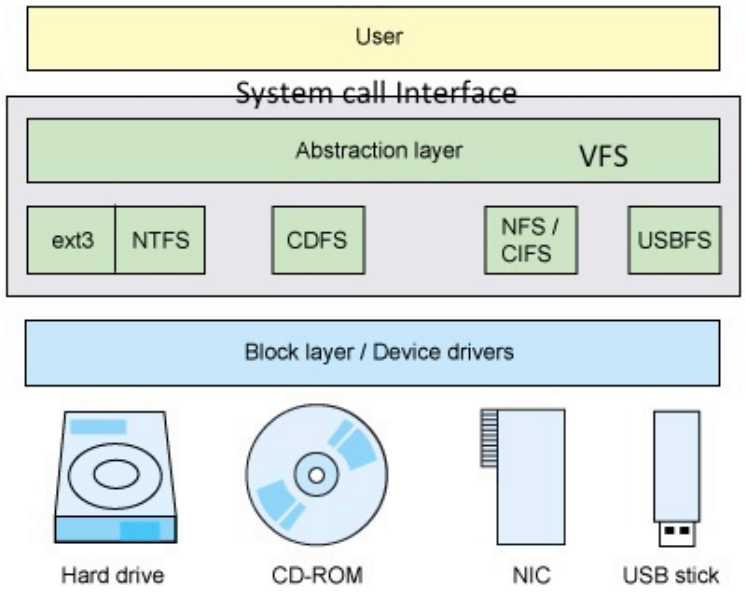
Virtual File System

A *virtual file system* (VFS) or *virtual filesystem switch* is an abstraction layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference. It can be used to bridge the differences in Windows, Mac OS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they are accessing.

The VFS provides a set of standard interfaces for upper-layer applications to perform file I/O over a diverse set of file systems. And it does it in a way that supports multiple concurrent file systems over one or more underlying devices. Additionally, these file systems need not be static but may come and go with the transient nature of the storage devices.

For example, a typical Linux desktop supports an ext3 file system on the available hard disk, as well as the ISO 9660 file system on an available CD-ROM (otherwise called the *CD-ROM file system,* or CDFS). As CD-ROMs are inserted and removed, the Linux kernel must adapt to these new file systems with different contents and structure. A remote file system can be accessed through the Network File System (NFS). At the same time, Linux can mount the NT File System (NTFS) partition of a Windows®/Linux dual-boot system from the local hard disk and read and write from it. Finally, a removable USB flash drive (UFD) can be hot-plugged, providing yet another file system. All the while, the same set of file I/O interfaces can be used over these devices, permitting the underlying file system and physical device to be abstracted away from the user.



The VFS provides the abstraction layer, separating the POSIX API from the details of how a particular file system implements that behavior. The key here is that *Open*, *Read*, *Write*, or *Close* API system calls work the same regardless of whether the underlying file system is ext3 or NTFS. VFS provides a common file model that the underlying file systems inherit (they must implement behaviors for the various POSIX API functions). A further abstraction, outside of the VFS, hides the underlying physical device (which could be a disk, partition of a disk, networked storage entity, memory, or any other medium able to store information—even transiently).

Adding a filesystem

Various objects play a role here. There are *file systems*, organized collections of files, usually on some disk partition. And there are *filesystem types*, abstract descriptions of the way data is organized in a filesystem of that type, like FAT16 or ext2. And there is code, perhaps a module, that implements the handling of file systems of a given type. Sometimes this code is called a low-level filesystem, low-level since it sits below the VFS.

A module implementing a filesystem type must announce its presence so that it can be used. Its task is (i) to have a name, (ii) to know how it is mounted, (iii) to know how to lookup files, (iv) to know how to find (read, write) file contents. This announcing is done using the call *register_filesystem(),* either at kernel initialization time or when the module is inserted.

The mount system call attaches a filesystem to the big file hierarchy at some indicated point. Ingredients needed: (i) a device that carries the filesystem (disk, partition, floppy, CDROM, SmartMedia card, ...), (ii) a directory where the filesystem on that device must be attached, (iii) a filesystem type.

VFS Internals

There are four major objects used to implement VFS: the superblock, the index node (or *inode*), the directory entry (or *dentry*), and file object.

Superblock: The *superblock* is the container for high-level metadata about a file system. The superblock is a structure that exists on disk (actually, multiple places on disk for redundancy) and also in memory. It provides the basis for dealing with the on-disk file system, as it defines the file system's managing parameters (for example, total number of blocks, free blocks, root index node).

Inode: Linux manages all objects in a file system through an object called an *inode* (short for *index node*). An inode can refer to a file or a directory or a symbolic link to another object. Note that because files are used to represent other types of objects, such as devices or memory, inodes are used to represent them also.

dentry: The hierarchical nature of a file system is managed by another object in VFS called a *dentry* object. A file system will have one root dentry (referenced in the superblock), this being the only dentry without a parent. All other dentries have parents, and some have children. For example, if a file is opened that's made up of /home/user/name, four dentry objects are created: one for the root /, one for the home entry of the root directory, one for the name entry of the user directory, and finally, one dentry for the name entry of the user directory. In this way, dentries map cleanly into the hierarchical file systems in use today. Note that the dentry objects exist only in file system memory and are not stored on disk. Only file system inodes are stored permanently, where dentry objects are used to improve performance.

File object: For each opened file in a Linux system, a file object exists. This object contains information specific to the open instance for a given user.

Object Relationships

Figure below shows a typical relationship between various objects. At the top is the open file object, which is referenced by a process's file descriptor list. The file object refers to a dentry object, which refers to an inode. Both the inode and dentry objects refer to the underlying super_block object. Multiple file objects may refer to the same dentry (as in the case of two users sharing the same file). Note also in in the figure that a dentry object refers to another dentry object. In this case, a directory refers to file, which in turn refers to the inode for the particular file.