

Overview of the Linux Virtual File System

Original author: Richard Gooch <rgooch@atnf.csiro.au>

Last updated on June 24, 2007.

Copyright (C) 1999 Richard Gooch

Copyright (C) 2005 Pekka Enberg

This file is released under the GPLv2.

Introduction

=====

The Virtual File System (also known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs. It also provides an abstraction within the kernel which allows different filesystem implementations to coexist.

VFS system calls `open(2)`, `stat(2)`, `read(2)`, `write(2)`, `chmod(2)` and so on are called from a process context. Filesystem locking is described in the document `Documentation/filesystems/Locking`.

Directory Entry Cache (dcache)

The VFS implements the `open(2)`, `stat(2)`, `chmod(2)`, and similar system calls. The pathname argument that is passed to them is used by the VFS to search through the directory entry cache (also known as the dentry cache or dcache). This provides a very fast look-up mechanism to translate a pathname (filename) into a specific dentry. Dentries live in RAM and are never saved to disc: they exist only for performance.

The dentry cache is meant to be a view into your entire filespace. As most computers cannot fit all dentries in the RAM at the same time, some bits of the cache are missing. In order to resolve your pathname into a dentry, the VFS may have to resort to creating dentries along the way, and then loading the inode. This is done by looking up the inode.

The Inode Object

An individual dentry usually has a pointer to an inode. Inodes are filesystem objects such as regular files, directories, FIFOs and other beasts. They live either on the disc (for block device filesystems) or in the memory (for pseudo filesystems). Inodes that live on the disc are copied into the memory when required and changes to the inode are written back to disc. A single inode can be pointed to by multiple dentries (hard links, for example, do this).

To look up an inode requires that the VFS calls the `lookup()` method of the parent directory inode. This method is installed by the specific filesystem implementation that the inode lives in. Once the VFS has

the required dentry (and hence the inode), we can do all those boring things like `open(2)` the file, or `stat(2)` it to peek at the inode data. The `stat(2)` operation is fairly simple: once the VFS has the dentry, it peeks at the inode data and passes some of it back to userspace.

The File Object

Opening a file requires another operation: allocation of a file structure (this is the kernel-side implementation of file descriptors). The freshly allocated file structure is initialized with a pointer to the dentry and a set of file operation member functions. These are taken from the inode data. The `open()` file method is then called so the specific filesystem implementation can do its work. You can see that this is another switch performed by the VFS. The file structure is placed into the file descriptor table for the process.

Reading, writing and closing files (and other assorted VFS operations) is done by using the userspace file descriptor to grab the appropriate file structure, and then calling the required file structure method to do whatever is required. For as long as the file is open, it keeps the dentry in use, which in turn means that the VFS inode is still in use.

Registering and Mounting a Filesystem

To register and unregister a filesystem, use the following API functions:

```
#include <linux/fs.h>

extern int register_filesystem(struct file_system_type *);
extern int unregister_filesystem(struct file_system_type *);
```

The passed `struct file_system_type` describes your filesystem. When a request is made to mount a filesystem onto a directory in your namespace, the VFS will call the appropriate `mount()` method for the specific filesystem. New `vfsmount` referring to the tree returned by `->mount()` will be attached to the mountpoint, so that when pathname resolution reaches the mountpoint it will jump into the root of that `vfsmount`.

You can see all filesystems that are registered to the kernel in the file `/proc/filesystems`.

struct file_system_type

This describes the filesystem. As of kernel 2.6.39, the following members are defined:

```
struct file_system_type {
    const char *name;
    int fs_flags;
    struct dentry *(*mount) (struct file_system_type *, int,
        const char *, void *);
```

```

    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
};

```

name: the name of the filesystem type, such as "ext2", "iso9660", "msdos" and so on

fs_flags: various flags (i.e. FS_REQUIRES_DEV, FS_NO_DCACHE, etc.)

mount: the method to call when a new instance of this filesystem should be mounted

kill_sb: the method to call when an instance of this filesystem should be shut down

owner: for internal VFS use: you should initialize this to THIS_MODULE in most cases.

next: for internal VFS use: you should initialize this to NULL

s_lock_key, s_umount_key: lockdep-specific

The mount() method has the following arguments:

struct file_system_type *fs_type: describes the filesystem, partly initialized by the specific filesystem code

int flags: mount flags

const char *dev_name: the device name we are mounting.

void *data: arbitrary mount options, usually comes as an ASCII string (see "Mount Options" section)

The mount() method must return the root dentry of the tree requested by caller. An active reference to its superblock must be grabbed and the superblock must be locked. On failure it should return ERR_PTR(error).

The arguments match those of mount(2) and their interpretation depends on filesystem type. E.g. for block filesystems, dev_name is interpreted as block device name, that device is opened and if it contains a suitable filesystem image the method creates and initializes struct super_block accordingly, returning its root dentry to caller.

->mount() may choose to return a subtree of existing filesystem - it doesn't have to create a new one. The main result from the caller's point of view is a reference to dentry at the root of (sub)tree to be attached; creation of new superblock is a common side effect.

The most interesting member of the superblock structure that the mount() method fills in is the "s_op" field. This is a pointer to a "struct super_operations" which describes the next level of the filesystem implementation.

Usually, a filesystem uses one of the generic mount() implementations

and provides a `fill_super()` callback instead. The generic variants are:

`mount_bdev`: mount a filesystem residing on a block device

`mount_nodev`: mount a filesystem that is not backed by a device

`mount_single`: mount a filesystem which shares the instance between all mounts

A `fill_super()` callback implementation has the following arguments:

`struct super_block *sb`: the superblock structure. The callback must initialize this properly.

`void *data`: arbitrary mount options, usually comes as an ASCII string (see "Mount Options" section)

`int silent`: whether or not to be silent on error

The Superblock Object

=====

A superblock object represents a mounted filesystem.

`struct super_operations`

This describes how the VFS can manipulate the superblock of your filesystem. As of kernel 2.6.22, the following members are defined:

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);

    void (*dirty_inode) (struct inode *, int flags);
    int (*write_inode) (struct inode *, int);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*freeze_fs) (struct super_block *);
    int (*unfreeze_fs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);

    int (*show_options)(struct seq_file *, struct dentry *);

    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
    int (*nr_cached_objects)(struct super_block *);
    void (*free_cached_objects)(struct super_block *, int);
};
```

All methods are called without any locks being held, unless otherwise

noted. This means that most methods can block safely. All methods are only called from a process context (i.e. not from an interrupt handler or bottom half).

`alloc_inode`: this method is called by `inode_alloc()` to allocate memory for struct inode and initialize it. If this function is not defined, a simple 'struct inode' is allocated. Normally `alloc_inode` will be used to allocate a larger structure which contains a 'struct inode' embedded within it.

`destroy_inode`: this method is called by `destroy_inode()` to release resources allocated for struct inode. It is only required if
->`alloc_inode` was defined and simply undoes anything done by
->`alloc_inode`.

`dirty_inode`: this method is called by the VFS to mark an inode dirty.

`write_inode`: this method is called when the VFS needs to write an inode to disc. The second parameter indicates whether the write should be synchronous or not, not all filesystems check this flag.

`drop_inode`: called when the last access to the inode is dropped, with the `inode->i_lock` spinlock held.

This method should be either NULL (normal UNIX filesystem semantics) or "generic_delete_inode" (for filesystems that do not want to cache inodes - causing "delete_inode" to always be called regardless of the value of `i_nlink`)

The "generic_delete_inode()" behavior is equivalent to the old practice of using "force_delete" in the `put_inode()` case, but does not have the races that the "force_delete()" approach had.

`delete_inode`: called when the VFS wants to delete an inode

`put_super`: called when the VFS wishes to free the superblock (i.e. unmount). This is called with the superblock lock held

`sync_fs`: called when VFS is writing out all dirty data associated with a superblock. The second parameter indicates whether the method should wait until the write out has been completed. Optional.

`freeze_fs`: called when VFS is locking a filesystem and forcing it into a consistent state. This method is currently used by the Logical Volume Manager (LVM).

`unfreeze_fs`: called when VFS is unlocking a filesystem and making it writable again.

`statfs`: called when the VFS needs to get filesystem statistics.

`remount_fs`: called when the filesystem is remounted. This is called with the kernel lock held

`clear_inode`: called then the VFS clears the inode. Optional

`umount_begin`: called when the VFS is unmounting a filesystem.

show_options: called by the VFS to show mount options for
/proc/<pid>/mounts. (see "Mount Options" section)

quota_read: called by the VFS to read from filesystem quota file.

quota_write: called by the VFS to write to filesystem quota file.

nr_cached_objects: called by the sb cache shrinking function for the
filesystem to return the number of freeable cached objects it contains.
Optional.

free_cache_objects: called by the sb cache shrinking function for the
filesystem to scan the number of objects indicated to try to free them.
Optional, but any filesystem implementing this method needs to also
implement ->nr_cached_objects for it to be called correctly.

We can't do anything with any errors that the filesystem might
encountered, hence the void return type. This will never be called if
the VM is trying to reclaim under GFP_NOFS conditions, hence this
method does not need to handle that situation itself.

Implementations must include conditional reschedule calls inside any
scanning loop that is done. This allows the VFS to determine
appropriate scan batch sizes without having to worry about whether
implementations will cause holdoff problems due to large scan batch
sizes.

Whoever sets up the inode is responsible for filling in the "i_op" field. This
is a pointer to a "struct inode_operations" which describes the methods that
can be performed on individual inodes.

The Inode Object

=====

An inode object represents an object within the filesystem.

struct inode_operations

This describes how the VFS can manipulate an inode in your
filesystem. As of kernel 2.6.22, the following members are defined:

```
struct inode_operations {
    int (*create) (struct inode *,struct dentry *, umode_t, bool);
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
    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 *,umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
    int (*rename) (struct inode *, struct dentry *,
                   struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char __user *,int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    void (*put_link) (struct dentry *, struct nameidata *, void *);
```

```

int (*permission) (struct inode *, int);
int (*get_acl)(struct inode *, int);
int (*setattr) (struct dentry *, struct iattr *);
int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
ssize_t (*listxattr) (struct dentry *, char *, size_t);
int (*removexattr) (struct dentry *, const char *);
void (*update_time)(struct inode *, struct timespec *, int);
int (*atomic_open)(struct inode *, struct dentry *,
                   struct file *, unsigned open_flag,
                   umode_t create_mode, int *opened);
};

```

Again, all methods are called without any locks being held, unless otherwise noted.

create: called by the `open(2)` and `creat(2)` system calls. Only required if you want to support regular files. The dentry you get should not have an inode (i.e. it should be a negative dentry). Here you will probably call `d_instantiate()` with the dentry and the newly created inode

lookup: called when the VFS needs to look up an inode in a parent directory. The name to look for is found in the dentry. This method must call `d_add()` to insert the found inode into the dentry. The "`i_count`" field in the inode structure should be incremented. If the named inode does not exist a NULL inode should be inserted into the dentry (this is called a negative dentry). Returning an error code from this routine must only be done on a real error, otherwise creating inodes with system calls like `create(2)`, `mknod(2)`, `mkdir(2)` and so on will fail. If you wish to overload the dentry methods then you should initialise the "`d_op`" field in the dentry; this is a pointer to a struct "`dentry_operations`". This method is called with the directory inode semaphore held

link: called by the `link(2)` system call. Only required if you want to support hard links. You will probably need to call `d_instantiate()` just as you would in the `create()` method

unlink: called by the `unlink(2)` system call. Only required if you want to support deleting inodes

symlink: called by the `symlink(2)` system call. Only required if you want to support symlinks. You will probably need to call `d_instantiate()` just as you would in the `create()` method

mkdir: called by the `mkdir(2)` system call. Only required if you want to support creating subdirectories. You will probably need to call `d_instantiate()` just as you would in the `create()` method

rmdir: called by the `rmdir(2)` system call. Only required if you want to support deleting subdirectories

mknod: called by the `mknod(2)` system call to create a device (char, block) inode or a named pipe (FIFO) or socket. Only required if you want to support creating these types of inodes. You will probably need to call `d_instantiate()` just as you would

in the create() method

rename: called by the rename(2) system call to rename the object to have the parent and name given by the second inode and dentry.

readlink: called by the readlink(2) system call. Only required if you want to support reading symbolic links

follow_link: called by the VFS to follow a symbolic link to the inode it points to. Only required if you want to support symbolic links. This method returns a void pointer cookie that is passed to put_link().

put_link: called by the VFS to release resources allocated by follow_link(). The cookie returned by follow_link() is passed to this method as the last parameter. It is used by filesystems such as NFS where page cache is not stable (i.e. page that was installed when the symbolic link walk started might not be in the page cache at the end of the walk).

permission: called by the VFS to check for access rights on a POSIX-like filesystem.

May be called in rcu-walk mode (mask & MAY_NOT_BLOCK). If in rcu-walk mode, the filesystem must check the permission without blocking or storing to the inode.

If a situation is encountered that rcu-walk cannot handle, return -ECHILD and it will be called again in ref-walk mode.

setattr: called by the VFS to set attributes for a file. This method is called by chmod(2) and related system calls.

getattr: called by the VFS to get attributes of a file. This method is called by stat(2) and related system calls.

setxattr: called by the VFS to set an extended attribute for a file. Extended attribute is a name:value pair associated with an inode. This method is called by setxattr(2) system call.

getxattr: called by the VFS to retrieve the value of an extended attribute name. This method is called by getxattr(2) function call.

listxattr: called by the VFS to list all extended attributes for a given file. This method is called by listxattr(2) system call.

removexattr: called by the VFS to remove an extended attribute from a file. This method is called by removexattr(2) system call.

update_time: called by the VFS to update a specific time or the i_version of an inode. If this is not defined the VFS will update the inode itself and call mark_inode_dirty_sync.

atomic_open: called on the last component of an open. Using this optional method the filesystem can look up, possibly create and open the file in one atomic operation. If it cannot perform this (e.g. the file type turned out to be wrong) it may signal this by returning 1 instead of

usual 0 or -ve . This method is only called if the last component is negative or needs lookup. Cached positive dentries are still handled by `f_op->open()`.

The Address Space Object =====

The address space object is used to group and manage pages in the page cache. It can be used to keep track of the pages in a file (or anything else) and also track the mapping of sections of the file into process address spaces.

There are a number of distinct yet related services that an address-space can provide. These include communicating memory pressure, page lookup by address, and keeping track of pages tagged as Dirty or Writeback.

The first can be used independently to the others. The VM can try to either write dirty pages in order to clean them, or release clean pages in order to reuse them. To do this it can call the `->writepage` method on dirty pages, and `->releasepage` on clean pages with `PagePrivate` set. Clean pages without `PagePrivate` and with no external references will be released without notice being given to the `address_space`.

To achieve this functionality, pages need to be placed on an LRU with `lru_cache_add` and `mark_page_active` needs to be called whenever the page is used.

Pages are normally kept in a radix tree index by `->index`. This tree maintains information about the `PG_Dirty` and `PG_Writeback` status of each page, so that pages with either of these flags can be found quickly.

The Dirty tag is primarily used by `mpage_writepages` - the default `->writepages` method. It uses the tag to find dirty pages to call `->writepage` on. If `mpage_writepages` is not used (i.e. the address provides its own `->writepages`), the `PAGECACHE_TAG_DIRTY` tag is almost unused. `write_inode_now` and `sync_inode` do use it (through `_sync_single_inode`) to check if `->writepages` has been successful in writing out the whole `address_space`.

The Writeback tag is used by `filemap*wait*` and `sync_page*` functions, via `filemap_fdatawait_range`, to wait for all writeback to complete. While waiting `->sync_page` (if defined) will be called on each page that is found to require writeback.

An `address_space` handler may attach extra information to a page, typically using the 'private' field in the 'struct page'. If such information is attached, the `PG_Private` flag should be set. This will cause various VM routines to make extra calls into the `address_space` handler to deal with that data.

An address space acts as an intermediate between storage and application. Data is read into the address space a whole page at a time, and provided to the application either by copying of the page, or by memory-mapping the page. Data is written into the address space by the application, and then

written-back to storage typically in whole pages, however the address_space has finer control of write sizes.

The read process essentially only requires 'readpage'. The write process is more complicated and uses write_begin/write_end or set_page_dirty to write data into the address_space, and writepage, sync_page, and writepages to writeback data to storage.

Adding and removing pages to/from an address_space is protected by the inode's i_mutex.

When data is written to a page, the PG_Dirty flag should be set. It typically remains set until writepage asks for it to be written. This should clear PG_Dirty and set PG_Writeback. It can be actually written at any point after PG_Dirty is clear. Once it is known to be safe, PG_Writeback is cleared.

Writeback makes use of a writeback_control structure...

```
struct address_space_operations
-----
```

This describes how the VFS can manipulate mapping of a file to page cache in your filesystem. As of kernel 2.6.22, the following members are defined:

```
struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    int (*sync_page)(struct page *);
    int (*writepages)(struct address_space *, struct writeback_control *);
    int (*set_page_dirty)(struct page *page);
    int (*readpages)(struct file *filp, struct address_space *mapping,
                     struct list_head *pages, unsigned nr_pages);
    int (*write_begin)(struct file *, struct address_space *mapping,
                       loff_t pos, unsigned len, unsigned flags,
                       struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping,
                     loff_t pos, unsigned len, unsigned copied,
                     struct page *page, void *fsdata);
    sector_t (*bmap)(struct address_space *, sector_t);
    int (*invalidatepage)(struct page *, unsigned long);
    int (*releasepage)(struct page *, int);
    void (*freepage)(struct page *);
    ssize_t (*direct_IO)(int, struct kiocb *, const struct iovec *iov,
                          loff_t offset, unsigned long nr_segs);
    struct page* (*get_xip_page)(struct address_space *, sector_t,
                                 int);
    /* migrate the contents of a page to the specified target */
    int (*migratepage)(struct page *, struct page *);
    int (*launder_page)(struct page *);
    int (*error_remove_page)(struct mapping *mapping, struct page *page);
    int (*swap_activate)(struct file *);
    int (*swap_deactivate)(struct file *);
};
```

writepage: called by the VM to write a dirty page to backing store. This may happen for data integrity reasons (i.e. 'sync'), or to free up memory (flush). The difference can be seen in wbc->sync_mode.

The PG_Dirty flag has been cleared and PageLocked is true. writepage should start writeout, should set PG_Writeback, and should make sure the page is unlocked, either synchronously or asynchronously when the write operation completes.

If wbc->sync_mode is WB_SYNC_NONE, ->writepage doesn't have to try too hard if there are problems, and may choose to write out other pages from the mapping if that is easier (e.g. due to internal dependencies). If it chooses not to start writeout, it should return AOP_WRITEPAGE_ACTIVATE so that the VM will not keep calling ->writepage on that page.

See the file "Locking" for more details.

readpage: called by the VM to read a page from backing store. The page will be Locked when readpage is called, and should be unlocked and marked uptodate once the read completes. If ->readpage discovers that it needs to unlock the page for some reason, it can do so, and then return AOP_TRUNCATED_PAGE. In this case, the page will be relocated, relocked and if that all succeeds, ->readpage will be called again.

sync_page: called by the VM to notify the backing store to perform all queued I/O operations for a page. I/O operations for other pages associated with this address_space object may also be performed.

This function is optional and is called only for pages with PG_Writeback set while waiting for the writeback to complete.

writepages: called by the VM to write out pages associated with the address_space object. If wbc->sync_mode is WBC_SYNC_ALL, then the writeback_control will specify a range of pages that must be written out. If it is WBC_SYNC_NONE, then a nr_to_write is given and that many pages should be written if possible. If no ->writepages is given, then mpage_writepages is used instead. This will choose pages from the address space that are tagged as DIRTY and will pass them to ->writepage.

set_page_dirty: called by the VM to set a page dirty. This is particularly needed if an address space attaches private data to a page, and that data needs to be updated when a page is dirtied. This is called, for example, when a memory mapped page gets modified. If defined, it should set the PageDirty flag, and the PAGECACHE_TAG_DIRTY tag in the radix tree.

readpages: called by the VM to read pages associated with the address_space object. This is essentially just a vector version of readpage. Instead of just one page, several pages are requested. readpages is only used for read-ahead, so read errors are ignored. If anything goes wrong, feel free to give up.

write_begin: Called by the generic buffered write code to ask the filesystem to prepare to write len bytes at the given offset in the file. The address_space should check that the write will be able to complete, by allocating space if necessary and doing any other internal housekeeping. If the write will update parts of any basic-blocks on

storage, then those blocks should be pre-read (if they haven't been read already) so that the updated blocks can be written out properly.

The filesystem must return the locked pagecache page for the specified offset, in *pagep, for the caller to write into.

It must be able to cope with short writes (where the length passed to write_begin is greater than the number of bytes copied into the page).

flags is a field for AOP_FLAG_xxx flags, described in include/linux/fs.h.

A void * may be returned in fsdata, which then gets passed into write_end.

Returns 0 on success; < 0 on failure (which is the error code), in which case write_end is not called.

write_end: After a successful write_begin, and data copy, write_end must be called. len is the original len passed to write_begin, and copied is the amount that was able to be copied (copied == len is always true if write_begin was called with the AOP_FLAG_UNINTERRUPTIBLE flag).

The filesystem must take care of unlocking the page and releasing its refcount, and updating i_size.

Returns < 0 on failure, otherwise the number of bytes (<= 'copied') that were able to be copied into pagecache.

bmap: called by the VFS to map a logical block offset within object to physical block number. This method is used by the FIBMAP ioctl and for working with swap-files. To be able to swap to a file, the file must have a stable mapping to a block device. The swap system does not go through the filesystem but instead uses bmap to find out where the blocks in the file are and uses those addresses directly.

invalidatepage: If a page has PagePrivate set, then invalidatepage will be called when part or all of the page is to be removed from the address space. This generally corresponds to either a truncation or a complete invalidation of the address space (in the latter case 'offset' will always be 0).

Any

private data associated with the page should be updated to reflect this truncation. If offset is 0, then the private data should be released, because the page must be able to be completely discarded. This may be done by calling the ->releasepage function, but in this case the release MUST succeed.

releasepage: releasepage is called on PagePrivate pages to indicate that the page should be freed if possible. ->releasepage should remove any private data from the page and clear the PagePrivate flag. If releasepage() fails for some reason, it must indicate failure with a 0 return value.

releasepage() is used in two distinct though related cases. The first is when the VM finds a clean page with no active users and wants to make it a free page. If ->releasepage succeeds, the

page will be removed from the address_space and become free.

The second case is when a request has been made to invalidate some or all pages in an address_space. This can happen through the fadvice(POSIX_FADV_DONTNEED) system call or by the filesystem explicitly requesting it as nfs and 9fs do (when they believe the cache may be out of date with storage) by calling invalidate_inode_pages2().

If the filesystem makes such a call, and needs to be certain that all pages are invalidated, then its releasepage will need to ensure this. Possibly it can clear the PageUptodate bit if it cannot free private data yet.

freepage: freepage is called once the page is no longer visible in the page cache in order to allow the cleanup of any private data. Since it may be called by the memory reclaimer, it should not assume that the original address_space mapping still exists, and it should not block.

direct_IO: called by the generic read/write routines to perform direct_IO - that is IO requests which bypass the page cache and transfer data directly between the storage and the application's address space.

get_xip_page: called by the VM to translate a block number to a page. The page is valid until the corresponding filesystem is unmounted. Filesystems that want to use execute-in-place (XIP) need to implement it. An example implementation can be found in fs/ext2/xip.c.

migrate_page: This is used to compact the physical memory usage. If the VM wants to relocate a page (maybe off a memory card that is signalling imminent failure) it will pass a new page and an old page to this function. migrate_page should transfer any private data across and update any references that it has to the page.

launder_page: Called before freeing a page - it writes back the dirty page. To prevent redirtying the page, it is kept locked during the whole operation.

error_remove_page: normally set to generic_error_remove_page if truncation is ok for this address space. Used for memory failure handling. Setting this implies you deal with pages going away under you, unless you have them locked or reference counts increased.

swap_activate: Called when swapon is used on a file to allocate space if necessary and pin the block lookup information in memory. A return value of zero indicates success, in which case this file can be used to back swapspace. The swapspace operations will be proxied to this address space's ->swap_{out,in} methods.

swap_deactivate: Called during swapoff on files where swap_activate was successful.

The File Object

=====

A file object represents a file opened by a process.

```
struct file_operations
-----
```

This describes how the VFS can manipulate an open file. As of kernel 3.5, the following members are defined:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (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 *, loff_t, loff_t, int datasync);
    int (*aio_fsync) (struct kiocb *, 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 (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    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);
    int (*check_flags) (int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write) (struct pipe_inode_info *, struct file *, size_t,
unsigned int);
    ssize_t (*splice_read) (struct file *, struct pipe_inode_info *, size_t,
unsigned int);
    int (*setlease) (struct file *, long arg, struct file_lock **);
    long (*fallocate) (struct file *, int mode, loff_t offset, loff_t len);
};
```

Again, all methods are called without any locks being held, unless otherwise noted.

llseek: called when the VFS needs to move the file position index

read: called by read(2) and related system calls

aio_read: called by io_submit(2) and other asynchronous I/O operations

write: called by write(2) and related system calls

aio_write: called by io_submit(2) and other asynchronous I/O operations

readdir: called when the VFS needs to read the directory contents

poll: called by the VFS when a process wants to check if there is activity on this file and (optionally) go to sleep until there is activity. Called by the select(2) and poll(2) system calls

unlocked_ioctl: called by the ioctl(2) system call.

compat_ioctl: called by the ioctl(2) system call when 32 bit system calls are used on 64 bit kernels.

mmap: called by the mmap(2) system call

open: called by the VFS when an inode should be opened. When the VFS opens a file, it creates a new "struct file". It then calls the open method for the newly allocated file structure. You might think that the open method really belongs in "struct inode_operations", and you may be right. I think it's done the way it is because it makes filesystems simpler to implement. The open() method is a good place to initialize the "private_data" member in the file structure if you want to point to a device structure

flush: called by the close(2) system call to flush a file

release: called when the last reference to an open file is closed

fsync: called by the fsync(2) system call

fasync: called by the fcntl(2) system call when asynchronous (non-blocking) mode is enabled for a file

lock: called by the fcntl(2) system call for F_GETLK, F_SETLK, and F_SETLKW commands

readv: called by the readv(2) system call

writew: called by the writew(2) system call

sendfile: called by the sendfile(2) system call

get_unmapped_area: called by the mmap(2) system call

check_flags: called by the fcntl(2) system call for F_SETFL command

flock: called by the flock(2) system call

splice_write: called by the VFS to splice data from a pipe to a file. This method is used by the splice(2) system call

splice_read: called by the VFS to splice data from file to a pipe. This method is used by the splice(2) system call

setlease: called by the VFS to set or release a file lock lease. setlease has the file_lock_lock held and must not sleep.

fallocate: called by the VFS to preallocate blocks or punch a hole.

Note that the file operations are implemented by the specific filesystem in which the inode resides. When opening a device node (character or block special) most filesystems will call special support routines in the VFS which will locate the required device driver information. These support routines replace the filesystem file operations with those for the device driver, and then proceed to call the new open() method for the file. This is how opening a device file in the filesystem eventually ends up calling the device driver open() method.

Directory Entry Cache (dcache)

=====

struct dentry_operations

This describes how a filesystem can overload the standard dentry operations. Dentries and the dcache are the domain of the VFS and the individual filesystem implementations. Device drivers have no business here. These methods may be set to NULL, as they are either optional or the VFS uses a default. As of kernel 2.6.22, the following members are defined:

```
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, unsigned int);
    int (*d_weak_revalidate)(struct dentry *, unsigned int);
    int (*d_hash)(const struct dentry *, const struct inode *,
                  struct qstr *);
    int (*d_compare)(const struct dentry *, const struct inode *,
                     const struct dentry *, const struct inode *,
                     unsigned int, const char *, const struct qstr *);
    int (*d_delete)(const struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
    char *(*d_dname)(struct dentry *, char *, int);
    struct vfsmount *(*d_automount)(struct path *);
    int (*d_manage)(struct dentry *, bool);
};
```

d_revalidate:

called when the VFS needs to revalidate a dentry. This is called whenever a name look-up finds a dentry in the dcache. Most local filesystems leave this as NULL, because all their dentries in the dcache are valid. Network filesystems are different since things can change on the server without the client necessarily being aware of it.

This function should return a positive value if the dentry is still valid, and zero or a negative error code if it isn't.

d_revalidate may be called in rcu-walk mode (flags & LOOKUP_RCU). If in rcu-walk mode, the filesystem must revalidate the dentry without blocking or storing to the dentry, d_parent and d_inode should not be used without care (because they can change and, in d_inode case, even become NULL under us).

If a situation is encountered that rcu-walk cannot handle, return -ECHILD and it will be called again in ref-walk mode.

`d_weak_revalidate`: called when the VFS needs to revalidate a "jumped" dentry. This is called when a path-walk ends at dentry that was not acquired by doing a lookup in the parent directory. This includes "/", "." and "..", as well as procfs-style symlinks and mountpoint traversal.

In this case, we are less concerned with whether the dentry is still fully correct, but rather that the inode is still valid. As with `d_revalidate`, most local filesystems will set this to NULL since their dcache entries are always valid.

This function has the same return code semantics as `d_revalidate`.

`d_weak_revalidate` is only called after leaving rcu-walk mode.

`d_hash`: called when the VFS adds a dentry to the hash table. The first dentry passed to `d_hash` is the parent directory that the name is to be hashed into. The inode is the dentry's inode.

Same locking and synchronisation rules as `d_compare` regarding what is safe to dereference etc.

`d_compare`: called to compare a dentry name with a given name. The first dentry is the parent of the dentry to be compared, the second is the parent's inode, then the dentry and inode (may be NULL) of the child dentry. `len` and `name` string are properties of the dentry to be compared. `qstr` is the name to compare it with.

Must be constant and idempotent, and should not take locks if possible, and should not store into the dentry or inodes. Should not dereference pointers outside the dentry or inodes without lots of care (eg. `d_parent`, `d_inode`, `d_name` should not be used).

However, our `vfsmount` is pinned, and RCU held, so the dentries and inodes won't disappear, neither will our `sb` or filesystem module. `->i_sb` and `->d_sb` may be used.

It is a tricky calling convention because it needs to be called under "rcu-walk", ie. without any locks or references on things.

`d_delete`: called when the last reference to a dentry is dropped and the dcache is deciding whether or not to cache it. Return 1 to delete immediately, or 0 to cache the dentry. Default is NULL which means to always cache a reachable dentry. `d_delete` must be constant and idempotent.

`d_release`: called when a dentry is really deallocated

`d_iput`: called when a dentry loses its inode (just prior to its being deallocated). The default when this is NULL is that the VFS calls `iput()`. If you define this method, you must call `iput()` yourself

`d_dname`: called when the pathname of a dentry should be generated. Useful for some pseudo filesystems (`sockfs`, `pipefs`, ...) to delay pathname generation. (Instead of doing it when dentry is created,

it's done only when the path is needed.). Real filesystems probably don't want to use it, because their dentries are present in global dcache hash, so their hash should be an invariant. As no lock is held, `d_dname()` should not try to modify the dentry itself, unless appropriate SMP safety is used. CAUTION : `d_path()` logic is quite tricky. The correct way to return for example "Hello" is to put it at the end of the buffer, and returns a pointer to the first char. `dynamic_dname()` helper function is provided to take care of this.

`d_automount`: called when an automount dentry is to be traversed (optional). This should create a new VFS mount record and return the record to the caller. The caller is supplied with a path parameter giving the automount directory to describe the automount target and the parent VFS mount record to provide inheritable mount parameters. NULL should be returned if someone else managed to make the automount first. If the `vfsmount` creation failed, then an error code should be returned. If `-EISDIR` is returned, then the directory will be treated as an ordinary directory and returned to `pathwalk` to continue walking.

If a `vfsmount` is returned, the caller will attempt to mount it on the mountpoint and will remove the `vfsmount` from its expiration list in the case of failure. The `vfsmount` should be returned with 2 refs on it to prevent automatic expiration - the caller will clean up the additional ref.

This function is only used if `DCACHE_NEED_AUTOMOUNT` is set on the dentry. This is set by `__d_instantiate()` if `S_AUTOMOUNT` is set on the inode being added.

`d_manage`: called to allow the filesystem to manage the transition from a dentry (optional). This allows autofs, for example, to hold up clients waiting to explore behind a 'mountpoint' whilst letting the daemon go past and construct the subtree there. 0 should be returned to let the calling process continue. `-EISDIR` can be returned to tell `pathwalk` to use this directory as an ordinary directory and to ignore anything mounted on it and not to check the automount flag. Any other error code will abort `pathwalk` completely.

If the 'rcu_walk' parameter is true, then the caller is doing a `pathwalk` in RCU-walk mode. Sleeping is not permitted in this mode, and the caller can be asked to leave it and call again by returning `-ECHILD`.

This function is only used if `DCACHE_MANAGE_TRANSIT` is set on the dentry being transitioned from.

Example :

```
static char *pipefs_dname(struct dentry *dent, char *buffer, int buflen)
{
    return dynamic_dname(dentry, buffer, buflen, "pipe:[%lu]",
                        dentry->d_inode->i_ino);
}
```

Each dentry has a pointer to its parent dentry, as well as a hash list of child dentries. Child dentries are basically like files in a directory.

Directory Entry Cache API

There are a number of functions defined which permit a filesystem to manipulate dentries:

`dget`: open a new handle for an existing dentry (this just increments the usage count)

`dput`: close a handle for a dentry (decrements the usage count). If the usage count drops to 0, and the dentry is still in its parent's hash, the "`d_delete`" method is called to check whether it should be cached. If it should not be cached, or if the dentry is not hashed, it is deleted. Otherwise cached dentries are put into an LRU list to be reclaimed on memory shortage.

`d_drop`: this unhashes a dentry from its parents hash list. A subsequent call to `dput()` will deallocate the dentry if its usage count drops to 0

`d_delete`: delete a dentry. If there are no other open references to the dentry then the dentry is turned into a negative dentry (the `d_iput()` method is called). If there are other references, then `d_drop()` is called instead

`d_add`: add a dentry to its parents hash list and then calls `d_instantiate()`

`d_instantiate`: add a dentry to the alias hash list for the inode and updates the "`d_inode`" member. The "`i_count`" member in the inode structure should be set/incremented. If the inode pointer is NULL, the dentry is called a "negative dentry". This function is commonly called when an inode is created for an existing negative dentry

`d_lookup`: look up a dentry given its parent and path name component. It looks up the child of that given name from the dcache hash table. If it is found, the reference count is incremented and the dentry is returned. The caller must use `dput()` to free the dentry when it finishes using it.

Mount Options

=====

Parsing options

On mount and remount the filesystem is passed a string containing a comma separated list of mount options. The options can have either of these forms:

option
option=value

The `<linux/parser.h>` header defines an API that helps parse these options. There are plenty of examples on how to use it in existing filesystems.

Showing options

If a filesystem accepts mount options, it must define `show_options()` to show all the currently active options. The rules are:

- options MUST be shown which are not default or their values differ from the default
- options MAY be shown which are enabled by default or have their default value

Options used only internally between a mount helper and the kernel (such as file descriptors), or which only have an effect during the mounting (such as ones controlling the creation of a journal) are exempt from the above rules.

The underlying reason for the above rules is to make sure, that a mount can be accurately replicated (e.g. umounting and mounting again) based on the information found in `/proc/mounts`.

A simple method of saving options at mount/remount time and showing them is provided with the `save_mount_options()` and `generic_show_options()` helper functions. Please note, that using these may have drawbacks. For more info see header comments for these functions in `fs/namespace.c`.

Resources =====

(Note some of these resources are not up-to-date with the latest kernel version.)

Creating Linux virtual filesystems. 2002
<<http://lwn.net/Articles/13325/>>

The Linux Virtual File-system Layer by Neil Brown. 1999
<<http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html>>

A tour of the Linux VFS by Michael K. Johnson. 1996
<<http://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html>>

A small trail through the Linux
kernel by Andries Brouwer. 2001
<<http://www.win.tue.nl/~aeb/linux/vfs/trail.html>>