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

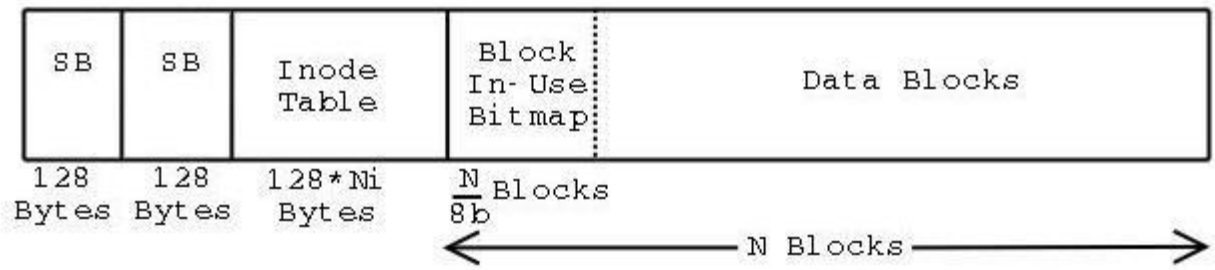
PRAMFS:

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

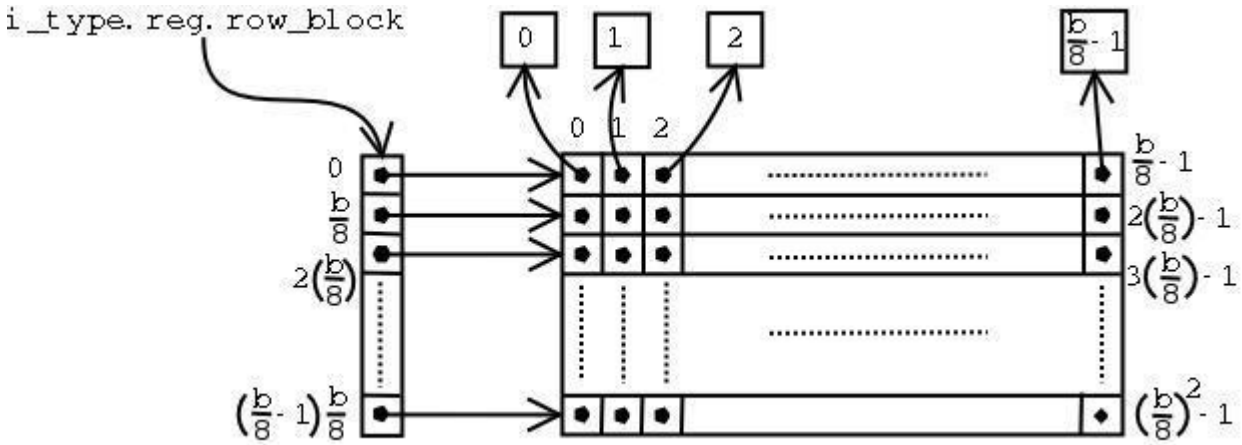
PRAMFS is write protected. The page table entries that map the backing-store RAM are normally marked read-only.




```
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
```