = PAGE\_CACHE\_SHIFT;

172 sb->s\_magic = RAMFS\_MAGIC;

173 sb->s\_op = &ramfs\_ops;

174 sb->s\_time\_gran = 1;

175 inode = ramfs\_get\_inode(sb, S\_IFDIR | 0755, 0);

176 if (!inode)

177 return -ENOMEM;

178

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

180 if (!root) {

181 iput(inode);

182 return -ENOMEM;

183 }

184 sb->s\_root = root;

185 return 0;

186 }

为rootfs文件系统创建一个根目录和inode

rootfs文件系统的超级块操作

159 static struct super\_operations ramfs\_ops = {

160 .statfs = simple\_statfs,

161 .drop\_inode = generic\_delete\_inode,

162 };

1612 /\*

1613 \* Replace the fs->{pwdmnt,pwd} with {mnt,dentry}. Put the old values.

1614 \* It can block. Requires the big lock held.

1615 \*/

1616 void set\_fs\_pwd(struct fs\_struct \*fs, struct vfsmount \*mnt,

1617 struct dentry \*dentry)

1618 {

1619 struct dentry \*old\_pwd;

1620 struct vfsmount \*old\_pwdmnt;

1621

1622 write\_lock(&fs->lock);

1623 old\_pwd = fs->pwd;

1624 old\_pwdmnt = fs->pwdmnt;

1625 fs->pwdmnt = mntget(mnt);

1626 fs->pwd = dget(dentry);

1627 write\_unlock(&fs->lock);

1628

1629 if (old\_pwd) {

1630 dput(old\_pwd);

1631 mntput(old\_pwdmnt);

1632 }

1633 }

1591 /\*

1592 \* Replace the fs->{rootmnt,root} with {mnt,dentry}. Put the old values.

1593 \* It can block. Requires the big lock held.

1594 \*/

1595 void set\_fs\_root(struct fs\_struct \*fs, struct vfsmount \*mnt,

1596 struct dentry \*dentry)

1597 {

1598 struct dentry \*old\_root;

1599 struct vfsmount \*old\_rootmnt;

1600 write\_lock(&fs->lock);

1601 old\_root = fs->root;

1602 old\_rootmnt = fs->rootmnt;

1603 fs->rootmnt = mntget(mnt);

1604 fs->root = dget(dentry);

1605 write\_unlock(&fs->lock);

1606 if (old\_root) {

1607 dput(old\_root);

1608 mntput(old\_rootmnt);

1609 }

1610 }

上述两函数就是设置进程0的根目录和当前工作目录设置为根文件系统。

1. 挂载实际跟文件系统

实际根文件系统的安装是由init内核线程完成的，而init内核线程是由start\_kernel函数中调用kernel\_thread()函数创建的。

start\_kernel();

--------------->

rest\_init();

--------------->

kernel\_thread(init, NULL, CLONE\_FS | CLONE\_SIGHAND);

--------------->

static int init(void \* unused)

--------------->

prepare\_namespace();

--------------->

mount\_root();

①、设置root\_device\_name变量，即要启动的设备文件名称。

②、设置ROOT\_DEV的主设备号和次设备号

③、调用mount\_root()函数

④、调用sys\_mount系统服务例程挂载实际根文件系统。

⑤、用flag=MS\_MOVE调用sys\_mount()移动安装点。

393 /\*

394 \* Prepare the namespace - decide what/where to mount, load ramdisks, etc.

395 \*/

396 void \_\_init prepare\_namespace(void)

397 {

398 int is\_floppy;

399

400 if (root\_delay) {

401 printk(KERN\_INFO "Waiting %dsec before mounting root device...\n",

402 root\_delay);

403 ssleep(root\_delay);

404 }

405

406 md\_run\_setup();

407

408 if (saved\_root\_name[0]) {

409 root\_device\_name = saved\_root\_name;

410 if (!strncmp(root\_device\_name, "mtd", 3)) {

411 mount\_block\_root(root\_device\_name, root\_mountflags);

412 goto out;

413 }

414 ROOT\_DEV = name\_to\_dev\_t(root\_device\_name);

415 if (strncmp(root\_device\_name, "/dev/", 5) == 0)

416 root\_device\_name += 5;

417 }

418

419 is\_floppy = MAJOR(ROOT\_DEV) == FLOPPY\_MAJOR;

420

421 if (initrd\_load())

422 goto out;

423

424 if (is\_floppy && rd\_doload && rd\_load\_disk(0))

425 ROOT\_DEV = Root\_RAM0;

426

427 mount\_root();

428 out:

429 sys\_mount(".", "/", NULL, MS\_MOVE, NULL);

430 sys\_chroot(".");

431 security\_sb\_post\_mountroot();

432 }

①、在create\_dev()中调用sys\_mknod()在rootfs根文件系统中创建设备文件/dev/root

②、调用mount\_block\_root()函数。

366 void \_\_init mount\_root(void)

367 {

368 #ifdef CONFIG\_ROOT\_NFS

369 if (MAJOR(ROOT\_DEV) == UNNAMED\_MAJOR) {

370 if (mount\_nfs\_root())

371 return;

372

373 printk(KERN\_ERR "VFS: Unable to mount root fs via NFS, trying floppy.\n");

374 ROOT\_DEV = Root\_FD0;

375 }

376 #endif

377 #ifdef CONFIG\_BLK\_DEV\_FD

378 if (MAJOR(ROOT\_DEV) == FLOPPY\_MAJOR) {

379 /\* rd\_doload is 2 for a dual initrd/ramload setup \*/

380 if (rd\_doload==2) {

381 if (rd\_load\_disk(1)) {

382 ROOT\_DEV = Root\_RAM1;

383 root\_device\_name = NULL;

384 }

385 } else

386 change\_floppy("root floppy");

387 }

388 #endif

389 create\_dev("/dev/root", ROOT\_DEV);

390 mount\_block\_root("/dev/root", root\_mountflags);

391 }

17 static inline int create\_dev(char \*name, dev\_t dev)

18 {

19 sys\_unlink(name);

20 return sys\_mknod(name, S\_IFBLK|0600, new\_encode\_dev(dev));

21 }

①、分配一个内核缓冲区，调用get\_fs\_names()从root\_fs\_names拷贝，或者从文件系统类型链表（由file\_systems变量保存链表头）获得文件系统类型名。

②、对设备以上一步得来的文件系统名称尝试进行挂载，调用do\_mount\_root()函数。

③、putname(fs\_names);释放缓冲区。

283 void \_\_init mount\_block\_root(char \*name, int flags)

284 {

285 char \*fs\_names = \_\_getname();

286 char \*p;

287 char b[BDEVNAME\_SIZE];

288

289 get\_fs\_names(fs\_names);

290 retry:

291 for (p = fs\_names; \*p; p += strlen(p)+1) {

292 int err = do\_mount\_root(name, p, flags, root\_mount\_data);

293 switch (err) {

294 case 0:

295 goto out;

296 case -EACCES:

297 flags |= MS\_RDONLY;

298 goto retry;

299 case -EINVAL:

300 continue;

301 }

302 /\*

303 \* Allow the user to distinguish between failed sys\_open

304 \* and bad superblock on root device.

305 \*/

306 \_\_bdevname(ROOT\_DEV, b);

307 printk("VFS: Cannot open root device \"%s\" or %s\n",

308 root\_device\_name, b);

309 printk("Please append a correct \"root=\" boot option\n");

310

311 panic("VFS: Unable to mount root fs on %s", b);

312 }

313

314 printk("No filesystem could mount root, tried: ");

315 for (p = fs\_names; \*p; p += strlen(p)+1)

316 printk(" %s", p);

317 printk("\n");

318 panic("VFS: Unable to mount root fs on %s", \_\_bdevname(ROOT\_DEV, b));

319 out:

320 putname(fs\_names);

321 }

下述述代码获取由设备名组成的字符列表并用’\n’隔开，返回字符串长度。

代码中有一个编程技巧导致比较难理解，即其中的for循环。

注意代码中p和page都是一个字符指针。循环终止的条件是该指针不为空，而不是指针指向的内容位字符串结束标志（\*p）。起始条件是p=page-1显然不为空，p和s指向同一地址空间，因为复制字符时，p指针总是>=s所以不会发生错误。

241 static void \_\_init get\_fs\_names(char \*page)

242 {

243 char \*s = page;

244

245 if (root\_fs\_names) {

246 strcpy(page, root\_fs\_names);

247 while (\*s++) {

248 if (s[-1] == ',')

249 s[-1] = '\0';

250 }

251 } else {

252 int len = get\_filesystem\_list(page);

253 char \*p, \*next;

254

255 page[len] = '\0';

256 for (p = page-1; p; p = next) {

257 next = strchr(++p, '\n');

258 if (\*p++ != '\t')

259 continue;

260 while ((\*s++ = \*p++) != '\n')

261 ;

262 s[-1] = '\0';

263 }

264 }

265 \*s = '\0';

266 }

201 int get\_filesystem\_list(char \* buf)

202 {

203 int len = 0;

204 struct file\_system\_type \* tmp;

205

206 read\_lock(&file\_systems\_lock);

207 tmp = file\_systems;

208 while (tmp && len < PAGE\_SIZE - 80) {

209 len += sprintf(buf+len, "%s\t%s\n",

210 (tmp->fs\_flags & FS\_REQUIRES\_DEV) ? "" : "nodev",

211 tmp->name);

212 tmp = tmp->next;

213 }

214 read\_unlock(&file\_systems\_lock);

215 return len;

216 }

①、调用sys\_mount()尝试进行挂载，如果挂载成功执行继续执行2；

②、调用sys\_chdir()改变进程的当前目录。

③、设置ROOT\_DEV为实际根文件系统的设备号。

268 static int \_\_init do\_mount\_root(char \*name, char \*fs, int flags, void \*data)

269 {

270 int err = sys\_mount(name, "/root", fs, flags, data);

271 if (err)

272 return err;

273

274 sys\_chdir("/root");

275 ROOT\_DEV = current->fs->pwdmnt->mnt\_sb->s\_dev;

276 printk("VFS: Mounted root (%s filesystem)%s.\n",

277 current->fs->pwdmnt->mnt\_sb->s\_type->name,

278 current->fs->pwdmnt->mnt\_sb->s\_flags & MS\_RDONLY ?

279 " readonly" : "");

280 return 0;

281 }

sys\_mount()操作和之前挂载普通文件系统的步骤一致，在此大致回顾一下：

①、拷贝参数。

②、调用do\_mount()。从flags中检出mnt\_flag，根据flags做不同的操作，在此肯定是调用do\_new\_mount()。

③、do\_new\_mount()中首先查找安装点‘/root’的相关信息，存放在nameidata结构的变量nd中。然后调用do\_kernel\_mount()获取一个名称位name的文件系统类型的vfsmount结构，该结构中有一个字段指向新分配的super\_block对象。

④、do\_new\_mount()继续调用do\_add\_mount()设置相关数据结构namespace，父子vfsmount，mount\_hashtable。

⑤、do\_mount()调用path\_release释放nd。

1. **卸载文件系统**
2. 从数据结构推测卸载操作要做的工作

在看具体代码之前，同样我们根据数据结构的关系来分析可能要做哪些操作:

①、将卸载的文件系统从mount\_hashtable表中删除。

②、设置父文件系统的vfsmount结构。

③、设置进程namespace数据结构。

④、如果该vfsmount结构所指的super\_block不再被引用，则释放。

⑤、如果安装的文件系统目录下有安装的子文件系统则需等待子文件系统卸载，如果强制卸载，则同子文件系统一起卸载。

⑥、安装时同样要查找安装点，同样存放在nd中，卸载完成之后同样要减少引用计数。

1. 内核代码摘录

①、通过\_\_user\_walk()调用最终得到安装点的信息，并存放在nd中。

②、判断查找到的安装点是否真的是文件系统的安装点。

③、调用check\_mnt()检查要卸载的文件系统是不是已经安装到进程命名空间中。

④、权限检查。

⑤、调用do\_umount()进行卸载。

632 /\*

633 \* Now umount can handle mount points as well as block devices.

634 \* This is important for filesystems which use unnamed block devices.

635 \*

636 \* We now support a flag for forced unmount like the other 'big iron'

637 \* unixes. Our API is identical to OSF/1 to avoid making a mess of AMD

638 \*/

639

640 asmlinkage long sys\_umount(char \_\_user \* name, int flags)

641 {

642 struct nameidata nd;

643 int retval;

644

645 retval = \_\_user\_walk(name, LOOKUP\_FOLLOW, &nd);

646 if (retval)

647 goto out;

648 retval = -EINVAL;

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

650 goto dput\_and\_out;

651 if (!check\_mnt(nd.mnt))

652 goto dput\_and\_out;

653

654 retval = -EPERM;

655 if (!capable(CAP\_SYS\_ADMIN))

656 goto dput\_and\_out;

657

658 retval = do\_umount(nd.mnt, flags);

659 dput\_and\_out:

660 path\_release\_on\_umount(&nd);

661 out:

662 return retval;

663 }

546 static int do\_umount(struct vfsmount \*mnt, int flags)

547 {

548 struct super\_block \*sb = mnt->mnt\_sb;

549 int retval;

550 LIST\_HEAD(umount\_list);

551

552 retval = security\_sb\_umount(mnt, flags);

553 if (retval)

554 return retval;

555

556 /\*

557 \* Allow userspace to request a mountpoint be expired rather than

558 \* unmounting unconditionally. Unmount only happens if:

559 \* (1) the mark is already set (the mark is cleared by mntput())

560 \* (2) the usage count == 1 [parent vfsmount] + 1 [sys\_umount]

561 \*/

562 if (flags & MNT\_EXPIRE) {

563 if (mnt == current->fs->rootmnt ||

564 flags & (MNT\_FORCE | MNT\_DETACH))

565 return -EINVAL;

566

567 if (atomic\_read(&mnt->mnt\_count) != 2)

568 return -EBUSY;

569

570 if (!xchg(&mnt->mnt\_expiry\_mark, 1))

571 return -EAGAIN;

572 }

574 /\*

575 \* If we may have to abort operations to get out of this

576 \* mount, and they will themselves hold resources we must

577 \* allow the fs to do things. In the Unix tradition of

578 \* 'Gee thats tricky lets do it in userspace' the umount\_begin

579 \* might fail to complete on the first run through as other tasks

580 \* must return, and the like. Thats for the mount program to worry

581 \* about for the moment.

582 \*/

583

584 lock\_kernel();

585 if (sb->s\_op->umount\_begin)

586 sb->s\_op->umount\_begin(mnt, flags);

587 unlock\_kernel();

588

589 /\*

590 \* No sense to grab the lock for this test, but test itself looks

591 \* somewhat bogus. Suggestions for better replacement?

592 \* Ho-hum... In principle, we might treat that as umount + switch

593 \* to rootfs. GC would eventually take care of the old vfsmount.

594 \* Actually it makes sense, especially if rootfs would contain a

595 \* /reboot - static binary that would close all descriptors and

596 \* call reboot(9). Then init(8) could umount root and exec /reboot.

597 \*/

598 if (mnt == current->fs->rootmnt && !(flags & MNT\_DETACH)) {

599 /\*

600 \* Special case for "unmounting" root ...

601 \* we just try to remount it readonly.

602 \*/

603 down\_write(&sb->s\_umount);

604 if (!(sb->s\_flags & MS\_RDONLY)) {

605 lock\_kernel();

606 DQUOT\_OFF(sb);

607 retval = do\_remount\_sb(sb, MS\_RDONLY, NULL, 0);

608 unlock\_kernel();

609 }

610 up\_write(&sb->s\_umount);

611 return retval;

612 }

613

614 down\_write(&namespace\_sem);

615 spin\_lock(&vfsmount\_lock);

616 event++;

617

618 retval = -EBUSY;

619 if (flags & MNT\_DETACH || !propagate\_mount\_busy(mnt, 2)) {

620 if (!list\_empty(&mnt->mnt\_list))

621 umount\_tree(mnt, 1, &umount\_list);

622 retval = 0;

623 }

624 spin\_unlock(&vfsmount\_lock);

625 if (retval)

626 security\_sb\_umount\_busy(mnt);

627 up\_write(&namespace\_sem);

628 release\_mounts(&umount\_list);

629 return retval;

630 }

524 void umount\_tree(struct vfsmount \*mnt, int propagate, struct list\_head \*kill)

525 {

526 struct vfsmount \*p;

527

528 for (p = mnt; p; p = next\_mnt(p, mnt))

529 list\_move(&p->mnt\_hash, kill);

530

531 if (propagate)

532 propagate\_umount(kill);

533

534 list\_for\_each\_entry(p, kill, mnt\_hash) {

535 list\_del\_init(&p->mnt\_expire);

536 list\_del\_init(&p->mnt\_list);

537 \_\_touch\_namespace(p->mnt\_namespace);

538 p->mnt\_namespace = NULL;

539 list\_del\_init(&p->mnt\_child);

540 if (p->mnt\_parent != p)

541 p->mnt\_mountpoint->d\_mounted--;

542 change\_mnt\_propagation(p, MS\_PRIVATE);

543 }

544 }

1. **路径名查找**

注：新版内核对路径名查找这块的函数名称和函数调用关系变化很大。

1. 基础知识

①、Linux中目录也是一个文件

②、目录项高速缓存用于加速文件查找

③、要对访问权限做检查

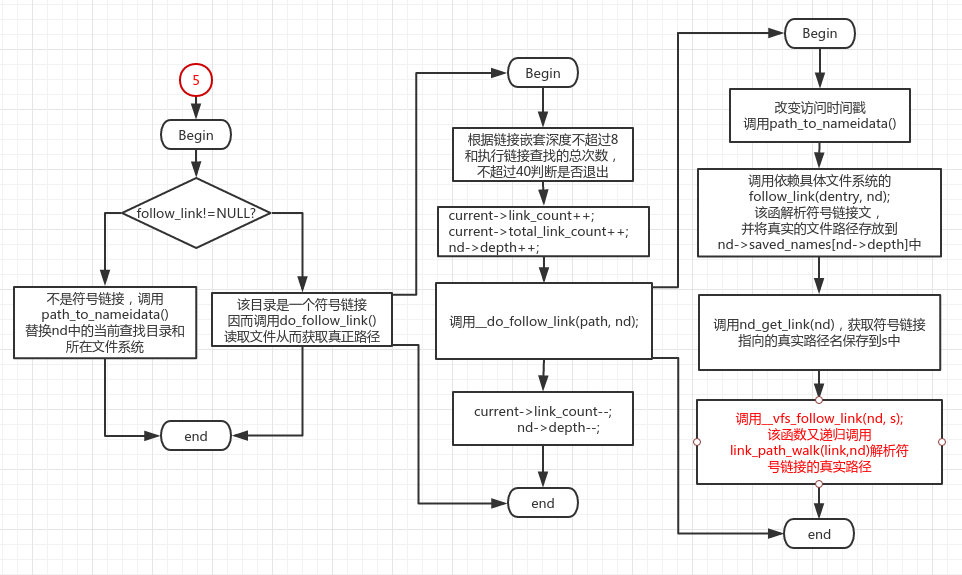
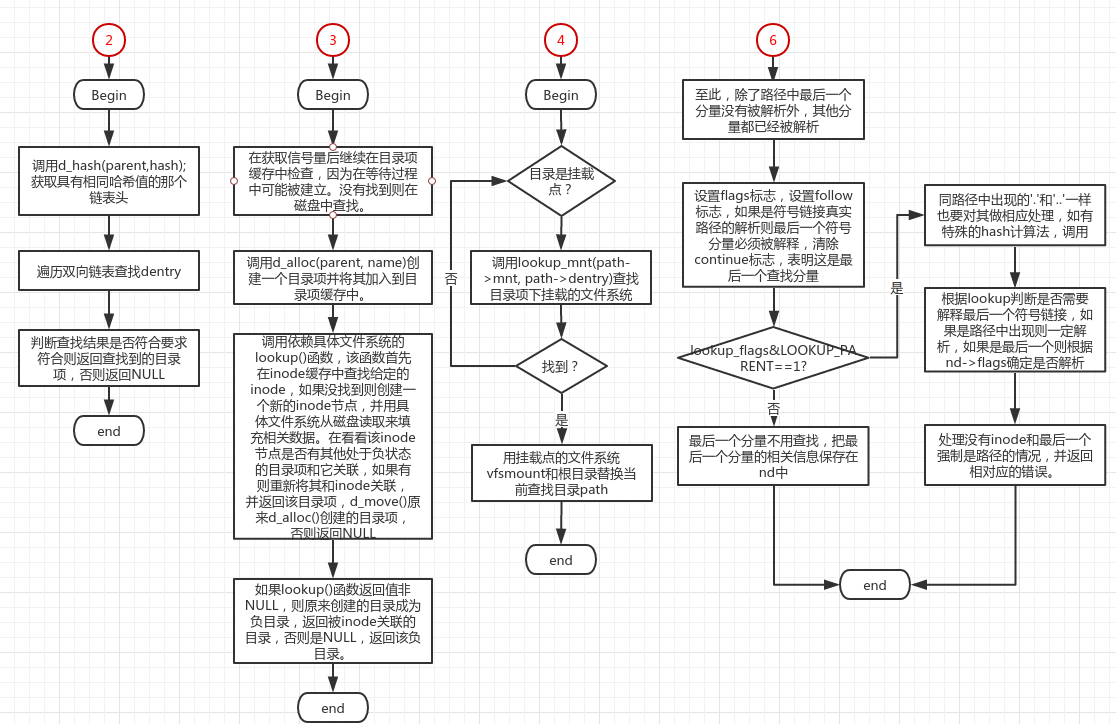
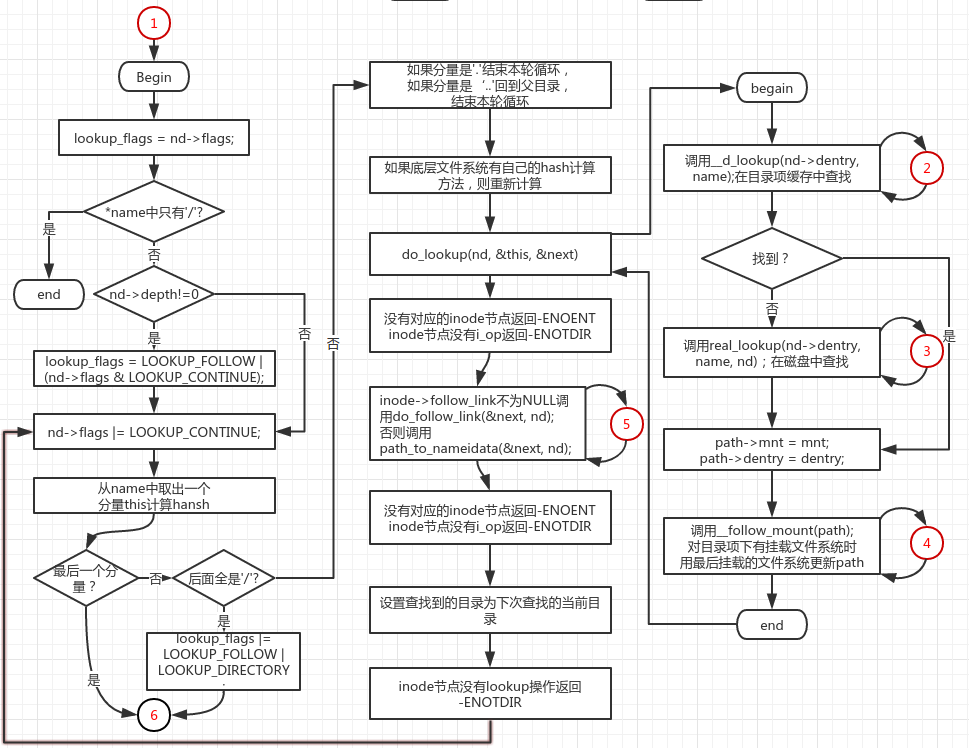
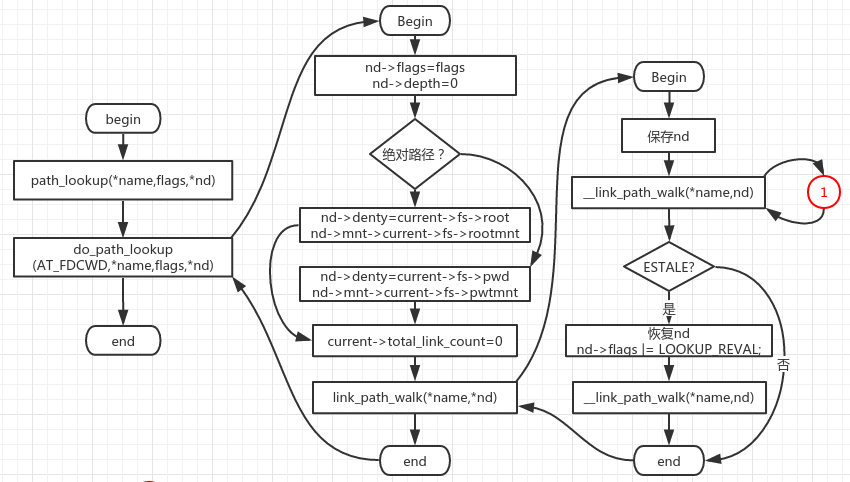
④、若是文件系统安装点，则应在文件系统中查找

⑤、要处理链接文件的相关问题

1. 代码流程图

路径查找工作是递归进行的。下面是整个查找过程的流程图，以供阅读代码时作为参考。

注：图中用红色圈标出来的是所链接的方框中的函数的具体执行过程。从①到⑥是主体过程，其他的都是对某一具体过程（函数）的分析。



1. 内核代码摘录

1147 int fastcall path\_lookup(const char \*name, unsigned int flags,

1148 struct nameidata \*nd)

1149 {

1150 return do\_path\_lookup(AT\_FDCWD, name, flags, nd);

1151 }

根据路径名是绝对路径和AT\_FDCWD标志设置不同的查找起始点。

①、设置nd->mnt和nd->dentry，分别表示查找其实点的目录项和所在的文件系统。为此要增加其引用计数。

②、调用link\_path\_walk()进行查找。

1078 /\* Returns 0 and nd will be valid on success; Retuns error, otherwise. \*/

1079 static int fastcall do\_path\_lookup(int dfd, const char \*name,

1080 unsigned int flags, struct nameidata \*nd)

1081 {

1082 int retval = 0;

1083 int fput\_needed;

1084 struct file \*file;

1085

1086 nd->last\_type = LAST\_ROOT; /\* if there are only slashes... \*/

1087 nd->flags = flags;

1088 nd->depth = 0;

1089

1090 if (\*name=='/') {

1091 read\_lock(&current->fs->lock);

1092 if (current->fs->altroot && !(nd->flags & LOOKUP\_NOALT)) {

1093 nd->mnt = mntget(current->fs->altrootmnt);

1094 nd->dentry = dget(current->fs->altroot);

1095 read\_unlock(&current->fs->lock);

1096 if (\_\_emul\_lookup\_dentry(name,nd))

1097 goto out; /\* found in altroot \*/

1098 read\_lock(&current->fs->lock);

1099 }

1100 nd->mnt = mntget(current->fs->rootmnt);

1101 nd->dentry = dget(current->fs->root);

1102 read\_unlock(&current->fs->lock);

1103 } else if (dfd == AT\_FDCWD) {

1104 read\_lock(&current->fs->lock);

1105 nd->mnt = mntget(current->fs->pwdmnt);

1106 nd->dentry = dget(current->fs->pwd);

1107 read\_unlock(&current->fs->lock);

1108 } else {

1109 struct dentry \*dentry;

1110

1111 file = fget\_light(dfd, &fput\_needed);

1112 retval = -EBADF;

1113 if (!file)

1114 goto out\_fail;

1115

1116 dentry = file->f\_dentry;

1117

1118 retval = -ENOTDIR;

1119 if (!S\_ISDIR(dentry->d\_inode->i\_mode))

1120 goto fput\_fail;

1121

1122 retval = file\_permission(file, MAY\_EXEC);

1123 if (retval)

1124 goto fput\_fail;

1125

1126 nd->mnt = mntget(file->f\_vfsmnt);

1127 nd->dentry = dget(dentry);

1128

1129 fput\_light(file, fput\_needed);

1130 }

1131 current->total\_link\_count = 0;

1132 retval = link\_path\_walk(name, nd);

1133 out:

1134 if (likely(retval == 0)) {

1135 if (unlikely(!audit\_dummy\_context() && nd && nd->dentry &&

1136 nd->dentry->d\_inode))

1137 audit\_inode(name, nd->dentry->d\_inode);

1138 }

1139 out\_fail:

1140 return retval;

1141

1142 fput\_fail:

1143 fput\_light(file, fput\_needed);

1144 goto out\_fail;

1145 }

①、调用\_\_link\_path\_wald()进行查找。

②、如果第一次查找失败，再次进行一次查找，这次不用目录项缓存，而是直接从磁盘中读取。

②、nd使用完成之后需要减少dentry和mnt的引用计数。

974 /\*

975 \* Wrapper to retry pathname resolution whenever the underlying

976 \* file system returns an ESTALE.

977 \*

978 \* Retry the whole path once, forcing real lookup requests

979 \* instead of relying on the dcache.

980 \*/

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

982 {

983 struct nameidata save = \*nd;

984 int result;

985

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

987 dget(save.dentry);

988 mntget(save.mnt);

989

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

991 if (result == -ESTALE) {

992 \*nd = save;

993 dget(nd->dentry);

994 mntget(nd->mnt);

995 nd->flags |= LOOKUP\_REVAL;

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

997 }

998

999 dput(save.dentry);

1000 mntput(save.mnt);

1001

1002 return result;

1003 }

由\_\_link\_path\_walk()进行实际的查找工作，流程如下：

1. 路径名中只有‘/’时所查找的目录项就是当前进程的根目录。
2. 如果nd->depth不为0，则表明程序在解析一个符号链接的真实路径，此时。如果再解析符号链接的真是路径时，最后一个分量如果是符号链接无论如何都要解析。
3. 在for循环中处理路径名，进行查找。
4. 设置LOOKUP\_CONTINUE标志。
5. 权限检查。
6. 计算该分量的hash值，如果底层文件系统有自己的hash计算方法则调用。
7. 判断处理的是否是最后一个分量，是则跳转，到跳转地清除LOOKUP\_CONTINUE标志。
8. 不是. .. 以及最后一个分量。调用do\_lookup()函数查找。
9. inode不存在跳转返回ENOENT错误(查找分量不是最后一个一定是目录，但没有inode，无法继续查找）。
10. 如果i\_op不存在，返回ENOTDIR错误，如果查找结果是链接，调用do\_follow\_link()
11. 检查是否是一个目录项对象（因为不是最后一个查找分量），根据该inode节点是否有lookup()操作，因为如果是一个目录则要调用该方法查找目录下的文件。猜猜在哪里。至此除最后一个分量外都已经查找完毕。
12. 如果i\_op->lookup是NULL则返回ENOTDIR错误。
13. 现在除了最后一个分量，其它的都被分析完毕，如果最后以‘/’结束，最后一个一定是目录或者目录项的符号链接，目录项的符号链接必须解析。所以设置lookup\_flags的LOOKUP\_FOLLOW和LOOKUP\_DIRECTORY标志。

781 /\*

782 \* Name resolution.

783 \* This is the basic name resolution function, turning a pathname into

784 \* the final dentry. We expect 'base' to be positive and a directory.

785 \*

786 \* Returns 0 and nd will have valid dentry and mnt on success.

787 \* Returns error and drops reference to input namei data on failure.

788 \*/

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

790 {

791 struct path next;

792 struct inode \*inode;

793 int err;

794 unsigned int lookup\_flags = nd->flags;

795

796 while (\*name=='/')

797 name++;

798 if (!\*name)

799 goto return\_reval;

800

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

802 if (nd->depth)

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

804

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

806 for(;;) {

807 unsigned long hash;

808 struct qstr this;

809 unsigned int c;

810

811 nd->flags |= LOOKUP\_CONTINUE;

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

813 if (err == -EAGAIN)

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

815 if (err)

816 break;

817

818 this.name = name;

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

820

821 hash = init\_name\_hash();

822 do {

823 name++;

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

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

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

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

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

829

830 /\* remove trailing slashes? \*/

831 if (!c)//是结束符到达最后一个跳转，否则是‘/’继续向下执行

832 goto last\_component;

833 while (\*++name == '/');//去除多余‘/’

834 if (!\*name)//是结束符，则是最后一个分量。

835 goto last\_with\_slashes;

836

837 /\*

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

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

840 \* parent relationships.

841 \*/

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

843 default:

844 break;

845 case 2:

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

847 break;

848 follow\_dotdot(nd);

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

850 /\* fallthrough \*/

851 case 1:

852 continue;

853 }

854 /\*

855 \* See if the low-level filesystem might want

856 \* to use its own hash..

857 \*/

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

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

860 if (err < 0)

861 break;

862 }

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

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

865 if (err)

866 break;

867

868 err = -ENOENT;

869 inode = next.dentry->d\_inode;

870 if (!inode)

871 goto out\_dput;

872 err = -ENOTDIR;

873 if (!inode->i\_op)

874 goto out\_dput;

875

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

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

878 if (err)

879 goto return\_err;

880 err = -ENOENT;

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

882 if (!inode)

883 break;

884 err = -ENOTDIR;

885 if (!inode->i\_op)

886 break;

887 } else

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

889 err = -ENOTDIR;

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

891 break;

892 continue;

893 /\* here ends the main loop \*/

894

895 last\_with\_slashes:

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

897 last\_component:

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

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

900 if (lookup\_flags & LOOKUP\_PARENT)

901 goto lookup\_parent;

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

903 default:

904 break;

905 case 2:

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

907 break;

908 follow\_dotdot(nd);

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

910 /\* fallthrough \*/

911 case 1:

912 goto return\_reval;

913 }

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

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

916 if (err < 0)

917 break;

918 }

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

920 if (err)

921 break;

922 inode = next.dentry->d\_inode;

923 if ((lookup\_flags & LOOKUP\_FOLLOW)

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

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

926 if (err)

927 goto return\_err;

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

929 } else

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

931 err = -ENOENT;

932 if (!inode)

933 break;

934 if (lookup\_flags & LOOKUP\_DIRECTORY) {

935 err = -ENOTDIR;

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

937 break;

938 }

939 goto return\_base;

940 lookup\_parent:

941 nd->last = this;

942 nd->last\_type = LAST\_NORM;

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

944 goto return\_base;

945 if (this.len == 1)

946 nd->last\_type = LAST\_DOT;

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

948 nd->last\_type = LAST\_DOTDOT;

949 else

950 goto return\_base;

951 return\_reval:

952 /\*

953 \* We bypassed the ordinary revalidation routines.

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

955 \*/

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

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

958 err = -ESTALE;

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

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

961 break;

962 }

963 return\_base:

964 return 0;

965 out\_dput:

966 dput\_path(&next, nd);

967 break;

968 }

969 path\_release(nd);

970 return\_err:

971 return err;

972 }

1. 调用\_\_d\_lookup()函数查找目录项。
2. 如果没有找到这样的目录项对象，则调用real\_lookup()从磁盘读入目录并建立索引节点加入到各自的高速缓存中。
3. 处理目录项存在于高速缓存中，但是与其对应的索引节点不存在的情况。
4. 找到之后将设置path的dentry和mnt。
5. 调用\_\_follow\_mount()查看找到的目录项对象是否是挂载点，如果是挂载点要进行替换，以便在挂载的文件系统下查找。

742 /\*

743 \* It's more convoluted than I'd like it to be, but... it's still fairly

744 \* small and for now I'd prefer to have fast path as straight as possible.

745 \* It \_is\_ time-critical.

746 \*/

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

748 struct path \*path)

749 {

750 struct vfsmount \*mnt = nd->mnt;

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

752

753 if (!dentry)

754 goto need\_lookup;

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

756 goto need\_revalidate;

757 done:

758 path->mnt = mnt;

759 path->dentry = dentry;

760 \_\_follow\_mount(path);

761 return 0;

762

763 need\_lookup:

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

765 if (IS\_ERR(dentry))

766 goto fail;

767 goto done;

768

769 need\_revalidate:

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

771 goto done;

772 if (d\_invalidate(dentry))

773 goto done;

774 dput(dentry);

775 goto need\_lookup;

776

777 fail:

778 return PTR\_ERR(dentry);

779 }

1. 调用d\_hash从dentry catch中获取散列链。
2. 遍历散列链进行查找，找到之后获取索继续进行验证是否是要查找的目录，如果不是返回NULL;
3. 如果不在hash中也返回NULL，否则增加引用计数，返回找到的目录项。
4. 调用d\_unhashed()判断是否在缓存中。

1057 struct dentry \* \_\_d\_lookup(struct dentry \* parent, struct qstr \* name)

1058 {

1059 unsigned int len = name->len;

1060 unsigned int hash = name->hash;

1061 const unsigned char \*str = name->name;

1062 struct hlist\_head \*head = d\_hash(parent,hash);

1063 struct dentry \*found = NULL;

1064 struct hlist\_node \*node;

1065 struct dentry \*dentry;

1066

1067 rcu\_read\_lock();

1068

1069 hlist\_for\_each\_entry\_rcu(dentry, node, head, d\_hash) {

1070 struct qstr \*qstr;

1071

1072 if (dentry->d\_name.hash != hash)

1073 continue;

1074 if (dentry->d\_parent != parent)

1075 continue;

1076

1077 spin\_lock(&dentry->d\_lock);

1078

1079 /\*

1080 \* Recheck the dentry after taking the lock - d\_move may have

1081 \* changed things. Don't bother checking the hash because we're

1082 \* about to compare the whole name anyway.

1083 \*/

1084 if (dentry->d\_parent != parent)

1085 goto next;

1086

1087 /\*

1088 \* It is safe to compare names since d\_move() cannot

1089 \* change the qstr (protected by d\_lock).

1090 \*/

1091 qstr = &dentry->d\_name;

1092 if (parent->d\_op && parent->d\_op->d\_compare) {

1093 if (parent->d\_op->d\_compare(parent, qstr, name))

1094 goto next;

1095 } else {

1096 if (qstr->len != len)

1097 goto next;

1098 if (memcmp(qstr->name, str, len))

1099 goto next;

1100 }

1101

1102 if (!d\_unhashed(dentry)) {

1103 atomic\_inc(&dentry->d\_count);

1104 found = dentry;

1105 }

1106 spin\_unlock(&dentry->d\_lock);

1107 break;

1108 next:

1109 spin\_unlock(&dentry->d\_lock);

1110 }

1111 rcu\_read\_unlock();

1112

1113 return found;

1114 }

895 static inline struct hlist\_head \*d\_hash(struct dentry \*parent,

896 unsigned long hash)

897 {

898 hash += ((unsigned long) parent ^ GOLDEN\_RATIO\_PRIME) / L1\_CACHE\_BYTES;

899 hash = hash ^ ((hash ^ GOLDEN\_RATIO\_PRIME) >> D\_HASHBITS);

900 return dentry\_hashtable + (hash & D\_HASHMASK);

901 }

322 /\*\*

323 \* d\_unhashed - is dentry hashed

324 \* @dentry: entry to check

325 \*

326 \* Returns true if the dentry passed is not currently hashed.

327 \*/

328

329 static inline int d\_unhashed(struct dentry \*dentry)

330 {

331 return (dentry->d\_flags & DCACHE\_UNHASHED);

332 }

下述函数对挂载点进行替换，以便在挂载的文件系统上进行查找。

1. 根据目录项的d\_mounted字段检查是否是挂载点。
2. 通过while循环找到最后被挂载的文件系统。调用lookup\_mnt找到挂载的文件系统vfsmount，通过在mount\_hashtable散列表中查找。
3. 增加引用计数防止被卸载。

654 /\* no need for dcache\_lock, as serialization is taken care in

655 \* namespace.c

656 \*/

657 static int \_\_follow\_mount(struct path \*path)

658 {

659 int res = 0;

660 while (d\_mountpoint(path->dentry)) {

661 struct vfsmount \*mounted = lookup\_mnt(path->mnt, path->dentry);

662 if (!mounted)

663 break;

664 dput(path->dentry);

665 if (res)

666 mntput(path->mnt);

667 path->mnt = mounted;

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

669 res = 1;

670 }

671 return res;

672 }

346 static inline int d\_mountpoint(struct dentry \*dentry)

347 {

348 return dentry->d\_mounted;

349 }

128 /\*

129 \* lookup\_mnt increments the ref count before returning

130 \* the vfsmount struct.

131 \*/

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

133 {

134 struct vfsmount \*child\_mnt;

135 spin\_lock(&vfsmount\_lock);

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

137 mntget(child\_mnt);

138 spin\_unlock(&vfsmount\_lock);

139 return child\_mnt;

140 }

103 /\*

104 \* find the first or last mount at @dentry on vfsmount @mnt depending on

105 \* @dir. If @dir is set return the first mount else return the last mount.

106 \*/

107 struct vfsmount \*\_\_lookup\_mnt(struct vfsmount \*mnt, struct dentry \*dentry,

108 int dir)

109 {

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

111 struct list\_head \*tmp = head;

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

113

114 for (;;) {

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

116 p = NULL;

117 if (tmp == head)

118 break;

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

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

121 found = p;

122 break;

123 }

124 }

125 return found;

126 }

1. 当所有的查找都失败后，调用上面的real\_lookup在磁盘中进行查找，并将在查找过程中得到的dentry和inode加入到hash中，以便下次查找时加速查找过程。
2. 该函数依赖底层具体的文件系统。
3. 该函数进行查找时要获取锁，当得到锁之后调用d\_lookup()函数在hash中查找，因为在等待锁的过程中可能已经建立起denty。
4. 如果没有找到则动态创建一个dentry，并进行相应的初始化。
5. 显然如果没有找到该文件（或目录），需要在磁盘中进行查找，这时就依赖于具体的文件系统，每个有效的目录项都对应一个inode。inode有一个i\_op字段指向一个函数列表，在创建这个目录项的时候被具体的文件系统初始化。其中有一个函数i\_op->lookup()实现在磁盘中查找给定的目录dentry，并设置dentry中的某些值。

438 /\*

439 \* This is called when everything else fails, and we actually have

440 \* to go to the low-level filesystem to find out what we should do..

441 \*

442 \* We get the directory semaphore, and after getting that we also

443 \* make sure that nobody added the entry to the dcache in the meantime..

444 \* SMP-safe

445 \*/

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

447 {

448 struct dentry \* result;

449 struct inode \*dir = parent->d\_inode;

450

451 mutex\_lock(&dir->i\_mutex);

452 /\*

453 \* First re-do the cached lookup just in case it was created

454 \* while we waited for the directory semaphore..

455 \*

456 \* FIXME! This could use version numbering or similar to

457 \* avoid unnecessary cache lookups.

458 \*

459 \* The "dcache\_lock" is purely to protect the RCU list walker

460 \* from concurrent renames at this point (we mustn't get false

461 \* negatives from the RCU list walk here, unlike the optimistic

462 \* fast walk).

463 \*

464 \* so doing d\_lookup() (with seqlock), instead of lockfree \_\_d\_lookup

465 \*/

466 result = d\_lookup(parent, name);

467 if (!result) {

468 struct dentry \* dentry = d\_alloc(parent, name);

469 result = ERR\_PTR(-ENOMEM);

470 if (dentry) {

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

472 if (result)

473 dput(dentry);

474 else

475 result = dentry;

476 }

477 mutex\_unlock(&dir->i\_mutex);

478 return result;

479 }

480

481 /\*

482 \* Uhhuh! Nasty case: the cache was re-populated while

483 \* we waited on the semaphore. Need to revalidate.

484 \*/

485 mutex\_unlock(&dir->i\_mutex);

486 if (result->d\_op && result->d\_op->d\_revalidate) {

487 if (!result->d\_op->d\_revalidate(result, nd) && !d\_invalidate(result)) {

488 dput(result);

489 result = ERR\_PTR(-ENOENT);

490 }

491 }

492 return result;

493 }

调用前面的\_\_d\_lookup()实现在hash中查找。

1014 /\*\*

1015 \* d\_lookup - search for a dentry

1016 \* @parent: parent dentry

1017 \* @name: qstr of name we wish to find

1018 \*

1019 \* Searches the children of the parent dentry for the name in question. If

1020 \* the dentry is found its reference count is incremented and the dentry

1021 \* is returned. The caller must use d\_put to free the entry when it has

1022 \* finished using it. %NULL is returned on failure.

1023 \*

1024 \* \_\_d\_lookup is dcache\_lock free. The hash list is protected using RCU.

1025 \* Memory barriers are used while updating and doing lockless traversal.

1026 \* To avoid races with d\_move while rename is happening, d\_lock is used.

1027 \*

1028 \* Overflows in memcmp(), while d\_move, are avoided by keeping the length

1029 \* and name pointer in one structure pointed by d\_qstr.

1030 \*

1031 \* rcu\_read\_lock() and rcu\_read\_unlock() are used to disable preemption while

1032 \* lookup is going on.

1033 \*

1034 \* dentry\_unused list is not updated even if lookup finds the required dentry

1035 \* in there. It is updated in places such as prune\_dcache, shrink\_dcache\_sb,

1036 \* select\_parent and \_\_dget\_locked. This laziness saves lookup from dcache\_lock

1037 \* acquisition.

1038 \*

1039 \* d\_lookup() is protected against the concurrent renames in some unrelated

1040 \* directory using the seqlockt\_t rename\_lock.

1041 \*/

1042

1043 struct dentry \* d\_lookup(struct dentry \* parent, struct qstr \* name)

1044 {

1045 struct dentry \* dentry = NULL;

1046 unsigned long seq;

1047

1048 do {

1049 seq = read\_seqbegin(&rename\_lock);

1050 dentry = \_\_d\_lookup(parent, name);

1051 if (dentry)

1052 break;

1053 } while (read\_seqretry(&rename\_lock, seq));

1054 return dentry;

1055 }

1. 从磁盘查找的过程中建立目录项-ext2

从磁盘中搜索并建立dentry和inode的操作主要由inode节点的操作函数完成。

ext2文件系统的inode操作

376 struct inode\_operations ext2\_dir\_inode\_operations = {

377 .create = ext2\_create,

378 .lookup = ext2\_lookup,

379 .link = ext2\_link,

380 .unlink = ext2\_unlink,

381 .symlink = ext2\_symlink,

382 .mkdir = ext2\_mkdir,

383 .rmdir = ext2\_rmdir,

384 .mknod = ext2\_mknod,

385 .rename = ext2\_rename,

386 #ifdef CONFIG\_EXT2\_FS\_XATTR

387 .setxattr = generic\_setxattr,

388 .getxattr = generic\_getxattr,

389 .listxattr = ext2\_listxattr,

390 .removexattr = generic\_removexattr,

391 #endif

392 .setattr = ext2\_setattr,

393 .permission = ext2\_permission,

394 };

由ext2\_lookup函数完成从磁盘查找文件的工作，并在此过程中建立目录项dentry和inode节点，并将其和dentry关联。

1. 创建一个inode。
2. 如果inode所对应的目录项没有在缓存中，则和新创建的目录项关联，否则删除（更确切应是交换和搜索到的目录项的一些字段）创建的目录项，将其与在缓存中的目录项关联。

51 /\*

52 \* Methods themselves.

53 \*/

54

55 static struct dentry \*ext2\_lookup(struct inode \* dir, struct dentry \*dentry, struct nameidata \*nd)

56 {

57 struct inode \* inode;

58 ino\_t ino;

59

60 if (dentry->d\_name.len > EXT2\_NAME\_LEN)

61 return ERR\_PTR(-ENAMETOOLONG);

62

63 ino = ext2\_inode\_by\_name(dir, dentry);

64 inode = NULL;

65 if (ino) {

66 inode = iget(dir->i\_sb, ino);

67 if (!inode)

68 return ERR\_PTR(-EACCES);

69 }

70 return d\_splice\_alias(inode, dentry);

71 }

968 /\*\*

969 \* d\_splice\_alias - splice a disconnected dentry into the tree if one exists

970 \* @inode: the inode which may have a disconnected dentry

971 \* @dentry: a negative dentry which we want to point to the inode.

972 \*

973 \* If inode is a directory and has a 'disconnected' dentry (i.e. IS\_ROOT and

974 \* DCACHE\_DISCONNECTED), then d\_move that in place of the given dentry

975 \* and return it, else simply d\_add the inode to the dentry and return NULL.

976 \*

977 \* This is needed in the lookup routine of any filesystem that is exportable

978 \* (via knfsd) so that we can build dcache paths to directories effectively.

979 \*

980 \* If a dentry was found and moved, then it is returned. Otherwise NULL

981 \* is returned. This matches the expected return value of ->lookup.

982 \*

983 \*/

984 struct dentry \*d\_splice\_alias(struct inode \*inode, struct dentry \*dentry)

985 {

986 struct dentry \*new = NULL;

987

988 if (inode) {

989 spin\_lock(&dcache\_lock);

990 new = \_\_d\_find\_alias(inode, 1);

991 if (new) {

992 BUG\_ON(!(new->d\_flags & DCACHE\_DISCONNECTED));

993 fsnotify\_d\_instantiate(new, inode);

994 spin\_unlock(&dcache\_lock);

995 security\_d\_instantiate(new, inode);

996 d\_rehash(dentry);

997 d\_move(new, dentry);

998 iput(inode);

999 } else {

1000 /\* d\_instantiate takes dcache\_lock, so we do it by hand \*/

1001 list\_add(&dentry->d\_alias, &inode->i\_dentry);

1002 dentry->d\_inode = inode;

1003 fsnotify\_d\_instantiate(dentry, inode);

1004 spin\_unlock(&dcache\_lock);

1005 security\_d\_instantiate(dentry, inode);

1006 d\_rehash(dentry);

1007 }

1008 } else

1009 d\_add(dentry, inode);

1010 return new;

1011 }

281 /\*\*

282 \* d\_find\_alias - grab a hashed alias of inode

283 \* @inode: inode in question

284 \* @want\_discon: flag, used by d\_splice\_alias, to request

285 \* that only a DISCONNECTED alias be returned.

286 \*

287 \* If inode has a hashed alias, or is a directory and has any alias,

288 \* acquire the reference to alias and return it. Otherwise return NULL.

289 \* Notice that if inode is a directory there can be only one alias and

290 \* it can be unhashed only if it has no children, or if it is the root

291 \* of a filesystem.

292 \*

293 \* If the inode has a DCACHE\_DISCONNECTED alias, then prefer

294 \* any other hashed alias over that one unless @want\_discon is set,

295 \* in which case only return a DCACHE\_DISCONNECTED alias.

296 \*/

297

298 static struct dentry \* \_\_d\_find\_alias(struct inode \*inode, int want\_discon)

299 {

300 struct list\_head \*head, \*next, \*tmp;

301 struct dentry \*alias, \*discon\_alias=NULL;

302

303 head = &inode->i\_dentry;

304 next = inode->i\_dentry.next;

305 while (next != head) {

306 tmp = next;

307 next = tmp->next;

308 prefetch(next);

309 alias = list\_entry(tmp, struct dentry, d\_alias);

310 if (S\_ISDIR(inode->i\_mode) || !d\_unhashed(alias)) {

311 if (alias->d\_flags & DCACHE\_DISCONNECTED)

312 discon\_alias = alias;

313 else if (!want\_discon) {

314 \_\_dget\_locked(alias);

315 return alias;

316 }

317 }

318 }

319 if (discon\_alias)

320 \_\_dget\_locked(discon\_alias);

321 return discon\_alias;

322 }

现在除了最后一个分量，其它的都被分析完毕，如果最后以‘/’结束，最后一个一定是目录或者目录项的符号链接。清除继续查找标志。

假设LOOKUP\_PARENT被置0。则执行下述操作。

1. 如果最后一个分量是’.’则终止执行并返回。
2. 如果是’..’则尝试回到父目录。
3. 否则在高速缓存中查找最后一个分量，调用do\_lookup();
4. 判断是否是一个挂载点。
5. 如果设置了LOOKUP\_FOLLOW标志，且有i\_op->follow\_link()则是一个符号链接。
6. 设置mnt和dentry。
7. dentry的d\_inode为空，路径名指向一个不存在的文件，返回ENOENT。
8. 如果设置了LOOKUP\_DIRECTORY标志，检查是不是一个目录。没有返回ENOTDIR。
9. 返回0，至此查找结束。

当LOOKUP\_PARENT被设置时。

1. 设置nd->last为最后一个分量。
2. 设置设置nd->last\_type。

当路径名称中包含符号链接时：

1. 1、解析到符号链接后要进行替换（要提取文件内容），以便在实际目录查找。
2. 2、要处理符号链接中出现的循环问题，和出现过多符号链接的问题，link\_count和total\_link\_count。
3. 3、调用的函数是do\_follow\_link()

602 /\*

603 \* This limits recursive symlink follows to 8, while

604 \* limiting consecutive symlinks to 40.

605 \*

606 \* Without that kind of total limit, nasty chains of consecutive

607 \* symlinks can cause almost arbitrarily long lookups.

608 \*/

609 static inline int do\_follow\_link(struct path \*path, struct nameidata \*nd)

610 {

611 int err = -ELOOP;

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

613 goto loop;

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

615 goto loop;

616 BUG\_ON(nd->depth >= MAX\_NESTED\_LINKS);

617 cond\_resched();

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

619 if (err)

620 goto loop;

621 current->link\_count++;

622 current->total\_link\_count++;

623 nd->depth++;

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

625 current->link\_count--;

626 nd->depth--;

627 return err;

628 loop:

629 dput\_path(path, nd);

630 path\_release(nd);

631 return err;

632 }

572 static \_\_always\_inline int \_\_do\_follow\_link(struct path \*path, struct nameidata \*nd)

573 {

574 int error;

575 void \*cookie;

576 struct dentry \*dentry = path->dentry;

577

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

579 nd\_set\_link(nd, NULL);

580

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

582 path\_to\_nameidata(path, nd);

583 dget(dentry);

584 }

585 mntget(path->mnt);

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

587 error = PTR\_ERR(cookie);

588 if (!IS\_ERR(cookie)) {

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

590 error = 0;

591 if (s)

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

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

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

595 }

596 dput(dentry);

597 mntput(path->mnt);

598

599 return error;

600 }

516 static \_\_always\_inline int \_\_vfs\_follow\_link(struct nameidata \*nd, const char \*link)

517 {

518 int res = 0;

519 char \*name;

520 if (IS\_ERR(link))

521 goto fail;

522

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

524 path\_release(nd);

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

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

527 goto out;

528 }

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

530 out:

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

532 return res;

533 /\*

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

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

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

537 \*/

538 name = \_\_getname();

539 if (unlikely(!name)) {

540 path\_release(nd);

541 return -ENOMEM;

542 }

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

544 nd->last.name = name;

545 return 0;

546 fail:

547 path\_release(nd);

548 return PTR\_ERR(link);

549 }