Anatomy of the Linux virtual file system switch

Abstractions and high-level concepts

M. Tim Jones (mtj@mtjones.com)
Independent author

31 August 2009

Linux® is the very definition of flexibility and extensibility. Take the virtual file system switch (VFS). You can create file systems on a variety of devices, from traditional disk, USB flash drives, memory, and other storage devices. You can even embed a file system within the context of another file system. Discover what makes the VFS so powerful, and learn its major interfaces and processes.

Read more by Tim Jones on developerWorks

- Tim's *Anatomy of...* articles
- All of Tim's articles on developerWorks

The flexibility and extensibility of support for Linux file systems is a direct result of an abstracted set of interfaces. At the core of that set of interfaces is the virtual file system switch (VFS).

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.

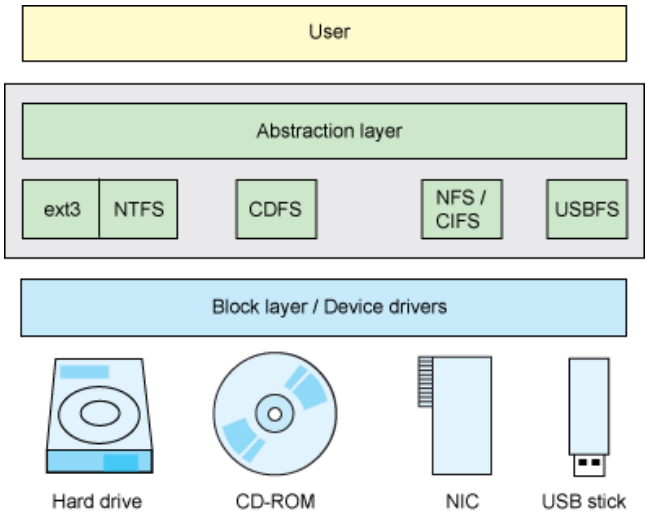
VFs or VFS?

You'll find VFS also defined as *virtual file system,* but *virtual file system switch* is a much more descriptive definition, as the layer switches (that is, multiplexes) requests across multiple file systems. The `/proc` file system adds even more confusion here, as it is commonly called a virtual file system.

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 (see Figure 1).

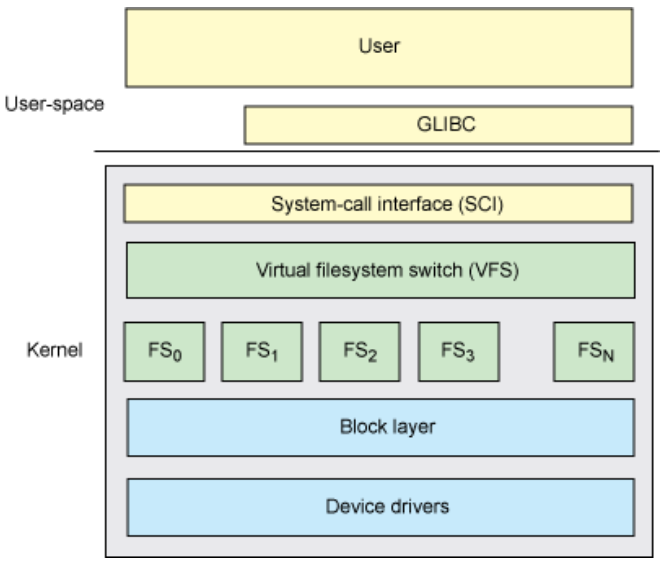
Figure 1. An abstraction layer provides a uniform interface over different file systems and storage devices



Layered abstractions

Now, let's add some concrete architecture to the abstract features that the Linux VFS provides. Figure 2 shows a high-level view of the Linux stack from the point of view of the VFS. Above the VFS is the standard kernel system-call interface (SCI). This interface allows calls from user-space to transition to the kernel (in different address spaces). In this domain, a user-space application invoking the POSIX `open` call passes through the GNU C library (`glibc`) into the kernel and into system call de-multiplexing. Eventually, the VFS is invoked using the call `sys_open`.

Figure 2. The layered architecture of the VFS



Earlier VFS implementations

Linux wasn't the first operating system to incorporate a virtual layer to support a common file model. Earlier VFS implementations include Sun's VFS (in SunOS version 2.0, circa 1985) and IBM and Microsoft®'s "Installable File System" for IBM OS/2. These approaches to virtualizing the file system layer paved the way for Linux's VFS.

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 Btrfs. 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).

In addition to abstracting the details of file operations from the underlying file systems, VFS ties the underlying block devices to the available file systems. Let's now look at the internals of the VFS to see how this works.

VFS internals

Before looking at the overall architecture of the VFS subsystem, let's have a look at the major objects that are used. This section explores the superblock, the index node (or *inode*), the directory entry (or *dentry*), and finally, the file object. Some additional elements, such as caches, are also important here, and I explore these later in the overall architecture.

Superblock

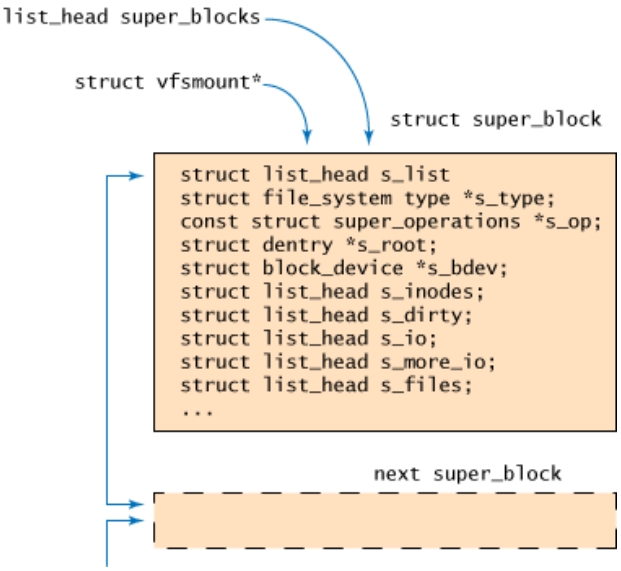
The *superblock* is the container for high-level metadata about a file system. The superblock is a structure that exists on disk (actually, multiple places on disk for redundancy) and also in memory. It provides the basis for dealing with the on-disk file system, as it defines the file system's managing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

On disk, the superblock provides information to the kernel on the structure of the file system on disk. In memory, the superblock provides the necessary information and state to manage the active (mounted) file system. Because Linux supports multiple concurrent file systems mounted at the same time, each `super_block` structure is maintained in a list (`super_blocks`, defined in ./linux/fs/super.c, with the structure defined in /linux/include/fs/fs.h).

Figure 3 provides a simplified view of the superblock and its elements. The `super_block` structure refers to a number of other structures that encapsulate other information. The `file_system_type` structure, for example, maintains the name of the file system (such as *ext3*) as well as various locks and functions to get and remove the `super_block`. The `file_system_type` object is managed through the well-known `register_file system` and `unregister_file system` functions (see ./linux/fs/file systems.c). The `super_operations` structure defines a number of functions for reading and writing inodes as well as higher-level operations (such as remounting). The root directory entry (`dentry`) object is cached here also, as is the block device on which this file system resides. Finally, a number of lists are provided for managing inodes, including `s_inodes` (a list of all inodes),

`s_dirty` (a list of all dirty inodes), `s_io` and `s_more_io` (parked for writeback), and `s_files` (the list of all opened files for a given file system).

Figure 3. Simplified view of the super_block structure and its related elements



Note that within the kernel, another management object called `vfsmount` provides information on mounted file systems. The list of these objects refers to the superblock and defines the mount point, name of the /dev device on which this file system resides, and other higher-level attachment information.

The index node (inode)

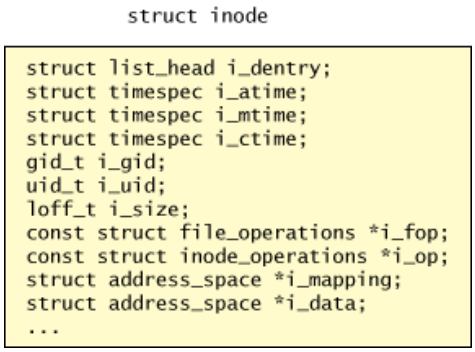
Linux manages all objects in a file system through an object called an *inode* (short for *index node*). An inode can refer to a file or a directory or a symbolic link to another object. Note that because files are used to represent other types of objects, such as devices or memory, inodes are used to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

Note that the inode I refer to here is the VFS layer inode (in-memory inode). Each file system also includes an inode that lives on disk and provides details about the object specific to the particular file system.

VFS inodes are allocated using the slab allocator (from the `inode_cache`; see Resources for a link to more information on the slab allocator). The inode consists of data and operations that describe the inode, its contents, and the variety of operations that are possible on it. Figure 4 is a simple illustration of a VFS inode consisting of a number of lists, one of which refers to the dentries that refer to this inode. Object-level metadata is included here, consisting of the familiar manipulation times (create time, access time, modify time), as are the owner and permission data (group-id, user-id, and permissions). The inode refers to the file operations that are possible on it, most of which map directly to the system-call interfaces (for example, `open`, `read`, `write`, and `flush`). There is also a reference to inode-specific operations (`create`, `lookup`, `link`, `mkdir`, and so on). Finally, there's a structure to manage the actual data for the object that is represented by an address

space object. An *address space object* is an object that manages the various pages for the inode within the page cache. The address space object is used to manage the pages for a file and also for mapping file sections into individual process address spaces. The address space object comes with its own set of operations (`writepage`, `readpage`, `releasepage`, and so on).

Figure 4. Simplified representation of the VFS inode



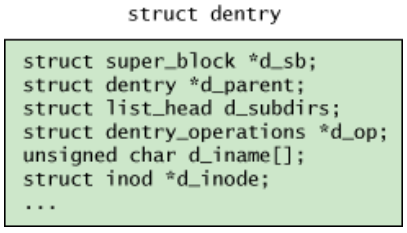
Note that all of this information can be found in ./linux/include/linux/fs.h.

Directory entry (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.

The dentry object is defined by the dentry structure (in ./linux/include/fs/dcache.h). It consists of a number of elements that track the relationship of the entry to other entries in the file system as well as physical data (such as the file name). A simplified view of the dentry object is shown in Figure 5. The dentry refers to the `super_block`, which defines the particular file system instance in which this object is contained. Next is the parent dentry (parent directory) of the object, followed by the children dentries contained within a list (if the object happens to be a directory). The operations for a dentry are then defined (consisting of operations such as `hash`, `compare`, `delete`, `release`, and so on). The name of the object is then defined, which is kept here in the dentry instead of the inode itself. Finally, a reference is provided to the VFS inode.

Figure 5. Simplified representation of the dentry object

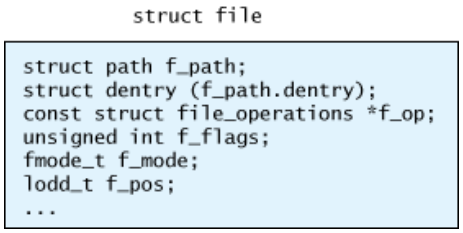


Note that the dentry objects exist only in file system memory and are not stored on disk. Only file system inodes are stored permanently, where dentry objects are used to improve performance. You can see the full description of the dentry structure in ./linux/include/dcache.h.

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. A very simplified view of the file object is provided in Figure 6. As shown, a `path` structure provides reference to both the `dentry` and `vfsmount`. A set of file operations is defined for each file, which are the well-known file operations (`open`, `close`, `read`, `write`, `flush`, and so on). A set of flags and permissions is defined (including group and owner). Finally, stateful data is defined for the particular file instance, such as the current offset into the file.

Figure 6. Simplified representation of the file object

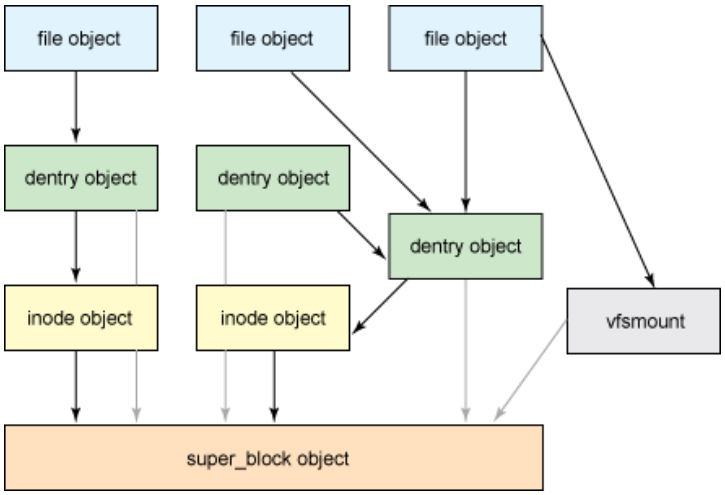


Object relationships

Now that I've reviewed the various important objects in the VFS layer, let's look at how they relate in a single diagram. Because I've explored the object in a bottom-up fashion so far in this article, I now look at the reverse from the user perspective (see Figure 7).

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 Figure 7 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.

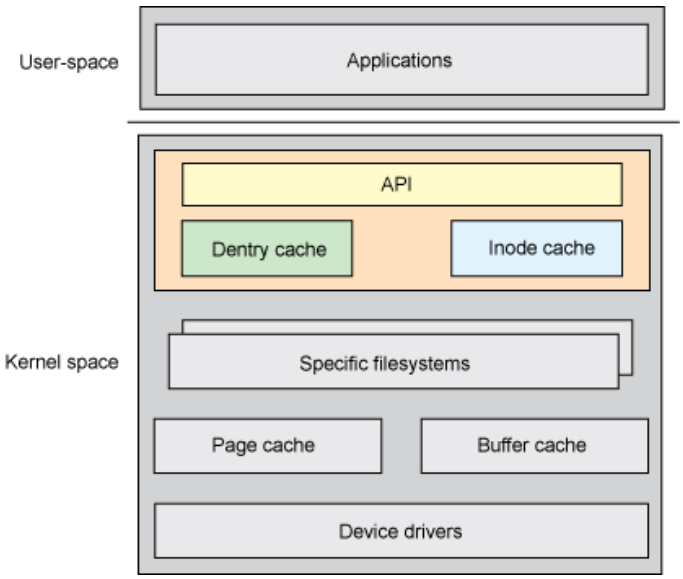
Figure 7. Relationships of major objects in the VFS



The VFS architecture

The internal architecture of the VFS is made up of a dispatching layer that provides the file system abstraction and a number of caches to improve the performance of file system operations. This section explores the internal architecture and how the major objects interact (see Figure 8).

Figure 8. High-level view of the VFS layer



The two major objects that are dynamically managed in the VFS include the `dentry` and `inode` objects. These are cached to improve the performance of accesses to the underlying file systems. When a file is opened, the dentry cache is populated with entries representing the directory levels representing the path. An inode for the object is also created representing the file. The dentry cache is built using a hash table and is hashed by the name of the object. Entries for the dentry cache are allocated from the `dentry_cache` slab allocator and use a least-recently-used (LRU) algorithm to prune entries when memory pressure exists. You can find the functions associated with the dentry cache in ./linux/fs/dcache.c (and ./linux/include/linux/dcache.h).

The inode cache is implemented as two lists and a hash table for faster lookup. The first list defines the inodes that are currently in use; the second list defines the inodes that are unused. Those inodes in use are also stored in the hash table. Individual inode cache objects are allocated from the `inode_cache` slab allocator. You can find the functions associated with the inode cache in ./linux/fs/inode.c (and ./linux/include/fs.h). From the implementation today, the dentry cache is the master of the inode cache. When a `dentry` object exists, an `inode` object will also exist in the inode cache. Lookups are performed on the dentry cache, which result in an object in the inode cache.

Going further

This article has scratched the surface of the VFS, its approach, and objects used to provide uniform access to differing file systems. Linux is scalable, flexible, and extensible from subsystems such as this. The Resources section provides details on where you can learn more.

Resources

Learn

- The Linux subsystem for managing file systems is large and complex. You can learn more about the larger file system subsystem in Tim's "Anatomy of the Linux file system" (developerWorks, October 2007).
- Tim's "Anatomy of ext4" (developerWorks, February 2009) gives an introduction to and overview of this next-generation journaling file system.
- Tim's "Anatomy of the Linux slab allocator" (developerWorks, May 2007) discusses how Linux manages memory through slab allocation.
- Dealing with the VFS from user-space involves the system-call interface, which manages commands' transitions from user-space to the kernel and back. You can learn more about this process in Tim's "Kernel command using Linux system calls" (developerWorks, March 2007).
- Another implementation of a VFS is provided in the IBM AIX® operating system. This Virtual File System Overview provides an introduction to the AIX VFS implementation and describes the various objects that are represented.
- The Linux VFS overview from the operating systems course at the Institut für Informatik provides a great introduction to VFS and explores its use by the ext2 file system. This presentation from the Indian Institute of Technology at Bombay provides a good short introduction to VFS.
- Read more of Tim's articles on developerWorks.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective.* His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.