VFS函数都在fs/目录下

namespace.c

open.c

read\_write.c

file.c

sync.c

。。。

与文件系统相关的函数声明基本上在include/linux/fs.h中

**文件系统的挂载：**

fs/namespace.c:

SYSCALL\_DEFINE5(mount, ...): sys\_mount()

     do\_mount()

          do\_new\_mount()

               fs\_get\_type();//通过调用find\_filesystem()根据文件系统名字查找注册的文件系统类型

               vfs\_kern\_mount()

                    mount\_fs()

                         type->mount();//struct file\_system\_type \*type

对于ext2文件系统，其type->mount = ext2\_mount，如下：

struct file\_system\_type{

...

.mount = ext2\_mount

...

}

挂载一般的块设备文件系统，使用mount\_bdev()

ext2\_mount = mount\_bdev

mount\_bdev();//fs/super.c

挂载基于内存的文件系统，使用mount\_nodev()

对于pmfs，pmfs\_mount = mount\_nodev

mount\_nodev()

     sget();//在本fs\_type中查找或创建一个新的super\_block，不过由mount\_nodev()调用的sget()只调用alloc\_super()创建一个新的super block，并将其加入超级块链表super\_blocks（获得文件系统对应在VFS层中的super　block后，便可以从中获得文件系统的超级块：super\_block->s\_fs\_info->xxxfs\_super\_block）

     fill\_super();//是各个文件系统自己定义的函数，作为参数传入mount\_nodev()，如对于pmfs，fill\_super = pmfs\_fill\_super分配inode，并根据inode生成根目录的dentry，将dentry和inode通过d\_add()加入到HASH表中

每当在某个目录加载一个文件系统时，系统会为此文件系统生成几个结构：

file\_system\_type, vfs\_mount, super\_block, 根dentry, inode

文件内容的缓存叫cache，inode之类的块缓存叫buffer

**虚拟文件系统ＶＦＳ：**

１、在同一个目录结构中，可以挂载若干种不同的文件系统。ＶＦＳ隐藏了它们的实现细节，为使用者提供统一的接口

２、目录结构本身并不是绝对的，每个进程可能会看到不一样的目录结构。目录结构是由“地址空间namespace”来描述的，不同的进程可能拥有不同的namespace，不同的namespace可能有着不同的目录结构（因为它们可能挂载了不同的文件系统）

ＶＦＳ的使用者是进程（用户访问文件系统总是需要启动进程）。描述进程的task\_struct结构定义在include/linux/sched.h中

struct task\_struct{

     ...

     struct fs\_struct \*fs;//filesystem information

     struct files\_struct \*files;//open file information

     ...

}

由task\_struct的结构可知，其中包含了文件系统的结构信息fs\_struct，以及打开的文件信息结构files\_struct，它描述了进程已打开的文件集合。

struct files\_struct{

...

struct fdtable \_\_rcu \*fdt;

...

}

struct fdtable{

...

unsigned long \*open\_fds;//已使用的fd位图

...

}

files\_struct结构维护了一个已打开文件所对应的file结构的指针数组，数组下标被用作用户程序操作已打开文件的句柄，即fd。files\_struct还维护着已使用的fd位图（files\_struct->fdt->open\_fds），以便在需要打开文件时，为其分配一个未使用的fd。  
file结构是一个已打开文件实例，每一个打开的文件都有一个与之相对应的struct file结构。用户程序通过fd索引到对应的file结构，再执行file结构的f\_op中对应的操作即可（如read、write）

struct file结构中包含有两个结构指针

struct file{

...

struct file\_operations \*f\_op;

...

struct address\_space \*f\_mapping;

...

}

file\_operations结构中包含的是与文件相关的各种操作：如打开、读、写等文件操作

struct address\_space:

该结构中有一个radix tree，包含所有该文件使用的页面，

还有一个struct address\_space\_operations结构指针（a\_ops），

address\_space\_operations结构中包含的是与页面（page）相关的操作：如读写页面，交换区页面等

要打开一个文件，首先需要文件路径，这个路径被‘／’拆分成多级，每一级都是一个文件（目录也是文件）

在寻找这个文件路径的一开始，我们需要一个起点。如果文件路径以‘／’开关，则以根目录为起点；否则以当前路径为起点。这两个可能的起点都保存在进程的task\_struct所对应的fs\_struct结构中，每个文件在目录结构中由目录项（dentry）结构来表示，“起点”本身也是一个dentry结构

struct fs\_struct{

     ...

     struct path root, pwd;

     ...

}

struct path{

     ...

     struct dentry \*dentry;

}

在shell中执行“cd”命令时，实际上就是改变了fs\_struct结构中代表当前路径的那个dentry

进程也可以通过chroot系统调用来改变fs\_struct结构中代表根路径的那个dentry。这样一来，这个dentry之上的那些路径对该进程将不可见。

Dentry:

作为文件的索引结构，若干dentry描绘了一个树型的目录结构，这就是用户所看到的目录结构。每个dentry结构中都有指向一个索引节点（inode）结构，后者才是实际描述这个文件信息的结构。而多个dentry可以指向同一个inode，这样就实现了link。

struct dentry{

     ...

     struct dentry \*d\_parent;//parent directory

     struct inode \*d\_inode;//

     const struct dentry\_operations \*d\_op;

     struct list\_head d\_subdirs;//子节点

     ...

}

dentry结构中实现了一组方法（struct dentry\_operations \*）d\_op，主要是用于匹配子节点。dentry实现了一个散列表，以全球查找子节点。

d\_op可能随文件系统类型的不同而不同，比如，散列方法可能不同，节点的匹配方法也可能不同（有的文件系统文件名大小写敏感，有的则不）。

寻找文件路径的过程就是在这个dentry树中不断查找子dentry，直到找到路径中的最后一个dentry的过程。

虽然dentry树描绘了文件系统的目录结构，但是，这些dentry结构并不是常驻内存的。整个目录结构可能会非常大，以至于内存根本装不下。

初始状态下，系统中只有代表根目录的dentry和它所指向的inode（这是在根文件系统挂载时生成的）。此时要打开一个文件，文件路径中对应的节点都是不存在的，根目录的dentry无法找到需要的子节点（它现在还没有子节点）。这时候就要通过inode－>i\_op中的lookup方法来寻找需要的inode的子节点，找到后（此时inode已被载入内存），再创建一个dentry与之联上。dentry的存在加速了对inode的查询。

既然整个目录结构可能不能全部载入内存，在内存中生成的dentry将在无从使用时被释放。d\_count字段记录了dentry的引用计数，引用为０时，dentry将被释放。

这里所谓的释放dentry并不是直接销毁并回收，而是将dentry放入一个ＬＲＵ队列（与对应的超级块相关联）。当队列过大，或系统内存紧缺时，最近最少使用的一些dentry才真正被释放。

这个ＬＲＵ队列就像是一个缓存池，加速了对重复的路径的访问。而当dentry被真正释放时，它所对应的inode将被减引用。如果引用为０，inode也被释放。

当寻找一个文件路径时，对于其中经历的每一个节点，有三种情况：

（１）对应的dentry引用计数尚未减为０，它们还在dentry树中，直接使用即可

（２）如果对应的dentry不在dentry树中，则试图从ＬＲＵ队列中寻找。ＬＲＵ队列中的dentry同时被散列到一个散列表中，以便查找。查找到需要的dentry后，这个dentry被从ＬＲＵ队列中拿出来，重新添加到dentry树中

（３）如果对应有dentry在ＬＲＵ队列中也找不到，则只好去文件系统的存储介质里面查找inode了。找到以后dentry被创建，并添加以dentry树中。

文件系统中的inode：

struct inode{

  ...

  int i\_mode; //文件类型（目录、块设备、字符设备、普通文件、权限等）  
     char i\_nlink; //文件硬链接数  
     char i\_uid; //属主ID（UID）  
     char i\_gid; //属主组ID（GID）  
     char i\_size; //文件大小  
     ...

}

关于链接文件：  
链接文件有两种：硬链接与软链接，硬链接相当于文件别名，用ls查看时会显示为正常文件，  
它们的inode号相同，会使i\_nlink增加，而在删除一个文件时，其i\_nlink数会减1，  
当其减为0时表示文件可以删除了，硬链接不能跨文件系统  
软链接的i\_mode字段中表示其为"l"（用ls查看时），创建软链接不会增加i\_nlink数，软链接就像快捷方式，  
其中存储的是其所指向文件的路径，可以跨文件系统，

文件读写流程：

fs/read\_write.c:

SYSCALL\_DEFINE3(read, unsigned int, fd, char \_\_user \*, buf, size\_t, count)

经过宏展开后就得到函数sys\_read()  
sys\_read():通过fd得到对应的file结构，然后调用vfs\_read()，最后根据vfs\_read()的返回值(读取的文件大小)修改文件f的位置指针pos;

vfs\_read():各种权限及文件锁的检查，然后调用file->f\_op->read()（若不存在则调用do\_sync\_read()）；file->f\_op是从对应的inode->i\_fop而来，而inode->i\_fop是由对应的文件系统类型在生成这个inode时赋予的，定义在相应的文件系统中的struct file\_operations结构中。在ext2中，file->f\_op->read = do\_sync\_read

*（下面这些都是传统文件系统的调用路径，而对于SCM上的文件系统，一般都是直接调用文件系统自己的read函数）*

do\_sync\_read()：是完成一次同步读，f\_op->aio\_read()是完成一次异步读。do\_sync\_read()会调用f\_op->aio\_read()，若其返回值是-EIOCBQUEUED，则进程睡眠，直到读完即可。但实际上对应磁盘文件的读，f\_op->aio\_read一般不会返回-EIOCBQUEUED，除非是设置了O\_DIRECT标志，或者是对于一些特殊的文件系统（如NFS这样的网络文件系统）；

f\_op->aio\_read()：它通常是由generic\_file\_aio\_read()或者其封装来实现的；

generic\_file\_aio\_read()：一次异步读可能包含多个读操作（对应于readv系统调用），对于其中的每一个，调用do\_generic\_file\_read()；

do\_generic\_file\_read()：不断地调用find\_get\_page()从radix树里面查找是否存在对应的page，且该页可用。是则从page里面读出所需要的数据，然后返回。否则调用file->f\_mapping->a\_ops->readpage()去读相应页面

file->f\_mapping->a\_ops是一个struct address\_space\_operations结构

file->f\_mapping是从对应inode->i\_mapping而来，而inode->i\_mapping->a\_ops是由对应的文件系统类型在生成这个inode时赋予的。而各个文件系统类型提供的a\_ops->readpage()函数一般是mpage\_readpage()函数的封装（ext2文件系统中ext2\_readpage=mpage\_readpage）

mpage\_readpage()：调用do\_mpage\_readpage()构造一个bio，然后调用mpage\_bio\_submit()将该bio提交

do\_mpage\_readpage()：根据page->index确定需要读的磁盘扇区号，然后构造一组bio。其中需要使用文件系统类型提供的get\_block函数（作为函数参数传递进来）来对应需要读取的磁盘扇区号。

mpage\_bio\_submit()：设置bio的结束回调bio->bio\_end\_io = mpage\_end\_io，然后调用submit\_bio()提交这组bio

到达块设备层block　device：block/blk-core.c

submit\_bio()：调用generic\_make\_request()将bio提交到磁盘驱动维护的请求队列中

generic\_make\_request()：对于每一个bio，通过bdev\_get\_queue()获取其对应的块设备文件对应的磁盘对象的请求队列q = bio->bi\_bdev->bd\_disk->queue，调用q->make\_request\_fn()将bio添加到队列

q->make\_request\_fn()：设备驱动程序在其初始化时会初始化这个request\_queue结构，并且设置q->make\_request\_fn和q->request\_fn。前者用于将一个bio组装成request添加到request\_queue，后者用于处理request\_queue中的请求。一般情况下，设备驱动通过调用blk\_init\_queue来初始化request\_queue，q->request\_fn需要给定，而q->make\_request\_fn使用了默认的\_\_make\_request。

sys\_write():与sys\_read()类似，依次调用vfs\_write()、do\_sync\_write()、f\_op->aio\_write()、generic\_file\_aio\_write()。

generic\_file\_aio\_write()：调用\_\_generic\_file\_aio\_write()来进行写的处理

\_\_generic\_file\_aio\_write()：调用generic\_file\_buffered\_write()

generic\_file\_buffered\_write()：调用generic\_perform\_write()

generic\_perform\_write()：一次异步写可能包含多个操作（对应于writev系统调用），对于其中牵涉的每一个page，调用file->f\_mapping->a\_ops->write\_begin()准备好需要写的磁盘高速缓存页面，然后将需要写的数据拷入其中，最后调用file->f\_mapping->a\_ops->write\_end()完成写

file->f\_mapping->a\_ops->write\_begin()函数一般是block\_write\_begin()函数的封装（ext2中ext2\_write\_begin()就是调用的block\_write\_begin()）

而file->f\_mapping->a\_ops->write\_end()函数一般是generic\_write\_end()函数的封装（ext2中就是这样）

block\_write\_begin()：调用grab\_cache\_page\_write\_begin()在radix树里面查找要被写的page，如果不存在则创建一个。然后调用\_\_block\_write\_begin()为这个page准备一组buffer\_head结构，用于描述组成这个page的数据块（利用其中的信息，可以生成对应的bio结构）

generic\_write\_end()：调用block\_write\_end()提交写请求，然后设置page的dirty标记

block\_write\_end()：调用\_\_block\_commit\_write()为page中的每一个buffer\_head结构设置dirty标记。

至此，write调用就要返回了。

文件打开流程：fs/open.c

SYSCALL\_DEFINE3(open,...)

sys\_open(): 调用do\_sys\_open();

do\_sys\_open(): 调用get\_unused\_fd\_flags()获得未使用的文件描述符，然后调用do\_filp\_open()进行路径查找，打开文件，将文件信息存储到struct file结构中

3.11内核的open系统调用流程：

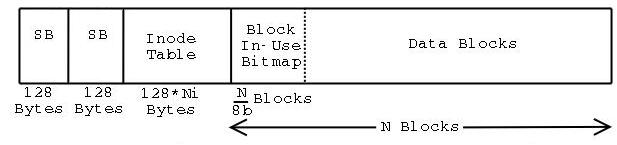
DCACHE\_OP\*全部为空  
  
  
do\_sys\_open();  
    do\_filp\_open();  
        path\_openat();//lookup rcu  
        path\_openat();//normal lookup  
        path\_openat();//lookup reval  
            path\_init();  
            link\_path\_walk();  
                不断walk\_component();  
                    lookup\_fast();  
                        //\_\_d\_lookup\_rcu();  
                        \_\_d\_lookup();//查找hash表找对应dentry,FIXME: 其内容实质上也是rcu，与\_\_d\_lookup\_rcu()重复？  
                    没找到就调用  
                    lookup\_slow();//  
                        \_\_lookup\_hash();  
                            lookup\_dcache();//  
                                d\_lookup();  
                                    \_\_d\_lookup();  
                                没找到就调用  
                                d\_alloc();//分配一个新的dentry，并且need\_lookup置位  
                            若dcache中没找到（need\_lookup置位）就调用  
                            lookup\_real();调用特定文件系统的lookup()，在文件系统中根据名字查找dentry  
            do\_last();  
                lookup\_open();//查找路径中的最后一个名称，也有可能创建它  
                    lookup\_dcache();//在dcache中查找  
                    若没找到就调用  
                    lookup\_real();//去文件系统中查找  
                    如果需要创建(O\_CREAT)  
                    vfs\_create();//调用具体文件系统的create()方法创建对应inode和dentry

**PRAMFS：**  
Protected and Persistent RAM Filesystem

The Persistent/Protected RAM Special Filesystem (PRAMFS) is a read/write filesystem that has been designed to address these issues. PRAMFS is targeted to fast I/O memory, and if the memory is non-volatile, the filesystem will be persistent.

PRAMFS使用direct I/O，并且文件IO总是同步的，当进行文件传输时不需要阻塞当前进程，因为PRAMFS存在于RAM中

PRAMFS is write protected. The page table entries that map the backing-store RAM are normally marked read-only.



struct pram\_dentry{

     \_\_be64 d\_next;//next dentry in this directory

     \_\_be64 d\_prev;//previous dentry in this directory

     \_\_be64 d\_parent;//parent directory

     char d\_name[0];//相当于字符串指针const char \*d\_name，指向文件名字符串

}

struct pram\_inode{

     ...

     union{

          struct {

               \_\_b64 row\_block;//ptr to row block of 2D block pointer array

          }

          struct {

               \_\_be64 head;//first entry in this directory

               \_\_be64 tail;//last entry in this directory

          }

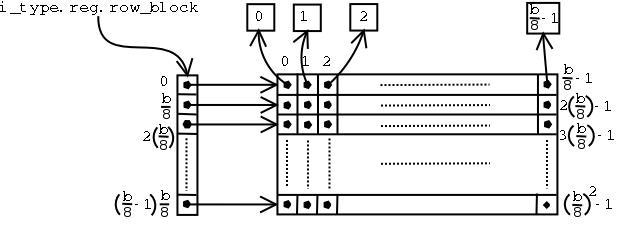
          ...

     }i\_type;

     struct pram\_dentry i\_d;//inode结构中直接包含目录项

}

PRAMFS中，只有普通文件拥有数据块（目录没有）。inode结构中的i\_type.reg.row\_block　指向一张２维数据块指针表的头，i\_type.reg.row\_block相当于EXT2文件系统中的i\_block[13]，即二次间接索引：row\_block指向一个block，该block又可容纳ｂ/８个数据块指针，每个数据块指针都指向一个新的block，新的block中存储着最终的数据块指针，其结构如下图所示：



一个目录下的所有inode都用一个双向链表连接，d\_next和d\_prev分别指向后一个和前一个目录，第一个inode的i\_prev和最后一个inode的i\_next为空

父目录的inode有两个指针，i\_type.dir.head和i\_type.dir.tail，分别指向双向链表的头和尾

pram\_add\_link()函数中

memcpy(pi->i\_d.d\_name, name, namelen);

pi->i\_d.d\_name[namelen] = '\0';

说明了PRAMFS是将文件名存储在pram\_dentry.d\_name指向的字符串中。  
创建inode

pram\_create()

     pram\_new\_inode()

     pram\_add\_nondir()

          pram\_add\_link()

          d\_instantiate();//填充inode信息