第八章　虚拟文件系统

作为一个最著名的自由软件，Linux 确实名不虚传，几乎处处体现了“自由”，你可以编译适合自己系统要求的内核，或者轻松添加别人开发的新的模块。只要你有实力，你还可以自己写一个新的 Linux 支持的文件系统。写一个新的文件系统虽然是一个“耸人听闻”的事，但 Linux 确实有这样一个特点，就是可以很方便地支持别的操作系统的文件系统，比如 Windows 的文件系统就被 Linux 所支持。Linux 不仅支持多种文件系统，而且还支持这些文件系统相互之间进行访问，这一切都要归功于神奇的虚拟文件系统。

8.1　概述

虚拟文件系统又称虚拟文件系统转换（Virual Filesystem Switch ，简称 VFS）。说它虚拟，是因为它所有的数据结构都是在运行以后才建立，并在卸载时删除，而在磁盘上并没有存储这些数据结构。如果只有 VFS，系统是无法工作的，因为它的这些数据结构不能凭空而来，只有与实际的文件系统，如 Ext2、Minix、MSDOS、VFAT 等相结合，才能开始工作，所以 VFS 并不是一个真正的文件系统。与 VFS 相对应，我们称 Ext2、Minix、MSDOS 等为具体文件系统。

1．虚拟文件系统的作用

在第一章 Linux 内核结构一节中，我们把 VFS 称为内核的一个子系统，其他子系统只与 VFS 打交道，而并不与具体文件系统发生联系。对具体文件系统来说，VFS 是一个管理者，而对内核的其他子系统来说，VFS 是它们与具体文件系统的一个接口，整个 Linux 中文件系统的逻辑关系如图 8.1 所示。

VFS 提供一个统一的接口（实际上就是 file_operatoin 数据结构，稍后介绍），一个具体文件系统要想被 Linux 支持，就必须按照这个接口编写自己的操作函数，而将自己的细节对内核其他子系统隐藏起来。因而，对内核其他子系统以及运行在操作系统之上的用户程序而言，所有的文件系统都是一样的。实际上，要支持一个新的文件系统，主要任务就是编写这些接口函数。

概括说来，VFS 主要有以下几个作用。

（1）对具体文件系统的数据结构进行抽象，以一种统一的数据结构进行管理。

（2）接受用户层的系统调用，例如 write、open、stat、link 等。

（3）支持多种具体文件系统之间相互访问。

（4）接受内核其他子系统的操作请求，特别是内存管理子系统。

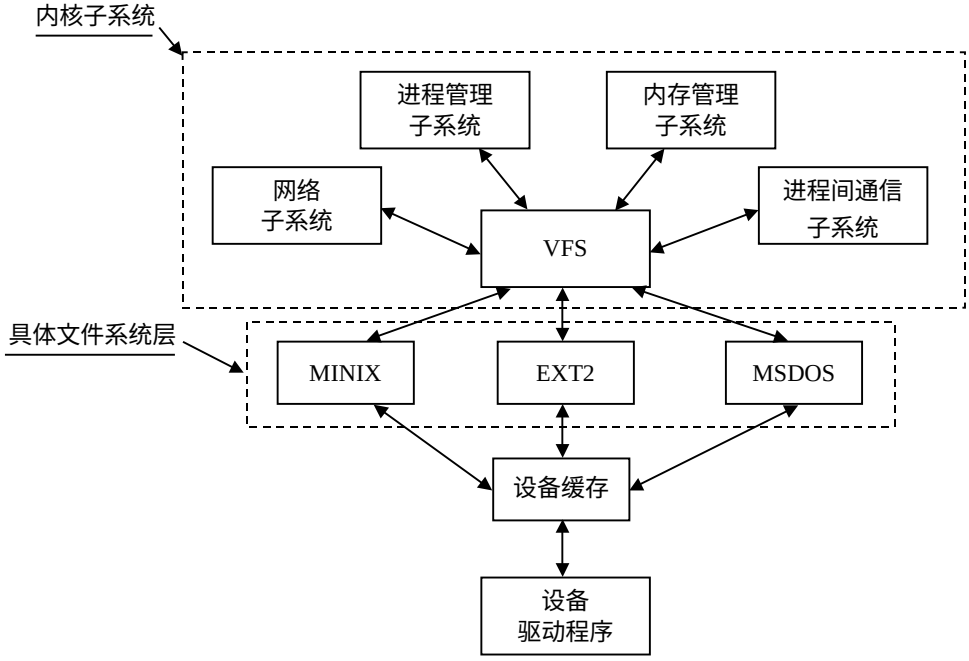

图 8.1　Linux 中文件系统的逻辑关系示意图

通过 VFS，Linux 可以支持很多种具体文件系统，表 8.1 是 Linux 支持的部分具体文件系统。

表 8.1　　Linux 支持的部分文件系统

文件系统	描述
Minix	Linux 最早支持的文件系统。主要缺点是最大 64MB 的磁盘分区和最长 14 个字符的文件名称的限制
Ext	第 1 个 Linux 专用的文件系统，支持 2GB 磁盘分区，255 字符的文件名称，但性能有问题
Xiafs	在 Minix 基础上发展起来，克服了 Minix 的主要缺点。但很快被更完善的文件系统取代
Ext2	当前实际上的 Linux 标准文件系统。性能强大，易扩充，可移植
Ext3	日志文件系统。Ext3 文件系统是对稳定的 Ext2 文件系统的改进
System V	UNIX 早期支持的文件系统，也有与 Minix 同样的限制
NFS	网络文件系统。使得用户可以像访问本地文件一样访问远程主机上的文件
ISO 9660	光盘使用的文件系统
/proc	一个反映内核运行情况的虚的文件系统，并不实际存在于磁盘上
Msdos	DOS 的文件系统，系统力图使它表现得像 UNIX
UMSDOS	该文件系统允许 MSDOS 文件系统可以当作 Linux 固有的文件系统一样使用
Vfat	fat 文件系统的扩展，支持长文件名
Ntfs	Windows NT 的文件系统
Hpfs	OS/2 的文件系统

2．VFS 所处理的系统调用

表 8.2 列出 VFS 的系统调用，这些系统调用涉及文件系统、常规文件、目录及符号链接。另外还有少数几个由 VFS 处理的其他系统调用：诸如 ioperm( )、ioctl( )、pipe( )和 mknod( )，涉及设备文件和管道文件，有些内容在下一章进行讨论。由 VFS 处理的最后一组系统调用，诸如 socket( )、connect( )、bind( )和 protocols( )，属于套接字系统调用并用于实现网络功能。

表 8.2　　VFS 的部分系统调用

系统调用名	功能
mount( )/ umount( )	安装/卸载文件系统
sysfs( )	获取文件系统信息
statfs( )/ fstatfs( ) /ustat( )	获取文件系统统计信息
chroot( )	更改根目录
chdir( ) /fchdir( ) /getcwd( )	更改当前目录
mkdir( ) /rmdir( )	创建/删除目录
getdents( ) /readdir( )/ link( ) unlink( ) /rename( )	对目录项进行操作
readlink( ) /symlink( )	对软链接进行操作
chown( ) /fchown( ) /lchown( )	更改文件所有者
chmod( )/ fchmod( ) /utime( )	更改文件属性
stat( ) /fstat( ) /lstat( ) access( )	读取文件状态
open( ) /close( ) /creat( ) /umask( )	打开/关闭文件
dup( ) /dup2( ) /fcntl( )	对文件描述符进行操作
select( ) /poll( )	异步 I/O 通信
truncate( ) /ftruncate( )	更改文件长度
lseek( ) /_llseek( )	更改文件指针
read( )/ write( ) /readv( ) /writev( ) sendfile( )	文件 I/O 操作
pread( )/ pwrite( )	搜索并访问文件
mmap( ) /munmap( )	文件内存映射
fdatasync( ) /fsync( ) /sync( )/ msync( )	同步访问文件数据
flock( )	处理文件锁

前面我们已经提到，VFS 是应用程序和具体的文件系统之间的一个层。不过，在某些情

况下，一个文件操作可能由 VFS 本身去执行，无需调用下一层程序。例如，当某个进程关闭一个打开的文件时，并不需要涉及磁盘上的相应文件，因此，VFS 只需释放对应的文件对象。类似地，如果系统调用 lseek()修改一个文件指针，而这个文件指针指向有关打开的文件与进程交互的一个属性，那么 VFS 只需修改对应的文件对象，而不必访问磁盘上的文件，因此，无需调用具体的文件系统子程序。从某种意义上说，可以把 VFS 看成“通用”文件系统，它在必要时依赖某种具体的文件系统。

8.2 VFS 中的数据结构

虚拟文件系统所隐含的主要思想在于引入了一个通用的文件模型，这个模型能够表示所有支持的文件系统。该模型严格遵守传统 UNIX 文件系统提供的文件模型。

你可以把通用文件模型看作是面向对象的，在这里，对象是一个软件结构，其中既定义了数据结构也定义了其上的操作方法。出于效率的考虑，Linux 的编码并未采用面向对象的程序设计语言（比如 C++）。因此对象作为数据结构来实现：数据结构中指向函数的域就对应于对象的方法。

通用文件模型由下列对象类型组成。

- 超级块（superblock）对象：存放系统中已安装文件系统的有关信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件系统控制块，也就是说，每个文件系统都有一个超级块对象。
- 索引节点（inode）对象：存放关于具体文件的一般信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件控制块（FCB），也就是说，每个文件都有一个索引节点对象。每个索引节点对象都有一个索引节点号，这个号唯一地标识某个文件系统中的指定文件。
- 目录项（dentry）对象：存放目录项与对应文件进行链接的信息。VFS 把每个目录看作一个由若干子目录和文件组成的常规文件。例如，在查找路径名/tmp/test 时，内核为根目录“/”创建一个目录项对象，为根目录下的 tmp 项创建一个第 2 级目录项对象，为/tmp 目录下的 test 项创建一个第 3 级目录项对象。
- 文件（file）对象：存放打开文件与进程之间进行交互的有关信息。这类信息仅当进程访问文件期间存在于内存中。

下面我们讨论超级块、索引节点、目录项及文件的数据结构，它们的共同特点有两个：

- 充分考虑到对多种具体文件系统的兼容性；
- 是“虚”的，也就是说只能存在于内存。

这正体现了 VFS 的特点，在下面的描述中，读者也许能体会到以上特点。

8.2.1 超级块

很多具体文件系统中都有超级块结构，超级块是这些文件系统中最重要的数据结构，它是来描述整个文件系统信息的，可以说是一个全局的数据结构。Minix、Ext2 等有超级块，

VFS 也有超级块，为了避免与后面介绍的 Ext2 超级块发生混淆，这里用 VFS 超级块来表示。VFS 超级块是各种具体文件系统在安装时建立的，并在这些文件系统卸载时自动删除，可见，VFS 超级块确实只存在于内存中,同时提到 VFS 超级块也应该说成是哪个具体文件系统的 VFS 超级块。VFS 超级块在 inculde/fs/fs.h 中定义，即数据结构 super_block，该结构及其主要域的含义如下：

```
struct super_block
{
/*************描述具体文件系统的整体信息的域*****************
       kdev_t s_dev;                  /* 包含该具体文件系统的块设备标识符。
例如，对于 /dev/hda1，其设备标识符为 0x301*/
     unsigned long s_blocksize;         /*该具体文件系统中数据块的大小，
以字节为单位 */
     unsigned char s_blocksize_bits;    /*块大小的值占用的位数，例如，
如果块大小为 1024 字节，则该值为 10*/
       unsigned long long      s_maxbytes;      /* 文件的最大长度 */
     unsigned long s_flags;             /* 安装标志*/
     unsigned long s_magic;             /*魔数，即该具体文件系统区别于其他
    文件系统的一个标志*/

/**************用于管理超级块的域******************/
    struct list_head   s_list;    /*指向超级块链表的指针*/
struct semaphore     s_lock       /*锁标志位，若置该位，则其他进程
                                不能对该超级块操作*/
    struct rw_semaphore     s_umount    /*对超级块读写时进行同步*/
unsigned char s_dirt;         /*脏位，若置该位，表明该超级块已被修改*/
struct dentry     *s_root; /*指向该具体文件系统安装目录的目录项*/
int                      s_count; /*对超级块的使用计数*/
atomic_t                 s_active;
struct list_head       s_dirty;         /*已修改的索引节点形成的链表 */
struct list_head         s_locked_inodes;/* 要进行同步的索引节点形成的链表*/
struct list_head         s_files
/************和具体文件系统相联系的域************************/
     struct file_system_type *s_type;                /*指向文件系统的
                                        file_system_type 数据结构的指针 */
     struct super_operations *s_op;            /*指向某个特定的具体文件系统的用
                                        于超级块操作的函数集合 */
struct dquot_operations *dq_op;              /* 指向某个特定的具体文件系统
                                        用于限额操作的函数集合 */
u;                                  /*一个共用体，其成员是各种文件系统
                                        的 fsname_sb_info 数据结构 */
};
```

所有超级块对象（每个已安装的文件系统都有一个超级块）以双向环形链表的形式链接在一起。链表中第一个元素和最后一个元素的地址分别存放在 super_blocks 变量的 s_list 域的 next 和 prev 域中。s_list 域的数据类型为 struct list_head，在超级块的 s_dirty 域以及内核的其他很多地方都可以找到这样的数据类型 ;这种数据类型仅仅包括指向链表中的前一个元素和后一个元素的指针。因此，超级块对象的 s_list 域包含指

向链表中两个相邻超级块对象的指针。图 8.2 说明了 list_head 元素、next 和 prev 是如何嵌入到超级块对象中的。

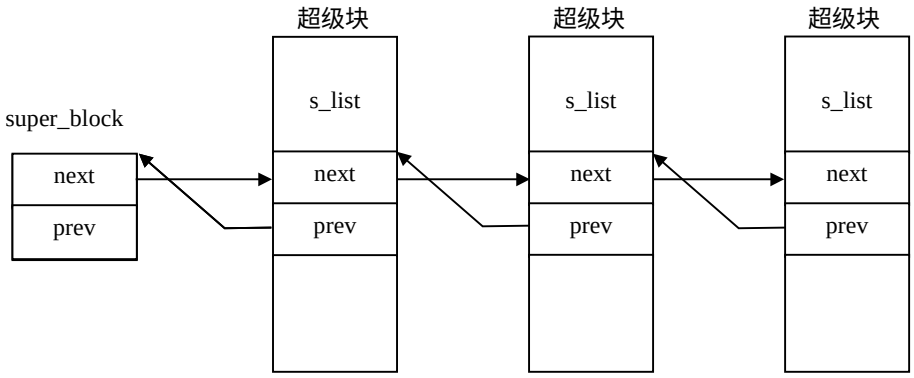

图 8.2　超级块链表

超级块最后一个 u 联合体域包括属于具体文件系统的超级块信息：

```
union {
        struct Minix_sb_info    Minix_sb;
        struct Ext2_sb_info     Ext2_sb;
        struct ext3_sb_info     ext3_sb;
        struct hpfs_sb_info     hpfs_sb;
        struct ntfs_sb_info     ntfs_sb;
        struct msdos_sb_info    msdos_sb;
        struct isofs_sb_info    isofs_sb;
        struct nfs_sb_info      nfs_sb;
        struct sysv_sb_info     sysv_sb;
        struct affs_sb_info     affs_sb;
        struct ufs_sb_info      ufs_sb;
        struct efs_sb_info      efs_sb;
        struct shmem_sb_info    shmem_sb;
        struct romfs_sb_info    romfs_sb;
        struct smb_sb_info      smbfs_sb;
        struct hfs_sb_info      hfs_sb;
        struct adfs_sb_info     adfs_sb;
        struct qnx4_sb_info     qnx4_sb;
        struct reiserfs_sb_info reiserfs_sb;
        struct bfs_sb_info      bfs_sb;
        struct udf_sb_info      udf_sb;
       struct ncp_sb_info      ncpfs_sb;
        struct usbdev_sb_info   usbdevfs_sb;
       struct jffs2_sb_info    jffs2_sb;
       struct cramfs_sb_info   cramfs_sb;
        void                    *generic_sbp;
} u;
```

通常，为了效率起见 u 域的数据被复制到内存。任何基于磁盘的文件系统都需要访问和更改自己的磁盘分配位示图，以便分配和释放磁盘块。VFS 允许这些文件系统直接对内存超

级块的 u 联合体域进行操作，无需访问磁盘。

但是，这种方法带来一个新问题：有可能 VFS 超级块最终不再与磁盘上相应的超级块同步。因此，有必要引入一个 s_dirt 标志，来表示该超级块是否是脏的，也就是说，磁盘上的数据是否必须要更新。缺乏同步还导致我们熟悉的一个问题：当一台机器的电源突然断开而用户来不及正常关闭系统时，就会出现文件系统崩溃。Linux 是通过周期性地将所有“脏”的超级块写回磁盘来减少该问题带来的危害。

与超级块关联的方法就是所谓的超级块操作。这些操作是由数据结构 super_operations 来描述的，该结构的起始地址存放在超级块的 s_op 域中，稍后将与其他对象的操作一块儿介绍。

8.2.2　VFS 的索引节点

文件系统处理文件所需要的所有信息都放在称为索引节点的数据结构中。文件名可以随时更改，但是索引节点对文件是唯一的，并且随文件的存在而存在。有关使用索引节点的原因将在下一章中进一步介绍，这里主要强调一点，具体文件系统的索引节点是存储在磁盘上的，是一种静态结构，要使用它，必须调入内存，填写 VFS 的索引节点，因此，也称 VFS 索引节点为动态节点。这里用 VFS 索引节点来避免与下一章的 Ext2 索引节点混淆。VFS 索引节点的数据结构 inode 在/includ/fs/fs.h 中定义如下（2.4.x 版本）：

```
  struct inode
{
    /**********描述索引节点高速缓存管理的域****************/
 struct list_head        i_hash;  ／*指向哈希链表的指针*/
 struct list_head        i_list;  /*指向索引节点链表的指针*/
 struct list_head        i_dentry;/*指向目录项链表的指针*/

 struct list_head        i_dirty_buffers;
 struct list_head        i_dirty_data_buffers;
 /**********描述文件信息的域****************/
    unsigned long i_ino;            /*索引节点号*/
    kdev_t        i_dev;           /*设备标识号 */
  umode_t     i_mode;         /*文件的类型与访问权限 */
  nlink_t     i_nlink;         /*与该节点建立链接的文件数 */
    uid_t         i_uid;           /*文件拥有者标识号*/
    gid_t         i_gid;            /*文件拥有者所在组的标识号*/
     kdev_t          i_rdev;           /*实际设备标识号*/
    off_t         i_size;           /*文件的大小（以字节为单位）*/
    unsigned long i_blksize;        /*块大小*/
    unsigned long i_blocks;         /*该文件所占块数*/
  time_t      i_atime;          /*文件的最后访问时间*/
    time_t        i_mtime;         /*文件的最后修改时间*/
    time_t        i_ctime;          /*节点的修改时间*/
    unsigned long i_version;        /*版本号*/
  struct semaphore   i_zombie;   /*僵死索引节点的信号量*/

/***********用于索引节点操作的域*****************/
```

```
    struct inode_operations  *i_op;  /*索引节点的操作*/
    struct super_block   *i_sb;     /*指向该文件系统超级块的指针 */
        atomic_t     i_count;     /*当前使用该节点的进程数。计数为 0，
    表明该节点可丢弃或被重新使用 */
      struct file_operations  *i_fop;  /*指向文件操作的指针 */

       unsigned char      i_lock;            /*该节点是否被锁定，用于同步操作中*/
     struct semaphore     i_sem;            /*指向用于同步操作的信号量结构*/
     wait_queue_head_t   *i_wait;          /*指向索引节点等待队列的指针*/
     unsigned char           i_dirt;          /*表明该节点是否被修改过，若已被修改，
                             则应当将该节点写回磁盘*/
    struct file_lock        *i_flock;      /*指向文件加锁链表的指针*/
    struct dquot            *i_dquot[MAXQUOTAS]; /*索引节点的磁盘限额*/
    /************用于分页机制的域*******************************/
    struct address_space    *i_mapping; ／* 把所有可交换的页面管理起来*/
         struct address_space    i_data;

    /**********以下几个域应当是联合体*************************************/
       struct list_head        i_devices;   /*设备文件形成的链表*/
       struct pipe_inode_info  i_pipe;     /*指向管道文件*/
       struct block_device     *i_bdev;   /*指向块设备文件的指针*/
       struct char_device      *i_cdev; /*指向字符设备文件的指针*/

    /************************其他域*****************************************/
    unsigned long           i_dnotify_mask; /* Directory notify events */
    struct dnotify_struct   *i_dnotify; /* for directory notifications */

    unsigned long           i_state;    /*索引节点的状态标志*/
    unsigned int            i_flags;    /*文件系统的安装标志*/
    unsigned char           i_sock;     /*如果是套接字文件则为真*/
    atomic_t                i_writecount; /*写进程的引用计数*/
    unsigned int            i_attr_flags; /*文件创建标志*/
    __u32                   i_generation  /*为以后的开发保留*/
    /************************各个具体文件系统的索引节点********************/
    union ;                             /*类似于超级块的一个共用体，其成员是各种具体文件系统
的 fsname_inode_info 数据结构 */
    }
```

对 inode 数据结构的进一步说明。

- 每个文件都有一个 inode，每个 inode 有一个索引节点号 i_ino。在同一个文件系统中，每个索引节点号都是唯一的，内核有时根据索引节点号的哈希值查找其 inode 结构。
- 每个文件都有个文件主,其最初的文件主是创建了这个文件的用户,但以后可以改变。每个用户都有一个用户组,且属于某个用户组,因此,inode 结构中就有相应的 i_uid、i_gid,以指明文件主的身份。
- inode 中有两个设备号，i_dev 和 i_rdev。首先，除特殊文件外，每个节点都存储在某个设备上，这就是 i_dev。其次，如果索引节点所代表的并不是常规文件，而是某个设备，那就还得有个设备号，这就是 i_rdev。
- 每当一个文件被访问时,系统都要在这个文件的 inode 中记下时间标记,这就是 inode

中与时间相关的几个域。

- 每个索引节点都会复制磁盘索引节点包含的一些数据，比如文件占用的磁盘块数。如果 i_state 域的值等于 I_DIRTY，该索引节点就是“脏”的，也就是说，对应的磁盘索引节点必须被更新。i_state 域的其他值有 I_LOCK（这意味着该索引节点对象已加锁），I_FREEING（这意味着该索引节点对象正在被释放）。每个索引节点对象总是出现在下列循环双向链表的某个链表中。

（1）未用索引节点链表。变量 inode_unused 的 next 域和 prev 域分别指向该链表中的首元素和尾元素。这个链表用做内存高速缓存。

（2）正在使用索引节点链表。变量 inode_in_use 指向该链表中的首元素和尾元素。

（3）脏索引节点链表。由相应超级块的 s_dirty 域指向该链表中的首元素和尾元素。

这 3 个链表都是通过索引节点的 i_list 域链接在一起的。

- 属于“正在使用”或“脏”链表的索引节点对象也同时存放在一个称为 inode_hashtable 链表中。哈希表加快了对索引节点对象的搜索，前提是系统内核要知道索引节点号及对应文件所在文件系统的超级块对象的地址。由于散列技术可能引发冲突，所以，索引节点对象设置一个 i_hash 域，其中包含向前和向后的两个指针，分别指向散列到同一地址的前一个索引节点和后一个索引节点；该域由此创建了由这些索引节点组成的一个双向链表。

与索引节点关联的方法也叫索引节点操作，由 inode_operations 结构来描述，该结构的地址存放在 i_op 域中，该结构也包括一个指向文件操作方法的指针。

8.2.3　目录项对象

每个文件除了有一个索引节点 inode 数据结构外，还有一个目录项 dentry（directory enrty）数据结构。dentry 结构中有个 d_inode 指针指向相应的 inode 结构。读者也许会问，既然 inode 结构和 dentry 结构都是对文件各方面属性的描述，那为什么不把这两个结构“合二为一”呢？这是因为二者所描述的目标不同，dentry 结构代表的是逻辑意义上的文件，所描述的是文件逻辑上的属性，因此，目录项对象在磁盘上并没有对应的映像；而 inode 结构代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统（如 Ext2），Ext2_ inode 结构在磁盘上就有对应的映像。所以说，一个索引节点对象可能对应多个目录项对象。

dentry 的定义在 include/linux/dcache.h 中：

```
struct dentry {
  atomic_t d_count;        /* 目录项引用计数器 */
  unsigned int d_flags;    /* 目录项标志 */
  struct inode  * d_inode;   /* 与文件名关联的索引节点 */
  struct dentry * d_parent;       /* 父目录的目录项 */
  struct list_head d_hash;        /* 目录项形成的哈希表 */
  struct list_head d_lru;         /*未使用的 LRU 链表 */
 struct list_head d_child;       /*父目录的子目录项所形成的链表 */
struct list_head d_subdirs;     /* 该目录项的子目录所形成的链表*/
struct list_head d_alias;       /* 索引节点别名的链表*/
```

```
        int d_mounted;                  /* 目录项的安装点 */
        struct qstr d_name;             /* 目录项名（可快速查找） */
        unsigned long d_time;           /* 由 d_revalidate 函数使用 */
        struct dentry_operations  *d_op; /* 目录项的函数集*/
        struct super_block * d_sb;      /* 目录项树的根 （即文件的超级块）*/
        unsigned long d_vfs_flags;
        void * d_fsdata;                /* 具体文件系统的数据 */
        unsigned char d_iname[DNAME_INLINE_LEN]; /* 短文件名 */
};
```

下面对 dentry 结构给出进一步的解释。

一个有效的 dentry 结构必定有一个 inode 结构，这是因为一个目录项要么代表着一个文件，要么代表着一个目录，而目录实际上也是文件。所以，只要 dentry 结构是有效的，则其指针 d_inode 必定指向一个 inode 结构。可是，反过来则不然，一个 inode 却可能对应着不止一个 dentry 结构；也就是说，一个文件可以有不止一个文件名或路径名。这是因为一个已经建立的文件可以被连接(link)到其他文件名。所以在 inode 结构中有一个队列 i_dentry，凡是代表着同一个文件的所有目录项都通过其 dentry 结构中的 d_alias 域挂入相应 inode 结构中的 i_dentry 队列。

在内核中有一个哈希表 dentry_hashtable ，是一个 list_head 的指针数组。一旦在内存中建立起一个目录节点的 dentry 结构，该 dentry 结构就通过其 d_hash 域链入哈希表中的某个队列中。

内核中还有一个队列 dentry_unused，凡是已经没有用户（count 域为 0）使用的 dentry 结构就通过其 d_lru 域挂入这个队列。

Dentry 结构中除了 d_alias、d_hash、d_lru 三个队列外，还有 d_vfsmnt、d_child 及 d_subdir 三个队列。其中 d_vfsmnt 仅在该 dentry 为一个安装点时才使用。另外，当该目录节点有父目录时，则其 dentry 结构就通过 d_child 挂入其父节点的 d_subdirs 队列中，同时又通过指针 d_parent 指向其父目录的 dentry 结构,而它自己各个子目录的 dentry 结构则挂在其 d_subdirs 域指向的队列中。

从上面的叙述可以看出，一个文件系统中所有目录项结构或组织为一个哈希表，或组织为一颗树，或按照某种需要组织为一个链表，这将为文件访问和文件路径搜索奠定下良好的基础。

8.2.4 与进程相关的文件结构

在具体介绍这几个结构以前，我们需要解释一下文件描述符、打开的文件描述、系统打开文件表、用户打开文件表的概念以及它们的联系。

1．文件对象

在 Linux 中，进程是通过文件描述符（file descriptors，简称 fd）而不是文件名来访问文件的，文件描述符实际上是一个整数。Linux 中规定每个进程最多能同时使用 NR_OPEN 个文件描述符，这个值在 fs.h 中定义，为 1024×1024（2.0 版中仅定义为 256）。

每个文件都有一个 32 位的数字来表示下一个读写的字节位置，这个数字叫做文件位置。

每次打开一个文件，除非明确要求，否则文件位置都被置为 0，即文件的开始处，此后的读或写操作都将从文件的开始处执行，但你可以通过执行系统调用 LSEEK（随机存储）对这个文件位置进行修改。Linux 中专门用了一个数据结构 file 来保存打开文件的文件位置，这个结构称为打开的文件描述（open file description）。这个数据结构的设置是煞费苦心的，因为它与进程的联系非常紧密，可以说这是 VFS 中一个比较难于理解的数据结构。

首先，为什么不把文件位置干脆存放在索引节点中，而要多此一举，设一个新的数据结构呢？我们知道，Linux 中的文件是能够共享的，假如把文件位置存放在索引节点中，则如果有两个或更多个进程同时打开同一个文件时，它们将去访问同一个索引节点，于是一个进程的 LSEEK 操作将影响到另一个进程的读操作，这显然是不允许也是不可想象的。

另一个想法是既然进程是通过文件描述符访问文件的，为什么不用一个与文件描述符数组相平行的数组来保存每个打开文件的文件位置？这个想法也是不能实现的，原因就在于在生成一个新进程时，子进程要共享父进程的所有信息，包括文件描述符数组。

我们知道，一个文件不仅可以被不同的进程分别打开，而且也可以被同一个进程先后多次打开。一个进程如果先后多次打开同一个文件，则每一次打开都要分配一个新的文件描述符，并且指向一个新的 file 结构，尽管它们都指向同一个索引节点，但是，如果一个子进程不和父进程共享同一个 file 结构，而是也如上面一样，分配一个新的 file 结构，会出现什么情况了？让我们来看一个例子。

假设有一个输出重定位到某文件 A 的 shell script（shell 脚本），我们知道，shell 是作为一个进程运行的，当它生成第 1 个子进程时，将以 0 作为 A 的文件位置开始输出，假设输出了 2KB 的数据，则现在文件位置为 2KB。然后，shell 继续读取脚本，生成另一个子进程，它要共享 shell 的 file 结构，也就是共享文件位置，所以第 2 个进程的文件位置是 2KB，将接着第 1 个进程输出内容的后面输出。如果 shell 不和子进程共享文件位置，则第 2 个进程就有可能重写第 1 个进程的输出了，这显然不是希望得到的结果。

至此，已经可以看出设置 file 结构的原因所在了。

file 结构中主要保存了文件位置，此外，还把指向该文件索引节点的指针也放在其中。file 结构形成一个双链表，称为系统打开文件表，其最大长度是 NR_FILE，在 fs.h 中定义为 8192。

file 结构在 include\linux\fs.h 中定义如下：

```
struct file
{
 struct list_head        f_list;    /*所有打开的文件形成一个链表*/
 struct dentry           *f_dentry; /*指向相关目录项的指针*/
 struct vfsmount         *f_vfsmnt; /*指向 VFS 安装点的指针*/
 struct file_operations  *f_op;     /*指向文件操作表的指针*/
  mode_t f_mode;                                  /*文件的打开模式*/
  loff_t f_pos;                                   /*文件的当前位置*/
     unsigned short f_flags;                        /*打开文件时所指定的标志*/
     unsigned short f_count;                          /*使用该结构的进程数*/
     unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
   /*预读标志、要预读的最多页面数、上次预读后的文件指针、预读的字节数以及
     预读的页面数*/
  int f_owner;                  /* 通过信号进行异步 I/O 数据的传送*/
```

```
    unsigned int         f_uid, f_gid;  /*用户的 UID 和 GID*/
    int                  f_error;       /*网络写操作的错误码*/

    unsigned long f_version;           /*版本号*/
      void *private_data;                          /* tty 驱动程序所需 */

};
```

每个文件对象总是包含在下列的一个双向循环链表之中。

- “未使用”文件对象的链表。该链表既可以用做文件对象的内存高速缓存，又可以当作超级用户的备用存储器，也就是说，即使系统的动态内存用完，也允许超级用户打开文件。由于这些对象是未使用的，它们的 f_count 域是 NULL，该链表首元素的地址存放在变量 free_list 中，内核必须确认该链表总是至少包含 NR_RESERVED_FILES 个对象，通常该值设为 10。
- “正在使用”文件对象的链表。该链表中的每个元素至少由一个进程使用，因此，各个元素的 f_count 域不会为 NULL，该链表中第一个元素的地址存放在变量 anon_list 中。

如果 VFS 需要分配一个新的文件对象，就调用函数 get_empty_filp( )。该函数检测‘未使用”文件对象链表的元素个数是否多于 NR_RESERVED_FILES，如果是，可以为新打开的文件使用其中的一个元素；如果没有，则退回到正常的内存分配。

2．用户打开文件表

每个进程用一个 files_struct 结构来记录文件描述符的使用情况，这个 files_struct 结构称为用户打开文件表，它是进程的私有数据。files_struct 结构在 include/linux/sched.h 中定义如下：

```
struct files_struct {
     atomic_t count;         /* 共享该表的进程数 */
     rwlock_t file_lock;      /* 保护以下的所有域,以免在
tsk->alloc_lock 中的嵌套*/
     int max_fds;            /*当前文件对象的最大数*/
     int max_fdset;         /*当前文件描述符的最大数*/
          int next_fd;           ／*已分配的文件描述符加 1*/
      struct file ** fd;       /* 指向文件对象指针数组的指针 */
      fd_set *close_on_exec;  /*指向执行 exec（ ）时需要关闭的文件描述符*/
      fd_set *open_fds;      /*指向打开文件描述符的指针*/
       fd_set close_on_exec_init;/* 执行 exec（ ）时需要关闭的文件描述符的初          值
集合*/
          fd_set open_fds_init;  /*文件描述符的初值集合*/
      struct file * fd_array[32];/* 文件对象指针的初始化数组*/
};
```

fd 域指向文件对象的指针数组。该数组的长度存放在 max_fds 域中。通常，fd 域指向 files_struct 结构的 fd_array 域，该域包括 32 个文件对象指针。如果进程打开的文件数目多于 32，内核就分配一个新的、更大的文件指针数组，并将其地址存放在 fd 域中；内核同时也更新 max_fds 域的值。

对于在 fd 数组中有入口地址的每个文件来说，数组的索引就是文件描述符（file descriptor)。通常，数组的第 1 个元素（索引为 0）是进程的标准输入文件，数组的第 2 个

元素（索引为 1）是进程的标准输出文件，数组的第 3 个元素（索引为 2）是进程的标准错误文件（参见图 8.3)。请注意，借助于 dup()、dup2()和 fcntl() 系统调用，两个文件描述符就可以指向同一个打开的文件，也就是说，数组的两个元素可能指向同一个文件对象。当用户使用 shell 结构（如 2>&1）将标准错误文件重定向到标准输出文件上时，用户总能看到这一点。

open_fds 域包含 open_fds_init 域的地址，open_fds_init 域表示当前已打开文件的文件描述符的位图。max_fdset 域存放位图中的位数。由于数据结构 fd_set 有 1024 位，通常不需要扩大位图的大小。不过，如果确实需要，内核仍能动态增加位图的大小，这非常类似文件对象的数组的情形。

当开始使用一个文件对象时调用内核提供的 fget()函数。这个函数接收文件描述符 fd 作为参数，返回在 current->files->fd[fd]中的地址，即对应文件对象的地址，如果没有任何文件与 fd 对应，则返回 NULL。在第 1 种情况下，fget()使文件对象引用计数器 f_count 的值增 1。

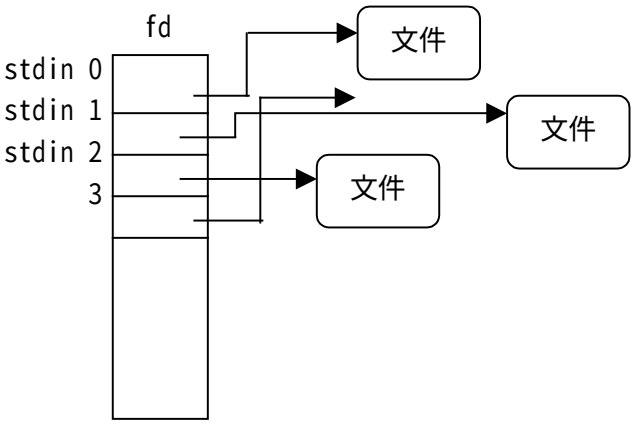

图 8.3　文件描述符数组

当内核完成对文件对象的使用时，调用内核提供的 fput() 函数。该函数将文件对象的地址作为参数，并递减文件对象引用计数器 f_count 的值，另外，如果这个域变为 NULL，该函数就调用文件操作的“ 释放 ”方法（如果已定义)，释放相应的目录项对象，并递减对应索引节点对象的 i_writeaccess 域的值（如果该文件是写打开)，最后，将该文件对象从“ 正在使用 ”链表移到“ 未使用 ”链表。

3．关于文件系统信息的 fs_struct 结构

fs_struct 结构在 2.4 以前的版本中在 include/linux/sched.h 中定义为：

```
struct fs_struct {
    atomic_t count;
    int umask;
    struct dentry * root, * pwd;
};
```

在 2.4 版本中，单独定义在 include/linux/fs_struct.h 中:

```
struct fs_struct {
         atomic_t count;
         rwlock_t lock;
         int umask;
        struct dentry * root, * pwd, * altroot;
         struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
};
```

count 域表示共享同一 fs_struct 表的进程数目。umask 域由 umask（）系统调用使用，用于为新创建的文件设置初始文件许可权。

fs_struct 中的 dentry 结构是对一个目录项的描述，root、pwd 及 altroot 三个指针都指向这个结构。其中，root 所指向的 dentry 结构代表着本进程所在的根目录，也就是在用户登录进入系统时所看到的根目录；pwd 指向进程当前所在的目录；而 altroot 则是为用户设置的替换根目录。实际运行时，这 3 个目录不一定都在同一个文件系统中。例如，进程的根目录通常是安装于“／”节点上的 Ext2 文件系统，而当前工作目录可能是安装于／msdos 的一个 DOS 文件系统。因此，fs_struct 结构中的 rootmnt、 pwdmnt 及 altrootmnt 就是对那 3 个目录的安装点的描述，安装点的数据结构为 vfsmount。

8.2.5 主要数据结构间的关系

前面我们介绍了超级块对象、索引节点对象、文件对象及目录项对象的数据结构。我们在此给出这些数据结构之间的联系。

超级块是对一个文件系统的描述；索引节点是对一个文件物理属性的描述；而目录项是对一个文件逻辑属性的描述。除此之外，文件与进程之间的关系是由另外的数据结构来描述的。一个进程所处的位置是由 fs_struct 来描述的，而一个进程（或用户）打开的文件是由 files_struct 来描述的，而整个系统所打开的文件是由 file 结构来描述。如图 8.4 给出了这些数据结构之间的关系。

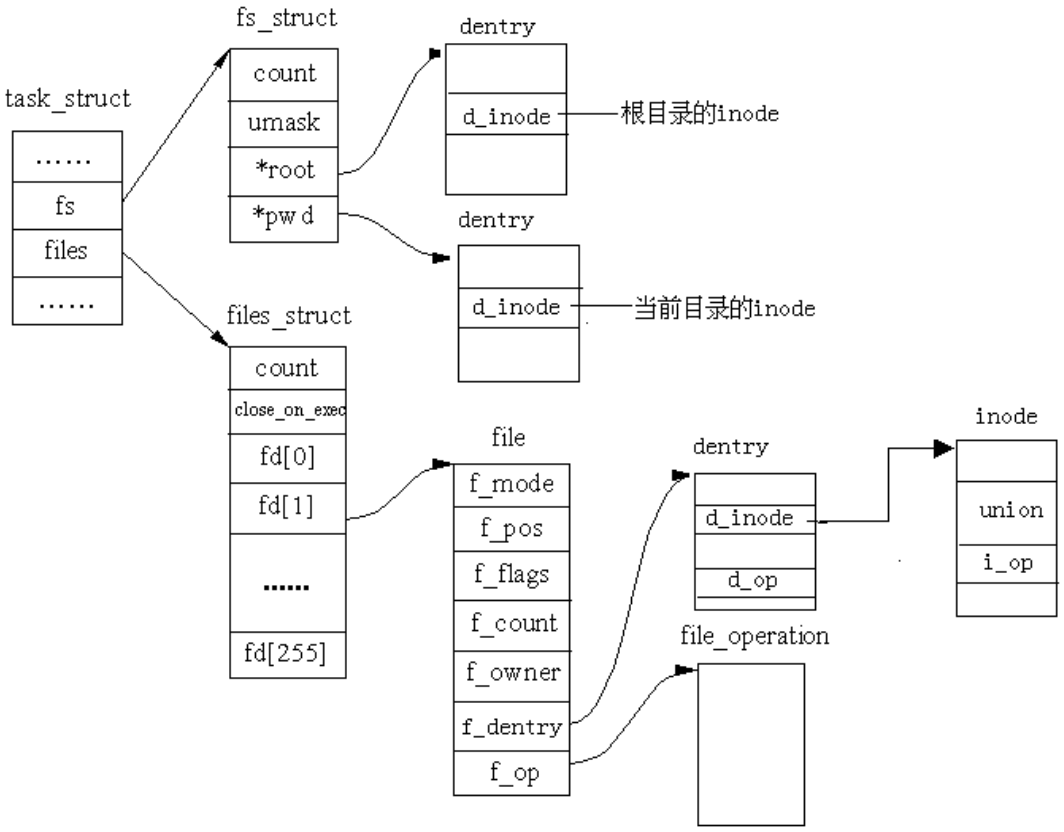

图 8.4　与进程联系的文件结构的关系示意图

8.2.6　有关操作的数据结构

VFS 毕竟是虚拟的，它无法涉及到具体文件系统的细节，所以必然在 VFS 和具体文件系统之间有一些接口，这就是 VFS 设计的一些有关操作的数据结构。这些数据结构就好像是一个标准，具体文件系统要想被 Linux 支持，就必须按这个标准来编写自己操作函数。实际上，也正是这样，各种 Linux 支持的具体文件系统都有一套自己的操作函数，在安装时，这些结构体的成员指针将被初始化，指向对应的函数。如果说 VFS 体现了 Linux 的优越性，那么这些数据结构的设计就体现了 VFS 的优越性所在。图 8.5 是 VFS 和具体文件系统的关系示意图。

这个示意图还无法完全反映这种关系。因为对每个具体文件系统来说，VFS 都有相应的数据结构对应，而不是如图中那样简单化了。

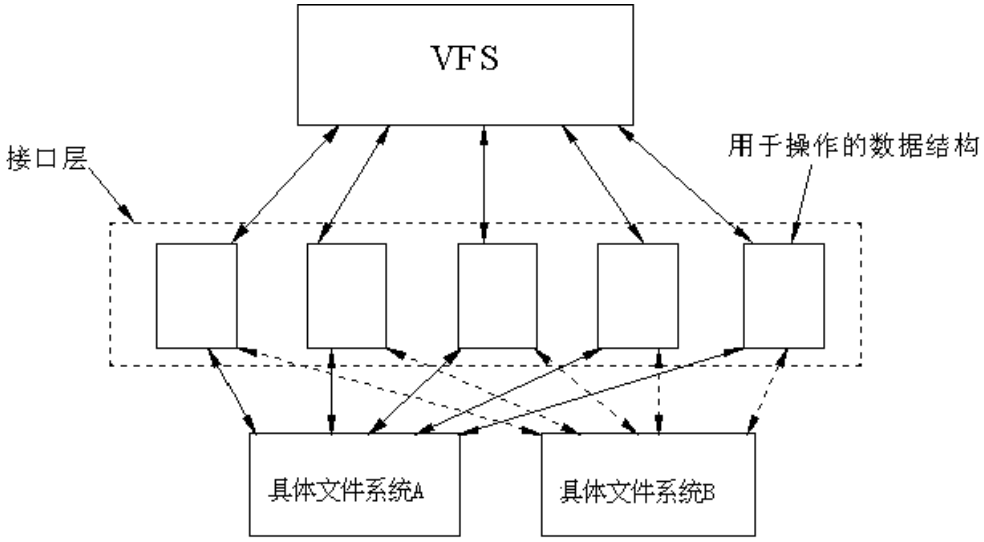

图 8.5　VFS 和具体文件系统的关系示意图

有关操作的数据结构主要有以下几个，分别用来操作 VFS 中的几个重要的数据结构。

1．超级块操作

超级块操作是由 super_operations 数据结构来描述的，该结构的起始地址存放在超级块的 s_op 域中。该结构定义于 fs.h 中：

```
/*
* NOTE: write_inode, delete_inode, clear_inode, put_inode can be called
* without the big kernel lock held in all filesystems.
*/
struct super_operations {
        void  (*read_inode)   (struct inode *) ;
        void  (*read_inode2)   (struct inode *, void *)  ;
        void  (*dirty_inode)   (struct inode *) ;
        void  (*write_inode)   (struct inode *, int) ;
        void  (*put_inode)   (struct inode *) ;
        void  (*delete_inode)   (struct inode *) ;
        void  (*put_super)   (struct super_block *) ;
        void  (*write_super)   (struct super_block *) ;
        void  (*write_super_lockfs)   (struct super_block *) ;
        void  (*unlockfs)   (struct super_block *) ;
        int  (*statfs)   (struct super_block *, struct statfs *) ;
        int  (*remount_fs)   (struct super_block *, int *, char *) ;
        void  (*clear_inode)   (struct inode *) ;
        void  (*umount_begin)   (struct super_block *) ;
}
```

其中的每个函数就叫做超级块的一个方法，表 8.3 给予描述。

表 8.3　　超级块对象的方法及其描述

函数形式	描述

Read_inode(inode)	inode 的地址是该函数的参数，inode 中的 i_no 域表示从磁盘要读取的具体文件系统的 inode，用磁盘上的数据填充参数 inode 的域
Dirty_inode(inode)	把 inode 标记为“脏”
Write_inode(inode)	用参数指定的 inode 更新某个文件系统的 inode。inode 的 i_ino 域标识指定磁盘上文件系统的索引节点
Put_inode(inode)	释放参数指定的索引节点对象。释放一个对象并不意味着释放内存，因为其他进程可能仍然在使用这个对象。该方法是可选的（即并非所有的文件系统都有相应的处理函数）
Delete_inode(inode)	删除那些包含文件、磁盘索引节点及 VFS 索引节点的数据块
Notify_change(dentry, iattr)	依照参数 iattr 的值修改索引节点的一些属性。如果没有定义该函数，VFS 转去执行 write_inode( ) 方法
Put_super(super)	释放超级块对象
Write_super(super)	将超级块的信息写回磁盘，该方法是可选的
Statfs(super, buf, bufsize)	将文件系统的统计信息填写在 buf 缓冲区中
Remount_fs(super, flags, data)	用新的选项重新安装文件系统（当某个安装选项必须被修改时进行调用）
Clear_inode(inode)	与 put_inode 类似，但同时也把索引节点对应文件中的数据占用的所有页释放
Umount_begin(super)	中断一个安装操作（只在网络文件系统中使用）

上面这些方法对所有的文件系统都是适用的，但对于一个具体的文件系统来说，可能只用到其中的几个方法。如果那些方法没有定义，则对应的域为空。

2．索引节点操作 inode_operations

索引节点操作是由 inode_operations 结构来描述的，主要是用来将 VFS 对索引节点的操作转化为具体文件系统处理相应操作的函数，在 fs.h 中描述如下：

```
struct inode_operations {
        int  (*create)   (struct inode *,struct dentry *,int) ;
        struct dentry *  (*lookup)   (struct inode *,struct dentry *) ;
        int  (*link)   (struct dentry *,struct inode *,struct dentry *) ;
        int  (*unlink)   (struct inode *,struct dentry *) ;
        int  (*symlink)   (struct inode *,struct dentry *,const char *) ;
        int  (*mkdir)   (struct inode *,struct dentry *,int) ;
        int  (*rmdir)   (struct inode *,struct dentry *) ;
        int  (*mknod)   (struct inode *,struct dentry *,int,int) ;
        int  (*rename)   (struct inode *, struct dentry *,
                        struct inode *, struct dentry *) ;
        int  (*readlink)   (struct dentry *, char *,int) ;
        int  (*follow_link)   (struct dentry *, struct nameidata *) ;
        void  (*truncate)   (struct inode *) ;
       int  (*permission)   (struct inode *, int) ;
```

```
        int  (*revalidate)   (struct dentry *) ;
        int  (*setattr)   (struct dentry *, struct iattr *) ;
        int  (*getattr)   (struct dentry *, struct iattr *) ;
};
```

表 8.4 所示为对索引节点的每个方法给予描述。

表 8.4　　索引节点对象的方法及其描述

函数形式	描述
Create(dir, dentry, mode)	在某个目录下，为与 dentry 目录项相关的常规文件创建一个新的磁盘索引节点
Lookup(dir, dentry)	查找索引节点所在的目录，这个索引节点所对应的文件名就包含在 dentry 目录项中
Link(old_dentry, dir, new_dentry)	创建一个新的名为 new_dentry 硬链接，这个新的硬连接指向 dir 目录下名为 old_dentry 的文件
unlink(dir, dentry)	从 dir 目录删除 dentry 目录项所指文件的硬链接
symlink(dir, dentry, symname)	在某个目录下，为与目录项相关的符号链创建一个新的索引节点
mkdir(dir, dentry, mode)	在某个目录下，为与目录项对应的目录创建一个新的索引节点
mknod(dir, dentry, mode, rdev)	在 dir 目录下，为与目录项对象相关的特殊文件创建一个新的磁盘索引节点。其中参数 mode 和 rdev 分别表示文件的类型和该设备的主码
rename(old_dir, old_dentry, new_dir, new_dentry)	将 old_dir 目录下的文件 old_dentry 移到 new_dir 目录下，新文件名包含在 new_dentry 指向的目录项中
readlink(dentry, buffer, buflen)	将 dentry 所指定的符号链中对应的文件路径名拷贝到 buffer 所指定的内存区
follow_link(inode, dir)	解释 inode 索引节点所指定的符号链；如果该符号链是相对路径名，从指定的 dir 目录开始进行查找
truncate(inode)	修改索引节点 inode 所指文件的长度。在调用该方法之前，必须将 inode 对象的 i_size 域设置为需要的新长度值
permission(inode, mask)	确认是否允许对 inode 索引节点所指的文件进行指定模式的访问
revalidate(dentry)	更新由目录项所指定文件的已缓存的属性（通常由网络文件系统调用）
Setattr (dentry, attr)	设置目录项的属性
Getattr (dentry, attr)	获得目录项的属性

以上这些方法均适用于所有的文件系统，但对某一个具体文件系统来说，可能只用到其中的一部分方法。例如，msdos 文件系统其公用索引节点的操作在 fs/msdos/namei.c 中定义如下：

```
struct inode_operations msdos_dir_inode_operations = {
    create:         msdos_create,
    lookup:         msdos_lookup,
    unlink:         msdos_unlink,
    mkdir:          msdos_mkdir,
```

```
        rmdir:          msdos_rmdir,
        rename:         msdos_rename,
        setattr:        fat_notify_change,
};
```

3．目录项操作

目录项操作是由 dentry_operations 数据结构来描述的，定义于 include/linux/dcache.h 中：

```
struct dentry_operations {
        int  (*d_revalidate)  (struct dentry *, int) ;
        int  (*d_hash)    (struct dentry *, struct qstr *) ;
        int  (*d_compare)    (struct dentry *, struct qstr *, struct qstr *) ;
        int  (*d_delete)  (struct dentry *) ;
        void  (*d_release)  (struct dentry *) ;
        void  (*d_iput)  (struct dentry *, struct inode *) ;
};
```

表 8.5 给出目录项对象的方法及其描述。

表 8.5　　目录项对象的方法及其描述

函数形成	描述
d_revalidate(dentry)	判定目录项是否有效。默认情况下，VFS 函数什么也不做，而网络文件系统可以指定自己的函数
d_hash(dentry, hash)	生成一个哈希值。对目录项哈希表而言，这是一个具体文件系统的哈希函数。参数 dentry 标识包含该路径分量的目录。参数 hash 指向一个结构，该结构包含要查找的路径名分量以及由 hash 函数生成的哈希值
d_compare(dir, name1, name2)	比较两个文件名。name1 应该属于 dir 所指目录。默认情况下，VFS 的这个函数就是常用的字符串匹配函数。不过，每个文件系统可用自己的方式实现这一方法。例如，MS-DOS 文件系统不区分大写和小写字母
d_delete(dentry)	如果对目录项的最后一个引用被删除（d_count 变为“0”），就调用该方法。默认情况下，VFS 的这个函数什么也不做
d_release(dentry)	当要释放一个目录项时（放入 slab 分配器），就调用该方法。默认情况下，VFS 的这个函数什么也不做
d_iput(dentry, ino)	当要丢弃目录项对应的索引节点时，就调用该方法。默认情况下，VFS 的这个函数调用 iput()释放索引节点

4．文件操作

文件操作是由 file_operations 结构来描述的，定义在 fs.h 中：

```
  /*
813  * NOTE:
814  * read, write, poll, fsync, readv, writev can be called
815  *   without the big kernel lock held in all filesystems.
  */
 struct file_operations {
      struct module *owner;
      loff_t  (*llseek)   (struct file *, loff_t, int) ;
      ssize_t  (*read)   (struct file *, char *, size_t, loff_t *) ;
```

```
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *) ;
        int (*readdir) (struct file *, void *, filldir_t) ;
        unsigned int (*poll) (struct file *, struct poll_table_struct *) ;
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long) ;
        int (*mmap) (struct file *, struct vm_area_struct *) ;
        int (*open) (struct inode *, struct file *) ;
        int (*flush) (struct file *) ;
        int (*release) (struct inode *, struct file *) ;
        int (*fsync) (struct file *, struct dentry *, int datasync) ;
        int (*fasync) (int, struct file *, int) ;
        int (*lock) (struct file *, int, struct file_lock *) ;
      ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *) ;
      ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *) ;
      ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int) ;
      unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long,
unsigned long, unsigned long) ;
    };
```

这个数据结构就是连接 VFS 文件操作与具体文件系统的文件操作之间的枢纽，也是编写设备驱动程序的重要接口，后面还会给出进一步的说明。对每个函数的描述如表 8.6 所示。

表 8.6　　文件操作的描述

函数形式	描述
Owner ()	指向模块的指针。只有驱动程序才把这个域置为 THIS_MODULE，文件系统一般忽略这个域
llseek(file, offset, whence)	修改文件指针
read(file, buf, count, offset)	从文件的 offset 处开始读出 count 个字节，然后增加*offset 的值
write(file, buf, count, offset)	从文件的*offset 处开始写入 count 个字节，然后增加*offset 的值
readdir(dir, dirent, filldir)	返回 dir 所指目录的下一个目录项，这个值存入参数 dirent；参数 filldir 存放一个辅助函数的地址，该函数可以提取目录项的各个域
poll(file, poll_table)	检查是否存在关于某文件的操作事件，如果没有则睡眠，直到发生该类操作事件为止
ioctl(inode, file, cmd, arg)	向一个基本硬件设备发送命令。该方法只适用于设备文件
mmap(file, vma)	执行文件的内存映射，并将这个映射放入进程的地址空间
open(inode, file)	通过创建一个新的文件而打开一个文件，并把它链接到相应的索引节点
flush(file)	当关闭对一个打开文件的引用时，就调用该方法。也就是说，减少该文件对象 f_count 域的值。该方法的实际用途依赖于具体文件系统
release(inode, file)	释放文件对象。当关闭对打开文件的最后一个引用时，也就是说，该文件对象 f_count 域的值变为 0 时，调用该方法
fsync(file, dentry)	将 file 文件在高速缓存中的全部数据写入磁盘
fasync(file, on)	通过信号来启用或禁用异步 I/O 通告

续表

函数形式	描述
check_media_change(dev)	检测自上次对设备文件操作以来是否存在介质的改变（可以对块设备使用这一方法，因为它支持可移动介质——比如软盘和 CD-ROM）
revalidate(dev)	恢复设备的一致性（由网络文件系统使用，这是在确认某个远程设备上的介质已被改变之后才使用）
lock(file, cmd, file_lock)	对 file 文件申请一个锁
readv(file, iovec, count, offset)	与 read()类似，所不同的是，readv () 把读入的数据放在多个缓冲区中（叫缓冲区向量）
Writev(file, buf, iovec, offset)	与 write()类似。所不同的是，writev()把数据写入多个缓冲区中（叫缓冲区向量）

VFS 中定义的这个 file_operations 数据结构相当于一个标准模板，对于一个具体的文件系统来说，可能只用到其中的一些函数。注意，2.2 和 2.4 版在对 file_operations 进行初始化时有所不同，在 2.2 版中，如果某个函数没有定义，则将其置为 NULL，如：

```
struct file_operations device_fops = {
    NULL,                   /* seek */
    device_read,        /* read */
    device_write,       /* write */
    NULL,                   /* readdir */
    NULL,                   /* poll */
    NULL,                   /* ioctl */
    NULL,                   /* mmap */
    device_open,        /* open */
    NULL,                   /* flush */
    device_release      /* release */
};
```

这是标准 C 的用法，在 2.4 版中，采用了 gcc 的扩展用法，如：

```
  struct file_operations device_fops = {
    read : device_read,             /* read */
    write : device_write,       /* write */
    open : device_open,             /* open */
    release : device_release    /* release */
};
```

这种方式显然简单明了，在设备驱动程序的开发中，经常会用到这种形式。

8.3　高速缓存

8.3.1　块高速缓存

Linux 支持的文件系统大多以块的形式组织文件，为了减少对物理块设备的访问，在文件以块的形式调入内存后，使用块高速缓存（buffer_cache）对它们进行管理。每个缓冲区由两部分组成，第 1 部分称为缓冲区首部，用数据结构 buffer_head 表示，第 2 部分是真正

的缓冲区内容（即所存储的数据）。由于缓冲区首部不与数据区域相连，数据区域独立存储。因而在缓冲区首部中，有一个指向数据的指针和一个缓冲区长度的字段。图 8.6 给出了一个缓冲区的格式。

缓冲区首部包含如下内容。

- 用于描述缓冲区内容的信息，包括：所在设备号、起始物理块号、包含在缓冲区中的字节数。
- 缓冲区状态的域：是否有有用数据、是否正在使用、重新利用之前是否要写回磁盘等。
- 用于管理的域。

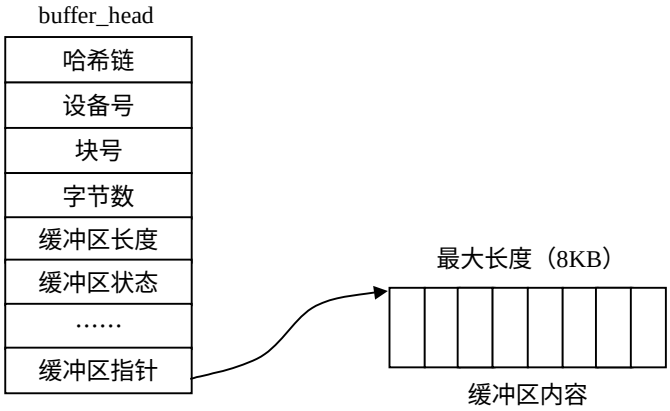

图 8.6　缓冲区格式

buffer-head 数据结构在 include\linux\fs.h 中定义如下：

```
/*
  * Try to keep the most commonly used fields in single cache lines  (16
  * bytes)  to improve performance.  This ordering should be
  * particularly beneficial on 32-bit processors.
  *
  * We use the first 16 bytes for the data which is used in searches
  * over the block hash lists  (ie. getblk ()  and friends) .
  *
  * The second 16 bytes we use for lru buffer scans, as used by
  * sync_buffers ()  and refill_freelist () .  -- sct
  */
struct buffer_head {
        /* First cache line: */
         struct buffer_head *b_next;     /* 哈希队列链表*/
         unsigned long b_blocknr;        /* 逻辑块号 */
         unsigned short b_size;          /* 块大小 */
         unsigned short b_list;          /* 本缓冲区所出现的链表 */
         kdev_t b_dev;                     /* 虚拟设备标示符（B_FREE = free）  */

         atomic_t b_count;               /* 块引用计数器 */
          kdev_t b_rdev;                  /* 实际设备标识符*/
         unsigned long b_state;          /* 缓冲区状态位图 */
unsigned long b_flushtime;      /* 对脏缓冲区进行刷新的时间*/
```

```
        struct buffer_head *b_next_free;/* 指向 lru/free 链表中的下一个元素 */
        struct buffer_head *b_prev_free;/* 指向链表中的上一个元素*/
 struct buffer_head *b_this_page;/* 每个页面中的缓冲区链表*/
        struct buffer_head *b_reqnext;  /*请求队列 */

        struct buffer_head **b_pprev;   /* 哈希队列的双向链表 */
       char * b_data;                  /* 指向数据块 */
       struct page *b_page;            /* 这个 bh 所映射的页面*/
void  (*b_end_io)  (struct buffer_head *bh, int uptodate) ; /* I/O 结束方法*/
        void *b_private;                /* 给 b_end_io 保留 */

       unsigned long b_rsector;        /* 缓冲区在磁盘上的实际位置*/
        wait_queue_head_t b_wait;     /* 缓冲区等待队列 */

        struct inode *       b_inode;
struct list_head    b_inode_buffers;   /* inode 脏缓冲区的循环链表*/
};
```

其中缓冲区状态在 fs.h 中定义为枚举类型：

```
/* bh state bits */
enum bh_state_bits {
        BH_Uptodate,    /* 如果缓冲区包含有效数据则置 1 */
        BH_Dirty,       /* 如果缓冲区数据被改变则置 1 */
        BH_Lock,        /* 如果缓冲区被锁定则置 1*/
        BH_Req,         /* 如果缓冲区数据无效则置 0  */
        BH_Mapped,      /* 如果缓冲区有一个磁盘映射则置 1 */
        BH_New,         /* 如果缓冲区为新且还没有被写出则置 1 */
        BH_Async,       /* 如果缓冲区是进行 end_buffer_io_async I/O 同步则置 1 */
        BH_Wait_IO,     /* 如果我们应该把这个缓冲区写出则置 1 */
        BH_launder,     /* 如果我们应该 " 清洗 " 这个缓冲区则置 1 */
        BH_JBD,         /* 如果与 journal_head 相连接则置 1 */

        BH_PrivateStart,/* 这不是一个状态位，但是，第 1 位由其他实体用于私有分配*/
}
```

显然一个缓冲区可以同时具有上述状态的几种。

块高速缓存的管理很复杂，下面先对空缓冲区、空闲缓冲区、正使用的缓冲区、缓冲区的大小以及缓冲区的类型作一个简短的介绍。

缓冲区可以分为两种，一种是包含了有效数据的，另一种是没有被使用的，即空缓冲区。

具有有效数据并不能表明某个缓冲区正在被使用，毕竟，在同一时间内，被进程访问的缓冲区（即处于使用状态）只有少数几个。当前没有被进程访问的有效缓冲区和空缓冲区称为空闲缓冲区。其实，buffer_head 结构中的 b_count 就可以反映出缓冲区是否处于使用状态。如果它为 0，则缓冲区是空闲的。大于 0，则缓冲区正被进程访问。

缓冲区的大小不是固定的，当前 Linux 支持 5 种大小的缓冲区，分别是 512、1024、2048、4096、8192 字节。Linux 所支持的文件系统都使用共同的块高速缓存，在同一时刻，块高速缓存中存在着来自不同物理设备的数据块，为了支持这些不同大小的数据块，Linux 使用了几种不同大小的缓冲区。

当前的 Linux 缓冲区有 3 种类型，在 include/linux/fs.h 中有如下的定义：

```
#define BUF_CLEAN        0       /*未使用的、干净的缓冲区*/
#define BUF_LOCKED       1       /*被锁定的缓冲区，正等待写入*/
#define BUF_DIRTY        2      /*脏的缓冲区，其中有有效数据，需要写回磁盘*/
```

VFS 使用了多个链表来管理块高速缓存中的缓冲区。

首先，对于包含了有效数据的缓冲区，用一个哈希表来管理，用 hash_table 来指向这个哈希表。哈希索引值由数据块号以及其所在的设备标识号计算（散列）得到。所以在 buffer_head 这个结构中有一些用于哈希表管理的域。使用哈希表可以迅速地查找到所要寻找的数据块所在的缓冲区。

对于每一种类型的未使用的有效缓冲区，系统还使用一个 LRU（最近最少使用）双链表管理，即 lru-list 链。由于共有 3 种类型的缓冲区，所以有 3 个这样的 LRU 链表。当需要访问某个数据块时，系统采取如下算法。

首先，根据数据块号和所在设备号在块高速缓存中查找，如果找到，则将它的 b_count 域加 1，因为这个域正是反映了当前使用这个缓冲区的进程数。如果这个缓冲区同时又处于某个 LRU 链中，则将它从 LRU 链中解开。

如果数据块还没有调入缓冲区，则系统必须进行磁盘 I/O 操作，将数据块调入块高速缓存，同时将空缓冲区分配一个给它。如果块高速缓存已满（即没有空缓冲区可供分配），则从某个 LRU 链首取下一个，先看是否置了“ 脏 ”位，如已置，则将它的内容写回磁盘。然后清空内容，将它分配给新的数据块。

在缓冲区使用完了后，将它的 b_count 域减 1，如果 b_count 变为 0，则将它放在某个 LRU 链尾，表示该缓冲区已可以重新利用。

为了配合以上这些操作，以及其他一些多块高速缓存的操作，系统另外使用了几个链表，主要是：

- 对于每一种大小的空闲缓冲区，系统使用一个链表管理，即 free_list 链。
- 对于空缓冲区，系统使用一个 unused_list 链管理。

以上几种链表都在 fs/buffer.c 定义。

Linux 中，用 bdflush 守护进程完成对块高速缓存的一般管理。bdflush 守护进程是一个简单的内核线程，在系统启动时运行，它在系统中注册的进程名称为 kflushd，你可以使用 ps 命令看到此系统进程。它的一个作用是监视块高速缓存中的“ 脏 ”缓冲区，在分配或丢弃缓冲区时，将对“ 脏 ”缓冲区数目作一个统计。通常情况下，该进程处于休眠状态，当块高速缓存中“ 脏 ”缓冲区的数目达到一定的比例，默认是 60%，该进程将被唤醒。但是，如果系统急需,则在任何时刻都可以唤醒这个进程。使用 update 命令可以看到和改变这个数值。

```
# update -d
```

当有数据写入缓冲区使之变成“ 脏 ”时，所有的“ 脏 ”缓冲区被连接到一个 BUF_DIRTY_LRU 链表中，bdflush 会将适当数目的缓冲区中的数据块写到磁盘上。这个数值的缺省值为 500，可以用 update 命令改变这个值。

另一个与块高速缓存管理相关的是 update 命令，它不仅仅是一个命令，还是一个后台进程。当以超级用户的身份运行时（在系统初始化时），它将周期性调用系统服务例程将老的“ 脏 ”缓冲区中内容“ 冲刷 ”到磁盘上去。它所完成的这个工作与 bdflush 类似，不同之处在于,当一个“ 脏 ”缓冲区完成这个操作后，它将把写入到磁盘上的时间标记到 buffer_head 结构中。update 每次运行时它将在系统的所有“ 脏 ”缓冲区中查找那些“ 冲刷 ”时间已经超

过一定期限的，这些过期缓冲区都要被写回磁盘。

8.3.2　索引节点高速缓存

VFS 也用了一个高速缓存来加快对索引节点的访问，和块高速缓存不同的一点是每个缓冲区不用再分为两个部分了，因为 inode 结构中已经有了类似于块高速缓存中缓冲区首部的域。索引节点高速缓存的实现代码全部在 fs/inode.c，这部分代码并没有随着内核版本的变化做更多的修改。

1．索引节点链表

每个索引节点可能处于哈希表中，也可能同时处于下列“类型”链表的一种中：

- “in_use”有效的索引节点，即 i_count > 0 且 i_nlink > 0（参看前面的 inode 结构）
- “dirty”类似于 “in_use” ，但还“脏”；
- “unused”有效的索引节点但还没使用，即 i_count = 0。

这几个链表定义如下：

```
static LIST_HEAD (inode_in_use) ;
static LIST_HEAD (inode_unused) ;
static struct list_head *inode_hashtable;
    static LIST_HEAD (anon_hash_chain) ; /* for inodes with NULL i_sb */
```

因此，索引节点高速缓存的结构概述如下。

- 全局哈希表 inode_hashtable，其中哈希值是根据每个超级块指针的值和 32 位索引节点号而得。对没有超级块的索引节点（inode->i_sb == NULL），则将其加入到 anon_hash_chain 链表的首部。例如，net/socket.c 中 sock_alloc()函数，通过调用 fs/inode.c 中 get_empty_inode()创建的套接字是一个匿名索引节点，这个节点就加入到了 anon_hash_chain 链表。
- 正在使用的索引节点链表。全局变量 inode_in_use 指向该链表中的首元素和尾元素。函数 get_empty_inode()获得一个空节点，get_new_inode()获得一个新节点，通过这两个函数新分配的索引节点就加入到这个链表中。
- 未用索引节点链表。全局变量 inode_unused 的 next 域 和 prev 域分别指向该链表中的首元素和尾元素。
- 脏索引节点链表。由相应超级块的 s_dirty 域指向该链表中的首元素和尾元素。
- 对 inode 对象的缓存，定义如下：

```
static kmem_cache_t * inode_cachep
```

这是一个 Slab 缓存，用于分配和释放索引节点对象。

索引节点的 i_hash 域指向哈希表，i_list 指向 in_use、unused 或 dirty 某个链表。所有这些链表都受单个自旋锁 inode_lock 的保护，其定义如下：

```
/*
* A simple spinlock to protect the list manipulations.
*
```

```
* NOTE! You also have to own the lock if you change
* the i_state of an inode while it is in use..
*/
static spinlock_t inode_lock = SPIN_LOCK_UNLOCKED;
```

索引节点高速缓存的初始化是由 inode_init()实现的，而这个函数是在系统启动时由 init/main.c 中的 start_kernel()函数调用的。inode_init（）只有一个参数，表示索引节点高速缓存所使用的物理页面数。因此，索引节点高速缓存可以根据可用物理内存的大小来进行配置，例如，如果物理内存足够大，就可以创建一个大的哈希表。

索引节点状态的信息存放在数据结构 inodes_stat_t 中，在 fs/fs.h 中定义如下：

```
struct inodes_stat_t {
     int nr_inodes;
     int nr_unused;
     int dummy[5];
 };
extern struct inodes_stat_t inodes_stat
```

用户程序可以通过/proc/sys/fs/inode-nr 和 /proc/sys/fs/inode-state 获得索引节点高速缓存中索引节点总数及未用索引节点数。

2．索引节点高速缓存的工作过程

为了帮助大家理解索引节点高速缓存如何工作，我们来跟踪一下在打开 Ext2 文件系统的一个常规文件时，相应索引节点的作用。

```
fd = open ("file", O_RDONLY) ;
close (fd) ;
```

open()系统调用是由 fs/open.c 中的 sys_open 函数实现的，而真正的工作是由 fs/open.c 中的 filp_open()函数完成的，filp_open()函数如下：

```
    struct file *filp_open (const char * filename, int flags, int mode)
{
        int namei_flags, error;
         struct nameidata nd;

        namei_flags = flags;
         if  ( (namei_flags+1)  & O_ACCMODE)
                 namei_flags++;
         if  (namei_flags & O_TRUNC)
                namei_flags |= 2;

        error = open_namei (filename, namei_flags, mode, &nd) ;
         if  (!error)
                 return dentry_open (nd.dentry, nd.mnt, flags) ;

         return ERR_PTR (error) ;
}
```

其中 nameidata 结构在 fs.h 中定义如下：

```
struct nameidata {
         struct dentry *dentry;
         struct vfsmount *mnt;
         struct qstr last;
```

```
        unsigned int flags;
        int last_type;
};
```

这个数据结构是临时性的，其中，我们主要关注 dentry 和 mnt 域。dentry 结构我们已经在前面介绍过,而 vfsmount 结构记录着所属文件系统的安装信息,例如文件系统的安装点、文件系统的根节点等。

filp_open（）主要调用以下两个函数。

（1）open_namei()：填充目标文件所在目录的 dentry 结构和所在文件系统的 vfsmount 结构。在 dentry 结构中 dentry->d_inode 就指向目标文件的索引节点。这个函数比较复杂和庞大，在此为了突出主题，后面我们只介绍与主题相关的内容。

（2）dentry_open()：建立目标文件的一个“上下文”，即 file 数据结构，并让它与当前进程的 task_strrct 结构挂上钩。同时，在这个函数中，调用了具体文件系统的打开函数，即 f_op->open()。该函数返回指向新建立的 file 结构的指针。

open_namei()函数通过 path_walk()与目录项高速缓存（即目录项哈希表）打交道，而 path_walk()又调用具体文件系统的 inode_operations->lookup()方法；该方法从磁盘找到并读入当前节点的目录项，然后通过 iget(sb，ino)，根据索引节点号从磁盘读入相应索引节点并在内存建立起相应的 inode 结构，这就到了我们讨论的索引节点高速缓存。

当索引节点读入内存后，通过调用 d_add(dentry，inode)，就将 dentry 结构和 inode 结构之间的链接关系建立起来。两个数据结构之间的联系是双向的。一方面，dentry 结构中的指针 d_inode 指向 inode 结构，这是一对一的关系，因为一个目录项只对应着一个文件。反之则不然，同一个文件可以有多个不同的文件名或路径（通过系统调用 link()建立，注意与符号连接的区别，那是由 symlink()建立的），所以从 inode 结构到 dentry 结构的方向是一对多的关系。因此,inode 结构的 i_ dentry 是个队列,dentry 结构通过其队列头部 d_alias 挂入相应 inode 结构的队列中。

为了进一步说明索引节点高速缓存，我们来进一步考察 iget（)。当我们打开一个文件时，就调用了 iget（）函数，而 iget 真正调用的是 iget4(sb，ino，NULL，NULL)函数，该函数代码如下：

```
struct inode *iget4 (struct super_block *sb, unsigned long ino, find_inode_t find_actor,
void *opaque)
{
        struct list_head * head = inode_hashtable + hash (sb,ino) ;
        struct inode * inode;

        spin_lock (&inode_lock) ;
        inode = find_inode (sb, ino, head, find_actor, opaque) ;
        if  (inode)  {
                __iget (inode) ;
                spin_unlock (&inode_lock) ;
                wait_on_inode (inode) ;
                return inode;
        }
        spin_unlock (&inode_lock) ;

        /*
```

```
         * get_new_inode ()  will do the right thing, re-trying the search
         * in case it had to block at any point.
        */
        return get_new_inode (sb, ino, head, find_actor, opaque) ;
}
```

下面对以上代码给出进一步的解释。

- inode 结构中有个哈希表 inode_hashtable，首先在 inode_lock 锁的保护下，通过 find_ inode 函数在哈希表中查找目标节点的 inode 结构，由于索引节点号只有在同一设备上时才是唯一的，因此，在哈希计算时要把索引节点所在设备的 super_block 结构的地址也结合进去。如果在哈希表中找到该节点，则其引用计数（i_count）加 1；如果 i_count 在增加之前为 0，说明该节点不“脏”，则该节点当前肯定处于 inode_unused list 队列中，于是，就把该节点从这个队列删除而插入 inode_in_use 队列；最后，把 inodes_stat.nr_unused 减 1。
- 如果该节点当前被加锁，则必须等待，直到解锁，以便确保 iget4()返回一个未加锁的节点。
- 如果在哈希表中没有找到该节点，说明目标节点的 inode 结构还不在内存，因此，调用 get_new_inode()从磁盘上读入相应的索引节点并建立起一个 inode 结构，并把该结构插入到哈希表中。
- 对 get_new_inode()给出进一步的说明,该函数从 slab 缓存区中分配一个新的 inode 结构，但是这个分配操作有可能出现阻塞，于是，就应当解除保护哈希表的 inode_lock 自旋锁，以便在哈希表中再次进行搜索。如果这次在哈希表中找到这个索引节点，就通过__iget 把该节点的引用计数加 1，并撤销新分配的节点；如果在哈希表中还没有找到，就使用新分配的索引节点。因此，把该索引节点的一些域先初始化为必须的值，然后调用具体文件系统的 sb->s_op->read_inode()域填充该节点的其他域。这就把我们从索引节点高速缓存带到了某个具体文件系统的代码中。当 s_op->read_inode()方法正在从磁盘读索引节点时，该节点被加锁（i_state = I_LOCK）；当 read_inode()返回时，该节点的锁被解除，并且唤醒所有等待者。

8.3.3 目录高速缓存

由于从磁盘读入一个目录项并构造相应的目录项对象需要花费大量的时间，所以，在完成对目录项对象的操作后，可能后面还要使用它，因此在内存仍保留它有重要的意义。例如，我们经常需要编辑文件，随后进行编译或编辑，然后打印或拷贝，再进行编辑，诸如此类的情况中，同一个文件需要被反复访问。

每个目录项对象属于以下 4 种状态之一。

- 空闲状态：处于该状态的目录项对象不包含有效的信息，还没有被 VFS 使用。它对应的内存区由 slab 分配器进行管理。
- 未使用状态：处于该状态的目录项对象当前还没有被内核使用。该对象的引用计数器 d_count 的值为 NULL。但其 d_inode 域仍然指向相关的索引节点。该目录项对象包含有效的信息，但为了在必要时回收内存，它的内容可能被丢弃。

- 正在使用状态：处于该状态的目录项对象当前正在被内核使用。该对象的引用计数器 d_count 的值为正数，而其 d_inode 域指向相关的索引节点对象。该目录项对象包含有效的信息，并且不能被丢弃。
- 负状态：与目录项相关的索引节点不复存在，那是因为相应的磁盘索引节点已被删除。该目录项对象的 d_inode 域置为 NULL，但该对象仍然被保存在目录项高速缓存中，以便后续对同一文件目录名的查找操作能够快速完成，术语“负的”容易使人误解，因为根本不涉及任何负值。

为了最大限度地提高处理这些目录项对象的效率，Linux 使用目录项高速缓存，它由以下两种类型的数据结构组成。

- 处于正在使用、未使用或负状态的目录项对象的集合。
- 一个哈希表，从中能够快速获取与给定的文件名和目录名对应的目录项对象。如果访问的对象不在目录项高速缓存中，哈希函数返回一个空值。

目录项高速缓存的作用也相当于索引节点高速缓存的控制器。内核内存中，目录项可能已经不使用，但与其相关的索引节点并不被丢弃，这是由于目录项高速缓存仍在使用它们，因此，索引节点的 i_count 域不为空。于是，这些索引节点对象还保存在 RAM 中，并能够借助相应的目录项快速引用它们。

所有“未使用” 目录项对象都存放在一个“最近最少使用”的双向链表中，该链表按照插入的时间排序。换句话说，最后释放的目录项对象放在链表的首部，所以最近最少使用的目录项对象总是靠近链表的尾部。一旦目录项高速缓存的空间开始变小，内核就从链表的尾部删除元素，使得多数最近经常使用的对象得以保留。LRU 链表的首元素和尾元素的地址存放在变量 dentry_unused 中的 next 域和 prev 域中。目录项对象的 d_lru 域包含的指针指向该链表中相邻目录的对象。

每个“正在使用”的目录项对象都被插入一个双向链表中，该链表由相应索引节点对象的 i_dentry 域所指向（由于每个索引节点可能与若干硬链接关联，所以需要一个链表）。目录项对象的 d_alias 域存放链表中相邻元素的地址。

当指向相应文件的最后一个硬链接被删除后，一个“正在使用”的目录项对象可能变成“负”状态。在这种情况下，该目录项对象被移到“未使用” 目录项对象组成的 LRU 链表中。每当内核缩减目录项高速缓存时，“负”状态目录项对象就朝着 LRU 链表的尾部移动，这样一来，这些对象就逐渐被释放。

哈希表是由 dentry_hashtable 数组实现的。数组中的每个元素是一个指向链表的指针，这种链表就是把具有相同哈希表值的目录项进行散列而形成的。该数组的长度取决于系统已安装 RAM 的数量。目录项对象的 d_hash 域包含指向具有相同 hash 值的链表中的相邻元素。哈希函数产生的值是由目录及文件名的目录项对象的地址计算出的。

8.4 文件系统的注册、安装与卸载

8.4.1 文件系统的注册

当内核被编译时，就已经确定了可以支持哪些文件系统，这些文件系统在系统引导时，在 VFS 中进行注册。如果文件系统是作为内核可装载的模块，则在实际安装时进行注册，并在模块卸载时注销。每个文件系统都有一个初始化例程，它的作用就是在 VFS 中进行注册，即填写一个叫做 file_system_type 的数据结构，该结构包含了文件系统的名称以及一个指向对应的 VFS 超级块读取例程的地址，所有已注册的文件系统的 file_system_type 结构形成一个链表，为区别后面将要说到的已安装的文件系统形成的另一个链表，我们把这个链表称为注册链表。图 8.7 所示就是内核中的 file_system_type 链表，链表头由 file_systems 变量指定。

图 8.7 仅示意性地说明系统中已安装的 3 个文件系统 Ext2、proc、iso9660 其 file_system_type 结构所形成的链表。当然，系统中实际安装的文件系统要更多。

file_system_type 的数据结构在 fs.h 中定义如下：

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block * (*read_super)   (struct super_block *, void *, int) ;
        struct module *owner;
        struct file_system_type * next;
        struct list_head fs_supers;
};
```

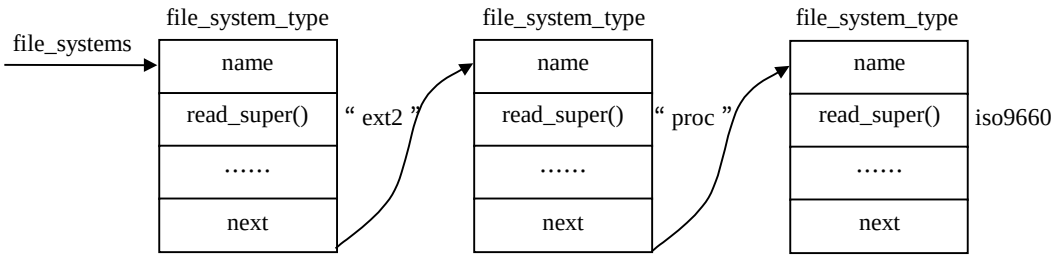

图 8.7 已注册的文件系统形成的链表

对其中几个域的说明如下。

- name：文件系统的类型名，以字符串的形式出现。
- fs_flags：指明具体文件系统的一些特性，有关标志定义于 fs.h 中：

```
     /* public flags for file_system_type */
     #define FS_REQUIRES_DEV 1
#define FS_NO_DCACHE    2 /* Only dcache the necessary things. */
     #define FS_NO_PRELIM    4 /* prevent preloading of dentries, even if
                              * FS_NO_DCACHE is not set.
                             */
```

```
#define FS_SINGLE       8 /* Filesystem that can have only one superblock */
#define FS_NOMOUNT      16 /* Never mount from userland */
#define FS_LITTER       32 /* Keeps the tree in dcache */
#define FS_ODD_RENAME   32768   /* Temporary stuff; will go away as soon
                                 * as nfs_rename() will be cleaned up
                                 */
```

对某些常用标志的说明如下。

（1）有些虚拟的文件系统，如 pipe、共享内存等，根本不允许由用户进程通过系统调用 mount()来安装。这样的文件系统其 fs_flags 中的 FS_NOMOUNT 标志位为 1。

（2）一般的文件系统类型要求有物理的设备作为其物质基础，其 fs_flags 中的 FS_REQUIRES_DEV 标志位为 1，这些文件系统如 Ext2、Minix、ufs 等。

（3）有些虚拟文件系统在安装了同类型中的第 1 个“设备”，从而创建了其超级块的 super_block 数据结构，在安装同一类型中的其他设备时就共享已存在的 super_block 结构，而不再有自己的超级块结构。此时 fs_flags 中的 FS_SINGLE 标志位为 1，表示整个文件系统只有一个超级块，而不像一般的文件系统类型那样，每个具体的设备上都有一个超级块。

- read_super：这是各种文件系统读入其超级块的函数指针。因为不同的文件系统其超级块不同，因此其读入函数也不同。
- owner：如果 file_system_type 所代表的文件系统是通过可安装模块实现的，则该指针指向代表着具体模块的 module 结构。如果文件系统是静态地链接到内核，则这个域为 NULL。实际上，你只需要把这个域置为 THIS_MODLUE （这是个一个宏），它就能自动地完成上述工作。
- next：把所有的 file_system_type 结构链接成单项链表的链接指针，变量 file_systems 指向这个链表。这个链表是一个临界资源，受 file_systems_lock 自旋读写锁的保护。
- fs_supers：这个域是 Linux 2.4.10 以后的内核版本中新增加的，这是一个双向链表。链表中的元素是超级块结构。如前说述，每个文件系统都有一个超级块，但有些文件系统可能被安装在不同的设备上，而且每个具体的设备都有一个超级块，这些超级块就形成一个双向链表。

搞清楚这个数据结构的各个域以后，就很容易理解下面的注册函数 register_filesystem()，该函数定义于 fs/super.c：

```
/**
*      register_filesystem - register a new filesystem
*      @fs: the file system structure
*
*      Adds the file system passed to the list of file systems the kernel
*      is aware of for mount and other syscalls. Returns 0 on success,
*      or a negative errno code on an error.
*
*      The &struct file_system_type that is passed is linked into the kernel
*      structures and must not be freed until the file system has been
*      unregistered.
 */

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;
}
```

find_filesystem () 函数在同一个文件中定义如下：

```
  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;
}
```

注意，对注册链表的操作必须互斥地进行，因此，对该链表的查找加了写锁 write_lock。

文件系统注册后，还可以撤消这个注册，即从注册链表中删除一个 file_system_type 结构，此后系统不再支持该种文件系统。fs/super.c 中的 unregister_filesystem()函数就是起这个作用的，它在执行成功后返回 0，如果注册链表中本来就没有指定的要删除的结构，则返回-1，其代码如下：

```
/**
*      unregister_filesystem - unregister a file system
*      @fs: filesystem to unregister
*
*      Remove a file system that was previously successfully registered
*      with the kernel. An error is returned if the file system is not found.
*      Zero is returned on a success.
*
*      Once this function has returned the &struct file_system_type structure
*      may be freed or reused.
*/

int unregister_filesystem (struct file_system_type * fs)
{
        struct file_system_type ** tmp;

         write_lock (&file_systems_lock) ;
         tmp = &file_systems;
```

```
        while (*tmp) {
                if (fs == *tmp) {
                        *tmp = fs->next;
                        fs->next = NULL;
                         write_unlock (&file_systems_lock) ;
                        return 0;
                 }
                tmp = & (*tmp) ->next;
        }
        write_unlock (&file_systems_lock) ;
        return -EINVAL;
  }
```

8.4.2　文件系统的安装

要使用一个文件系统，仅仅注册是不行的，还必须安装这个文件系统。在安装 Linux 时，硬盘上已经有一个分区安装了 Ext2 文件系统，它是作为根文件系统的，根文件系统在启动时自动安装。其实，在系统启动后你所看到的文件系统，都是在启动时安装的。如果需要自己（一般是超级用户）安装文件系统，则需要指定 3 种信息：文件系统的名称、包含文件系统的物理块设备、文件系统在已有文件系统中的安装点。例如：

```
$ mount -t iso9660 /dev/hdc /mnt/cdrom
```

其中，iso9660 就是文件系统的名称，/dev/hdc 是包含文件系统的物理块设备，/mnt/cdrom 是将要安装到的目录，即安装点。从这个例子可以看出，安装一个文件系统实际上是安装一个物理设备。

把一个文件系统（或设备）安装到一个目录点时要用到的主要数据结构为 vfsmount，定义于 include/linux/mount.h 中：

```
   struct vfsmount
{
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;    /* fs we are mounted on */
        struct dentry *mnt_mountpoint;  /* dentry of mountpoint */
        struct dentry *mnt_root;        /* root of the mounted tree */
         struct super_block *mnt_sb;     /* pointer to superblock */
         struct list_head mnt_mounts;    /* list of children, anchored here */
         struct list_head mnt_child;     /* and going through their mnt_child */
         atomic_t mnt_count;
        int mnt_flags;
         char *mnt_devname;              /* Name of device e.g. /dev/dsk/hda1 */
         struct list_head mnt_list;
};
```

下面对结构中的主要域给予进一步说明。

- 为了对系统中的所有安装点进行快速查找，内核把它们按哈希表来组织，mnt_hash 就是形成哈希表的队列指针。
- mnt_mountpoint 是指向安装点 dentry 结构的指针。而 dentry 指针指向安装点所在目录树中根目录的 dentry 结构。

- mnt_parent 是指向上一层安装点的指针。如果当前的安装点没有上一层安装点（如根设备），则这个指针为 NULL。同时，vfsmount 结构中还有 mnt_mounts 和 mnt_child 两个队列头，只要上一层 vfsmount 结构存在，就把当前 vfsmount 结构中 mnt_child 链入上一层 vfsmount 结构的 mnt_mounts 队列中。这样就形成一个设备安装的树结构，从一个 vfsmount 结构的 mnt_mounts 队列开始，可以找到所有直接或间接安装在这个安装点上的其他设备。
- mnt_sb 指向所安装设备的超级块结构 super_blaock。
- mnt_list 是指向 vfsmount 结构所形成链表的头指针。

另外，系统还定义了 vfsmntlist 变量，指向 mnt_list 队列。对这个数据结构的进一步理解请看后面文件系统安装的具体实现过程。

文件系统的安装选项，也就是 vfsmount 结构中的安装标志 mnt_flags 在 linux/fs.h 中定义如下：

```
/*
* These are the fs-independent mount-flags: up to 32 flags are supported
*/
 #define MS_RDONLY        1      /* Mount read-only */
 #define MS_NOSUID        2      /* Ignore suid and sgid bits */
 #define MS_NODEV         4     /* Disallow access to device special files */
 #define MS_NOEXEC        8      /* Disallow program execution */
 #define MS_SYNCHRONOUS  16      /* Writes are synced at once */
 #define MS_REMOUNT      32      /* Alter flags of a mounted FS */
 #define MS_MANDLOCK     64      /* Allow mandatory locks on an FS */
 #define MS_NOATIME      1024    /* Do not update access times. */
 #define MS_NODIRATIME   2048    /* Do not update directory access times */
 #define MS_BIND         4096
 #define MS_MOVE         8192
 #define MS_REC          16384
#define MS_VERBOSE      32768
#define MS_ACTIVE        (1<<30)
#define MS_NOUSER        (1<<31)
```

从定义可以看出，每个标志对应 32 位中的一位。安装标志是针对整个文件系统中的所有文件的。例如，如果 MS_NOSUID 标志为 1，则整个文件系统中所有可执行文件的 suid 标志位都不起作用了。其他安装标志的具体含义在后面介绍 do_mount()函数代码时再进一步介绍。

1．安装根文件系统

每个文件系统都有它自己的根目录，如果某个文件系统（如 Ext2）的根目录是系统目录树的根目录，那么该文件系统称为根文件系统。而其他文件系统可以安装在系统的目录树上，把这些文件系统要插入的那些目录就称为安装点。

当系统启动时，就要在变量 ROOT_DEV 中寻找包含根文件系统的磁盘主码。当编译内核或向最初的启动装入程序传递一个合适的选项时，根文件系统可以被指定为/dev 目录下的一个设备文件。类似地，根文件系统的安装标志存放在 root_mountflags 变量中。用户可以指定这些标志，这是通过对已编译的内核映像执行/sbin/rdev 外部程序，或者向最初的启动装入程序传递一个合适的选项来达到的。根文件系统的安装函数为 mount_root()。

2．安装一个常规文件系统

一旦在系统中安装了根文件系统，就可以安装其他的文件系统。每个文件系统都可以安装在系统目录树中的一个目录上。

前面我们介绍了以命令方式来安装文件系统，在用户程序中要安装一个文件系统则可以调用 mount（）系统调用。mount()系统调用在内核的实现函数为 sys_mount()，其代码在 fs/namespace.c 中。

```
asmlinkage long sys_mount(char * dev_name, char * dir_name, char * type,
                          unsigned long flags, void * data)
{
         int retval;
         unsigned long data_page;
         unsigned long type_page;
         unsigned long dev_page;
         char *dir_page;

         retval = copy_mount_options  (type, &type_page) ;
         if  (retval < 0)
                 return retval;

         dir_page = getname (dir_name) ;
         retval = PTR_ERR (dir_page) ;
         if  (IS_ERR (dir_page) )
                 goto out1;

        retval = copy_mount_options  (dev_name, &dev_page) ;
        if  (retval < 0)
                 goto out2;

         retval = copy_mount_options  (data, &data_page) ;
         if  (retval < 0)
                 goto out3;

         lock_kernel () ;
         retval = do_mount ( (char*) dev_page, dir_page,  (char*) type_page,
                           flags,  (void*) data_page) ;
         unlock_kernel () ;
         free_page (data_page) ;

out3:
         free_page (dev_page) ;
out2:
         putname (dir_page) ;
out1:
         free_page (type_page) ;
         return retval;
}
```

下面给出进一步的解释。

- 参数 dev_name 为待安装文件系统所在设备的路径名，如果不需要就为空（例如，当

待安装的是基于网络的文件系统时）；dir_name 则是安装点（空闲目录）的路径名；type 是文件系统的类型，必须是已注册文件系统的字符串名（如“ Ext2 ”，“ MSDOS ”等）；flags 是安装模式，如前面所述。data 指向一个与文件系统相关的数据结构（可以为 NULL）。

- copy_mount_options（）和 getname()函数将结构形式或字符串形式的参数值从用户空间拷贝到内核空间。这些参数值的长度均以一个页面为限，但是 getname()在复制时遇到字符串结尾符“ \0 ”就停止，并返回指向该字符串的指针；而 copy_mount_options（）则拷贝整个页面，并返回该页面的起始地址。

该函数调用的主要函数为 do_mount()，do_mount()执行期间要加内核锁，不过这个锁是针对 SMP，我们暂不考虑。do_mount()的实现代码在 fs/namespace.c 中：

```
long do_mount (char * dev_name, char * dir_name, char *type_page,
                  unsigned long flags, void *data_page)
{
        struct nameidata nd;
        int retval = 0;
        int mnt_flags = 0;

        /* Discard magic */
         if  ( (flags & MS_MGC_MSK)  == MS_MGC_VAL)
                flags &= ~MS_MGC_MSK;

         /* Basic sanity checks */

         if  (!dir_name || !*dir_name || !memchr (dir_name, 0, PAGE_SIZE) )
                 return -EINVAL;
         if  (dev_name && !memchr (dev_name, 0, PAGE_SIZE) )
                 return -EINVAL;

        /* Separate the per-mountpoint flags */
         if  (flags & MS_NOSUID)
                 mnt_flags |= MNT_NOSUID;
         if  (flags & MS_NODEV)
                 mnt_flags |= MNT_NODEV;
         if  (flags & MS_NOEXEC)
                mnt_flags |= MNT_NOEXEC;
        flags &= ~ (MS_NOSUID|MS_NOEXEC|MS_NODEV) ;

        /* ... and get the mountpoint */
         if  (path_init (dir_name, LOOKUP_FOLLOW|LOOKUP_POSITIVE, &nd) )
                retval = path_walk (dir_name, &nd) ;
         if  (retval)
                 return retval;

        if  (flags & MS_REMOUNT)
                retval = do_remount (&nd, flags & ~MS_REMOUNT, mnt_flags,
                                     data_page) ;
         else if  (flags & MS_BIND)
                 retval = do_loopback (&nd, dev_name, flags & MS_REC) ;
         else if  (flags & MS_MOVE)
                 retval = do_move_mount (&nd, dev_name) ;
```

```
        else
                retval = do_add_mount (&nd, type_page, flags, mnt_flags,
                                       dev_name, data_page) ;
        path_release (&nd) ;
        return retval;
}
```

下面对函数中的主要代码给予解释。

- MS_MGC_VAL 和 MS_MGC_MSK 是在以前的版本中定义的安装标志和掩码，现在的安装标志中已经不使用这些魔数了，因此，当还有这个魔数时，则丢弃它。
- 对参数 dir_name 和 dev_name 进行基本检查，注意“ !dir_name ” 和“ !*dir_name ”的不同，前者指指向字符串的指针为不为空，而后者指字符串不为空。memchr()函数在指定长度的字符串中寻找指定的字符，如果字符串中没有结尾符“ \0 ”，也是一种错误。前面已说过，对于基于网络的文件系统 dev_name 可以为空。
- 把安装标志为 MS_NOSUID、MS_NOEXEC 和 MS_NODEV 的 3 个标志位从 flags 分离出来，放在局部安装标志变量 mnt_flags 中。
- 函数 path_init () 和 path_walk () 寻找安装点的 dentry 数据结构，找到的 dentry 结构存放在局部变量 nd 的 dentry 域中。
- 如果 flags 中的 MS_REMOUNT 标志位为 1，就表示所要求的只是改变一个原已安装设备的安装方式，例如从“ 只读“ 安装方式改为“ 可写 ”安装方式，这是通过调用 do_remount () 函数完成的。
- 如果 flags 中的 MS_BIND 标志位为 1，就表示把一个“ 回接 ”设备捆绑到另一个对象上。回接设备是一种特殊的设备（虚拟设备），而实际上并不是一种真正设备，而是一种机制，这种机制提供了把回接设备回接到某个可访问的常规文件或块设备的手段。通常在/dev 目录中有/dev/loop0 和/dev/loop1 两个回接设备文件。调用 do_loopback () 来实现回接设备的安装。
- 如果 flags 中的 MS_MOVE 标志位为 1，就表示把一个已安装的设备可以移到另一个安装点，这是通过调用 do_move_mount () 函数来实现的。
- 如果不是以上 3 种情况，那就是一般的安装请求，于是把安装点加入到目录树中，这是通过调用 do_add_mount () 函数来实现的，而 do_add_mount () 首先调用 do_kern_mount () 函数形成一个安装点，该函数的代码在 fs/super.c 中：

```
    struct vfsmount *do_kern_mount (char *type, int flags, char *name, void *data)
{
        struct file_system_type * fstype;
        struct vfsmount *mnt = NULL;
        struct super_block *sb;

        if  (!type || !memchr (type, 0, PAGE_SIZE) )
                return ERR_PTR (-EINVAL) ;

        /* we need capabilities... */
        if  (!capable (CAP_SYS_ADMIN) )
                return ERR_PTR (-EPERM) ;

        /* ... filesystem driver... */
```

```
        fstype = get_fs_type (type) ;
       if  (!fstype)
              return ERR_PTR (-ENODEV) ;

        /* ... allocated vfsmount... */
       mnt = alloc_vfsmnt () ;
        if  (!mnt)  {
               mnt = ERR_PTR (-ENOMEM) ;
              goto fs_out;
        }
        set_devname (mnt, name) ;
        /* get locked superblock */
        if  (fstype->fs_flags & FS_REQUIRES_DEV)
               sb = get_sb_bdev (fstype, name, flags, data) ;
        else if  (fstype->fs_flags & FS_SINGLE)
               sb = get_sb_single (fstype, flags, data) ;
        else
               sb = get_sb_nodev (fstype, flags, data) ;

        if  (IS_ERR (sb) )  {
               free_vfsmnt (mnt) ;
               mnt =  (struct vfsmount *) sb;
               goto fs_out;
       }
        if  (fstype->fs_flags & FS_NOMOUNT)
               sb->s_flags |= MS_NOUSER;

        mnt->mnt_sb = sb;
        mnt->mnt_root = dget (sb->s_root) ;
        mnt->mnt_mountpoint = mnt->mnt_root;
        mnt->mnt_parent = mnt;
        up_write (&sb->s_umount) ;
fs_out:
        put_filesystem (fstype) ;
        return mnt;
}
```

对该函数的解释如下。

- 只有系统管理员才具有安装一个设备的权力,因此首先要检查当前进程是否具有这种权限。
- get_fs_type () 函数根据具体文件系统的类型名在 file_system_file 链表中找到相应的结构。
- alloc_vfsmnt () 函数调用 slab 分配器给类型为 vfsmount 结构的局部变量 mnt 分配空间，并进行相应的初始化。
- set_devname () 函数设置设备名。
- 一般的文件系统类型要求有物理的设备作为其物质基础，如果 fs_flags 中的 FS_REQUIRES_DEV 标志位为 1，说明这就是正常的文件系统类型，如 Ext2、mnix 等。对于这种文件系统类型，通过调用 get_sb_bdev () 从待安装设备上读其超级块。
- 如果 fs_flags 中的 FS_SINGLE 标志位为 1，说明整个文件系统只有一个类型，也就

是说，这是一种虚拟的文件系统类型。这种文件类型在安装了同类型的第 1 个“ 设备 ” ，通过调用 get_sb_single （） 创建了超级块 super_block 结构后，再安装的同类型设备就共享这个数据结构。但是像 Ext2 这样的文件系统类型在每个具体设备上都有一个超级块。

- 还有些文件系统类型的 fs_flags 中的 FS_NOMOUNT、FS_REUIRE_DEV 以及 FS_SINGLE 标志位全都为 0，那么这些所谓的文件系统其实是“ 虚拟的 ”，通常只是用来实现某种机制或者规程，所以根本就没有对应的物理设备。对于这样的文件系统类型都是通过 get_sb_nodev （） 来生成一个 super_block 结构的。
- 如果文件类型 fs_flags 的 FS_NOMOUNT 标志位为 1，说明根本就没有用户进行安装，因此，把超级块中的 MS_NOUSER 标志位置 1。
- mnt->mnt_sb 指向所安装设备的超级块 sb；mnt->mnt_root 指向其超级块的根 b->s_root，dget （） 函数把 dentry 的引用计数 count 加 1；mnt->mnt_mountpoint 也指向超级块的根，而 mnt->mnt_parent 指向自己。到此为止，仅仅形成了一个安装点，但还没有把这个安装点挂接在目录树上。

下面我们来看 do_add_mount （） 的代码：

```
   static int do_add_mount (struct nameidata *nd, char *type, int flags,
                            int mnt_flags, char *name, void *data)
{
         struct vfsmount *mnt = do_kern_mount (type, flags, name, data) ;
         int err = PTR_ERR (mnt) ;

         if  (IS_ERR (mnt) )
                goto out;

         down (&mount_sem) ;
         /* Something was mounted here while we slept */
         while (d_mountpoint (nd->dentry)  && follow_down (&nd->mnt, &nd->dentry) )
                 ;
         err = -EINVAL;
         if  (!check_mnt (nd->mnt) )
                goto unlock;

        /* Refuse the same filesystem on the same mount point */
        err = -EBUSY;
         if  (nd->mnt->mnt_sb == mnt->mnt_sb && nd->mnt->mnt_root == nd->dentry)
                 goto unlock;

        mnt->mnt_flags = mnt_flags;
        err = graft_tree (mnt, nd) ;
unlock:
         up (&mount_sem) ;
         mntput (mnt) ;
 out:
         return err;
   }
```

下面是对以上代码的解释。

- 首先检查 do_kern_mount （） 所形成的安装点是否有效。
- 在 do_mount()函数中，path_init()和 path_walk()函数已经找到了安装点的 dentry

结构、inode 结构以及 vfsmount 结构，并存放在类型为 nameidata 的局部变量 nd 中，在 do_add_mount()中通过参数传递了过来。

- 但是，在 do_kern_mount () 函数中从设备上读入超级块的过程是个较为漫长的过程，当前进程在等待从设备上读入超级块的过程中几乎可肯定要睡眠，这样就有可能另一个进程捷足先登抢先将另一个设备安装到了同一个安装点上。d_mountpoint () 函数就是检查是否发生了这种情况。如果确实发生了这种情况，其对策就是调用 follow_down () 前进到已安装设备的根节点，并且通过 while 循环进一步检测新的安装点，直到找到一个空安装点为止。
- 如果在同一个安装点上要安装两个同样的文件系统，则出错。
- 调用 graft_tree () 把 mnt 与安装树挂接起来，完成最终的安装。
- 至此，设备的安装就完成了。

8.4.3 文件系统的卸载

如果文件系统中的文件当前正在使用，该文件系统是不能被卸载的。如果文件系统中的文件或目录正在使用，则 VFS 索引节点高速缓存中可能包含相应的 VFS 索引节点。根据文件系统所在设备的标识符，检查在索引节点高速缓存中是否有来自该文件系统的 VFS 索引节点，如果有且使用计数大于 0，则说明该文件系统正在被使用，因此，该文件系统不能被卸载。否则，查看对应的 VFS 超级块，如果该文件系统的 VFS 超级块标志为“脏”，则必须将超级块信息写回磁盘。上述过程结束之后，对应的 VFS 超级块被释放，vfsmount 数据结构将从 vfsmntlist 链表中断开并被释放。具体的实现代码为 fs/super.c 中的 sysy_umount () 函数，在此不再进行详细的讨论。

8.5 限额机制

设想一下，如果对用户不采取某些限制措施，则任一用户可能用完文件系统的所有可用空间，在某些环境中，这种情况是不能接受的。Linux 中为了限制一个用户可获得的文件资源数量，使用了限额机制。限额机制对一个用户可分配的文件数目和可使用的磁盘空间设置了限制。系统管理员能分别为每一用户设置限额。

限制有软限制和硬限制之分，硬限制是绝对不允许超过的，而软限制则由系统管理员来确定。当用户占用的资源超过软限制时，系统开始启动定时机制，并在用户的终端上显示警告信息，但并不终止用户进程的运行，如果在规定时间内用户没有采取积极措施改正这一问题，则软限制将被强迫转化为硬限制，用户的进程将被中止。这个规定的时间可以由系统管理员来设置，默认为一周，在 include/linux/quota.h 中有如下宏定义：

```
#define MAX_IQ_TIME           604800 /*  (7*24*60*60)  =1 周 */
#define MAX_DQ_TIME           604800  /*  (7*24*60*60)  =1 周 */
```

分别是超过索引节点软限制的最长允许时间和超过块的软限制的最长允许时间。

下面看一下在 Linux 中，限额机制具体是怎样实现的。

首先，在编译内核时，要选择“支持限额机制”一项，默认情况下，Linux 不使用限额

机制。如果使用了限额机制，每一个安装的文件系统都与一个限额文件相联系，限额文件通常驻留在文件系统的根目录里，它实际是一组以用户标识号来索引的限额记录，每个限额记录可称为一个限额块，其数据结构如下（在 inclue/linux/quota.h 中定义）：

```
struct dqblk {
    __u32 dqb_bhardlimit;   /* 块的硬限制*/
    __u32 dqb_bsoftlimit;     /* 块的软限制 */
    __u32 dqb_curblocks;      /* 当前占有的块数 */
    __u32 dqb_ihardlimit;    /* 索引节点的硬限制 */
    __u32 dqb_isoftlimit;    /* 索引节点的软限制 */
    __u32 dqb_curinodes;     /* 当前占用的索引节点数 */
    time_t dqb_btime;       /* 块的软限制变为硬限制前，剩余的警告次数*/
    time_t dqb_itime;     /* 索引节点的软限制变为硬限制前，剩余的警告次数 */
};
```

限额块调入内存后，用哈希表来管理，这就要用到另一个结构 dquot（也在 inclue/linux/quota.h 中定义），其数据结构如下：

```
   struct dquot {
         struct list_head dq_hash;       /*在内存的哈希表*/
         struct list_head dq_inuse;      /*正在使用的限额块组成的链表*/
         struct list_head dq_free;       /* 空闲限额块组成的链表 */
         wait_queue_head_t dq_wait_lock; /* 指向加锁限额块的等待队列*/
         wait_queue_head_t dq_wait_free; /* 指向未用限额块的等待队列*/
         int dq_count;                   /* 引用计数 */

         /* fields after this point are cleared when invalidating */
         struct super_block *dq_sb;      /* superblock this applies to */
         unsigned int dq_id;             /* ID this applies to  (uid, gid)  */
         kdev_t dq_dev;                  /* Device this applies to */
         short dq_type;                  /* Type of quota */
         short dq_flags;                 /* See DQ_* */
         unsigned long dq_referenced;    /* Number of times this dquot was
                                            referenced during its lifetime */
         struct dqblk dq_dqb;            /* Diskquota usage */
};
```

哈希表是用文件系统所在的设备号和用户标识号为散列关键值的。

vfs 的索引节点结构中有一个指向 dquot 结构的指针。也就是说，调入内存的索引节点都要与相应的 dquot 结构联系，dquot 结构中，引用计数就是反映了当前有几个索引节点与之联系，只有在引用计数为 0 时，才将该结构放入空闲链表中。

如果使用了限额机制，则当有新的块分配请求，系统要以文件拥有者的标识号为索引去查找限额文件中相应的限额块，如果限额并没有满，则接受请求，并把它加入使用计数中。如果已达到或超过限额，则拒绝请求，并返回错误信息。

下面是为一个用户设置限额的具体实现方法。

（1）检查/etc/fstab ，如果没有提供限额机制，则该文件类似下面这样。

```
/dev/hda1 / Ext2 defaults 1 1
/dev/hda2 /home Ext2 defaults 1 2
```

（2）为了设置用户 user1 在目录 /home/user1 下所占用磁盘空间和使用文件数的限额，将/etc/fstab 改成像下面这样。

```
/dev/hda1 / Ext2 defaults 1 1
/dev/hda2 /home Ext2 defaults,usrquota 1 2
```

(3) 以 root 登录，在需要设置限额的分区目录下创建空文件 quota.user。

```
#touch /home/quota.user
#chmod 600 /home/quota.user
```

(4) 重新启动机器。

(5) 为指定的用户分配磁盘空间和最多存放文件个数。

```
# edquota -u user1
Quota for user user1
/dev/hda2: blocks in use:10, limits  (soft=4000,hard=5400)
inodes in use : 400, limits  (soft=1200,hard=1600)
```

你只需对 limits 那一项进行修改即可。

用#quota user1 可以查看用户 user1 的磁盘限额设置情况。

8.6 具体文件系统举例

如前所述，每种文件系统类型都有个 file_system_type 结构，而结构中的 fs_flags 则由各种标志位组成，这些标志位表明了具体文件系统类型的特性，也决定着这种文件系统的安装过程。以物理设备为基础的常规文件系统类型（如 Ext2、Minix 等），由用户进程通过系统调用 mount()来安装，而有些没有物理设备对应的文件系统（如 pipe、共享内存区等），由内核通过 kern_mount()来安装。

内核代码中提供了两个用来建立 file_system_type 结构的宏，其定义在 fs.h 中：

```
   #define DECLARE_FSTYPE (var,type,read,flags)  \
struct file_system_type var = { \
         name:           type, \
         read_super:     read, \
         fs_flags:       flags, \
        owner:          THIS_MODULE, \
}

 #define DECLARE_FSTYPE_DEV (var,type,read)  \
         DECLARE_FSTYPE (var,type,read,FS_REQUIRES_DEV)
```

一般常规的文件系统类型都通过 DECLARE_FSTYPE_DEV 建立其结构，因为它们的 FS_REQUIRES_DEV 标志位为 1，而其他标志位为 0。相比之下，特殊的、虚拟的文件系统类型大多直接通过 DECLARE_FSTYPE 建立其结构，因为它们的 fs_flags 是特殊的。

8.6.1 管道文件系统 pipefs

pipefs 是一种简单的、虚拟的文件系统类型，因为它没有对应的物理设备，因此其安装时不需要块设备，在第十章将看到，大部分文件系统是以模块的形式来实现的。该文件系统相关的代码在 fs/pipe.c 中：

```
static DECLARE_FSTYPE (pipe_fs_type, "pipefs", pipefs_read_super,
                FS_NOMOUNT|FS_SINGLE) ;
```

```
static int __init init_pipe_fs (void)
{
        int err = register_filesystem (&pipe_fs_type) ;
        if  (!err)  {
                pipe_mnt = kern_mount (&pipe_fs_type) ;
                err = PTR_ERR (pipe_mnt) ;
                if  (IS_ERR (pipe_mnt) )
                        unregister_filesystem (&pipe_fs_type) ;
                else
                        err = 0;
        }
        return err;
}

static void __exit exit_pipe_fs (void)
{
       unregister_filesystem (&pipe_fs_type) ;
       mntput (pipe_mnt) ;
    }

module_init (init_pipe_fs)
module_exit (exit_pipe_fs)
```

pipefs 文件系统是作为一个模块来安装的，其中 module_init () 是模块的初始化函数，module_exit () 是模块的卸载函数，其更详细的解释将在第十章给出。

从 DECLARE_FSTYPE () 宏定义可以看出，pipefs 文件系统的 FS_NOMOUNT 和 FS_SINGLE 标志位为 1，这就意味着该文件系统不能从用户空间进行安装，并且在整个系统范围内只有一个超级块。FS_SINGLE 标志也意味着在通过 register_filesystem()成功地注册了该文件系统后，应该通过 kern_mount()来安装。

register_filesystem()函数把 pipe_fs_type 链接到 file_systems 链表，因此，你可以通过读/proc/filesystems 找到“ pipefs ”入口点，在那里，“ nodev ”标志表示没有设置 FS_REQUIRES_DEV 标志，即该文件系统没有对应的物理设备。

kern_mount()类似于 do_mount()，用来安装 pipefs 文件系统。当安装出现错误时，则调用 unregister_filesystem () 把 pipe_fs_type 从 file_systems 链表中拆除。

现在，pipefs 文件系统已被注册，并成为内核中的一个模块，从此我们就可以使用它了。pipefs 文件系统的入口点就是 pipe()系统调用，其内核实现函数为 sys_pipe()，而真正的工作是调用 do_pipe()函数来完成的，其代码在/fs/pipe.c 中，我们同时给出了对代码的注释。

```
  int do_pipe (int *fd)
   {
        struct qstr this;
        char name[32];
        struct dentry *dentry;
        struct inode * inode;
            struct file *f1, *f2;    /*进程对每个已打开文件的操作是通过 file 结构进行的。一个管道实际上就是一个存在于内存的文件，对这个文件的操作要通过两个已打开的文件进行，f1、f2 分别代表该管道的两端。*/
```

```
        int error;
         int i,j;

        error = -ENFILE;
       f1 = get_empty_filp () ;          /*管道两端各分配一个 file 结构*/

        if  (!f1)
                goto no_files;

        f2 = get_empty_filp () ;
        if  (!f2)
                goto close_f1;

              inode = get_pipe_inode () ;  /*每个文件都有一个 inode 结构。由于管道文件在管道创建之前并不存在，因此，在创建管道时临时创建一个 inode 结构。*/

        if  (!inode)
                goto close_f12;

        error = get_unused_fd () ;       ／* 分配打开文件号*/
      if  (error < 0)
                goto close_f12_inode;
        i = error;

        error = get_unused_fd () ;
        if  (error < 0)
                goto close_f12_inode_i;
        j = error;

        error = -ENOMEM;
        sprintf (name, "[%lu]", inode->i_ino) ;
        this.name = name;
        this.len = strlen (name) ;
        this.hash = inode->i_ino; /* will go */
        dentry = d_alloc(pipe_mnt->mnt_sb->s_root, &this);  /* File 结构中有个指针 f_dentry 指向所打开文件的目录项 dentry 结构，而 dentry 中有个指针指向相应的 inode 结构。所以，调用 d_alloc() 分配一个目录项是为了把 file 结构与 inode 结构联系起来。*

        if  (!dentry)
                goto close_f12_inode_i_j;
       dentry->d_op = &pipefs_dentry_operations;

        d_add (dentry, inode) ;  ／*使已分配的 inode 结构与已分配的目录项结构挂勾*／

       f1->f_vfsmnt = f2->f_vfsmnt = mntget (mntget (pipe_mnt) ) ;   /* pipe_mnt 就是在 init_pipe_fs()中所获得的指向 vfsmount 结构的指针，因为这个结构多了两个使用者，因此调用两次 mntget()使其引用计数加 2 *／

       f1->f_dentry = f2->f_dentry = dget (dentry) ; ／*让两个已打开文件中的 f_dentry 指针都指向这个目录项，并使目录项的引用计数加 1*／

                /* read file */
```

```
            f1->f_pos = f2->f_pos = 0;
            f1->f_flags = O_RDONLY;
            f1->f_op = &read_pipe_fops;
            f1->f_mode = 1;
            f1->f_version = 0;

            /* write file */
            f2->f_flags = O_WRONLY;
            f2->f_op = &write_pipe_fops;
            f2->f_mode = 2;
            f2->f_version = 0;

                fd_install (i, f1) ;                    /*将已打开文件结构与分配得的打开文件
号相关联（打开文件号只在一个进程的范围内有效）。*/

            fd_install (j, f2) ;
            fd[0] = i;    ／*使得 fd[0]为管道读出端的打开文件号*/
            fd[1] = j;    ／*使得 fd[1]为管道写出端的打开文件号*/

           return 0;
       ／*以下为释放各种资源*/
     close_f12_inode_i_j:
            put_unused_fd (j) ;
     close_f12_inode_i:
            put_unused_fd (i) ;
     close_f12_inode:
            free_page ( (unsigned long)  PIPE_BASE (*inode) ) ;
            kfree (inode->i_pipe) ;
            inode->i_pipe = NULL;
            iput (inode) ;
     close_f12:
            put_filp (f2) ;
     close_f1:
            put_filp (f1) ;
    no_files:
           return error;
     }
```

下面对管道的单向性再做进一步的说明。从代码看出，把 f1 一端设置成“只读（O_RDONLY）”，另一端则设置成“只写（O_WRONLY）”。同时，两端的文件操作也分别设置成 read_pipe_fops 和 write_pipe_fops，其定义于 pipe.c 中：

```
        struct file_operations read_pipe_fops = {
            llseek:         pipe_lseek,
            read:           pipe_read,
            write:          bad_pipe_w,
            poll:           pipe_poll,
            ioctl:          pipe_ioctl,
            open:           pipe_read_open,
           release:        pipe_read_release,
     };

    struct file_operations write_pipe_fops = {
```

```
        llseek:         pipe_lseek,
        read:           bad_pipe_r,
        write:          pipe_write,
        poll:           pipe_poll,
        ioctl:          pipe_ioctl,
        open:           pipe_write_open,
        release:        pipe_write_release,
};
```

在 read_pipe_fops () 中的写操作函数为 bad_pipe_w ()，而在 write_pipe_fops () 中的读操作函数为 bad_pipe_r ()，这两个函数分别返回一个出错代码。尽管代表着管道两端的两个已打开文件一个只能读，一个只能写。但是，另一方面，这两个逻辑上已打开的文件指向同一个 inode，即用作管道的缓冲区，显然，这个缓冲区既支持读也支持写。这进一步说明了 file 、inode 及 dentry 之间的不同和联系。

8.6.2 磁盘文件系统 BFS

BFS 是 Berkeley fast File System 的简写，即柏克莱快速文件系统，是一种简单的基于磁盘的文件系统。BFS 将磁盘的分区分割为许多的柱面群，每一个柱面群依磁盘的大小，包含了 1～32 个相邻的柱面。BFS 模块的相关代码在 fs/bfs/inode.c 中：

```
static DECLARE_FSTYPE_DEV (bfs_fs_type, "bfs", bfs_read_super) ;

static int __init init_bfs_fs (void)
{
        return register_filesystem (&bfs_fs_type) ;
}

static void __exit exit_bfs_fs (void)
{
        unregister_filesystem (&bfs_fs_type) ;
}

module_init (init_bfs_fs)
module_exit (exit_bfs_fs)
```

宏 DECLARE_FSTYPE_DEV()把 bfs_fs_type 文件类型的标志置为 FS_REQUIRES_DEV，表示 BFS 需要一个实际的块设备来进行安装。

模块的初始化函数调用 register_filesystem () 向 VFS 注册该文件系统，调用 unregister_filesystem () 注销该文件系统。

一旦文件系统被注册，我们就可以安装它，在安装一个文件系统时就会调用 fs_type->read_super()方法来读其超级块，具体到 BFS 文件系统则是调用 fs/bfs/inode.c 中的 bfs_read_super()函数，该函数的原型为：

```
static struct super_block * bfs_read_super (struct super_block * s,
    void * data, int silent)
```

其中参数 s 是指向 super_block 的数据结构，在调用这个函数之前，该结构已被进行了一定的初始化，例如其 s_dev 域已经有了具体设备的设备号。但是，结构中的大部分内容还没有设置。而这个函数要做的工作就是从磁盘上读入该文件系统的超级块，并根据其内容设

置这个 super_block 数据结构。另一个指针 data 的使用，因文件系统而异，对于 BFS 文件系统来说并没有使用这个参数。而参数 silent，则表示在读超级块的过程中是否详细地报告出错信息。

现在，我们又回到在 VFS 级的调用函数 fs/super.c 中的 read_super()。在 read_super() 成功返回以后，VFS 就获得了对该文件系统模块的引用。

接下来，我们来考察以下在对该文件系统进行 I/O 操作时都发生些什么事情。我们已经考察过，iget(sb, ino)根据索引节点号从磁盘读入相应索引节点并在内存建立起相应的 inode 结构，在建立 inode 结构的过程中，还要做其他的事情，比如读 inode->i_op 和 inode->i_fop；打开一个文件就意味着把 inode->i_fop 拷贝到 file->f_op。

8.7　文件系统的系统调用

有关文件系统的系统调用有十几个，下面选其中几个简单地加以介绍。

8.7.1　open 系统调用

进程要访问一个文件，必须首先获得一个文件描述符，这是通过 open 系统调用来完成的。文件描述符是有限的资源，所以在不用时应该及时释放。

该系统调用是用来获得欲访问文件的文件描述符，如果文件并不存在，则还可以用它来创建一个新文件。其函数为 sys_open()，在 fs/open.c 中定义，函数如下：

```
asmlinkage long sys_open (const char * filename, int flags, int mode)
{
        char * tmp;
       int fd, error;

#if BITS_PER_LONG != 32
        flags |= O_LARGEFILE;
#endif
        tmp = getname (filename) ;
        fd = PTR_ERR (tmp) ;
        if  (!IS_ERR (tmp) )  {
                fd = get_unused_fd () ;
                if  (fd >= 0)  {
                       struct file *f = filp_open (tmp, flags, mode) ;
                        error = PTR_ERR (f) ;
                        if  (IS_ERR (f) )
                                goto out_error;
                        fd_install (fd, f) ;
                }
out:
                putname (tmp) ;
        }
        return fd;
```

```
out_error:
        put_unused_fd (fd) ;
        fd = error;
         goto out;
}
```

1．入口参数

(1) filename：　　欲打开文件的路径。

(2) flags：　　规定如何打开该文件，它必须取下列 3 个值之一。

O_RDONLY　　以只读方式打开文件

O_WRONLY　　以只写方式打开文件

O_RDWR　　以读和写的方式打开文件

此外，还可以用或运算对下列标志值任意组合。

O_CREAT　　打开文件，如果文件不存在则建立文件

O_EXCL　　如果已经置 O_CREAT 且文件存在，则强制 open()失败

O_TRUNC　　将文件的长度截为 0

O_APPEND　　强制 write()从文件尾开始

对于终端文件，这 4 个标志是没有任何意义的，另提供了两个新的标志。

O_NOCTTY　　停止这个终端作为控制终端

O_NONBLOCK　　使 open()、read()、write()不被阻塞。

这些标志的符号名称在/include/asmi386/fcntl.h 中定义。

(3) mode：这个参数实际上是可选的，如果用 open()创建一个新文件，则要用到该参数，它用来规定对该文件的所有者、文件的用户组和系统中其他用户的访问权限位。它用或运算对下列符号常量建立所需的组合。

S_IRUSR　　文件所有者的读权限位

S_IWUSR　　文件所有者的写权限位

S_IXUSR　　文件所有者的执行权限位

S_IRGRP　　文件用户组的读权限位

S_IWGRP　　文件用户组的写权限位

S_IXGRP　　文件用户组的执行权限位

S_IROTH　　文件其他用户的读权限位

S_IWOTH　　文件其他用户的写权限位

S_IXOTH　　文件其他用户的执行权限位

这些标志的符号名称在/include/linux/stat.h 中定义。

2．出口参数

返回一个文件描述符。

3．执行过程

sys_open()主要是调用 filp_open ()，这个函数也在 fs/open.c 中，这已在前面做过介绍。

从当前进程的 files_struct 结构的 fd 数组中找到第 1 个未使用项，使其指向 file 结构，将该项的下标作为文件描述符返回，结束。

在以上过程中，如果出错，则将分配的文件描述符、file 结构收回，inode 也被释放，函数返回一个负数以示出错，其中 PTR_ERR（）和 IS_ERR（）是出错处理函数，下一章将给予介绍。

8.7.2　read 系统调用

如果通过 open 调用获得一个文件描述符，而且是用 O_RDONLY 或 O_RDWR 标志打开的，就可以用 read 系统调用从该文件中读取字节。其内核函数在 fs/read_write.c 中定义：

```
asmlinkage ssize_t sys_read (unsigned int fd, char * buf, size_t count)
{
        ssize_t ret;
        struct file * file;

        ret = -EBADF;
        file = fget (fd) ;
        if  (file)  {
                if  (file->f_mode & FMODE_READ)  {
                        ret      =      locks_verify_area      (     FLOCK_VERIFY_READ,
file->f_dentry->d_inode,
                                                file, file->f_pos, count) ;
                        if  (!ret)  {
                                ssize_t  (*read)  (struct file *, char *, size_t, loff_t *) ;
                                ret = -EINVAL;
                                if  (file->f_op &&  (read = file->f_op->read)  != NULL)
                                        ret = read (file, buf, count, &file->f_pos) ;
                         }
                }
                 if  (ret > 0)
                        dnotify_parent (file->f_dentry, DN_ACCESS) ;
                fput (file) ;
       }
         return ret;
}
```

1．入口参数

(1) fd：要读的文件的文件描述符。

(2) buf：指向用户内存区中用来存储将读取字节的区域的指针。

(3) count：欲读的字节数。

2．出口参数

返回一个整数。在出错时返回-1；否则返回所读的字节数，通常这个数就是 count 值，但如果请求的字节数超过剩余的字节数，则返回实际读的字节数，例如文件的当前位置在文件尾，则返回值为 0。

3．执行过程

(1) 函数 fget()根据打开文件号 fd 找到该文件已打开文件的 file 结构。

(2) 取得了目标文件的 file 结构指针，并确认文件是以只读方式打开后，还要检查文件从当前位置 f_pos 开始的 count 个字节是否对读操作加上了“强制锁”，这是通过调用 locks_verify_area（）函数完成的，其代码在 fs.h 中。

(3) 通过了对强制锁的检查后，就是读操作本身了。可想而知，不同的文件系统有不同的读操作，具体的文件系统通过 file_operations 结构提供用于读操作的函数指针。就 Ext2 文件系统来说，它有两个这样的结构，一个是 Ext2_file_operations，另一个是 Ext2_dir_operations，视操作的目标为文件或目录而选择其一，在打开文件时，操作结构就安装在其 file 结构中。对于常规文件，这个函数指针指向 generic_file_read（），其代码在 mm/filemap.c 中。

(4) 如果读操作的返回值大于 0，说明出错，则调用 dnotify_parent（）报告错误，并释放文件描述符、file 结构、inode 结构。

8.7.3 fcntl 系统调用

这个系统调用功能比较多，可以执行多种操作，其内核函数在 fs/fcntl.c 中定义。

1．入口参数

(1) fd：欲访问文件的文件描述符。

(2) cmd：要执行的操作的命令，这个参数定义了 10 个标志，下面介绍其中的 5 个，F_DUPFD、F_GETFD、F_SETFD、F_GETFL 和 F_SETFL

(3) arg：可选，主要根据 cmd 来决定是否需要。

2．出口参数：根据第二个参数（cmd）的不同，这个返回值也不一样

3．函数功能

如果第二个参数（cmd）取值是 F_DUPFD，则进行复制文件描述符的操作。它需要用到第三个参数 arg，这时 arg 是一个文件描述符，fcntl(fd，F_DUPFD，arg)在 files_struct 结构中从指定的 arg 开始搜索空闲的文件描述符，找到第一个后，将 fd 的内容复制进来，然后将新找到的文件描述符返回。

第二个参数（cmd）取值是 F_GETFD，则返回 files_struct 结构中 close_on_exec 的值。

无需第三个参数。

第二个参数（cmd）取值是 F_SETFD，则需要第三个参数，若 arg 最低位为 1，则对 close_on_exec 置位，否则清除 close_on_exec。

第二个参数（cmd）取值是 F_GETFL，则用来读取 open 系统调用第二个参数设置的标志，即文件的打开方式（O_RDONLY，O_WRONLY，O_APPEND 等），它不需要第三个参数。实际上上这时函数返回的是 file 结构中的 flags 域。

第二个参数（cmd）取值是 F_SETFL，则用来对 open 系统调用第二个参数设置的标志进行改变，但是它只能对 O_APPEND 和 O_NONBLOCK 标志进行改变，这时需要第三个参数 arg，用来确定如何改变。函数返回 0 表示操作成功，否则返回-1，并置一个错

8.8　Linux 2.4 文件系统的移植问题

Linux 内核源代码 2.2.x 版本及 2.4.x 版本之间有一定的变化，我们把 2.2.x 版本叫旧版，把 2.4.x 叫新版。下面给出文件系统的变化。

1．模块处理方式发生了变化

模块的初始化方式有所变化，在旧版本中的初始化方式如下（我们把某个文件系统叫做 myfs）：

```
static struct file_system_type myfs_fs_type =
      { "myfs", FS_REQUIRES_DEV, myfs_read_super, NULL };

__initfunc (int init_myfs_fs (void) )  {
   return register_filesystem (&myfs_fs_type) ;
}

  #ifdef MODULE
 EXPORT_NO_SYMBOLS;

  int init_module (void)  {
      return init_myfs_fs () ;
 }

 void cleanup_module (void)  {
      unregister_filesystem (&myfs_fs_type) ;
 }
 #endif
```

另外，MOD_INC_USE_COUNT 宏在文件系统的 read_super()中被调用，而 MOD_DEC_USE_COUNT 宏在 put_super()中被调用。新版中的初始化方法如下：

```
static DECLARE_FSTYPE_DEV (myfs_fs_type, "myfs", myfs_read_super) ;

static int __init init_myfs_fs (void)  {
   return register_filesystem (&myfs_fs_type) ;
}
```

```
static void __exit exit_myfs_fs (void)  {
   unregister_filesystem (&myfs_fs_type) ;
}

EXPORT_NO_SYMBOLS;
module_init (init_myfs_fs) ;
module_exit (exit_myfs_fs) ;
```

MOD_XXX_USE_COUNT 现在由 VFS 文件系统在注册时进行处理。从代码可以看出，新版本的处理方式更具可读性，而本质上并没有多大变化。

2．大文件的支持（Large File Support，LFS）

VFS 现在支持 64 位文件（仅适用于 x86 和 Sparc 平台）：

- 使用 64 位类型 loff_t
- 但内核还不支持 64 位的 getrlimit()和 setrlimit()系统调用；
- glibc 库支持 getrlimit64() 和 setrlimit64 ()。

3．新的错误处理方式

旧版中错误处理如下：

```
if  (!dir || !dir->i_nlink)  {
   *err = -EPERM;
   return NULL;
   }
```

在新版中，对错误的处理调用了 ERR_PTR()，PTR_ERR() 和 IS_ERR()函数：

```
if  (!dir || !dir->i_nlink)
    return ERR_PTR (-EPERM) ;
```

这几个错误处理函数定义于 include/linux/fs.h 中：

```
 static inline void *ERR_PTR (long error)
{
        return  (void *)  error;
}

 static inline long PTR_ERR (const void *ptr)
 {
        return  (long)  ptr;
 }

 static inline long IS_ERR (const void *ptr)
 {
        return  (unsigned long) ptr >  (unsigned long) -1000L;
}
```

4．inode 结构

在旧版中，file_operations 在 inode_operations 结构中定义，而在新版中已移到 inode 结构中，即：

```
struct file_operations *i_fop;
```

其引用形式为：inode->i_fop。

另外，inode 中 count 类型的定义也有所改变：

旧版：int i_count;

新版：atomic_t i_count;

这种类型的定义是与体系结构独立的，因此，对这些变量的访问就是原子操作，例如对变量 v 的读和设置函数为：

```
atomic_read (v) ;
atomic_set (v, value) ;
```

这种形式也应用于 file 结构 和 dentry 结构。

5．dentry 高速缓存

在旧版中，删除一个 dentry 的函数形式为：

```
void (*d_delete) (struct dentry *) ;
```

在新版中，删除一个 dentry 的函数形式为：

```
int (*d_delete) (struct dentry *) ;
```

返回一个整数表示删除的成功与否。

在旧版中，分配一个根目录项的函数形式为：

```
struct dentry *d_alloc_root (struct inode *, struct dentry *) ;
```

在新版中，分配一个根目录项的函数形式为：

```
struct dentry *d_alloc_root (struct inode *) ;
```

6．VFS 操作

在旧版中，filldir 帮助函数的形式为：

```
typedef int (*filldir_t) (void *, const char *, int, off_t, ino_t) ;
```

在新版中，filldir 帮助函数的形式为：

```
typedef int (*filldir_t) (void *, const char *, int, off_t, ino_t, unsigned) ;
```

新增加的参数表示文件类型，其定义于 fs.h 中：

```
/*
 * File types
 */
#define DT_UNKNOWN     0
#define DT_FIFO        1
#define DT_CHR         2
#define DT_DIR         4
#define DT_BLK         6
#define DT_REG         8
#define DT_LNK         10
#define DT_SOCK        12
#define DT_WHT         14
```

7．各种操作结构的指定方式发生变化

所有操作结构的指定方式发生了变化，旧版中的形式为：

```
struct file_operations myfs_file_operations = {
   myfs_file_lseek,
   generic_file_read,
```

```
        generic_file_write,
        NULL,
        NULL,
        myfs_ioctl,
        NULL,
    };
```

新版中的形式为：

```
    struct file_operations myfs_file_operations = {
        llseek: myfs_file_lseek,
        read: generic_file_read,
        write: generic_file_write,
        ioctl: myfs_ioctl,
    };
```

这种形式使用了 GNU C 语言的扩展形式，符合 ISO C99 标准，C99 标准指定的初始化者的形式为：

```
    struct foo {
        int foo;
        long bar;
    };

    struct foo x = { .bar = 3, .foo = 4 };
```

8. VFS 的 file_operations

在 fsync () 函数中增加了一个新的参数；如果这个参数被设置，则不干预对时间标记的刷新：

旧版：int (*fsync) (struct file *, struct dentry *);

新版：int (*fsync) (struct file *, struct dentry *, int);

另外，原来 file_operations 中的两个操作：

```
int  (*check_media_change)   (kdev_t dev) ;
          int  (*revalidate)   (kdev_t dev) ;
```

被移到一个新的结构 block_device_operations：

```
    struct block_device_operations {
        int  (*open)   (struct inode *, struct file *) ;
        int  (*release)   (struct inode *, struct file *) ;
        int  (*ioctl)  (struct inode *, struct file *, unsigned, unsigned long) ;
        int  (*check_media_change)  (kdev_t) ;
        int  (*revalidate)   (kdev_t) ;
    };
```

这个新的结构成为 inode 结构中一个新的域：

新版：struct block_device *i_bdev;

于是在 block_device 中有一个指向这个操作结构的指针：

```
    struct block_device {
        struct list_head bd_hash;
        atomic_t bd_count;
        dev_t bd_dev;
        atomic_t bd_openers;
```

```
    const struct
    block_device_operations *bd_op;
    struct semaphore bd_sem;
  };
```

另外，file_operations（）中还增加了以下两个函数。在不用保持大内核锁的情况下，所有文件系统都可以调用这两个函数。这两个函数还实现了 readv() 和 writev()系统调用。

```
ssize_t  (*readv)   (struct file *, const struct iovec *, unsigned long, loff_t *) ;
ssize_t  (*writev)   (struct file *, const struct iovec *, unsigned long, loff_t *) ;
```

9. VFS 的 inode_operations

file_operations 指针从 inode_operations 结构移到 inode 结构。

inode_operations 结构中，follow_link（）函数的形式也发生了变化：

旧版中：struct dentry * (*follow_link) (struct dentry *, struct dentry *, unsigned int);

新版中：int (*follow_link) (struct dentry *, struct nameidata *);

从调用形式可以看出，第一个参数仍然相同，新版参数的 nameidata 结构中包含了旧版中的最后两个参数：

```
struct nameidata {
  struct dentry *dentry;
  struct vfsmount *mnt;
  struct qstr last;
  unsigned int flags;
  int last_type;
};
```

inode_operations 结构中还增加了以下两个函数：

```
int  (*setattr)   (struct dentry *, struct iattr *) ;
int  (*getattr)   (struct dentry *, struct iattr *) ;
```

这两个函数（实际上仅仅是 setattr()）取代了旧版中 superblock 操作中的 notify_change()函数。另外，下列 5 个函数已经全部被取消：

```
int  (*readpage)   (struct file *, struct page *) ;
int  (*writepage)   (struct file *, struct page *) ;
int  (*updatepage)   (struct file *, struct page *, unsigned long, unsigned int, int) ;
int  (*bmap)   (struct inode *,int) ;
int  (*smap)   (struct inode *,int) ;
```

10. VFS 的 super_operations

在 super_operations 结构中，write_inode（）函数的形式有所变化：

旧版中：void (*write_inode) (struct inode *);

新版中：void (*write_inode) (struct inode *, int);

新增加的参数是一个布尔标志，用来决定是否把 inode 同步地写到磁盘。

相反，statfs() 函数中少一个参数，因为 statfs 结构的大小是没有必要的。

旧版中：int (*statfs) (struct super_block *, struct statfs *, int);

新版中：int (*statfs) (struct super_block *, struct statfs *);

最后，notify_change（）函数被 getattr()和 setattr()函数取代。