Linux 2.6 VFS Locking

Rink Springer

February 21, 2007

Abstract

This short article is an introduction to the underlying structures of the Linux Virtual File System, and focuses on the actual locking of these structures. Only virtual filesystem structures are considered, all other VFS aspects (such as file-locking, caching and filesystem driver integration) are not discussed.

1 Introduction

During the mid eighties, a need arouse for a more generic abstraction to filesystem drivers. Previously, an operating system usually could access only one kind of filesystem. However, as different filesystems starting to become popular for difference applications, the Virtual File System, or VFS, was designed to provide a generic interface between each filesystem an the operating system kernel.

This article will detail the bare basics of the Linux VFS, discuss the relationship between the various components and detail the locking primitives that are used to prevent unwanted corruption of the structures used. It will not discuss VFS details (we refer to [Spr05] for a more in-depth discussion on this subject), neither file-locking (which is conceptually different from VFS locking) or VFS buffering. We refer to code that actually implements the filesystem to VFS mapping as a *filesystem driver*.

Finally, this article discusses Linux kernel 2.6.19; most of the information should be applicable to both previous and upcoming 2.6 releases, although minor details might be altered.

2 Locking primitives

Linux 2.6 supports the following locking primitives [Lov05, Rus03]:

- Atomic operations
  Architecture-specific atomic integer operations are provided, these can be found in the `include/asm/atomic.h` include file. They are used to atomically read, set, add, subtract and compare a variable. These are mainly used for resource counting, and can be used in any context.

- Spinlocks
  These provide an inexpensive way to keep a CPU busy-waiting for a resource (spinning around the lock), implemented in `include/asm/spinlock.h`. A special variant called *reader/writer spinlocks* is also available, where a resource may have several reader locks, but only one writer lock may be active (and only when there are no reader locks).
- Mutexes
  Short for **mut**ual **ex**clusion, these are small locks that will cause the task waiting for them to be suspended (sleeping), as opposed to spinlocks where the CPU will remain waiting until the resource is available. This means they cannot be used in any context where sleeping is forbidden.
- Semaphores
  These closely resemble mutexes, but allow the number of holders at any point in time to be specified. They will sleep whenever a lock is acquired which cannot be honored.

3 VFS organization

As this article solely focuses on how the VFS is locked, all details regarding caching and such will not be discussed. A basic introduction on VFS caching can be found in [Lov05]; a more thorough description is provided in [Spr05].

The Linux VFS consists of several key structures:

- `struct vfsmount` (`mount.h`)
  Contains details of a mounted filesystem. Every process has its own list of mounted filesystems; these are usually inherited when executing a new process.
- `struct super_block` (`fs.h`)
  A superblock structure is the relation between a block device and a filesystem. It contains references to the filesystem driver used for this block device.
- `struct dentry` (`dcache.h`)
  Short for **d**irectory **entry**, a dentry links a filename to an inode. Per standard UNIX semantics, an inode can have multiple names referencing to it (such as hard links).
- `struct inode` (`fs.h`)
  An inode contains information about a file or directory, such as the owner's user id, group id, file permissions, file length etc.
- `struct file` (`fs.h`)
  This is a single, open file, as opened by a process. It contains information on the current file pointer position, as well as other information such as the owner's userid and groupid when the file was opened.

The ERD on the next page details the relationship between these structures, as well as listing all revelant items.

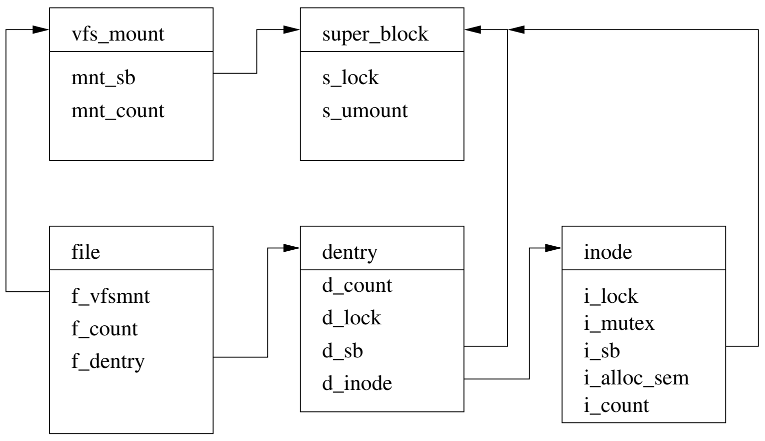

Figure 1: Relationships between key VFS structures

4 Locking

As stated in the introduction, locking is vital to ensure correct system operation. Fine-grained locking is used in order to protect only those data structures that need to be protected, in order to maximize performance by allowing operations not influenced by a certain structure to continue.

4.1 struct vfsmount

As stated above, this list is usually on a per-process basis; and almost all information is read-only. The few times that it must be updated (for example, when new mount(2)'s or umount(2)'s are being processed), this whole list is protected using the `vfsmount_lock` spinlock.

Each vfsmount structure uses the `mnt_count` field as refence counter, which is updated using atomic operations. Once this number is zero, the vfsmount structure is eligible to be removed and cleaned.

4.2 struct super_block

All super_block structures are stored in the `super_blocks` list, which is protected using the `sb_lock` spinlock. This lock only protects the list of the superblocks, and not each super_block itself; finer-grained locking is used for this purpose, in the form of the `s_lock` mutex. This lock is under control of the filesystem driver itself, as the kernel will not touch the other fields (this makes sense, as the kernel should not suddenly change the block device or block size of a mounted filesystem).

The semaphore `s_umount` is used to protect the filesystem from multiple mount operations, and is acquired throughout the mounting/umounting/flushing code paths.

4.3 struct dentry

Each dentry structure is locked using a per-dentry `d_lock` spinlock. It is acquired for every read and update throughout the code.

Dentries have an atomic value stored in the `d_count` field; this field is used for reference counting. Once all references are dropped, the dentry will be considered for garbage collection.

4.4 struct inode

Inodes have three locks. The `i_lock` spinlock is used to protect the file block and byte count, and several filesystems also use this to set filesystem-specific flags per inode.

The `i_mutex` is a mutex which is used for pretty much everything else; this is mainly due to the fact that filesystem operations can potentially cause the caller to sleep (for example, if an inode is to be removed, perhaps the directory entry has to be retrieved from disk before it can be updated).

The `i_alloc_sem` spinlock is not quite clear; it appears to be used for the file notification mechanism, as well as for O_DIRECT direct operations. However, the precise nature eludes the author; the spinlock seems to be explicitly acquired and freed only in the cases as stated in the previous sentence, and appears to be implied by the `i_lock` spinlock.

Finally, a field `i_count` is used for reference counting. This field is atomically incremented and decremented as needed; once all references are dropped, the inode will be considered for garbage collection.

4.5 struct file

All file structures are stored as a per-superblock list, where they are protected using the global `files_lock` spinlock.

As most other structures, the file structure has an atomic value `i_count`, which is used to keep track of how many references to this particular file exist. Once this value reaches the value of zero, the file structure will be recycled as nothing is actively pointing to it.

5 Closing words

As outlined in the paper, only a handful of key structures make up the VFS layer. These all have very specific locks, with close to no functional overlap. The goal of this paper was to provide a basic overview of which structure is locked, and ought to provide potential VFS developers with a concise overview of the VFS locking internals.

Linux guarantees the appropriate locks will be held upon calling filesystem driver code; this helps keep filesystem drivers concise and compact. Filesystem drivers are always allowed to acquire/free extra locks where needed.

References

[Lov05] Robert Love. *Linux Kernel Development*. Novell Press, 2005.

[Rus03] Rusty Russell. Unreliable guide to locking. Kernel source file Documentation/DocBook/kernel-locking.tmpl, 2003.

[Spr05] Rink Springer. Technical report: Limefs realtime file system. Technical report, 2005. http://rink.nu/downloads/publications/philips-graduation-report.pdf.