A Stackable File System Interface For Linux

Erez Zadok and Ion Badulescu
Computer Science Department, Columbia University
{ezk,ion}@cs.columbia.edu

Abstract

Linux is a popular operating system that is rapidly evolving due to being Open Source and having many developers. The Linux kernel comes with more than two dozen file systems, all of which are *native*: they access device drivers directly. Native file systems are harder to develop. Stackable file systems, however, are easier to develop because they use existing file systems and interfaces.

This paper describes a stackable *wrapper* file system we wrote for Linux, called *Wrapfs*. This file system requires a single small kernel change (one likely to be incorporated in future kernel releases), and can serve as a template from which other stackable file systems can be written. Wrapfs takes care of most interfacing to the kernel, freeing developers to concentrate on the core semantic issues of their new file system. As examples, we describe several file systems we wrote using Wrapfs.

We also detail the implementation of key operations in the Linux stackable vnode interface, and the change we made to the Linux kernel to support stackable file system modules. Though we have repeated this work for both Linux 2.0 and 2.1/2.2, this paper will concentrate and report results for the 2.1.129 kernel. The overhead imposed by Wrapfs is only 5–7%.

1 Introduction

Most file systems fall into two categories: (1) kernel resident *native* file systems that interact directly with lower level media such as disks[11] and networks[16], and (2) user-level file systems that are based on an NFS server such as the Amd automounter[13].

Native kernel-resident file systems are difficult to develop and debug because they interact directly with device drivers, and require deep understanding of operating systems internals. User-level file systems are easier to develop and debug, but suffer from poor performance due to the extra number of context switches that take place in order to serve a user request.

We advocate a third category: kernel-resident stackable file systems, that is based on the *Virtual File System* (VFS). This model results in file systems with performance close to that of native kernel-resident file systems, and development effort matching that of user-level file systems. Such stackable file systems can be written from our template *wrapper* file system — Wrapfs. Wrapfs takes care of interfacing with the rest of the kernel; it provides the developer with simple hooks to modify or inspect file data, file names, and file attributes. Wrapfs can be mounted on top of one or more existing directories, and act as an intermediary between the user accessing the mount point and the lower level file system it is mounted on. Wrapfs can then transparently change the behavior of the file system as seen by users, while keeping the underlying media unaware of the upper-level changes.

1.1 The Stackable Vnode Interface

Wrapfs is implemented as a stackable vnode interface. A *Virtual Node* or *vnode* (known in Linux as a memory *inode*) is a data structure used within Unix-based operating systems to represent an open file, directory, device, or other entity (e.g., socket) that can appear in the file system namespace. A vnode does not expose what type of physical file system it implements. Thus, the *vnode interface* allows higher level operating system modules to perform operations on vnodes uniformly.

One notable improvement to the vnode concept is *vnode stacking*[8, 14, 18], a technique for modularizing file system functions by allowing one vnode interface to call another. Before stacking existed, there was only a single vnode interface; higher level operating system code called the vnode interface which in turn called code for a specific file system. With vnode stacking, several instances of the vnode interface may exist and may call each other in sequence: the code for a certain operation at stack level N typically calls the corresponding operation at level $N-1$, and so on.

Figure 1 shows the structure for a simple, single-level,

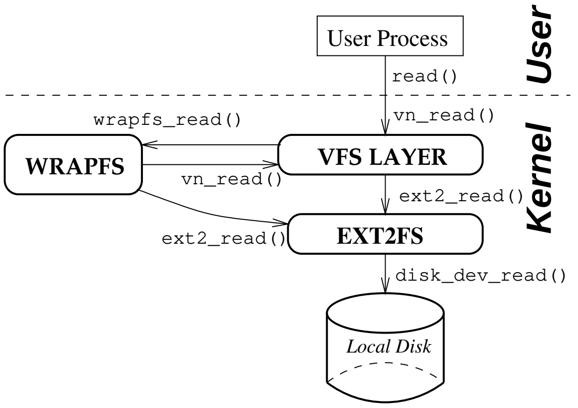

Figure 1: A Vnode Stackable File System

stackable wrapper file system. System calls are translated into vnode level calls, and those invoke their Wrapfs equivalents. Wrapfs again invokes generic vnode operations, and the latter call their respective lower level file system specific operations such as EXT2FS. Wrapfs can also call the lower level file system directly, by invoking the respective vnode operations of the lower vnode. It accomplishes that without knowing who or what type of lower file system it is calling.

The rest of this paper is organized as follows. Section 2 discusses the design of Wrapfs. Section 3 details Wrapfs' implementation. Section 4 describes four examples of file systems written using the Wrapfs template. Section 5 evaluates their performance and portability. We survey related works in Section 6 and conclude in Section 7.

2 Design

The design of Wrapfs concentrated on the following:

1. Simplifying the developer API so that it addresses most of the needs of users developing file systems using Wrapfs.
2. Adding a stackable vnode interface to Linux with minimal changes to the kernel, and with no changes to other file systems.
3. Keeping the performance overhead of Wrapfs as low as possible.

The first two points are discussed below. Performance is addressed in Section 5.

2.1 Developer API

There are three parts of a file system that developers wish to manipulate: file data, file names, and file attributes. Of those, data and names are the most important and also the hardest to handle. File data is difficult to manipulate because there are many different functions that use them such as read and write, and the memory-mapping (MMAP) ones; various functions manipulate files of different sizes at different offsets. File names are complicated to use not just because many functions use them, but also because the directory reading function, `readdir`, is a restartable function.

We created four functions that Wrapfs developers can use. These four functions address the manipulation of file data and file names:

1. **encode_data**: takes a buffer of 4KB or 8KB size (typical page size), and returns another buffer. The returned buffer has the encoded data of the incoming buffer. For example, an encryption file system can encrypt the incoming data into the outgoing data buffer. This function also returns a status code indicating any possible error (negative integer) or the number of bytes successfully encoded.
2. **decode_data**: is the inverse function of `encode_data` and otherwise has the same behavior. An encryption file system, for example, can use this to decrypt a block of data.
3. **encode_filename**: takes a file name string as input and returns a newly allocated and encoded file name of any length. It also returns a status code indicating either an error (negative integer) or the number of bytes in the new string. For example, a file system that converts between Unix and MS-DOS file names can use this function to encode long mixed-case Unix file names into short 8.3-format upper-case names used in MS-DOS.
4. **decode_filename**: is the inverse function of `encode_filename` and otherwise has the same behavior.

With the above functions available, file system developers that use Wrapfs as a template can implement most of the desired functionality of their file system in a few places and not have to worry about the rest.

File system developers may also manipulate file attributes such as ownership and modes. For example, a simple intrusion avoidance file system can prevent setting the setuid bit on any root-owned executables. Such a file system can declare certain important and seldom changing binaries (such as `/bin/login`) as immutable, to deny a potential attacker from replacing them with trojans, and may even require an authentication key to modify them. Inspecting or changing file attributes in Linux is easy, as they are trivially available by dereferencing the inode structure's fields. Therefore, we decided not to create a special API for manipulating attributes, so as not to hinder performance for something that is easily accessible.

2.2 Kernel Issues

Without stackable file system support, the divisions between file system specific code and the more general (upper) code are relatively clear, as depicted in Figure 2. When

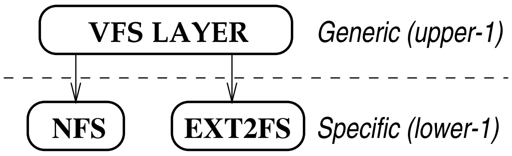

Figure 2: Normal File System Boundaries

a stackable file system such as Wrapfs is added to the kernel, these boundaries are obscured, as seen in Figure 3. Wrapfs assumes a dual responsibility: it must appear to the

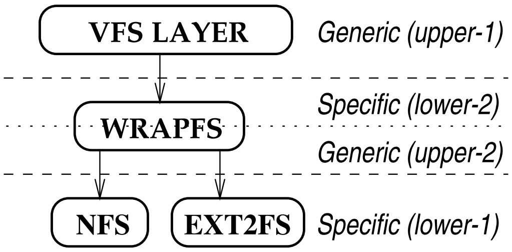

Figure 3: File System Boundaries with Wrapfs

layer above it (upper-1) as a native file system (lower-2), and at the same time it must treat the lower level native file system (lower-1) as a generic vnode layer (upper-2).

This dual role presents a serious challenge to the design of Wrapfs. The file system boundary as depicted in Figure 2 does not divide the file system code into two completely independent sections. A lot of state is exchanged and assumed by both the generic (upper) code and native (lower) file systems. These two parts must agree on who allocates and frees memory buffers, who creates and releases locks, who increases and decreases reference counts of various objects, and so on. This coordinated effort between the upper and lower halves of the file system must be perfectly maintained by Wrapfs in its interaction with them.

2.2.1 Call Sequence and Existence

The Linux vnode interface contains several classes of functions:

- **mandatory**: these are functions that must be implemented by each file system. For example, the `read_inode` superblock operation which is used to initialize a newly created inode (read its fields from the mounted file system).
- **semi-optional**: functions that must either be implemented specifically by the file system, or set to use a generic version offered for all common file systems. For example, the `read` file operation can be implemented by the specific file system, or it can be set to a general purpose read function called `generic_file_read` which offers read functionality for file systems that use the page cache.
- **optional**: functions that can be safely left unimplemented. For example, the inode `readlink` function is necessary only for file systems that support symbolic links.
- **dependent**: these are functions whose implementation or existence depends on other functions. For example, if the file operation `read` is implemented using `generic_file_read`, then the inode operation `readpage` must also be implemented. In this case, all reading in that file system is performed using the MMAP interface.

Wrapfs was designed to accurately reproduce the aforementioned call sequence and existence checking of the various classes of file system functions.

2.2.2 Data Structures

There are five primary data structures that are used in Linux file systems:

1. **super_block**: represents an instance of a mounted file system (also known as `struct vfs` in BSD).
2. **inode**: represents a file object in memory (also known as `struct vnode` in BSD).
3. **dentry**: represents an inode that is cached in the Directory Cache (dcache) and also includes its name. This structure is extended in Linux 2.1, and combines several older facilities that existed in Linux 2.0. A dentry is an abstraction that is higher than an inode. A *negative dentry* is one which does not (yet) contain a valid inode; otherwise, the dentry contains a pointer to its corresponding inode.
4. **file**: represents an open file or directory object that is in use by a process. A file is an abstraction that is one level higher than the dentry. The file structure contains a valid pointer to a dentry.
5. **vm_area_struct**: represents custom per-process virtual memory manager page-fault handlers.

The key point that enables stacking is that each of the major data structures used in the file system contain a field into which file system specific data can be stored. Wrapfs uses that private field to store several pieces of information, especially a pointer to the corresponding lower level file system's object. Figure 4 shows the connections between some objects in Wrapfs and their corresponding objects in the stacked-on file system, as well as the regular connections between the objects within the same layer. When a file system operation in Wrapfs is called, it finds the corresponding lower level's object from the current one, and repeats the same operation on the lower object.

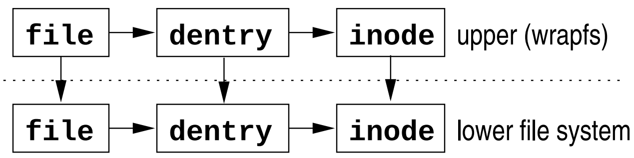

Figure 4: Connections Between Wrapfs and the Stacked-on File System

Figure 4 also suggests one additional complication that Wrapfs must deal with carefully — reference counts. Whenever more than one file system object refers to a single instance of another object, Linux employs a traditional reference counter in the referred-to object (possibly with a corresponding mutex lock variable to guarantee atomic updates to the reference counter). Within a single file system layer, each of the file, dentry, and inode objects for the same file will have a reference count of one. With Wrapfs in place, however, the dentry and inode objects of the lower level file system must have a reference count of two, since there are two distinct objects referring to each. These additional pointers between objects are ironically necessary to keep Wrapfs as independent from other layers as possible. The horizontal arrows in Figure 4 represent links that are part of the Linux file system interface and cannot be avoided. The vertical arrows represent those that are necessary for stacking. The higher reference counts ensure that the lower level file system and its objects could not disappear and leave Wrapfs's objects pointing to invalid objects.

2.2.3 Caching

Wrapfs keeps independent copies of its own data structures and objects. For example, each dentry contains the component name of the file it represents. (In an encryption file system, for example, the upper dentry will contain the cleartext name while the lower dentry contain the ciphertext name.) We pursued this independence and designed Wrapfs to be as separate as possible from the file system layers above and below it. This means that Wrapfs keeps its own copies of cached objects, reference counts, and memory mapped pages — allocating and freeing these as necessary.

Such a design not only promotes greater independence, but also improves performance, as data is served off of a cache at the top of the stack. Cache incoherency could result if pages at different layers are modified independently[7]. We therefore decided that higher layers would be more authoritative. For example, when writing to disk, cached pages for the same file in Wrapfs overwrite their EXT2 counterparts. This policy correlates with the most common case of cache access, through the uppermost layer.

3 Implementation

Each of the five primary data structures used in the Linux VFS contains an operations vector describing all of the functions that can be applied to an instance of that data structure. We describe the implementation of these operations not based on the data structure they belong to, but based on one of five implementation categories:

1. mounting and unmounting a file system
2. functions creating new objects
3. data manipulation functions
4. functions that use file names
5. miscellaneous functions

For conciseness, we describe the implementation of the 1–2 most important functions in each category. Readers are referred to other documentation[2] for a description of the rest of the file system operations in Linux, and to Wrapfs's sources for their implementation.

There are two important auxiliary functions in Wrapfs. The first function, `interpose`, takes a lower level dentry and a wrapfs dentry, and creates the links between them and their inodes. When done, the Wrapfs dentry is said to be *interposed* on top of the dentry for the lower level file system. The interpose function also allocates a new Wrapfs inode, initializes it, and increases the reference counts of the dentries in use. The second important auxiliary function is called `hidden_dentry` and is the opposite of interpose. It retrieves the lower level (hidden) dentry from a Wrapfs dentry. The hidden dentry is stored in the private data field of `struct dentry`.

3.1 Mounting and Unmounting

The function `read_super` performs all of the important actions that occur when mounting Wrapfs. It sets the operations vector of the superblock to that of Wrapfs's, allocates a new root dentry (the root of the mounted file system), and finally calls `interpose` to link the root dentry to that of the mount point. This is vital for lookups since they are relative to a given directory (see Section 3.2). From that point on, every lookup within the Wrapfs file system will use Wrapfs's own operations.

3.2 Creating New Objects

Several inode functions result in the creation of new inodes and dentries: `lookup`, `link`, `symlink`, `mkdir`, and `mknod`. The `lookup` function is the most complex in this group because it also has to handle negative dentries (ones that do not yet contain valid inodes). Lookup is given a directory inode to look in, and a dentry (containing the pathname) to look for. It proceeds as follows:

1. encode the file name it was given using `encode_filename` and get a new one.

2. find the lower level (hidden) dentry from the Wrapfs dentry.
3. call Linux's primary lookup function, called `lookup_dentry`, to locate the encoded file name in the hidden dentry. Return a new dentry (or one found in the directory cache, dcache) upon success.
4. if the new dentry is negative, interpose it on top of the hidden dentry and return.
5. if the new dentry is not negative, interpose it and the inodes it refers to, as seen in Figure 4.

3.3 Data Manipulation

File data can be manipulated in one of two ways: (1) the traditional read and write interface can be used to read or write any number of bytes starting at any given offset in a file, and (2) the MMAP interface can be used to map pages of files into a process that can use them as normal data buffers. The MMAP interface can manipulate only whole pages and on page boundaries. Since MMAP support is vital for executing binaries, we decided to manipulate data in Wrapfs in whole pages.

Reading data turned out to be easy. We set the file `read` function to the general purpose `generic_file_read` function, and were subsequently required to implement only our version of the `readpage` inode operation. Readpage is asked to retrieve one page in a given opened file. Our implementation looks for a page with the same offset in the hidden file. If it cannot find one, Wrapfs's readpage allocates a new one. It proceeds by calling the lower file system's readpage function to get the page's data, and then it decodes the data from the hidden page into the Wrapfs page. Finally, Wrapfs's readpage function mimics some of the functionality that `generic_file_read` performs: it unlocks the page, marks it as referenced, and wakes up anyone who might be waiting for that page.

3.4 File Name Manipulation

As mentioned in Section 3.2, we use the call to `encode_filename` at every file system function that is given a file name and has to pass it to the lower level file system, such as `rmdir`. There are only two places where file names are decoded: `readlink` needs to decode the target of a symlink after having read it from the lower level file system, and `readdir` needs to decode each file name read from a directory. Readdir is implemented in a similar fashion to other Linux file systems, by using a callback function called "filldir" that is used to process one file name at a time.

3.5 Miscellaneous Functions

In Section 3.3 we described some MMAP functions that handle file data. Other than those, we had to implement three MMAP-related functions that are part of the `vm_area_struct`, but only for shared memory-mapped pages: `vm_open`, `vm_close`, and `vm_shared_unmap`. We implemented them to properly support multiple (shared) mappings to the same page. Shared pages have increased reference counts and they must be handled carefully (see Figure 4 and Section 2.2.2). The rest of the `vm_area_struct` functions were left implemented or unimplemented as defined by the generic operations vectors of this structure.

This implementation underscored the only change, albeit a crucial one, that we had to make to the Linux kernel. The data structure `vm_area_struct` is the only one (as of kernel 2.1.129) that does not contain a private data field into which we can store a link from our Wrapfs vm_area object to the hidden one of the lower level file system. This change was necessary to support stacking.[1]

All other functions that had to be implemented reproduce the functionality of the generic (upper) level vnode code (see Section 2.2) and follow a similar procedure: for each object passed to the function, they find the corresponding object in the lower level file system, and repeat the same operation on the lower level objects.

4 Examples

This section details the design and implementation of four sample file systems we wrote using Wrapfs:

1. **Lofs**: is a loopback mount file system such as the one available in Solaris[19].
2. **Rot13fs**: is a trivial encryption file system that encrypts file data.
3. **Cryptfs**: is a strong encryption file system that also encrypts file names.
4. **Usenetfs**: breaks large flat article directories, most often found in very active news spools, into deeper directory hierarchies, so as to improve access time to individual files.

These examples are merely experimental file systems intended to illustrate the kinds of file systems that can be written using Wrapfs. We do not consider them to be complete solutions. There are many potential enhancements to our examples.

4.1 Lofs

Lofs[2] provides access to a directory of one file system from another, without using symbolic links. It is most often used by automounters to provide a consistent name space for all

[1]We submitted this small change for inclusion in future versions of Linux.

[2]Our Linux 2.0 ports of lofs and Wrapfs were based on an older prototype available in http://www.kvack.org/~blah/lofs/.

local and remote file systems, and by chroot-ed processes to access portions of a file system outside the chroot-ed environment

This trivial file system was actually implemented by removing unnecessary code from Wrapfs. A loop-back file system does not need to manipulate data or file names. We removed all of the hooks that called `encode_data`, `decode_data`, `encode_filename`, and `decode_filename`. This was done to improve performance by avoiding unnecessary copying.

4.2 Rot13fs

Before we embarked on a strong encryption file system, described in the next section, we implemented one using a trivial encryption algorithm. We decided at this stage to encrypt only file data. The implementation was simple: we filled in `encode_data` and `decode_data` with the same rot13 algorithm (since the algorithm is symmetric).

4.3 Cryptfs

Cryptfs uses Blowfish[17] — a 64 bit block cipher that is fast, compact, and simple. We used 128 bit keys. Cryptfs uses Blowfish in Cipher Block Chaining (CBC) mode, so we can encrypt whole blocks. We start a new encryption sequence for each block (using a fixed Initialization Vector, IV) and encrypt whole pages (4KB or 8KB) together. That is, within each page, bytes depend on the preceding ones. To accomplish this part, we modified a free reference implementation (SSLeay) of Blowfish, and put the right calls to encrypt and decrypt a block of data into `encode_data` and `decode_data`, respectively.

Next, we decided to encrypt file names as well (other than "." and ".." so as to keep the lower level file system intact). Once again, we placed the right calls to encrypt and decrypt file names into the respective `encode_filename` and `decode_filename` functions. Applying encryption to file names may result in names containing characters that are illegal in Unix file names (such as nulls and forward slashes "/"). To solve this, we also uuencode file names after encrypting them, and uudecode them before decrypting them.

Key management was the last important design and implementation issue for Cryptfs. We decided that only the root user will be allowed to mount an instance of Cryptfs, but could not automatically encrypt or decrypt files. We implemented a simple ioctl in Cryptfs for setting keys. A user tool prompts for a passphrase and using that ioctl, sends an MD5 hash of the passphrase to a mounted instance of Cryptfs. To thwart an attacker who gains access to a user's account or to root privileges, Cryptfs maintains keys in an in-memory data structure that associates keys not with UIDs alone but with the combination of UID and session ID. To succeed in acquiring or changing a user's key, an attacker would not only have to break into an account, but also arrange for his processes to have the same session ID as the process that originally received the user's passphrase. Since session IDs are set by login shells and inherited by forked processes, a user would normally have to authorize themselves only once in a shell. From this shell they could run most other programs that would work transparently and safely with the same encryption key.

Details of the design and implementation of Cryptfs are available as a separate technical report[23].

4.4 Usenetfs

Busy traditional Usenet news servers could have large directories containing many thousands of articles in directories representing very active newsgroups such as *control.cancel* and *misc.jobs.offered*. Unix directory searches are linear and unsorted, resulting in significant delays processing articles in these large newsgroups. We found that over 88% of typical file system operations that our departmental news server performs are for looking up articles. Usenetfs improves the performance of looking up and manipulating files in such large flat directories, by breaking the structure into smaller directories.

Since article names are composed of sequential numbers, Usenetfs takes advantage of this to generate a simple hash function. After some experimentation, we decided to create a hierarchy consisting of one thousand directories as depicted in Figure 5. We therefore distribute articles across 1000 directories named 000 through 999. Since article numbers are sequential, we maximize the distribution by computing the final directory into which the article will go based on three lesser significant digits, skipping the least significant one. For example, the article named `control/cancel/123456` is placed into the directory `control/cancel/345/`. The article name itself does not change; it only gets moved one level down. We picked the directory based on the second, third, and fourth digits of the article number to allow for some amount of *clustering*. By not using the least significant digit we cluster ten sequential articles together: the ten articles 123450–123459 get placed in the same directory. This increases the chances of kernel cache hits due to the likelihood of sequential access of these articles, a further performance improvement. In general, every article numbered X..XYYYZ gets placed in a directory named YYY.

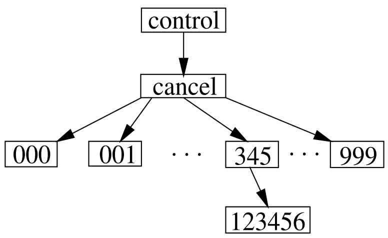

Figure 5: A Usenetfs Managed Newsgroup

Additionally, we decided to use a seldom used mode bit for directories, the setuid bit, to flag a directory as managed by Usenetfs. Using this bit allows the news administrator to control which directory is managed by Usenetfs and which is not, using a simple `chmod` command.

The implementation of Usenetfs concentrated in two places. First, we implemented the `encode_filename` and `decode_filename` to convert normal file names into their extended deeper-hierarchy forms, and back (but only for directories that were flagged as managed by Usenetfs). Second, we updated the `readdir` function to iterate over all possible subdirectories from 000 to 999, and perform a (smaller) directory read within each one.

Details of the design and implementation of Usenetfs are available as a separate technical report[22].

5 Performance

5.1 Wrapfs and Cryptfs

For most of our tests, we included figures for a native disk-based file system because disk hardware performance can be a significant factor. Since Cryptfs is a stackable file system, we included figures for Wrapfs and for Lofs, to be used as a base for evaluating the cost of stacking. When using lofs, Wrapfs, or Cryptfs, we mounted them over a local disk-based file system. CFS[3] and TCFS[4] are two encryption file systems based on NFS, so we also included the performance of native NFS. All NFS mounts used the local host as both server and client (i.e., mounting `localhost:/path` on `/mnt`), and used protocol version 2 over a UDP transport, with a user-space NFS server[3]. CFS was configured to use Blowfish (same as Cryptfs), but we had to configure TCFS to use DES, because it does not support Blowfish.

For the first set of tests, we measured the time it took to perform 10 successive builds of a large package (Amutils[20]) and averaged the elapsed times. These results are listed in Table 1. For these tests, the standard deviation did not exceed 0.8% of the mean. Lofs is only 1.1–1.2% slower than the native disk based file system. Wrapfs adds an overhead of 4.7–6.8%, but that is comparable to the 3–10% degradation previously reported for null-layer stackable file systems[8, 18] and is the cost of copying data pages and file names.

File System	SPARC 5	Intel P5/90
ext2	1097.0	524.2
lofs	1110.1	530.6
wrapfs	1148.4	559.8
cryptfs	1258.0	628.1
nfs	1440.1	772.3
cfs	1486.1	839.8
tcfs	2092.3	1307.4

Table 1: Time to Build a Large Package (Sec)

[3]Universal NFSD 2.2betaXX included in the RedHat 5.2 distribution.

Wrapfs is the baseline for evaluating the performance impact of the encryption algorithm, because the only difference between Wrapfs and Cryptfs is that the latter encrypts and decrypts data and file names. Cryptfs adds an overhead of 9.5–12.2% over Wrapfs. That is a significant overhead but is unavoidable. It is the cost of the Blowfish encryption code, which, while designed as a fast software cipher, is still CPU intensive.

Next, we compare the three encryption file systems. Cryptfs is 40–52% faster than TCFS. Since TCFS uses DES and Cryptfs uses Blowfish, however, it is more proper to compare Cryptfs to CFS. Still, Cryptfs is 12–30% faster than CFS. Because both CFS and Cryptfs use the same encryption algorithm, most of the difference between them stems from the extra context switches that CFS incurs.

For the second set of tests we performed microbenchmarks on the file systems listed in Table 1, specifically reading and writing of small and large files. These tests were designed to isolate and show the performance difference between Cryptfs, CFS, and TCFS for individual file system operations. Table 2 summarizes some of these results.

File	Writes		Reads	
System	1024×8KB	8×1MB	1024×8KB	8×1MB
cryptfs	9.27	8.33	0.26	0.34
cfs	101.90	50.84	0.89	8.77
tcfs	110.86	84.64	6.45	7.94

Table 2: x86 Times for Read and Write Calls (Sec)

A complete and detailed analysis of the results listed in Table 2 is beyond the scope of this paper, and will have to take into account the size and effectiveness of the operating system's page and buffer caches. Nevertheless, these results clearly show that Cryptfs improves performance from as little as 43% to as much as over an order of magnitude. Additional performance analysis of Cryptfs is available elsewhere[23].

5.2 Usenetfs

To test the performance of Usenetfs, we setup a test Usenet news server and configured it with test directories of increasingly greater number of files in each. Then we compared the performance of typical news server operations when these large directories were managed by Usenetfs and when they were not (i.e., straight onto ext2fs).

We performed 1000 random lookups of articles in large directories. When the directory had fewer than 2000 articles, Usenetfs added a small overhead of 70–80 milliseconds. The performance of ext2fs continued to degrade linearly, and when the directory had over 250,000 articles, performance of Usenetfs was over 100 times faster. When we performed sequential lookups, thus involving kernel caches, Usenetfs's performance was only two times better than ext2fs's for directories with 500 or more articles.

The results for deleting and adding new articles showed that Usenetfs' performance remained almost flat for all directory sizes we tested, while ext2fs's performance degraded linearly. With just 10,000 articles in the directory, adding or deleting articles was more than 10 times faster with Usenetfs.

Since Usenetfs uses 1000 more directories for managed ones, we expected the performance of reading a directory to be worse. Usenetfs takes an almost constant 500 milliseconds to read a managed directory, while ext2fs once again degraded linearly. It is not until there are over 100,000 articles in the directory, that Usenetfs's readdir is faster than ext2fs's. Although Usenetfs's performance also starts degrading linearly after a certain directory size, this is not a problem because the algorithm can be easily tuned and extended.

The last test we performed took into account all of the above factors. Once again, we built a large package on a busy news server that was configured to manage the top 6 newsgroups using Usenetfs. This test was designed to measure the reserve capacity on the news server, or how much more free did the CPU become due to using Usenetfs. With Usenetfs, compile times improved by an average of 22%. During periods of heavy activity on the news server, such as article expirations, compile times improved by a factor of 2–3. Additional performance analysis of Usenetfs is available elsewhere[22].

5.3 Portability

Table 3 shows the overall estimated times that it took us to develop the file systems mentioned in this paper. Since the first ports were for Linux 2.0, they took longer as we were also learning our way around Linux and stackable file systems in general. The bulk of the time was spent initially on porting the Wrapfs template. Using this template, other filesystems were implemented faster.

File Systems	Linux 2.0	Linux 2.1/2.2
wrapfs	2 weeks	1 week
lofs	1 hour	30 minutes
rot13fs	2 hours	1 hour
cryptfs	1 week	1 day
usenetfs	2 days	1 day

Table 3: Time to Develop and Port File Systems

Another interesting measure of the complexity of Wrapfs is the size of the code. The total number of source code lines for Wrapfs in Linux 2.0 is 2157, but that number grew to by more than 50% to 3279 lines when we ported Wrapfs to the 2.1 kernel. This is a testament to the unfortunate complexity that Linux 2.1 added, mostly due to the integration with the dentry concept.

6 Related Work

6.1 Other Stackable File Systems

Vnode stacking was first implemented by Rosenthal (in SunOS 4.1) around 1990[15]. A few other works followed Rosenthal, such as further prototypes for extensible file systems in SunOS[18], and the Ficus layered file system[6, 9] at UCLA.

Several newer operating systems offer a stackable file system interface. Such operating systems have the potential of easy development of file systems offering a wider range of services. Their main disadvantages are that they are not portable enough, not sufficiently developed or stable, or they are not available for common use. Also, new operating systems with new file system interfaces are not likely to perform as well as ones that are several years older.

The *Herd of Unix-Replacing Daemons* (HURD) from the Free Software Foundation (FSF) is a set of servers, running on the Mach 3.0 microkernel[1], that collectively provide a Unix-like environment. HURD file systems are implemented at the user level. The HURD introduced the concept of a translator. A translator is a program that can be attached to a pathname and perform specialized services when that pathname is accessed. Writing a new translator is a matter of implementing a well defined file access interface and filling in such operations as opening files, looking up file names, creating directories, etc.

Spring is an object-oriented research operating system built by Sun Microsystems Laboratories[12]. It was designed as a set of cooperating servers on top of a microkernel. Spring provides several generic modules that offer services useful to a file system: caching, coherency, I/O, memory mapping, object naming, and security. Writing a file system for Spring entails defining the operations to be applied on the file objects. Operations not defined are inherited from their parent object.

One work that has resulted from Spring is the Solaris MC (Multi-Computer) File System[10]. It borrows the object-oriented interfaces from Spring and integrates them with the existing Solaris vnode interface to provide a distributed file system infrastructure through a special file system called *pxfs*, the Proxy File System. Solaris MC provides all of the benefits that come with Spring, while requiring little or no change to existing file systems; those can be gradually ported over time. Solaris MC was designed to perform well in a closely coupled cluster environment (not a general network) and requires high performance networks and nodes.

6.2 Other News File Systems

The Cyclic News File System (CNFS)[5] stores articles in a few large files or in a raw block device, recycling their storage when reaching the end of the buffer. CNFS avoids most of the overhead of traditional FFS-like[11] file systems, because it reduces the need for many synchronous meta-data updates. CNFS reports an order of magnitude reduction in disk activity. CNFS is part of the INN 2.x Usenet server news software. In the long run, CNFS offers a superior solution to the performance problems of news servers than Usenetfs. Migrating from traditional news spools and software to ones using CNFS, however, is a time consuming process. Furthermore, since CNFS no longer uses traditional file systems, it is not possible to NFS-export the news spool to other hosts; non-NNTP compliant news readers cannot work with CNFS, but they can with Usenetfs.

Reiserfs[4] is a file system available only for Linux that uses balanced trees to optimize performance and space utilization for files of any size and file names. Being a native disk-based file system, and using complex algorithms, Reiserfs improves performance significantly, but it is hard to develop and debug.

6.3 Other Encryption File Systems

CFS[3] is a portable user-level cryptographic file system based on NFS. It is used to encrypt any local or remote directory on a system, accessible via a different mount point and a user-attached directory. Users first create a secure directory and choose the encryption algorithm and key to use. A wide choice of ciphers is available and great care was taken to ensure a high degree of security. CFS's performance is limited by the number of context switches that must be performed and the encryption algorithm used.

TCFS[4] is a modified client-side NFS kernel module that communicates with a remote NFS server. TCFS is available only for Linux systems, and both client and server must run on Linux. TCFS allows finer grained control over encryption; individual files or directories can be encrypted by turning on or off a special flag.

7 Conclusions

Wrapfs and the examples in this paper show that useful, non-trivial vnode stackable file systems can be implemented under Linux with minor changes to the rest of the operating system, and with no changes to other file systems. Better performance was achieved by running the file systems in the kernel instead of at user-level.

Estimating the complexity of software is a difficult task. Nevertheless, it is our assertion that with Wrapfs, other non-trivial file systems built from it can be prototyped in a matter of hours or days. We also estimate that Wrapfs can be ported to any operating system in less than one month, as long as the file system has a vnode interface that provides a private opaque field for each of the major data structures used in the file system. In comparison, traditional file system development often takes many months to several years.

Wrapfs saves developers from dealing with many kernel related issues, and allows them to concentrate on the specifics of the file system they are developing. We hope that with Wrapfs and the example file systems we have built, other developers would be able to prototype new file systems to try new ideas, and develop fully working ones — bringing the complexity of file system development down to the level of common user-level software.

We believe that a truly stackable file system interface could significantly improve portability, especially if adopted by the main Unix vendors and developers. If such an interface becomes popular, it might result in many more practical file systems developed. We hope through this paper to have proven the usefulness and practicality of non-trivial stackable file systems.

One item we would like to add to Wrapfs is support for stackable file systems that change the size of the data, such as with compression. Several initial designs we made indicated a greater level of code complexity and serious performance impact. We opted to hold off on such support for the initial implementation of Wrapfs.

[4] http://www.idiom.com/~beverly/reiserfs.html

8 Acknowledgments

We would like to thank Fred Korz, Seth Robertson, and especially Dan Duchamp for their help in reviewing this paper and offering concrete suggestions that made this work better. This work was partially made possible thanks to NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *USENIX Conference Proceedings* (Atlanta, GA), pages 93–112. USENIX, Summer 1986.

[2] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. The Linux File System. In *Linux Kernel Internals, Second Edition*, pages 152–73. Addison-Wesley, 1998.

[3] M. Blaze. A Cryptographic File System for Unix. *Proceedings of the first ACM Conference on Computer and Communications Security* (Fairfax, VA). ACM, November, 1993.

[4] G. Cattaneo and G. Persiano. Design and Implementation of a Transparent Cryptographic File System for Unix. Unpublished Technical Report. Dip. Informatica ed Appl, Università di

Salerno, 8 July 1997. Available via ftp in ftp://edu-gw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz.

[5] S. L. Fritchie. The Cyclic News Filesystem: Getting INN To Do More With Less. *System Administration (LISA XI) Conference* (San Diego, California), pages 99–111. USENIX, 26-31 October 1997.

[6] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[7] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Association for Computing Machinery SIGOPS, 3–6 December 1995.

[8] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *Transactions on Computing Systems*, **12**(1):58–89. (New York, New York), ACM, February, 1994.

[9] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Technical report CSD-910007. University of California, Los Angeles, March 1991.

[10] V. Matena, Y. A. Khalidi, and K. Shirriff. Solaris MC File System Framework. Technical Reports TR-96-57. Sun Labs, October 1996. Available http://www.sunlabs.com/technical-reports/1996/abstract-57.html.

[11] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[12] J. G. Mitchel, J. J. Giobbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conference Proceedings* (San Francisco, California). CompCon, 1994.

[13] J.-S. Pendry and N. Williams. Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha. Imperial College of Science, Technology, and Medicine, London, England, March 1991.

[14] D. S. H. Rosenthal. Requirements for a “Stacking” Vnode/VFS Interface. Unix International document SD-01-02-N014. UNIX International, 1992.

[15] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conference Proceedings* (Anaheim, CA), pages 107–18. USENIX, Summer 1990.

[16] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Association Summer Conference Proceedings of 1985* (11-14 June 1985, Portland, OR), pages 119–30. USENIX Association, El Cerrito, CA, 1985.

[17] B. Schneier. Blowfish. In *Applied Cryptography, Second Edition*, pages 336–9. John Wiley & Sons, 1996.

[18] G. C. Skinner and T. K. Wong. ”Stacking” Vnodes: A Progress Report. *USENIX Conference Proceedings* (Cincinnati, OH), pages 161–74. USENIX, Summer 1993.

[19] SMCC. lofs – loopback virtual file system. SunOS 5.5.1 Reference Manual, Section 7. Sun Microsystems, Incorporated, 20 March 1992.

[20] E. Zadok. Am-utils (4.4BSD Automounter Utilities). User Manual, for Am-utils version 6.0a16. Columbia University, 22 April 1998. Available http://www.cs.columbia.edu/~ezk/am-utils/.

[21] E. Zadok. *FiST: A File System Component Compiler*. PhD thesis, published as Technical Report CUCS-033-97 (Ph.D. Thesis Proposal). Computer Science Department, Columbia University, 27 April 1997. Available http://www.cs.columbia.edu/~library/.

[22] E. Zadok and I. Badulescu. Usenetfs: A Stackable File System for Large Article Directories. Technical Report CUCS-022-98. Computer Science Department, Columbia University, 23 June 1998. Available http://www.cs.columbia.edu/~library/.

[23] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 28 July 1998. Available http://www.cs.columbia.edu/~library/.

9 Author Information

Erez Zadok is an Ph.D. candidate in the Computer Science Department at Columbia University. His primary interests include operating systems and file systems. The work described in this paper was first mentioned in his Ph.D. thesis proposal[21].

Ion Badulescu holds a B.A. from Columbia University. His primary interests include operating systems, networking, compilers, and languages.

To access sources for the software described in this paper, as well as related technical reports, see *http://www.cs.columbia.edu/~ezk/research/software/*