析 Linux 中的 VFS 文件

[关闭 [x]](http://www.ibm.com/developerworks/cn/linux/l-vfs/#close)

Ricard Chen，男，感兴趣的领域：Linux 系统内核，BIOS，文件系统，XScale 等。读者可以通过email: [ricard\_chen@yahoo.com](mailto:ricard_chen@yahoo.com) 和他联系。

2005 年 4 月 01 日

根文件系统的挂载过程分析

**宝德技术研究院 李磊**

# rootfs的挂载过程

## rootfs的注册

rootfs的挂载从函数init\_rootfs()开始.这个函数只是做了简单的注册.

**fs/ramfs/inode.c**

int \_\_init init\_rootfs(void)

{

return register\_filesystem(&rootfs\_fs\_type);

}

函数的具体实现如下:

**fs/filesystems.c**

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

{

int res = 0;

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

if (!fs)

return -EINVAL;

if (fs->next)

return -EBUSY;

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

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

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

if (\*p)

res = -EBUSY;

else

\*p = fs;

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

return res;

}

过程非常简单,首先初始化fs->fs\_super, 再调用find\_filesystem检测系统中是否有相同名称的文件系统,如果没有,就将fs插入到系统中,如果有,则返回错误值.

**fs/filesystems.c**

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

{

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

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

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

break;

return p;

}

全局变量file\_systems是维护着系统的文件系统类型的单链表头,这个链表的成员是file\_system\_type类型的指针,每系统注册一个新的文件系统都是尾插这个链表,调用者需要一个成员的地址才能将一个新的文件系统注册到系统中,故此函数的返回值是一个file\_system\_type类型的二级指针.

## rootfs的挂载

rootfs的挂载是通过init\_mount\_tree()这个函数完成的

**fs/namespace.c**

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

{

struct vfsmount \*mnt;

struct namespace \*namespace;

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

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

if (IS\_ERR(mnt))

panic("Can't create rootfs");

通过以上代码就为do\_kern\_mount分配了一个vfsmount的指针mnt.

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

if (!namespace)

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

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

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

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

namespace->event = 0;

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

namespace->root = mnt;

mnt->mnt\_namespace = namespace;

init\_task.namespace = namespace;

read\_lock(&tasklist\_lock);

do\_each\_thread(g, p) {

get\_namespace(namespace);

p->namespace = namespace;

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

read\_unlock(&tasklist\_lock);

set\_fs\_pwd(current->fs,namespace->root,

namespace->root->mnt\_root);

set\_fs\_root(current->fs,namespace->root,

namespace->root->mnt\_root);

}

这段代码主要作用是分配一个namespace,把分配好的namespace和mnt关联起来, 再将current->fs下的pwd,pwdmnt换成namespace->root和namespace->root->mnt\_root. 将current->fs下的root,rootmnt换成namespace->root和namespace->root->mnt\_root.

在rootfs挂载中,处理文件系统的是通过函数do\_kern\_mount完成的.

**fs/super.c**

struct vfsmount \*

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

\*data)

{

//查找是否有匹配的文件系统,如果没有就加载相关模块

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

struct vfsmount \*mnt;

if (!type)

return ERR\_PTR(-ENODEV);

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

put\_filesystem(type);

return mnt;

}

这个函数首先调用get\_fs\_type在系统中找出名称为fstype的file\_system\_type指针.再将file\_system\_type指针传入vfs\_kern\_mount返回指向vfsmount的指针.

struct file\_system\_type \*get\_fs\_type(const char \*name)

{

struct file\_system\_type \*fs;

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

fs = \*(find\_filesystem(name));

//try\_module\_get成功返回1,表示模块处于活动状态

if (fs && !try\_module\_get(fs->owner))

//如果fs已经在内核中注册过了,并且模块处于非活动状态

fs = NULL;

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

if (!fs && (request\_module("%s", name) == 0)) {

//如果fs为NULL, 并且插入模块成功

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

//再次搜索fs

fs = \*(find\_filesystem(name));

if (fs && !try\_module\_get(fs->owner))

fs = NULL;

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

}

return fs;

}

真正的挂载过程由函数vfs\_kern\_mount实现,去掉一部分我们目前不太关心的代码:

**fs/super.c**

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;

//分配并初始化一个vfsmount

mnt = alloc\_vfsmnt(name);

if (!mnt)

goto out;

if (data) {

...

}

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

if (error < 0)

goto out\_free\_secdata;

...

//fs挂载前的mountpoint就指向自己的根目录

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

//fs挂载前parent指向自己

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);

}

此函数的工作有:

1.分配并初始化一个vfsmount指针.

2.调用特定文件系统的获取superblock的方法,分配一个superblock,并将superblock和vfsmount关联起来

3.处理根目录及父vfsmount关系的问题.

第一点由函数alloc\_vfsmnt完成:

**fs/namespace.c**

struct vfsmount \*alloc\_vfsmnt(const char \*name)

{

struct vfsmount \*mnt = kmem\_cache\_alloc(mnt\_cache, GFP\_KERNEL);

if (mnt) {

memset(mnt, 0, sizeof(struct vfsmount));

atomic\_set(&mnt->mnt\_count, 1);

INIT\_LIST\_HEAD(&mnt->mnt\_hash);

INIT\_LIST\_HEAD(&mnt->mnt\_child);

INIT\_LIST\_HEAD(&mnt->mnt\_mounts);

INIT\_LIST\_HEAD(&mnt->mnt\_list);

INIT\_LIST\_HEAD(&mnt->mnt\_expire);

INIT\_LIST\_HEAD(&mnt->mnt\_share);

INIT\_LIST\_HEAD(&mnt->mnt\_slave\_list);

INIT\_LIST\_HEAD(&mnt->mnt\_slave);

if (name) {

int size = strlen(name) + 1;

char \*newname = kmalloc(size, GFP\_KERNEL);

if (newname) {

memcpy(newname, name, size);

mnt->mnt\_devname = newname;

}

}

}

return mnt;

}

首先通过kmem\_cache\_alloc分配一个vfsmount指针.然后对自己的引用计数置1,接着初始化所有与链表相关的成员.最后给mnt中的挂载设备赋值.

第2点通过回调函数完成,这里的回调函数是rootfs\_get\_sb.

**fs/ramfs/inode.c**

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

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

{

return get\_sb\_nodev(fs\_type,flags|MS\_NOUSER,data,

ramfs\_fill\_super,mnt);

}

get\_sb\_nodev的调用过程有些复杂, 大体过程如图1所示.



图1

**fs/super.c**

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

int flags, void \*data,

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

struct vfsmount \*mnt)

{

int error;

struct super\_block \*s = sget(fs\_type, NULL, set\_anon\_super, NULL);

if (IS\_ERR(s))

return PTR\_ERR(s);

s->s\_flags = flags;

//不同的fs调用不同的方法填充sb,如op,inode的赋值

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

if (error) {

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

deactivate\_super(s);

return error;

}

//激活sb

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

//设置fs的sb和rootdir,关联sb和vfsmount

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

}

这个函数首先通过sget获取一个super block, 接着通过回调函数fill\_super填充super block,当前fill\_super为ramfs\_fill\_super.最后将vfsmount和处理完毕的super block关联起来.

下面分析sget这个函数,由get\_sb\_nodev传入的test函数指针为NULL,我们省略部分目前无需关心的代码.

**fs/super.c**

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

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

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

void \*data)

{

struct super\_block \*s = NULL;

struct list\_head \*p;

int err;

retry:

spin\_lock(&sb\_lock);

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

...

}

if (!s) {

spin\_unlock(&sb\_lock);

//分配并初始化一个sb

s = alloc\_super(type);

if (!s)

return ERR\_PTR(-ENOMEM);

goto retry;

}

err = set(s, data);

if (err) {

spin\_unlock(&sb\_lock);

destroy\_super(s);

return ERR\_PTR(err);

}

s->s\_type = type;

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

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

//把sb头插到file\_system\_type中的sb链表中

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

spin\_unlock(&sb\_lock);

get\_filesystem(type);

return s;

}

首先, 函数调用了alloc\_super来分配一个初始化过的super block, 然后调用回调函数set, 当前情况是set\_anon\_super, 来分配一个设备标识符.再给super block中的用来记录文件系统类型的成员s\_type赋值. s用来标识文件系统了名称, 通过strlcpy将其赋值.全局变量super\_blocks是一个链表头,用于管理系统中所有的super block, 所以当一个super block分配完成后也需要将插入全局变量super\_blocks的链表中.另外,在系统中可能存在多个设备使用同一文件系统,文件系统中就专门有个成员fs\_supers维护这样一个链表,链表的元素都是使用同一文件系统的super block.在super block初始化完成后也需要将自己加入file\_system\_type的维护super block的链表中.

分配super\_block的函数是alloc\_super,下面代码过滤了部分目前无需关系的部分:

**fs/super.c**

static struct super\_block \*alloc\_super(struct file\_system\_type \*type)

{

struct super\_block \*s = kzalloc(sizeof(struct super\_block), GFP\_USER);

static struct super\_operations default\_op;

if (s) {

if (security\_sb\_alloc(s)) {

...

}

INIT\_LIST\_HEAD(&s->s\_dirty);

INIT\_LIST\_HEAD(&s->s\_io);

INIT\_LIST\_HEAD(&s->s\_files);

INIT\_LIST\_HEAD(&s->s\_instances);

INIT\_HLIST\_HEAD(&s->s\_anon);

INIT\_LIST\_HEAD(&s->s\_inodes);

init\_rwsem(&s->s\_umount);

mutex\_init(&s->s\_lock);

...

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

s->s\_count = S\_BIAS;

atomic\_set(&s->s\_active, 1);

mutex\_init(&s->s\_vfs\_rename\_mutex);

mutex\_init(&s->s\_dquot.dqio\_mutex);

mutex\_init(&s->s\_dquot.dqonoff\_mutex);

init\_rwsem(&s->s\_dquot.dqptr\_sem);

init\_waitqueue\_head(&s->s\_wait\_unfrozen);

s->s\_maxbytes = MAX\_NON\_LFS;

s->dq\_op = sb\_dquot\_ops;

s->s\_qcop = sb\_quotactl\_ops;

s->s\_op = &default\_op;

s->s\_time\_gran = 1000000000;

}

out:

return s;

}

这个函数的实现比较简单, 首先分配一个struct super\_block指针,然后对super bock的链表相关成员初始化,再对相关的互斥锁信号量初始化,初始化磁盘限额的操作函数,指定默认的super block操作函数,设定时间戳粒度(以ns为单位).super block有2个计数s\_active, s\_count.其中s\_active用来记录当前活动的super block,也就是被mount多少次, s\_active大于0表示super block的成员s\_root有效. s\_count用来记录被引用的次数, s\_count大于0,成员s\_root不一定有效.这里由于历史原因只要s\_active大于0, 不论具体多少, s\_count就从S\_BIAS开始计数.在2010.3.22以后的版本中,这种蹩脚的计数方式就被剔除了.

函数set\_anon\_super,主要是获取设备号,这里的设备号如何获取的,笔者目前不清楚,待以后有空研究idr机制再补充.

**fs/super.c**

int set\_anon\_super(struct super\_block \*s, void \*data)

{

int dev;

int error;

retry:

if (idr\_pre\_get(&unnamed\_dev\_idr, GFP\_ATOMIC) == 0)

return -ENOMEM;

spin\_lock(&unnamed\_dev\_lock);

error = idr\_get\_new(&unnamed\_dev\_idr, NULL, &dev);

spin\_unlock(&unnamed\_dev\_lock);

if (error == -EAGAIN)

/\* We raced and lost with another CPU. \*/

goto retry;

else if (error)

return -EAGAIN;

if ((dev & MAX\_ID\_MASK) == (1 << MINORBITS)) {

spin\_lock(&unnamed\_dev\_lock);

idr\_remove(&unnamed\_dev\_idr, dev);

spin\_unlock(&unnamed\_dev\_lock);

return -EMFILE;

}

s->s\_dev = MKDEV(0, dev & MINORMASK);

return 0;

}

在sget之后就需要调用回调函数填充super block, 对于rootfs, 用于填充super block的函数是ramfs\_fill\_super:

**fs/ramfs/inode.c**

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

silent)

{

struct inode \* inode;

struct dentry \* root;

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

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

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

sb->s\_magic = RAMFS\_MAGIC;

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

sb->s\_time\_gran = 1;

//建立根目录的inode

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

if (!inode)

return -ENOMEM;

//建立根目录的dentry

root = d\_alloc\_root(inode);

if (!root) {

iput(inode);

return -ENOMEM;

}

//sb的s\_root指向刚建立的根目录

sb->s\_root = root;

return 0;

}

这个函数针对ramfs对super block的文件系统相关的成员进行赋值.分配了inode和dentry, 并把inode和dentry和super block关联起来.

ramfs获取inode是通过调用ramfs\_get\_inode实现的:

**fs/ramfs/inode.c**

struct inode \*ramfs\_get\_inode(struct super\_block \*sb, int mode, dev\_t dev)

{

struct inode \* inode = new\_inode(sb);

if (inode) {

//文件类型赋值

inode->i\_mode = mode;

inode->i\_uid = current->fsuid;

inode->i\_gid = current->fsgid;

inode->i\_blksize = PAGE\_CACHE\_SIZE;

inode->i\_blocks = 0;

inode->i\_mapping->a\_ops = &ramfs\_aops;

inode->i\_mapping->backing\_dev\_info=&ramfs\_backing\_dev\_info;

inode->i\_atime=inode->i\_mtime=inode->i\_ctime = CURRENT\_TIME;

switch (mode & S\_IFMT) {

default:

//特殊文件

init\_special\_inode(inode, mode, dev);

break;

case S\_IFREG:

//普通文件

inode->i\_op = &ramfs\_file\_inode\_operations;

inode->i\_fop = &ramfs\_file\_operations;

break;

case S\_IFDIR:

//目录

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

inode->i\_fop = &simple\_dir\_operations;

inode->i\_nlink++;

break;

case S\_IFLNK:

//符号链接

inode->i\_op = &page\_symlink\_inode\_operations;

break;

}

}

return inode;

}

这个函数首先分配一个inode, 如果分配成功就先把inode中文件系统无关的成员赋值,如inode的权限, uid, gid, blocksize等.然后判断inode代表文件的类型,给不同类型的文件指定不同的inode操作函数和文件操作函数.

下面我们再来看看函数new\_inode是如何获得一个inode的.

struct inode \*new\_inode(struct super\_block \*sb)

{

static unsigned long last\_ino;

struct inode \* inode;

spin\_lock\_prefetch(&inode\_lock);

inode = alloc\_inode(sb);

if (inode) {

spin\_lock(&inode\_lock);

inodes\_stat.nr\_inodes++;

list\_add(&inode->i\_list, &inode\_in\_use);

//将刚刚分配的inode加入sb的inode链表中

list\_add(&inode->i\_sb\_list, &sb->s\_inodes);

inode->i\_ino = ++last\_ino;

inode->i\_state = 0;

spin\_unlock(&inode\_lock);

}

return inode;

}

这个函数首先调用alloc\_inode分配inode, 如果分配成功,用于统计系统inode信息的全局变量inodes\_stat的成员nr\_inodes就会自加1.当前分配的这个inode是ramfs根目录的inode, 马上就要使用,故把此inode加入链表inode\_in\_use.这个链表维护的是系统中正在使用的inode.接着要把inode加入相关联的super block的用与记录inode的链表中. 这里要注意的是last\_ino是用static标记的, 函数退出后局部变量的值不会清除.rootfs的inode->i\_ino从3开始,前2个inode给了sysfs.

到最后分配inode的函数就是alloc\_inode了:

**fs/inode.c**

static struct inode \*alloc\_inode(struct super\_block \*sb)

{

static const struct address\_space\_operations empty\_aops;

static struct inode\_operations empty\_iops;

static const struct file\_operations empty\_fops;

struct inode \*inode;

if (sb->s\_op->alloc\_inode)

inode = sb->s\_op->alloc\_inode(sb);

else

inode = (struct inode \*) kmem\_cache\_alloc(inode\_cachep, SLAB\_KERNEL);

if (inode) {

struct address\_space \* const mapping = &inode->i\_data;

//inode的i\_sb指向sb

inode->i\_sb = sb;

inode->i\_blkbits = sb->s\_blocksize\_bits;

inode->i\_flags = 0;

atomic\_set(&inode->i\_count, 1);

inode->i\_op = &empty\_iops;

inode->i\_fop = &empty\_fops;

inode->i\_nlink = 1;

atomic\_set(&inode->i\_writecount, 0);

inode->i\_size = 0;

inode->i\_blocks = 0;

inode->i\_bytes = 0;

inode->i\_generation = 0;

#ifdef CONFIG\_QUOTA

memset(&inode->i\_dquot, 0, sizeof(inode->i\_dquot));

#endif

inode->i\_pipe = NULL;

inode->i\_bdev = NULL;

inode->i\_cdev = NULL;

inode->i\_rdev = 0;

inode->i\_security = NULL;

inode->dirtied\_when = 0;

if (security\_inode\_alloc(inode)) {

...

}

mapping->a\_ops = &empty\_aops;

mapping->host = inode;

mapping->flags = 0;

mapping\_set\_gfp\_mask(mapping, GFP\_HIGHUSER);

mapping->assoc\_mapping = NULL;

mapping->backing\_dev\_info = &default\_backing\_dev\_info;

/\*

\* If the block\_device provides a backing\_dev\_info for client

\* inodes then use that. Otherwise the inode share the bdev's

\* backing\_dev\_info.

\*/

if (sb->s\_bdev) {

struct backing\_dev\_info \*bdi;

bdi = sb->s\_bdev->bd\_inode\_backing\_dev\_info;

if (!bdi)

bdi = sb->s\_bdev->bd\_inode->i\_mapping->backing\_dev\_info;

mapping->backing\_dev\_info = bdi;

}

memset(&inode->u, 0, sizeof(inode->u));

inode->i\_mapping = mapping;

}

return inode;

}

这个函数首先判断super block中有没有分配inode的方法,如果有就用super block提供的,如果没有就用kmem\_cache\_alloc分配.接着对inode成员赋值,这里分配的是一个文件系统无关的inode, 故很多成员置0, 操作函数也是一些空操作. mapping相关的部分笔者目前还没研究,以后会补充.根目录的目录项的建立过程和inode的建立过程差不多,这里不再赘述.根目录的dentry分配完成后需要和根目录的inode和super block关联起来.

函数get\_sb\_nodev的最后将super bloc标记成活动的,然后将mnt的mnt\_sb指向分配完毕的super block, mnt的mnt\_root指向根目录的dentry.

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

{

mnt->mnt\_sb = sb;

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

return 0;

}

# 实际根文件系统的挂载

实际的根文件系统一般是内核启动后挂载的initramfs(2.4的内核用的是initrd).对于PC来说,这是一个临时的根文件系统.由于存储设备及文件系统多种多样,内核不能把所有的存储设备及文件系统加入内核,这样会大大增加内核的大小.内核开发者就提出使用initrd, initrd中保存实际根文件系统中将要使用的存储设备驱动.内核起来后就先挂载initrd, 后续文件系统的挂载由initrd中的init去执行.再后来基于文件系统层的initramfs就取代了基于块设备层的initrd了.但是在很多嵌入式中,initramfs就是最终的根文件系统.

initramfs的使用请查考文档--- initramfs的使用.docx.经过上一章节,根文件系统已经构建起来了,挂载目录树,super block, 根目录项等一些重要的数据结构已经建立.接下来的工作就是将initramfs中的内容导入到根文件系统中.这部分的工作在fs/main.c中的init函数中完成.

**init/main.c**

static int init(void \* unused)

{

...

/\*

\* Do this before initcalls, because some drivers want to access

\* firmware files.

\*/

populate\_rootfs();

do\_basic\_setup();

/\*

\* check if there is an early userspace init. If yes, let it do

\* all

\* the work

\*/

if (!ramdisk\_execute\_command)

ramdisk\_execute\_command = "/init";

//sys\_access返回0, 表示执行成功

if (sys\_access((const char \_\_user \*) ramdisk\_execute\_command,

0) != 0) {

ramdisk\_execute\_command = NULL;

prepare\_namespace();

}

/\*

\* Ok, we have completed the initial bootup, and

\* we're essentially up and running. Get rid of the

\* initmem segments and start the user-mode stuff..

\*/

free\_initmem();

unlock\_kernel();

mark\_rodata\_ro();

system\_state = SYSTEM\_RUNNING;

numa\_default\_policy();

if (sys\_open((const char \_\_user \*) "/dev/console", O\_RDWR, 0) <

0)

printk(KERN\_WARNING "Warning: unable to open an initial

console.\n");

(void) sys\_dup(0);

(void) sys\_dup(0);

if (ramdisk\_execute\_command) {

run\_init\_process(ramdisk\_execute\_command);

printk(KERN\_WARNING "Failed to execute %s\n",

ramdisk\_execute\_command);

}

/\*

\* We try each of these until one succeeds.

\*

\* The Bourne shell can be used instead of init if we are

\* trying to recover a really broken machine.

\*/

if (execute\_command) {

run\_init\_process(execute\_command);

printk(KERN\_WARNING "Failed to execute %s. Attempting "

"defaults...\n", execute\_command);

}

run\_init\_process("/sbin/init");

run\_init\_process("/etc/init");

run\_init\_process("/bin/init");

run\_init\_process("/bin/sh");

panic("No init found. Try passing init= option to kernel.");

}

加压initramfs的动作在函数populate\_rootfs中完成.bootloader的启动参数中没有指定rdinit的情况下, ramdisk\_execute\_command是空的.这时候ramdisk\_execute\_command被指定为/init

**init/initramfs.c**

1 void \_\_init populate\_rootfs(void)

2 {

3 char \*err = unpack\_to\_rootfs(\_\_initramfs\_start,

4 \_\_initramfs\_end - \_\_initramfs\_start, 0);

5

6 if (err)

7 panic(err);

8 #ifdef CONFIG\_BLK\_DEV\_INITRD

9 if (initrd\_start) {

10 #ifdef CONFIG\_BLK\_DEV\_RAM

11 int fd;

12 printk(KERN\_INFO "checking if image is initramfs...");

13 //判断加载的内容是不是cpio-initrd

14 err = unpack\_to\_rootfs((char \*)initrd\_start,

15 initrd\_end - initrd\_start, 1);

16 if (!err) {

17 //如果是cpio-initrd, 就将内容释放到rootfs中

18 printk(" it is\n");

19 unpack\_to\_rootfs((char \*)initrd\_start,

20 initrd\_end - initrd\_start, 0);

21 free\_initrd();

22 return;

23 }

24 //如果不是,就认为这是一个image-initrd,将其内容保存到

//initrd.imag中

25 printk("it isn't (%s); looks like an initrd\n", err);

26 fd = sys\_open("/initrd.image", O\_WRONLY|O\_CREAT, 0700);

27 if (fd >= 0) {

28 sys\_write(fd, (char \*)initrd\_start,

29 initrd\_end - initrd\_start);

30 sys\_close(fd);

31 free\_initrd();

32 }

33 #else

34 //如果不支持ramdisk,就一定不支持image-initrd了

35 printk(KERN\_INFO "Unpacking initramfs...");

36 err = unpack\_to\_rootfs((char \*)initrd\_start,

37 initrd\_end - initrd\_start, 0);

38 if (err)

39 panic(err);

40 printk(" done\n");

41 free\_initrd();

42 #endif

在没有指定initramfs的情况下,内核编译时会产生一个默认的initramfs,并把这个initramfs放入内核镜像的\_\_initramfs\_start至\_\_initramfs\_end的段中.函数第3行就是将这个默认的initramfs解压.函数static char \* \_\_init unpack\_to\_rootfs(char \*buf, unsigned len, int check\_only)有两个功能.当check\_only为1时,就检查buf中的内容是不是一个通过cpio生成的ramfs.当check\_only为0时,就进行压缩包的释放.如果initrd\_start不为空,表示bootloader给内核传入了initramfs/initrd.第14行,判断initrd\_start开始的地址中保存的是否是一个cpio-initrd.如果是一个cpio-initrd, 在19行就将cpio解压到根目录中,然后将存放cpio包的地址释放.如果bootloader传入的是一个cpio-initrd,在26行就会在根目录下创建一个initrd.img.同时将initrd\_start至initrd\_end中的内容写入这个文件,最后存放cpio包的地址也会被释放.经过populate\_rootfs, 在根目录的实际内容就是在/boot/grub/grub.conf中记录的那个initrd.

将这个initrd解压后可以发现一个名为init的脚本文件,这个文件是一个nash脚本.这个脚本除了创建一些内存文件系统,加载一些ko文件,我们关系的就是下面这部分

73 echo Creating root device.

74 mkrootdev -t ext3 -o defaults,ro sda2

75 echo Mounting root filesystem.

76 mount /sysroot

77 echo Setting up other filesystems.

78 setuproot

79 echo Switching to new root and running init.

80 switchroot

以上代码就将最终的根文件系统挂载起来.

如果initramrd不是cpio压缩而是img形式的, 这种情况暂时不做讨论.