Kernel Support for Stackable File Systems

Josef Sipek, Yiannis Pericleous, and Erez Zadok
Stony Brook University

Appears in the proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)

Abstract

Although it is now possible to use stackable (layered) file systems in Linux, there are several issues that should be addressed to make stacking more reliable and efficient. To support stacking properly, some changes to the VFS and VM subsystems will be required. In this paper, we discuss some of the issues and solutions proposed at the Linux Storage and Filesystems workshop in February 2007, our ongoing work on stacking support for Linux, and our progress on several particular stackable file systems.

1 Introduction

A stackable (layered) file system is a file system that does not store data itself. Instead, it uses another file system for its storage. We call the stackable file system the *upper* file system, and the file systems it stacks on top of the *lower* file systems.

Although it is now possible to use stackable file systems, a number of issues should be addressed to improve file system stacking reliability and efficiency. The Linux kernel VFS was not designed with file system stacking in mind, and therefore it comes as no surprise that supporting stacking properly will require some changes to the VFS and VM subsystems.

We use eCryptfs and Unionfs as the example stackable file systems to cover both linear and fanout stacking, respectively.

eCryptfs is a cryptographic file system for Linux that stacks on top of existing file systems. It provides functionality similar to that of GnuPG, except that encrypting and decrypting the data is transparent to the application [1–3].

Unionfs is a stackable file system that presents a series of directories (branches) from different file systems as one virtual directory, as specified by the user. This is commonly referred to as namespace unification. Previous publications [4–6] provide detailed description and some possible use cases.

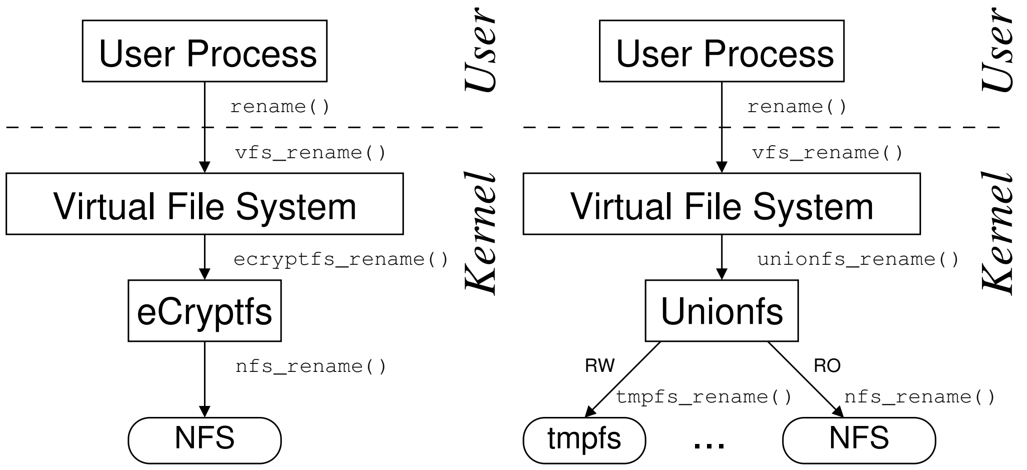
Both eCryptfs and Unionfs are based on the FiST stackable file system templates, which provide support for layering over a single directory [7]. As shown in Figures 1(a) and 1(b), the kernel's VFS is responsible for dispatching file-system–related system calls to the appropriate file system. To the VFS, a stackable file system appears as if it were a standard file system. However, instead of storing or retrieving data, a stackable file system passes calls down to lower-level file systems. In this scenario, NFS is used as a lower-level file system, but any file system can be used to store the data as well (e.g., Ext2, Ext3, Reiserfs, SQUASHFS, isofs, `tmpfs`, etc.).

To the lower-level file systems, a stackable file system appears as if it were the VFS. Stackable file system development can be difficult because the file system must adhere to the conventions of both the file systems for processing VFS calls, and of the VFS for making VFS calls.

Without kernel support, stackable file systems suffer from inherent cache coherency problems. These issues can be divided into two categories: (1) data coherency of the page cache contents, and (2) meta-data coherency of the `dentry` and `inode` caches. Changes to the VFS and the stackable file systems are required to remedy these problems.

Moreover, `lockdep`, the in-kernel lock validator, "observes" and maps all locking rules as they dynamically occur, as triggered by the kernel's natural use of locks (spinlocks, rwlocks, mutexes, and rwsems). Whenever the lock validator subsystem detects a new locking scenario, it validates this new rule against the existing set of rules. Unfortunately, stackable file systems need to lock many of the VFS objects in a recursive manner, triggering `lockdep` warnings.

To maintain the upper to lower file system map-



(a) eCryptfs layers over a single directory. (b) Unionfs layers over multiple directories.

Figure 1: The user processes issue system calls, which the kernel's virtual file system (VFS) directs to stackable file systems. Stackable file systems in turn pass the calls down to lower-level file systems (e.g., tmpfs or NFS).

ping of kernel objects (such as `dentrys`, `inodes`, etc.), many stackable file systems share much of the basic infrastructure. The 2.6.20 kernel introduced `fs/stack.c`, a new file that contains several helper functions useful to stackable file systems.

The rest of this paper is organized as follows. In Section 2 we discuss the cache coherency issues. In Section 3 we discuss the importance of locking order. In Section 4 we discuss `fsstack`, the emerging Linux kernel support for stacking. In Section 5, we discuss a persistent store prototype we developed for Unionfs, which can be of use to others. Finally, we conclude in Section 6.

2 Cache Coherency

There are two different cache coherency issues that stackable file systems must overcome: data and meta-data.

2.1 Data Coherency

Typically, the upper file system maintains its own set of pages used for the page cache. Under ideal conditions, all the changes to the data go through the upper file system. Therefore, either the upper file system's `write` inode operation or the `writepage` address space operation will have a chance to transform the data as necessary (e.g., eCryptfs needs to encrypt all writes) and write it to the lower file system's pages.

Data incoherency occurs when data is written to the lower pages directly, without the stacked file system's knowledge. There are two possible solutions to this problem:

Weak cache coherency : NFS also suffers from cache coherency issues as the data on the server may be changed by either another client or a local server process. NFS uses a number of assertions that are checked before cached data is used. If any of these assertions fail, the cached data is invalidated. One such assertion is a comparison of the `ctime` with what is cached, and invalidating any potentially out-of-date information.

Strong cache coherency : Another possible solution to the cache coherency problem is to modify the VFS and VM to effectively inform the stackable file systems that the data has changed on the lower file system. There are several different ways of accomplishing this, but all involve maintaining pointers from lower VFS objects to *all* upper ones. Regardless of how this is implemented, the VFS/VM must traverse a dependency graph of VFS objects, invalidate all pages belonging to the corresponding upper addresses spaces, and `sync` all of the pages that are part of the lower address spaces.

Both approaches have benefits and drawbacks.

The major benefit of the weak consistency approach is that the VFS does not have to be modified at all. The major downside is that *every* stackable file system needs to contain a number of these checks. Even if helper functions are created, calls to these functions need to be placed throughout the stackable file systems. This leads to code duplication, which we try to address with `fsstack` (see Section 4).

The most significant benefit of the stronger coherency approach is the fact that it guarantees that the caches are always coherent. At the same time, it requires that the file system use the page cache properly, and that the file system supplies a valid address space operations vector. Some file systems do not meet these requirements. For example, GPFS (created by IBM) only has `readpage` and `writepage`, but does not have any other address space operations. If the cache coherency is maintained at the page-cache level, the semantics of using a lower file system that does not define the needed operations would be unclear.

2.2 Meta-Data Coherency

Similar to the page cache, many VFS objects, such as the cached `inode` and `dentry` objects, may become inconsistent. The meta-data contained in these caches includes the {`a`,`c`,`m`}`times`, file size, etc.

Just as with data consistency, either a strong or a weak cache coherency model may be used to prevent the upper and lower VFS objects from disagreeing on the file system state. The benefits and drawbacks stated previously apply here as well (e.g., weak coherency requires code duplication in most stackable file systems).

2.3 File Revalidation

The VFS currently allows for `dentry` revalidation. NFS and other network file system are the few users of this. A useful addition to this `dentry` revalidation operation would be an equivalent `file` operation. Given a `struct file`, this would allow the file system to check for validity and repair any inconsistencies.

Unionfs works around the lack of `file` revalidation by calling its own helper function in the appropriate `struct file` operations. The reason Unionfs requires this is due to the possibility of a branch management operation changing the number or order of branches, and the lower `struct file` pointers need to be updated.

3 Locking Order

Since stackable file systems must behave as both a file system and the VFS, they need to lock many of the VFS objects in a recursive manner, triggering warnings about potential deadlocks. The in-kernel lock validator, `lockdep`, dynamically monitors the kernel's usage of locks (spinlocks, rwlocks, mutexes and rwsems) and creates rules. Whenever the lock validator subsystem detects a new locking scenario, it validates this new rule against the existing set of rules.

The `lockdep` system is aware of locking dependency chains, such as: parent→child→xattr→quota. However, it does not understand that a stackable file system may cause recursion in the VFS. For example, the VFS may indirectly (but safely) call itself; `vfs_readdir` can call a stackable file system on one directory, which can in turn call `vfs_readdir` again on other lower directories. Each time `vfs_readdir` is called, the corresponding `i_mutex` is taken. This triggers a `lockdep` warning, as it considers this situation a potential place for a deadlock, and warns accordingly. In other words, `lockdep` needs to be informed of the hierarchies between stacked file systems. This, however, would require adding a "`stacked`" argument to many functions in the VFS, and passing that information to `lockdep`.

4 `fsstack`

The code duplication found in many stackable file systems (such as eCryptfs, Unionfs, and our upcoming cachefs) is another problem. The 2.6.20 kernel introduced `fs/stack.c`, a new file, which contains several useful helper functions. We are working on further abstractions to the stacking API in Linux.

Each stackable file system must maintain a set of pointers from the upper (stackable file system) objects to the lower objects. For example, each Unionfs `inode` maintains a series of lower `inode` pointers.

Currently, there are two ways to keep track of lower objects. Linear (one lower pointer) and fan-out (several lower pointers). Fan-out is the more interesting case, as linear stacking is just a special case of fan-out, with only one branch. Quite frequently, Unionfs needs to traverse all the branches. This creates the need for a `for_each_branch` macro (analogous to `for_each_node`), which would decide when to terminate.

A "reference" stackable file system, much like NullFS in many BSDs, would allow stackable file system authors to easily create new stackable file systems for Linux. This reference file system should use as many of the `fsstack` interfaces as possible. Currently, the closest thing to this is Wrapfs [7], which can be generated from FiST. Unfortunately, the generated code does not follow proper coding style and general code cleanliness.

In Section 2.1, we considered a weaker form of the cache coherency model. This model suffers from the fact that a large number of the coherency checks (e.g., checking the {`a,c,m`}`times`) will need to be duplicated in each stackable file system. Using `fsstack` avoids this problem by making use of generic functions to perform operations common to all stackable file systems. The code necessary to invalidate and revalidate the upper file system objects could be shared by several file systems. However, *each* file system must call these helper functions. If a bug is discovered in one stackable file system (e.g., a helper function should be called but is not), the fix may have to be ported to other file systems.

Stackable file systems must behave as a file system from the point of view of the VFS, yet they must behave as the VFS from the point of view of the file systems it is stacked on top of. Generally, the most complex code occurs in the file system lookup code. One idea, proposed at the 2007 Linux Storage and Filesystem workshop, was to divide the current VFS lookup code into two portions, and to allow the file system to override part of the functionality via a new `inode` operation. The default operation would have functionality identical to the current lookup code. The flexibility allowed by this code refactoring would simplify some of the code in more complex file systems. For example, Ext2 could use the generic lookup code provided by the VFS, while a file system requiring more complex lookup code, such as Unionfs, can provide its own lookup helper which performs the necessary operations (e.g., to perform namespace unification).

5 On Disk Format (ODF)

We have developed an On Disk Format (ODF) to help Unionfs 2.0 persistently store any meta-data it needs, such as whiteouts. ODF is a small file system on a partition or loop device. The ODF has helped us resolve most of the critical issues that Unionfs 1.x had faced, such as namespace pollution, `inode` persistence, `readdir` consistency and efficiency, and more. Since, all the meta-data is kept in a separate file system instead of in the branches themselves, Unionfs can be stacked on top of itself and have overlapping branches.

Such a format can be used by any stackable file system that needs to store meta-data persistently. For example, a versioning file system may use it to store information about the current version of a file, a caching file system may use it to store information about the status of the cached files. Unionfs benefits in many ways as well, for example there is no namespace pollution, and Unionfs can be stacked on itself.

By keeping all the meta-data together in a separate file system, simply creating an in-kernel mount can be used to easily hide it from the user, assuring that the user will not temper with the meta-data. Also, it becomes easier to backup the state of the file system by simply creating a backup of the ODF file system. If a file system which statically allocates `inode` tables is used, the user must estimate the number of `inodes` and data blocks the ODF will need before hand. Using a file system which allocates `inode` blocks dynamically (e.g., XFS) fixes this problem. This is a shortcoming of the file system, and not ODF itself.

The ODF can use another file system, such as Ext2 or XFS, to store the meta-data. Another possibility we are looking at for the future is to build an ODF file system that will have complete control of how it stores this meta-data, thus allowing us to make it more efficient, flexible and reusable.

6 Conclusion

Stackable file systems can be used today on Linux. There are some issues which should be addressed to increase their reliability and efficiency. The major issues include the data and meta-data cache coherency between the upper and lower file systems, code duplication between stackable file systems, and the recursive nature of stacking causing `lockdep` to warn about what it infers are possible deadlocks. Addressing these issues will require changes to the VFS/VM.

7 Acknowledgements

The ideas presented in this paper were inspired and motivated by numerous discussions with the following people: Russel Catalan, Dave Chinner, Bruce Fields, Steve French, Christoph Hellwig, Eric Van Hensbergen, Val Henson, Chuck Lever, Andrew Morton, Trond Myklebust, Eric Sandeen, Theodore Ts'o, Al Viro, Peter Zijlstra, and many others.

This work was partially made possibly by NSF Trusted Computing Award CCR-0310493.

References

[1] M. Halcrow. eCryptfs: a stacked cryptographic filesystem. *Linux Journal*, (156):54–58, April 2007.

[2] M. A. Halcrow. Demands, Solutions, and Improvements for Linux Filesystem Security. In *Proceedings of the 2004 Linux Symposium*, pages 269–286, Ottawa, Canada, July 2004. Linux Symposium.

[3] M. A. Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, pages 201–218, Ottawa, Canada, July 2005. Linux Symposium.

[4] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok. UnionFS: User- and Community-oriented Development of a Unification Filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, Ottawa, Canada, July 2006.

[5] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[6] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, (128):24–29, December 2004.

[7] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.