Overview of the Linux Virtual File System

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

Last updated on June 24, 2007.

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 device onto a directory in your filespace, the VFS will call the appropriate get_sb() method for the specific filesystem. The dentry for the mount point will then be updated to point to the root inode for the new filesystem.

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 2.6.22, the following members are defined:

```
struct file_system_type {
        const char *name;
        int fs_flags;
        int (*get_sb) (struct file_system_type *, int,
                       const char *, void *, struct vfsmount *);
        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.)

  get_sb: 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 unmounted

  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 get_sb() 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)

  struct vfsmount *mnt: a vfs-internal representation of a mount point

The get_sb() method must determine if the block device specified in the dev_name and fs_type contains a filesystem of the type the method supports. If it succeeds in opening the named block device, it initializes a struct super_block descriptor for the filesystem contained by the block device. On failure it returns an error.

The most interesting member of the superblock structure that the get_sb() 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 get_sb() implementations
and provides a fill_super() method instead. The generic methods are:

  get_sb_bdev: mount a filesystem residing on a block device

  get_sb_nodev: mount a filesystem that is not backed by a device

  get_sb_single: mount a filesystem which shares the instance between
	all mounts

A fill_super() method implementation has the following arguments:

  struct super_block *sb: the superblock structure. The method fill_super()
	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 (*write_inode) (struct inode *, int);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_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 vfsmount *);

        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);
};
```

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_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

  write_super: called when the VFS superblock needs to be written to
	disc. This method is optional

  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.

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 *,int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct 
nameidata *);
        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,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 *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int, struct nameidata *);
        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 (*truncate_range)(struct inode *, loff_t, loff_t);
};
```

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_dop" 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).

  truncate: Deprecated. This will not be called if ->setsize is defined.
	Called by the VFS to change the size of a file.  The
	i_size field of the inode is set to the desired size by the
	VFS before this method is called.  This method is called by
	the truncate(2) system call and related functionality.

	Note: ->truncate and vmtruncate are deprecated. Do not add new
	instances/calls of these. Filesystems should be converted to do their
	truncate sequence via ->setattr().

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

  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.

  truncate_range: a method provided by the underlying filesystem to truncate a
	range of blocks , i.e. punch a hole somewhere in a file.
```

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);
        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);
};
```

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 it
        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.  It may also remove the page from the
        address_space.  If this fails for some reason, it may indicate
        failure with a 0 return value.
        This 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.

  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.
```

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
2.6.22, 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 *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
        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 *, 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);
};
```

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

  ioctl: called by the ioctl(2) system call

```
  unlocked_ioctl: called by the ioctl(2) system call. Filesystems that do not
        require the BKL should use this method instead of the ioctl() above.

  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

  writev: called by the writev(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

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 *, struct nameidata *);
	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 *);
	char *(*d_dname)(struct dentry *, char *, int);
};
```

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 filesystems leave this as NULL, because all their dentries in the dcache are valid

d_hash: called when the VFS adds a dentry to the hash table

d_compare: called when a dentry should be compared with another

d_delete: called when the last reference to a dentry is deleted. This means no-one is using the dentry, however it is still valid and in the dcache

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 dont 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.

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, the "d_delete" method is called and the dentry is placed on the unused list if the dentry is still in its parents hash list. Putting the dentry on the unused list just means that if the system needs some RAM, it goes through the unused list of dentries and deallocates them. If the dentry has already been unhashed and the usage count drops to 0, in this case the dentry is deallocated after the "d_delete" method is called

- 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.

For further information on dentry locking, please refer to the document
Documentation/filesystems/dentry-locking.txt.

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>