Anatomy of the Linux file system

A layered structure-based review

M. Tim Jones
Consultant Engineer
Emulex Corp.

30 October 2007

When it comes to file systems, Linux® is the Swiss Army knife of operating systems. Linux supports a large number of file systems, from journaling to clustering to cryptographic. Linux is a wonderful platform for using standard and more exotic file systems and also for developing file systems. This article explores the virtual file system (VFS)—sometimes called the virtual filesystem switch—in the Linux kernel and then reviews some of the major structures that tie file systems together.

Basic file system architecture

The Linux file system architecture is an interesting example of abstracting complexity. Using a common set of API functions, a large variety of file systems can be supported on a large variety of storage devices. Take, for example, the `read` function call, which allows some number of bytes to be read from a given file descriptor. The `read` function is unaware of file system types, such as ext3 or NFS. It is also unaware of the particular storage medium upon which the file system is mounted, such as AT Attachment Packet Interface (ATAPI) disk, Serial-Attached SCSI (SAS) disk, or Serial Advanced Technology Attachment (SATA) disk. Yet, when the `read` function is called for an open file, the data is returned as expected. This article explores how this is done and investigates the major structures of the Linux file system layer.

What is a file system?

I'll start with an answer to the most basic question, the definition of a file system. A file system is an organization of data and metadata on a storage device. With a vague definition like that, you know that the code required to support this will be interesting. As I mentioned, there are many types of file systems and media. With all of this variation, you can expect that the Linux file system interface is implemented as a layered architecture, separating the user interface layer from the file system implementation from the drivers that manipulate the storage devices.

File systems as protocols

Another way to think about a file system is as a protocol. Just as network protocols (such as IP) give meaning to the streams of data traversing the Internet, file systems give meaning to the data on a particular storage medium.

Mounting

Associating a file system to a storage device in Linux is a process called *mounting*. The `mount` command is used to attach a file system to the current file system hierarchy (root). During a mount, you provide a file system type, a file system, and a mount point.

To illustrate the capabilities of the Linux file system layer (and the use of mount), create a file system in a file within the current file system. This is accomplished first by creating a file of a given size using `dd` (copy a file using /dev/zero as the source) -- in other words, a file initialized with zeros, as shown in Listing 1.

Listing 1. Creating an initialized file

```
$ dd if=/dev/zero of=file.img bs=1k count=10000
10000+0 records in
10000+0 records out
$
```

You now have a file called file.img that's 10MB. Use the `losetup` command to associate a loop device with the file (making it look like a block device instead of just a regular file within the file system):

```
$ losetup /dev/loop0 file.img
$
```

With the file now appearing as a block device (represented by /dev/loop0), create a file system on the device with `mke2fs`. This command creates a new second ext2 file system of the defined size, as shown in Listing 2.

Listing 2. Creating an ext2 file system with the loop device

```
$ mke2fs -c /dev/loop0 10000
mke2fs 1.35 (28-Feb-2004)
max_blocks 1024000, rsv_groups = 1250, rsv_gdb = 39
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
2512 inodes, 10000 blocks
500 blocks (5.00%) reserved for the super user
...
$
```

The file.img file, represented by the loop device (`/dev/loop0`), is now mounted to the mount point /mnt/point1 using the `mount` command. Note the specification of the file system as `ext2`. When mounted, you can treat this mount point as a new file system using an `ls` command, as shown in Listing 3.

Listing 3. Creating a mount point and mounting the file system through the loop device

```
$ mkdir /mnt/point1
$ mount -t ext2 /dev/loop0 /mnt/point1
$ ls /mnt/point1
lost+found
$
```

As shown in Listing 4, you can continue this process by creating a new file within the new mounted file system, associating it with a loop device, and creating another file system on it.

Listing 4. Creating a new loop file system within a loop file system

```
$ dd if=/dev/zero of=/mnt/point1/file.img bs=1k count=1000
1000+0 records in
1000+0 records out
$ losetup /dev/loop1 /mnt/point1/file.img
$ mke2fs -c /dev/loop1 1000
mke2fs 1.35 (28-Feb-2004)
max_blocks 1024000, rsv_groups = 125, rsv_gdb = 3
Filesystem label=
...
$ mkdir /mnt/point2
$ mount -t ext2 /dev/loop1 /mnt/point2
$ ls /mnt/point2
lost+found
$ ls /mnt/point1
file.img lost+found
$
```

From this simple demonstration, it's easy to see how powerful the Linux file system (and the loop device) can be. You can use this same approach to create encrypted file systems with the loop device on a file. This is useful to protect your data by transiently mounting your file using the loop device when needed.

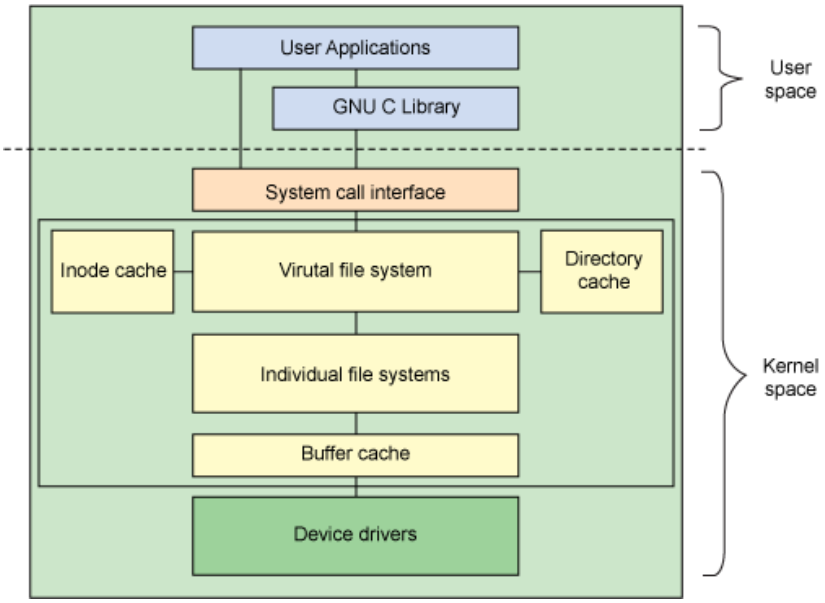
File system architecture

Now that you've seen file system construction in action, I'll get back to the architecture of the Linux file system layer. This article views the Linux file system from two perspectives. The first view is from the perspective of the high-level architecture. The second view digs in a little deeper and explores the file system layer from the major structures that implement it.

High-level architecture

While the majority of the file system code exists in the kernel (except for user-space file systems, which I'll note later), the architecture shown in Figure 1 shows the relationships between the major file system- related components in both user space and the kernel.

Figure 1. Architectural view of the Linux file system components



User space contains the applications (for this example, the user of the file system) and the GNU C Library (glibc), which provides the user interface for the file system calls (open, read, write, close). The system call interface acts as a switch, funneling system calls from user space to the appropriate endpoints in kernel space.

The VFS is the primary interface to the underlying file systems. This component exports a set of interfaces and then abstracts them to the individual file systems, which may behave very differently from one another. Two caches exist for file system objects (inodes and dentries), which I'll define shortly. Each provides a pool of recently-used file system objects.

Each individual file system implementation, such as ext2, JFS, and so on, exports a common set of interfaces that is used (and expected) by the VFS. The buffer cache buffers requests between the file systems and the block devices that they manipulate. For example, read and write requests to the underlying device drivers migrate through the buffer cache. This allows the requests to be cached there for faster access (rather than going back out to the physical device). The buffer cache is managed as a set of least recently used (LRU) lists. Note that you can use the `sync` command to flush the buffer cache out to the storage media (force all unwritten data out to the device drivers and, subsequently, to the storage device).

What is a block device?

A block device is one in which the data that moves to and from it occurs in blocks (such as disk sectors) and supports attributes such as buffering and random access behavior (is not required to read blocks sequentially, but can access any block at any time). Block devices include hard drives, CD-ROMs, and RAM disks. This is in contrast to character devices, which differ in that they do not have a physically-addressable media. Character devices include serial ports and tape devices, in which data is streamed character by character.

That's the 20,000-foot view of the VFS and file system components. Now I'll look at the major structures that implement this subsystem.

Major structures

Linux views all file systems from the perspective of a common set of objects. These objects are the superblock, inode, dentry, and file. At the root of each file system is the superblock, which describes and maintains state for the file system. Every object that is managed within a file system (file or directory) is represented in Linux as an inode. The inode contains all the metadata to manage objects in the file system (including the operations that are possible on it). Another set of structures, called dentries, is used to translate between names and inodes, for which a directory cache exists to keep the most-recently used around. The dentry also maintains relationships between directories and files for traversing file systems. Finally, a VFS file represents an open file (keeps state for the open file such as the write offset, and so on).

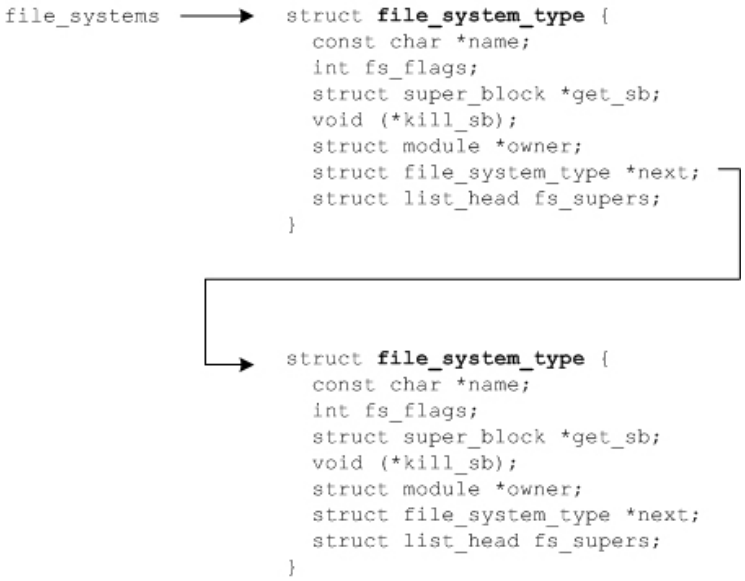
Virtual file system layer

The VFS acts as the root level of the file-system interface. The VFS keeps track of the currently-supported file systems, as well as those file systems that are currently mounted.

File systems can be dynamically added or removed from Linux using a set of registration functions. The kernel keeps a list of currently-supported file systems, which can be viewed from user space through the /proc file system. This virtual file also shows the devices currently associated with the file systems. To add a new file system to Linux, `register_filesystem` is called. This takes a single argument defining the reference to a file system structure (`file_system_type`), which defines the name of the file system, a set of attributes, and two superblock functions. A file system can also be unregistered.

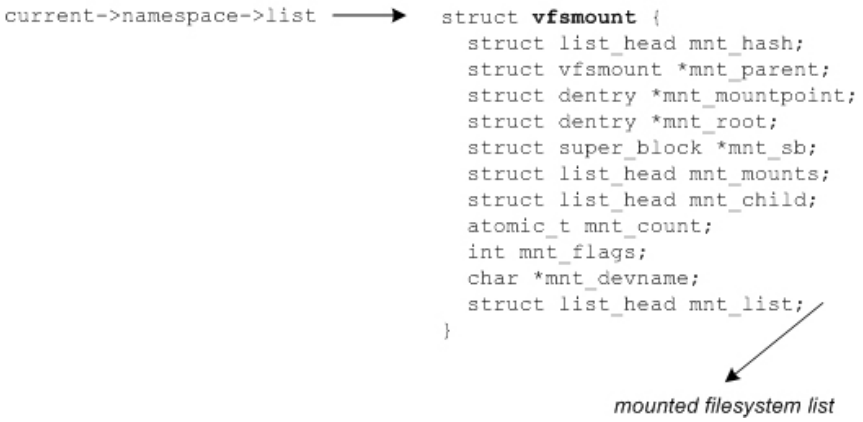
Registering a new file system places the new file system and its pertinent information onto a file_systems list (see Figure 2 and linux/include/linux/mount.h). This list defines the file systems that can be supported. You can view this list by typing `cat /proc/filesystems` at the command line.

Figure 2. File systems registered with the kernel



Another structure maintained in the VFS is the mounted file systems (see Figure 3). This provides the file systems that are currently mounted (see linux/include/linux/fs.h). This links to the `superblock` structure, which I'll explore next.

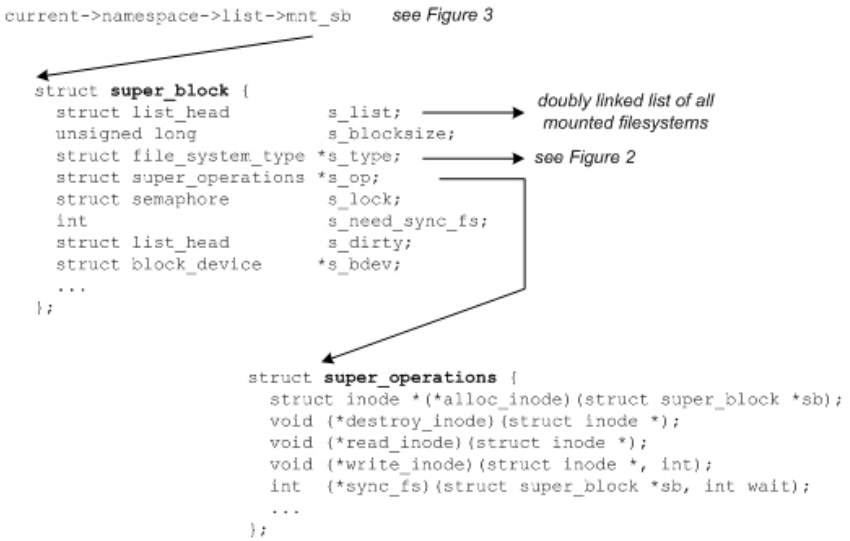
Figure 3. The mounted file systems list



Superblock

The superblock is a structure that represents a file system. It includes the necessary information to manage the file system during operation. It includes the file system name (such as ext2), the size of the file system and its state, a reference to the block device, and metadata information (such as free lists and so on). The superblock is typically stored on the storage medium but can be created in real time if one doesn't exist. You can find the superblock structure (see Figure 4) in ./linux/include/linux/fs.h.

Figure 4. The superblock structure and inode operations



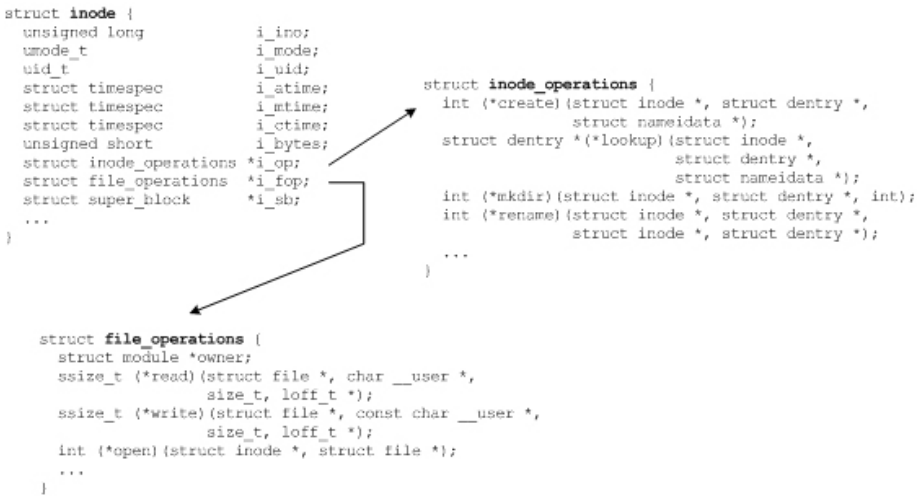
One important element of the superblock is a definition of the superblock operations. This structure defines the set of functions for managing inodes within the file system. For example, inodes can be allocated with `alloc_inode` or deleted with `destroy_inode`. You can read and write

inodes with `read_inode` and `write_inode` or sync the file system with `sync_fs`. You can find the `super_operations` structure in ./linux/include/linux/fs.h. Each file system provides its own inode methods, which implement the operations and provide the common abstraction to the VFS layer.

inode and dentry

The inode represents an object in the file system with a unique identifier. The individual file systems provide methods for translating a filename into a unique inode identifier and then to an inode reference. A portion of the inode structure is shown in Figure 5 along with a couple of the related structures. Note in particular the `inode_operations` and `file_operations`. Each of these structures refers to the individual operations that may be performed on the inode. For example, `inode_operations` define those operations that operate directly on the inode and `file_operations` refer to those methods related to files and directories (the standard system calls).

Figure 5. The inode structure and its associated operations



The most-recently used inodes and dentries are kept in the inode and directory cache respectively. Note that for each inode in the inode cache there is a corresponding dentry in the directory cache. You can find the `inode` and `dentry` structures defined in ./linux/include/linux/fs.h.

Buffer cache

Except for the individual file system implementations (which can be found at ./linux/fs), the bottom of the file system layer is the buffer cache. This element keeps track of read and write requests from the individual file system implementations and the physical devices (through the device drivers). For efficiency, Linux maintains a cache of the requests to avoid having to go back out to the physical device for all requests. Instead, the most-recently used buffers (pages) are cached here and can be quickly provided back to the individual file systems.

Interesting file systems

This article spent no time exploring the individual file systems that are available within Linux, but it's worth note here, at least in passing. Linux supports a wide range of file systems, from the

old file systems such as MINIX, MS-DOS, and ext2. Linux also supports the new journaling file systems such as ext3, JFS, and ReiserFS. Additionally, Linux supports cryptographic file systems such as CFS and virtual file system such as /proc.

One final file system worth noting is the Filesystem in Userspace, or FUSE. This is an interesting project that allows you to route file system requests through the VFS back into user space. So if you've ever toyed with the idea of creating your own file system, this is a great way to start.

Summary

While the file system implementation is anything but trivial, it's a great example of a scalable and extensible architecture. The file system architecture has evolved over the years but has successfully supported many different types of file systems and many types of target storage devices. Using a plug-in based architecture with multiple levels of function indirection, it will be interesting to watch the evolution of the Linux file system in the near future.

Resources

Learn

- The proc file system provides a novel scheme for communicating between user space and the kernel through a virtual file system. "Access the Linux kernel using the /proc filesystem" (developerWorks, March 2006) introduces you to the /proc virtual file system and demonstrates its use.
- The Linux system call interface provides the means to transition control between user space and the kernel to invoke kernel API functions. "Kernel command using Linux system calls" (developerWorks, 2007) explores the Linux system call interface.
- Yolinux.com maintains a great list of Linux file systems, clustered file systems, and performance compute clusters. You can also find a complete list of Linux file systems in the File systems HOWTO. Xenotime provides another option with descriptions of a large number of file systems.
- For more information on programming Linux in user space, check out *GNU/Linux Application Programming*, written by the author of this article.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- 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

- The Filesystem in Userspace (FUSE) is a kernel module that enables development of file systems in user space. The file system driver implementation routes requests from the VFS back to user space. It's a great way to experiment with file system development without resorting to kernel development. If you're into Python, you can write a file system with this language as well using LUFS-Python.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the author

M. Tim Jones

M. Tim Jones is an embedded software architect and the author of *GNU/Linux Application Programming*, *AI Application Programming*, 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.