# FiST: A System for Stackable File-System Code Generation

**Erez Zadok**

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

**COLUMBIA UNIVERSITY**

May, 2001

# ABSTRACT

## FiST: A System for Stackable File-System Code Generation

Erez Zadok

File systems often need to evolve and require many changes to support new features. Traditional file-system development is difficult because most of the work is done in the kernel—a hostile development environment where progress is slow, debugging is difficult, and simple mistakes can crash systems. Kernel work also requires deep understanding of system internals, resulting in developers spending a lot of time becoming familiar with the system's details. Furthermore, any file system written for one system requires significant effort to port to another system. Stackable file systems promise to ease the development of file systems by offering a mechanism for incremental development building on existing file systems. Unfortunately, existing stacking methods often require writing complex low-level kernel code that is specific to a single operating system platform and also difficult to port.

We propose a new language, *FiST*, to describe stackable file systems. FiST uses operations common to file-system interfaces and familiar to user-level developers: creating a directory, reading a file, removing a file, listing the contents of a directory, etc. From a single description, FiST's compiler produces file-system modules for multiple platforms. FiST does that with the assistance of platform-specific stackable templates. The templates handle many of the internal details of operating systems, and free developers from dealing with these internals. The templates support many features: data copying and file name copying useful for applications that want to modify them; size-changing file systems such as compression; fan-out for access to multiple file systems from one layer; and more. The FiST language compiler uses the templates as a basis for producing code for a new file system, by inserting, removing, or modifying code in the templates.

This dissertation describes the design, implementation, and evaluation of FiST. Our thesis is that it is possible to extend file system functionality in a portable way without changing existing kernels. This is possible because the FiST language uses file-system functions that are common across many systems, while the templates execute in-kernel operating systems specific functions unchanged. We built several file systems using FiST on Solaris, FreeBSD, and Linux. Our experiences with these file systems show the following benefits: average code size is reduced ten times as compared to writing code given another null-layer stackable file system; average development time is reduced seven times compared to writing using another null-layer stackable file system; performance overhead of stacking is only 1–2% per layer.

# Contents

# List of Tables

# List of Figures

To Martha, the most understanding person.

# Chapter 1

# Introduction

File systems have proven to be useful in enriching system functionality. The abstraction of folders with files containing data is natural for use with existing file browsers, text editors, and other tools. Modifying file systems is a popular method of adding new functionality requested by users. For example, it is desirable to extend existing file systems to include new security features such as encryption and Access Control Lists, or to improve file-system reliability and redundancy through load-balancing and replication techniques. As the number of different computer systems, networks, and users continues to grow at astounding rates, it becomes ever more important to be able to develop new file systems and extend existing ones quickly to accommodate the ever-changing needs of users.

Current practices of developing file systems, however, lag behind other software technologies (e.g., object-oriented modular programming). Developing file systems is very difficult and involved. Developers often use existing code for native in-kernel file systems as a starting point [52, 71]. Such file systems are difficult to write and port because they depend on many operating-system specifics, and they often contain many lines of complex operating-systems code, as seen in Table 1.1. Writing in-kernel file systems demands thorough understanding of kernel internals, and this takes a long time to learn. Consequently, only a small community of kernel experts is able to develop in-kernel file systems.

| **Media Type** | **Common File System** | **Avg. Code Size** (C lines) |
|---|---|---|
| Hard Disks | UFS, FFS, EXT2FS | 5,000–20,000 |
| Network | NFS | 6,000–30,000 |
| CD-ROM | HSFS, ISO-9660 | 3,000–6,000 |
| Floppy | PCFS, MS-DOS | 5,000–6,000 |

Table 1.1: Common native unix file systems and code sizes for each medium. We counted those over the last few major releases of Solaris, Linux, and FreeBSD.

To improve the file-system development process, some developers have suggested writing and running them outside the kernel. User-level file systems are easier to develop and port because they reside outside the kernel [53]. However, their performance is poor due to the extra context switches these file systems must incur. These context switches can affect performance by as much as an order of magnitude [83, 84]. In addition, the reliability of these user-level file servers is poor because they must contend for system resources with all other processes on the system—resources such as memory, swap space, CPU cycles, etc. If the system becomes busy because of normal user activities, user-level file server processes can get descheduled, swapped out, or even killed. When user-level file servers are interrupted, the whole system gradually becomes unusable as processes that need access to the file system hang. As a result, user-level file systems have not become suitable replacements for commercial or production systems.

*Stackable file systems* [64] promise to speed file-system development by providing an extensible file-system interface. This extensibility allows new features to be added incrementally. Several extensible interfaces have been proposed and a few have been implemented [31, 52, 63, 69]. To improve performance and reliability, these stackable file systems were designed to run in the kernel. Unfortunately, using these stackable interfaces still requires writing lots of complex C kernel code that is specific to a single operating system. In addition, stackable file system code is also difficult to port from system to system because it depends on the specifics of the stackable interface and the operating system: all past proposals for stackable interfaces defined different APIs. Finally, these proposals advocated rewriting existing operating and existing file systems. This affected both their reliability, because new code is often less stable, and their performance; the changes made to systems resulted in a 3–10% overall performance degradation, even when stackable file systems were *not* in use. As a result, stacking interfaces are rarely available in modern operating systems, and are seldom used in production systems.

Newer proposals for extensible file-system interfaces include research operating systems such as Spring [44], GNU HURD [12], and Plan 9 [54, 55, 57]. All of these offered well-defined interfaces for extending file-system functionality, some using object-oriented technologies. The two main problems with these operating systems are (1) they are not generally released, supported, or available for production use, and (2) they all define different mechanisms for extending file-system functionality, resulting in a difficult porting effort. These systems, therefore, still do not help alleviate the problems of file-system development.

One additional problem with all extensible file-system interfaces is the lack of high-level support. For extensible filing systems to be useful, developers need libraries of common file-system operations that can be used in the same way the C library `libc` provides a common set of operations that are useful for building applications. Common operations often desired by file-system developers include compression and encryption of files, copying files, extending file and directory attributes, reading and writing new files as easily as can be done in user level, and more. This high-level functionality is missing from past extensible filing systems.

In short, the file-system development process is still long and costly. Existing methods for extending file systems do not help speed up this process significantly.

## 1.1 Our Approach

Past approaches to simplify file-system development, as discussed above, could not achieve both performance and portability. To perform well, a file system should run in the kernel, not at user level. Kernel code, however, is much more difficult to write and port than user-level code. To ease the problems of developing and porting stackable file systems that perform well, we propose a high-level language to describe such file systems. We combine this language with compiler-like techniques to generate a high performance implementation which enhances portability. There are three benefits to using a language:

1. **Simplicity:** A file-system language can provide familiar higher-level primitives that simplify file-system development. The language can also define suitable defaults automatically. These reduce the amount of code that developers need to write, and lessen their need for extensive knowledge of kernel internals, allowing even non-experts to develop file systems.
2. **Portability:** A language can describe file systems using an interface abstraction that is common to many operating systems. The language compiler can bridge the gaps among different systems' interfaces. From a single description of a file system, we could generate file system code for different platforms. This improves portability considerably. At the same time, however, the language can allow developers to take advantage of system-specific features.
3. **Specialization:** A language allows developers to customize the file system to their needs. Instead of having one large and complex file system with many features that may be configured and turned on or off, the compiler can

produce special-purpose file systems. This improves performance and reduces memory usage because a specialized file system includes only necessary code.

This dissertation describes the design and implementation of *FiST*, a *File-System Translator* language for stackable file systems. FiST lets developers describe stackable file systems at a high level, using operations common to file-system interfaces. With FiST, developers need only describe the core functionality of their file systems. The FiST language code generator, *fistgen*, generates kernel file-system modules for several platforms using a single description. FiST currently supports Solaris, FreeBSD, and Linux.[1]

To assist fistgen with generating stackable file systems, we created a minimal stackable file-system template called Basefs. Basefs adds stacking functionality missing from the underlying systems and relieves fistgen from dealing with many platform-dependent aspects of file systems. Basefs does not require changes to the kernel or existing file systems. Instead, Basefs calls kernel functions directly as needed. Basefs's main function is to handle many kernel details relating to stacking. Basefs provides simple hooks for fistgen to insert, remove, or replace code that performs common tasks desired by file-system developers, such as modifying file data or inspecting file names. That way, fistgen can produce file-system code for any platform that Basefs has been ported to. The hooks also allow fistgen to include only the necessary code, improving performance and reducing kernel memory usage.

We built several file systems using FiST. Our experiences with these file systems shows the following benefits of FiST compared with other stackable file systems:

- Average code size is reduced ten times as compared to writing code given another null-layer stackable file system.
- Average development time is reduced seven times compared to writing using another null-layer stackable file system.
- When a stackable file system is mounted on top of any other file system, the stackable file system adds a performance overhead of only 1–2% for accessing the other file system.

Unlike other stacking systems, however, there is no performance overhead for native file systems.

## 1.2 Contributions

The primary conceptual contribution of this work is in the creation of a new programming model for file-system development called FiST: a domain-specific language for stackable file systems. This is the first time a high-level language has been used to describe stackable file systems. From a single FiST description we generate code for different platforms. We achieved this portability since FiST uses an API that combines common features from several vnode interfaces. FiST saves its developers from dealing with many kernel internals, and lets developers concentrate on the core issues of the file system they are developing. FiST reduces the learning curve for non-experts involved in writing file systems—leading to the simplification of the file system development process.

Our technical contributions include the following:

1. **File-System Language:** A high-level file-system language that abstracts file-system interfaces across several platforms, offering a common file system API. The language includes syntax for one file system to call another, for manipulating file data, names, and attributes transparently, for creating and storing new file attributes, and more [86].
2. **Portable Stacking:** Stackable templates that handle numerous kernel details on several systems without kernel modifications. The templates deal with many of the portability issues by exporting an intermediary API to the FiST language compiler (fistgen) such that it does not have to worry about kernel details. This is a working system that we implemented on three platforms: Linux, Solaris, and FreeBSD [84].

[1] We are planning to port FiST to other systems that do not support Unix-like vnodes, especially Windows NT. See Section 10.1.

3. **File-System–Programming Model:** A new programming model for file systems. In this model, developers manipulate one or more file-system operations. For each operation, programmers can control the code that runs before the main part of the operation executes, the actual main part, and the code that runs after the main part of the operation executes.

4. **Transparent Stacking APIs:** Our templates implement a fully working null layer stackable file system, all without changing current file systems, and while minimizing kernel changes. We were able to achieve this by making the templates act as both a stacked-on (lower) file system and one that stacks on others (upper). We accomplished this by making the templates call other kernel code as needed, and ensuring that the templates simulate the rest of the kernel's behavior accurately. That way, our stackable file systems behave transparently to kernels and other file systems. Not changing existing kernels and existing file systems meant that their performance was unchanged: if our stacking is not in use, these systems behave and perform identically as before. In other words, our stackable interfaces do not have to be tightly integrated with the rest of the operating system.

5. **Size-Changing Algorithms:** Our system can generate stackable file systems that change data sizes, such as for compression. We have designed an efficient new algorithm to support size-changing file systems as stackable file systems. No other extensible file system in the past was able to demonstrate a working implementation of such size-changing stackable file systems [80].

6. **File-System–Development Libraries:** Our system comes with high-level support for common file-system operations desired by developers but not available in kernels. This is a library of functions for use in development of typical new file systems. Our library of functions include compression and encryption of files, copying files, reading and writing files as easily as done in user-level C programs, extending the attributes of files and directories, storing additional attributes persistently, extending the functionality of system calls by creating new ioctl(2) calls that can exchange arbitrary information with user-level programs, and more.

7. **New File Systems:** We developed and released several file systems using FiST. Along with the templates, these file systems serve as excellent educational tools to teach others how to write file systems, and to help understand the intricacies of the details of those systems [81, 82, 83].

## 1.3 Thesis Organization

The rest of this dissertation is organized as follows. Chapter 2 provides background information and surveys related work. Chapter 3 describes the design of the FiST system. We detail the design of the FiST language in Chapter 4 and the details of the templates in Chapter 5. We discuss our stacking support for size-changing stackable file system in Chapter 6. Important implementation details are described in Chapter 7. In Chapter 8 we describe the design and implementation of several file systems built using FiST. We evaluate the FiST system in Chapter 9 and conclude with a summary in Chapter 10. We follow with several appendices. Appendix A describes the FiST language specification. Appendix B lists the complete code for some of the more involved file systems we described in Chapter 8. Appendix C provides a tutorial on vnode interfaces.

# Chapter 2

# Background

There are many operating systems, and many new file systems have been proposed, but only a handful of file systems are in regular use. We begin this chapter by describing how file-system development is categorized—all the forms which one can view file-system development. We continue the chapter with a history of the evolution of file systems in general and the vnode interface in particular, and attempt to explain why so few file systems are used in practice. To a large degree, the reasons overlap with the limitations that FiST is intended to remove. We conclude this chapter with a survey of work related to development of file systems in environments that facilitate extensibility.

## 2.1 Evolution of File Systems Development

In this section we trace the historical development of file systems. We start with the pre-stacking days, go through the first stacking efforts, and move on to more general-purpose extensible file systems.

First, however, we define two basic terms that we will use throughout this dissertation:

**File**: A file is a storage data object along with its attributes. For example, the list of user names and their passwords is the data of an object. One attribute of such an object can be its owner: `root`; another attribute can be its size in bytes.

**File System**: A file system is a collection of file objects with the operations that can be performed on these files. For example, a file system knows how to arrange a collection of files on a media such as a hard disk or a floppy. The file system also knows how to apply file operations to those objects, such as reading a file, listing the names of files, deleting a file, etc.

### 2.1.1 Native File Systems

Native file systems are part of the operating system and call device drivers directly. These file systems are usually aware of and often optimized for specific device characteristics, as shown in Figure 2.1.

Examples of such file systems include:

- The Berkeley Fast File System (FFS) [42] for physical disks.
- Sun Microsystems's UFS [7], an optimized version of FFS.
- The LFS *log-structured* file system, optimized for sequential writes [62] on hard disks.
- NFS [51, 65], a file system that uses the network.

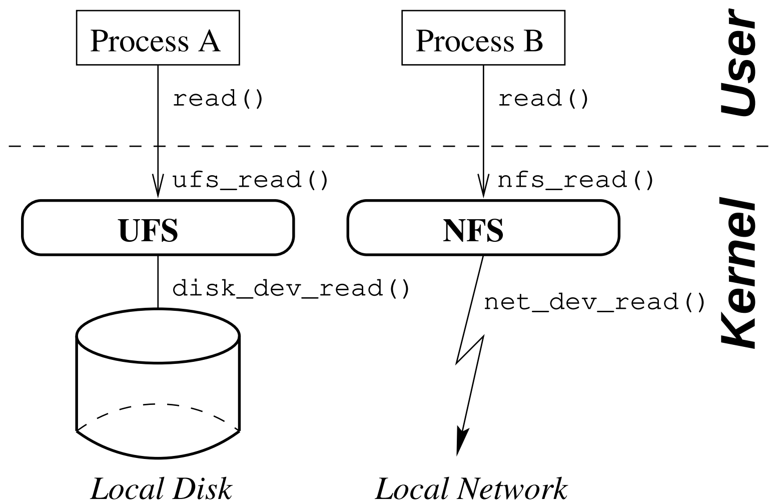



Figure 2.1: Native file systems reside in the kernel and interact with device drivers directly. They perform well, but are difficult to develop and port.

- The High-Sierra file system (HSFS, ISO9660) for CD-ROMs [34].
- Memory-based file systems such as Rio [16].
- The FAT-based file system originally developed for DOS [75], and later adapted for Unix machines to access a floppy as a native PC-based file system (PCFS) [22].

Such file systems are difficult to develop and port because they are coupled to the surrounding operating system: system- call handlers call the file system code and the file-system code calls device drivers. Native file systems perform very well because there is very little code that runs between them and the devices that store the files, and these file systems are able to employ device-specific optimizations.

These file systems have been heavily optimized by vendors for particular combinations of hard disks and Unix workloads. Many vendors share the implementation basis for the same file systems. As a result, we find only a handful in use. For example, while many Unix vendors have their own version of a disk-based local file system, these are in most cases only small variations of the Berkeley FFS.

### 2.1.2 User-Level File Systems

Since developing file systems in the kernel is difficult, it was suggested that file systems should be developed in user level [13]. These file systems reside outside the kernel. They are implemented either as a process or as a run-time library. Most such file systems are accessed via the NFS protocol. That is, the process that implements them registers with the kernel as an NFS server, although the files it manages are not necessarily remote.

The primary benefits of user-level file systems are easier development, easier debugging, and greater portability. This is because they are written using the usual user-level programming languages and development tools. However, user-level file systems suffer from inherently poor performance. Figure 2.2 shows how many steps it takes the system to satisfy an access request through a user-level file server. Each crossing of the dashed line requires a context switch and, sometimes, a data copy. Context switch and data copies can degrade performance by as much as an order of magnitude [83].

Examples of out-of-kernel file systems are the Amd [53, 73] and `automountd` [13] automounters, Blaze's CFS encrypting file system [8], and Amd derivatives including Hlfsd [85], AutoCacher [43], and Restore-o-Mounter [46].

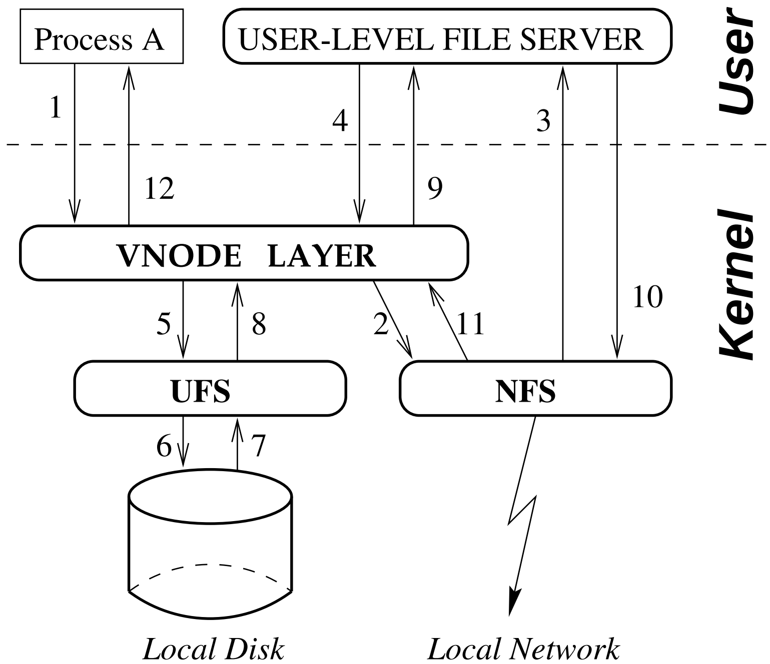



Figure 2.2: User-level file systems are easier to develop and port, but suffer from poor performance due to the greater number of context switches and data copies that occur when serving user requests.

A few file systems at the user level have been implemented as user-level libraries. One such example is Systas [39], a file system for Linux that adds an extra measure of flexibility by allowing users to write Scheme code to implement the file-system semantics. Another, also for Linux, is Userfs [21]. For example, to write a new file system using Userfs, the implementor fills in a set of C++ stub file-system calls—the file system's versions of `open`, `close`, `lookup`, `read`, `write`, `unlink`, etc. Developers have all the flexibility of user-level C++ programs. Then, they compile their code and link it with the provided Userfs run-time library. The library provides the file-system driver engine and the necessary linkage to special kernel hooks. The result is a process that implements the file system. When the process is run, the kernel diverts file-system calls to that custom-linked user-level process that linked with the library.

Such flexibility is very appealing. Unfortunately, the two examples just mentioned are limited to Linux and cannot be easily ported to other operating systems because they require special kernel support that is available only for Linux; that kernel support cannot be easily ported to other operating systems because it depends on specifics of the Linux kernel. Also, such file-system development still requires the user to write a full implementation of each file-system call.

### 2.1.3 The Vnode Interface

As more and more native file systems were being developed, it became apparent that different file systems duplicated large portions of their code—code that assumed that the file system was the only one running. The vnode interface was invented over a decade ago to facilitate the implementation of multiple file systems in one operating system [38], and it has been very successful at that. It is now universally present in Unix operating systems. Readers not familiar with the vnode interface may refer to Appendix C for a tutorial on the subject.

To access a particular file system, processes make system calls that get translated into vnode interface calls, as depicted in Figure 2.3.

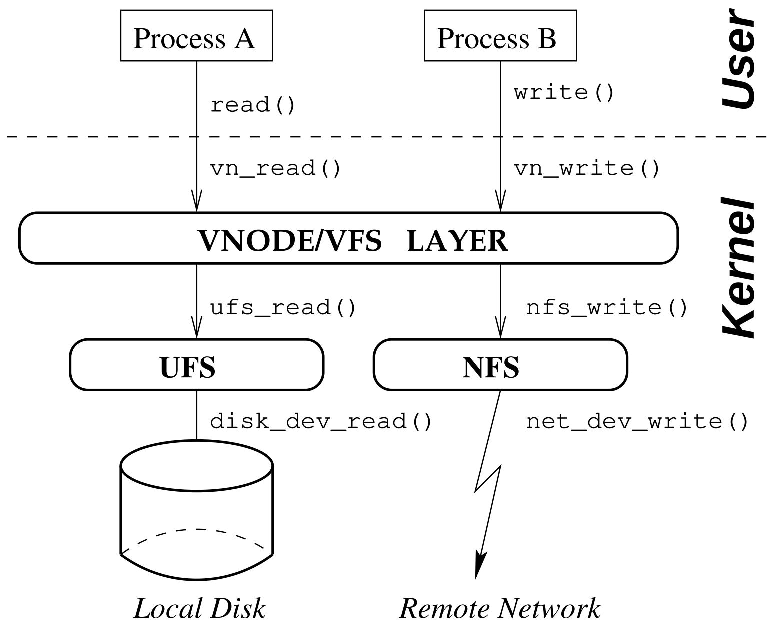



Figure 2.3: In Vnode-based file systems, a system call is translated first to a generic Virtual File System (VFS) call, and the VFS in turn makes the call to the specific file system.

There is a lot of code in the kernel that deals with file systems. Some of this code is specific to the file system and relates to its storage. For example, a disk-based file system contains code to access disk device drivers and the block I/O subsystem, while a network file system contains code to exchange data over a network and ensure its integrity on both the sender's and receiver's sides.

There is a generic section of file-system code in the (Unix) kernel, called the *Virtual File System* (VFS) [1]. The VFS is also often called the *upper-level* file-system code because it is a layer of abstraction above the file-system–specific code. In particular, when system calls begin executing in the kernel's context, the kernel then executes VFS code for those system calls. The VFS then decides which file system to pass the operation onto. The VFS is generic in that it does not contain code specific to any one file system; instead, it calls the predefined file-system functions that were given to it by specific (lower level) file systems.

A *Virtual Node* (Vnode) is a handle to a file maintained by a running kernel. This handle is a data structure that contains useful information associated with the file object, such as the file's owner, size, last modification date, and more. The vnode object also contains a list of functions that can be applied to the file object itself. These functions form a vector of operations that are defined by the file system to which the file belongs.

Vnodes are the primary objects manipulated by the VFS. The VFS creates and destroys vnodes. It fills them with pertinent information, some of which is gathered from specific file systems by handing the vnode object to a lower level file system. The VFS treats vnodes generically without knowing exactly which file system they belong to.

The *Vnode Interface* is an API that defines all of the possible operations that a file system implements. This interface is often internal to the kernel, and resides in between the VFS and lower-level file systems. Since the VFS implements generic functionality, it does not know of the specifics of any one file system. Therefore, new file systems must adhere to the conventions set by the VFS; these conventions specify the names, prototypes, return values, and expected behavior from all functions that a file system can implement.

[1] While vnodes are a Unix concept, Windows NT has a similar concept. See Section 2.1.8.

The designers of the original vnode interface envisioned *pluggable* file-system modules [60], but this capability was not present at the beginning. Through the 1980s Sun made at least three revisions of the interface designed to enhance pluggability [64]. However, during the same period Sun lost control of the vnode definition as other operating-system vendors made slight, incompatible, changes to their vnode interfaces.

Vnode-based file systems are hard to write, port, and maintain. However, they perform well because they reside in the kernel. Such file systems are often written from scratch because they interact with many operating-system specifics.

### 2.1.4 A Stackable Vnode Interface

A Vnode usually contains opaque information that tells the VFS how to perform operations on that file. These operations often go straight into device-driver code or networking code that is used to store the files of the file system.

One notable improvement to the vnode concept is *vnode stacking* [31, 63, 69], a technique for modularizing file-system functions. Stacking is the idea that a vnode object that normally relates—or points—to low-level file system code, may in fact point to another vnode, perhaps even more than one vnode. This idea allows one vnode interface to call another. However, to support stacking, all vendors had to change their original vnode interface significantly. This work often involved major changes to the rest of the operating system to support stacking, and included rewriting existing file systems to a newer stackable interface.

Before stacking existed, there was only a single vnode interface. Higher-level operating-systems code called the vnode interface which in turn called code for a specific file system. With vnode stacking, several vnode interfaces may exist and they may call each other in sequence: the code for a certain operation at stack-level $N$ calls the corresponding operation at level $N + 1$, and so on. For each level, or file system, the operation and data may be changed and passed to the next level down.

A *stackable file system* is one that stacks its vnodes on top of another file system. Such a file system's default behavior is to take a vnode object it received from its caller, change the object's data and attributes as it sees fit, and then call the file system(s) below it.

A regular VFS defines a file system API defining the operations that it expects file systems to implement, calling conventions, and more. However, regular VFSs are only concerned with the API *below* them, the interface to the file systems they call. The file systems they call are only concerned with the device drivers they must call.

A *Stackable VFS* defines a symmetric file system API: the operations and conventions of the file system's callers and callees are identical. The VFS itself must not assume anything about the hierarchy of file systems it calls. In other words, stackable file systems are said to be transparent above and below them.

As a simple example, consider Figure 2.4. Here, we use two layers in the stack. When users perform file-system operations such as reading or writing files, their data goes first through the top-most layer: compression with Gzipfs. After the data has been compressed, it is moved to the next layer down: encryption via Cryptfs. Finally, Cryptfs moves the data to the next layer down, and the data is stored on a disk-based file system, UFS.

More generally than a single stack, vnodes can be *composed*. That is, vnodes need not form a simple linear order, but can branch. This branching is provided by a single vnode calling, or being called from, multiple vnodes. These configurations are called *fan-out* and *fan-in*, respectively, described in more detail in Section 2.1.4.2. Composition creates an directed acyclic graph (DAG) of file systems. (It is important to avoid cycles in file system composition, so as to prevent deadlocks and infinite loops.)

As another example of the utility of vnode stacking, consider the complex caching file system (Cachefs) shown in Figure 2.5. Here, files are accessed from a compressed (Gzipfs), replicated (Replicfs), file system and cached in an encrypted (Cryptfs), compressed file system. One of the replicas of the source file system is itself encrypted, presumably with a key different from that of the encrypted cache. The cache is stored in a UFS [7] physical file system. Each of the three replicas is stored in a different type of physical file system, UFS, NFS, and PCFS [22].

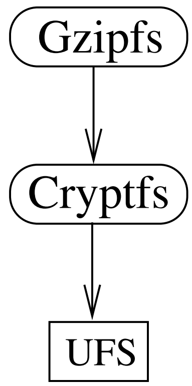



Figure 2.4: A simple file system composed of two stackable layers: first data is compressed via Gzipfs, and then it gets encrypted via Cryptfs, before heading down to stable storage.

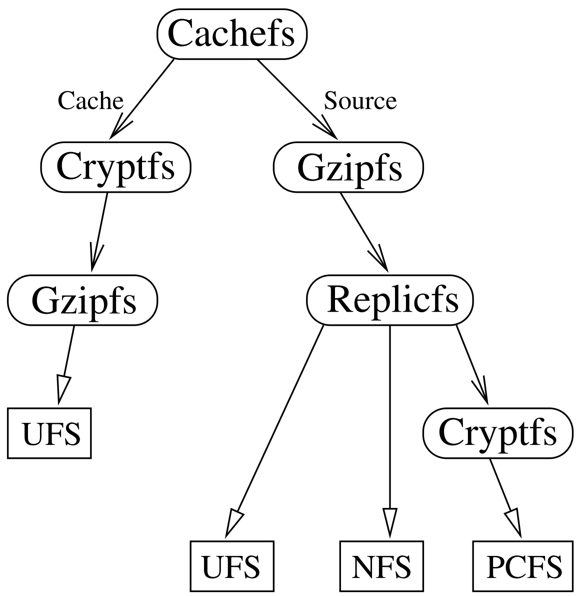



Figure 2.5: A complex file system composed of several stackable modules that can be used repeatedly: caching, encryption, compression, replication, and so on.

One could design a single file system that includes all of this functionality. However, the result would be complex and difficult to debug and maintain. Alternatively, one could decompose such a file system into a set of components:

1. A caching file system that copies from a source file system and caches in a target file system.
2. A cryptographic file system that decrypts as it reads and encrypts as it writes.
3. A compressing file system that decompresses as it reads and compresses as it writes.
4. A replicated file system that provides consistency control among copies spread across three file systems.

These components can be combined in many ways provided that they are written to call and be callable by other, unknown, components. Figure 2.5 shows how the cryptographic file system can stack on top of either a physical file system (PCFS) or a non-physical one (Gzipfs). Vnode stacking facilitates this design concept by providing a convenient inter-component interface.

Building file systems by stacking components carries the expected advantages of greater modularity, easier debugging, and scalability. BSD 4.4's Nullfs is a C template that is useful as a starting point for developing stackable file systems, making it easier to write new file systems. The primary disadvantage of layered file systems is performance. Crossing the vnode interface is overhead, albeit small (1–10%) since this occurs in the kernel [83, 84].

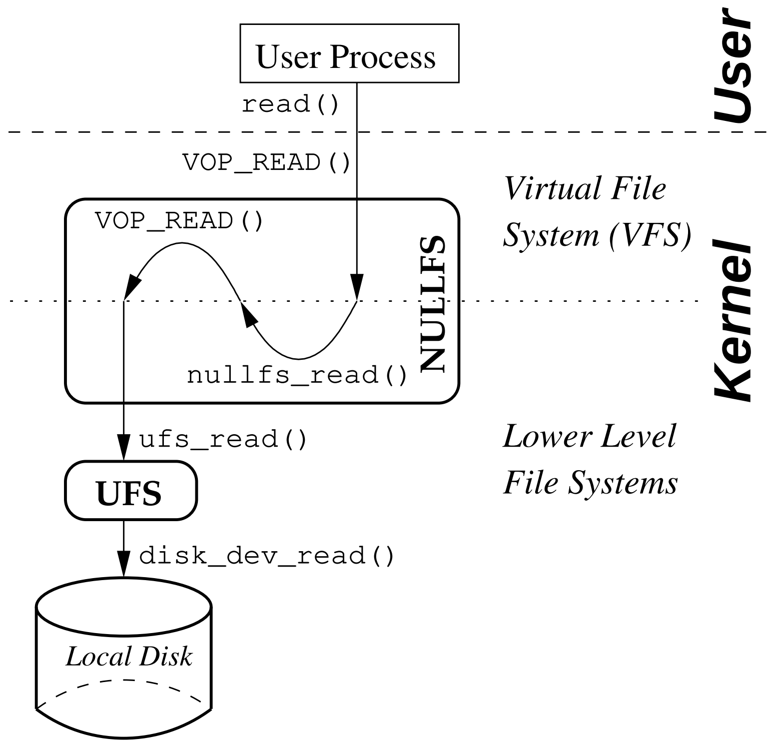



Figure 2.6: A vnode stackable file system showing how system calls are translated into Nullfs calls by the VFS. Nullfs in turn calls the corresponding lower-level file system's operations.

Figure 2.6 shows the structure for a simple, single-level stackable null-layer file system, using BSD 4.4's Nullfs. System calls are translated into VFS calls, which in turn invoke their Nullfs equivalents. Nullfs again invokes generic VFS operations, and the latter call their respective *lower-level* file-system operations. Nullfs calls the lower-level file system without knowing what it is.

#### 2.1.4.1 First Stacking Interfaces

Researchers and developers have always needed an environment where they can quickly prototype and test new file-system ideas. Several earlier works attempted to provide the necessary flexibility. Apollo's I/O system was extensible through user-level libraries that changed the behavior of the application linking with them [58]. Now, modern support for shared libraries [25] permits new functionality to be loaded by the run-time linker. One of the first attempts to extend file system functionality was Bershad's *watchdogs* [6], a mechanism for trapping file-system operations and running user-written code as part of the operation. Later, Webber implemented file system interface extensions that allow user-level file servers [78] to add new functionality. All of these previous attempts at file-system extensibility were ad-hoc—not solutions that were fully integrated into the operating system, and certainly not general purpose solutions.

Vnode stacking was first implemented by Rosenthal in SunOS 4.1 around 1990 [64]. His work was both the first implementation of the pluggability concept and also a clean-up effort in response to changes that had been required to support integration of SunOS and System V and to merge the file system's buffer cache with the virtual memory system. Because it focused on the universally available vnode interface, Rosenthal's stacking model was not ad hoc, unlike earlier efforts, and held promise to become a standard file-system–extension mechanism.

With vnode stacking, a vnode now represented a file open in a particular file system. If $N$ file systems are stacked, a single file is represented by $N$ vnodes, one for each file system. The vnodes are chained together. A vnode interface operation proceeds from the head of the chain to the tail, operating on each vnode, and aborting if an error occurs. This mechanism, which is similar to the way Stream I/O modules [59] operate, is depicted in Figure 2.7.

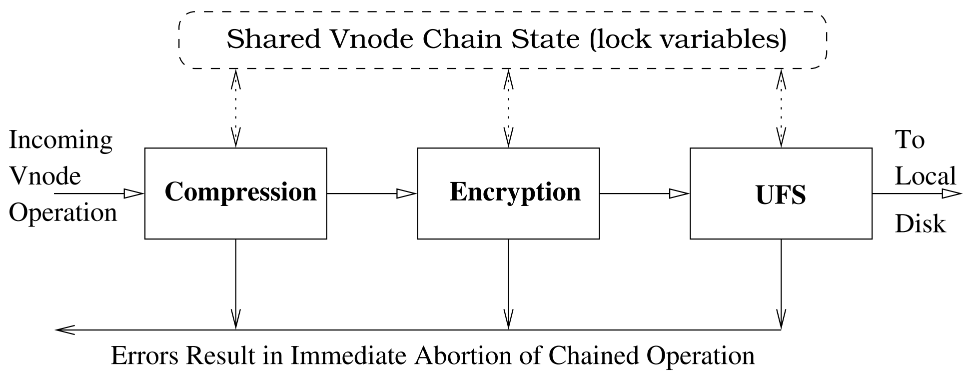



Figure 2.7: Typical Propagation of a vnode operation in a chained architecture. Vnodes in the same chain share a single lock. Vnode operations proceed from the head of the chain until they reach its tail or an error occurs.

This simple interface alone was capable of combining several instances of existing UFS or NFS file systems to provide replication, caching, and fall-back file systems, among other services. Rosenthal built a prototype of his proposed interface in the SunOS 4.1 kernel, but was not satisfied with his design and implementation for several reasons: locking techniques were inadequate, the VFS interface had not been redesigned to fit the new model, and multi-threading issues were not considered. In addition, Rosenthal wanted to implement more file-system modules so as to get more experience with the interface. Rosenthal's interface was never made public nor incorporated into Sun's operating systems.

A few similar works followed Rosenthal. Skinner and Wong developed further prototypes for extended file systems in SunOS [69], which also were never made public or incorporated in Sun's operating system. Around the same time, Guy and Heidemann developed slightly more generalized stacking in the Ficus layered file system [28, 30] at UCLA. Heidemann's work was similar to Rosenthal's. Heidemann's work, however, was made public and eventually made it into BSD 4.4.

#### 2.1.4.2 Fanning in Stackable File Systems

Traditional stackable file systems create a single linear stack of mounts, each one hiding the one file system below it. More general stacking allows for a DAG-like mount structure, as well as for direct access to any layer [31, 63]. This interesting aspect of stackable file systems is called *fanning*, as shown in Figure 2.8. Fanning offer features desired by file-system developers that are only feasible with stackable file systems. A *fan-out* allows the mounted file system to access two or more mounts below. A fan-out is useful for example in replicated, load-balancing, unifying [52], or caching file systems [69]. As discussed in Section 4.2, since FiST uses stacking file systems, it benefits from these added features.

Note that stackable file systems can only call the file systems mounted immediately below it. They cannot directly call file systems further down below. That is, a file system at level $N$ can call the file systems at level $N-1$, but not the file systems at level $N-2$. Furthermore, stackable file systems cannot call file systems above it; in fact, they do not know which or how many file systems are stacked above it. These restrictions are necessary to maintain modular independence and transparency among layers.

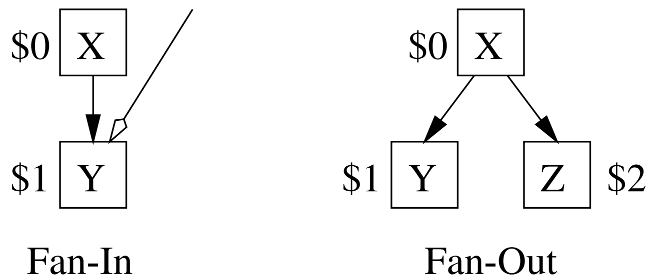



Figure 2.8: Fanning in stackable file systems. A fan-in file system allows direct access to upper and lower layers. A fan-out file system can access two or more file systems directly. The “$” references are explained in Chapter 4.

A *fan-in* allows a process to access lower-level mounts directly, as can also be seen in Figure 2.9. This can be useful when fast access to the lower-level data is needed. For example, in an encryption file system, a backup utility can backup the data faster (and more securely) by accessing the ciphertext files in the lower-level file system. If fan-in is not used, the mounted file system will overlay the mounted directory with the mount point. An overlay mount hides the lower level file system. This can be useful for some security applications. For example, our ACL file system (Section 8.2) hides certain important files from normal view and is able to control who can manipulate those files and how.

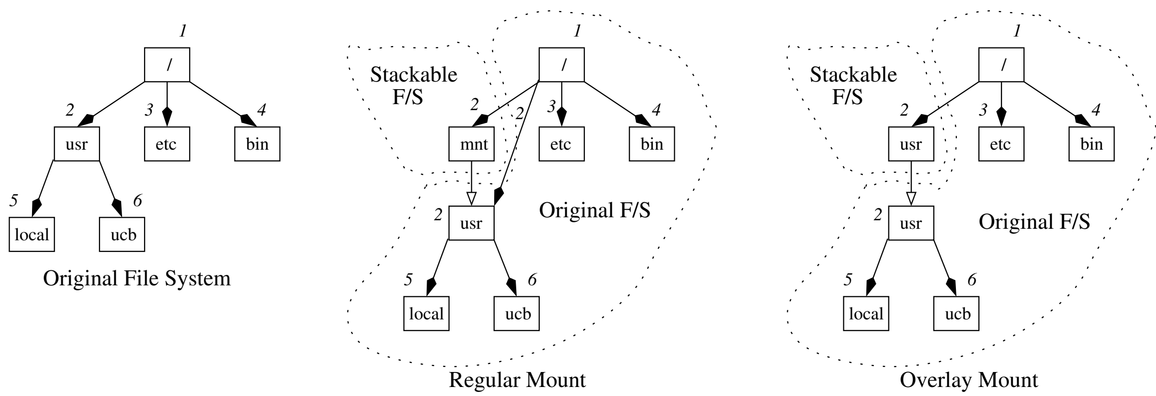



Figure 2.9: Stackable file-system mounts can keep the mounted directory exposed (fan-in allowed) or overlay the mounted directory (fan-in disallowed).

#### 2.1.4.3 Interposition and Composition

A few years later, Rosenthal, Skinner, and Wong established new terminology for the field [63, 69]. They expanded the term *stacking* in to *interposition* and *composition.* The term “stacking” was considered at once to have too many implications, to be too vague, and to imply only a linear LIFO structure with no fan-in or fan-out.

Interposition was coined as the new term for stacking. The defining papers [63, 69] explain a particular implementation of interposition based on a new definition of vnodes. The new vnode contains only the public fields of the old vnode. A new data structure called a *pvnode* contains the private fields of the old vnode. A *vnode chain* now becomes a single vnode (providing a unique identity for the file) plus a *chain*[2] of linked pvnodes. Interposed functionality is represented by one pvnode per open file.

Pvnodes may contain pointers to other vnodes, with the effect that all the linked vnodes may need to be regarded as a single object. This effect is called composition. Composition, in particular, requires the following two capabilities [63]:

1. The ability to lock a complete interposition chain with one operation.
2. Treating an interposition chain as an atomic unit. An operation that failed midway should result in undoing anything that was done when the operation began at the head of the chain.

Figure 2.10 shows this structure for a compressing, encrypting file system that uses UFS as its persistent storage. For each of the three file-system layers in the stack, there is one pvnode. Each pvnode contains a pointer back to the file system that it represents, so that the correct operations vector is used. The three pvnodes are linked together in the order of the stack from the top to the bottom. The head of the stack is referenced from a single vnode structure. The purpose of this restructuring that Skinner & Wong had proposed was so that the three pvnodes could be used as one composed entity (shown here as a dashed enclosing box) that could be locked using a single lock variable in the new vnode structure.

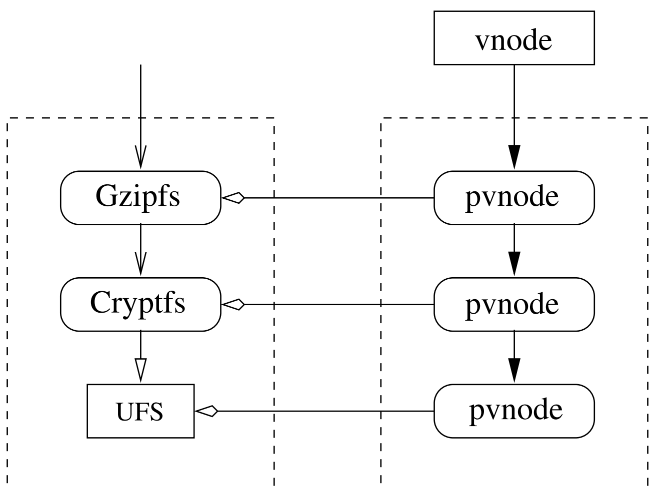



Figure 2.10: Vnode composition using private vnodes, pvnodes. Pvnodes can refer to any file system, while a single vnode handles a chain of pvnodes, allowing a single lock to operate on a complete chain of stacked file systems.

The linked data structures created by interposition and the corresponding complex semantics arising from composition complicate concurrency control and failure recovery. One concurrency control problem is how to lock an arbitrarily long interposition chain as cheaply as possible. Another, harder, problem is how to lock more than one chain for multi-vnode operations.

[2] Actually a DAG, to provide fan-in and fan-out.

The failure recovery problem arises from composition. If a multi-vnode operation fails midway, it is vital to rollback the operations that have succeeded. Both Rosenthal and Skinner discuss adapting the database concept of atomic transactions. Specifically, each pvnode would contain routines to *abort*, *commit*, and *prepare* [3] the effects of operations on it. However, probably because of the complexity involved, no one has yet implemented transactions in support of composition. Consequently, stacks of interposed file systems may have failure behavior that is different from single file systems.

#### 2.1.4.4 4.4 BSD's Nullfs

4.4 BSD includes a file system called *Nullfs*, which was directly derived from Heidemann's work at UCLA [30]. BSD's Nullfs does not create any new infrastructure for stacking; all it does is allow mounting one part of the file system in a different location. It proved useful as a template from which 4.4 BSD's Union file system was written [52]. The latter was developed by extending Nullfs to merge the mount-point file system and the mounted one, rather than blindly forward vnode and VFS operations to the new mount point.

The main contribution of 4.4 BSD to stacking is that it used an existing vnode interface in a manner similar to Sun Microsystems's Lofs. In fact, the way to write stackable file systems in 4.4 BSD is to take the template code for their Nullfs, and adapt it to one's needs. This template approach simplifies development somewhat, but still, difficult kernel code must be written in C, and no high-level functionality is available. Furthermore, this code is not portable to other systems. File systems based on Nullfs, however, suffer only a small performance degradation since they execute in the kernel.

#### 2.1.4.5 Programmed Logic Corp.'s StackFS

Programmed Logic Corp. is a company specializing in storage products. Among their offerings are a compression file system, a 64-bit file system, a high-throughput file system utilizing transactions, and a stackable file system. PLC's StackFS [56] is very similar to BSD 4.4's Nullfs.

StackFS allows for different modules to be inserted in a variety of ways to provide new functionality. Modules offering 64-bit access, mirroring, union, hierarchical storage management (HSM), FTP, caching, and others are available. Several modules can be loaded in a stack fashion into StackFS. The only organization available is a single stack; that is, each file system performs its task and then passes on the vnode operation to the one it is stacked on top of, until the lowest stacked file system accesses the native file system (UFS or NFS).

There is no support for fan-in or fan-out. StackFS does not have facilities for high-level functionality. There is no language available for producing modules that will work within StackFS. PLC's work is not portable to other systems. Still, PLC's products are the only known commercially available stackable file systems written on top of an existing stacking infrastructure.

### 2.1.5 HURD

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 that collectively provide a Unix-like environment [12]. HURD file systems are implemented at user level, much the same as in Mach [2] and CHORUS [1].

The novel concept introduced by HURD is that of the translator. A translator is a program that can be attached to a pathname and perform specialized services when that pathname is accessed.

For example, in HURD there is no need for the `ftp` program. Instead, a translator for ftp service is attached to a pathname, for example, `/ftp`. To access, say, the latest sources for HURD itself, one could `cd` to the directory: `/ftp/prep.ai.mit.edu/pub/gnu` and copy the file `hurd-0.1.tar.gz`. Common Unix commands such as `ls`,

[3] In transaction terminology, "prepare" means to stop processing and prepare to either commit or abort.

cp, and rm work normally when applied to remote ftp-accessed files. The ftp translator takes care of logging into the remote server, translating FTP protocol commands to file system commands, and returning result codes back to the user.

Originally, a translator-like idea was used by the "Alex" work and allowed for example transparent ftp access via a file-system interface [14].

#### 2.1.5.1 How to Write a Translator

HURD defines a common interface for translators. The operations in this interface are much closer to the user's view of a file than the kernel's, in many cases resembling Unix commands:

- `file_chown` to change owner and or group.
- `file_chflags` to change file flags.
- `file_utimes` to change access and modify times.
- `file_lock` to apply or manipulate advisory locks.
- `dir_lookup` to translate a pathname.
- `dir_mkdir` to create a new directory.

HURD also includes a few operations not available in the vnode interface, but which have often been wished for:

- `file_notice_changes` to send notification when a file changes.
- `dir_notice_changes` to send notification when a directory changes.
- `file_getlinknode` to get the other names of a hard-linked file.
- `dir_mkfile` to create a new file without linking it into the file system. This is useful for temporary files, for preventing premature access to partially written files, and also for security reasons.
- `file_set_translator` to attach a translator to a point in the name space.

We have listed only some of HURD's file and directory operations, but even an exhaustive list is not as long as the VFS and vnode interfaces listed in Appendix sections C.2 and C.4.

HURD comes with library implementations for disk-based and network-based translators. Users wishing to write new translators can link with `libdiskfs.a` or `libnetfs.a` respectively. If different semantics are desired, only those necessary functions must be modified and relinked. HURD also comes with `libtrivfs.a`, a trivial template library for file system translators, useful when one needs to write a complete translator from scratch.

The idea of translators is akin to stacking a special file system on top of a single directory: the new file system can perform similar functions to that of a translator. However, the API for writing translators was designed especially for extensibility, while the vnode interface was not. Therefore, writing translators is simpler than writing new file systems using the vnode interface. While writing translators in HURD is relatively simple, developers must still write lots of C code, implementing every function in the API. Furthermore, this code is not portable to other systems.

HURD is unlikely ever to include a "standard" vnode interface. For political and copyright reasons, HURD was designed and built using free software and standards, with the emphasis on changing anything that could be improved. This undoubtedly will limit its popularity. That, coupled with the very different programming interface it offers, means that there is less of a chance that HURD will provide vnode-like code translation like FiST. Nevertheless, HURD offers an interface that is comparable to the vnode one and more.

One final problem with HURD is that it was based on Mach, a message-passing micro-kernel operating system that suffers from poor performance due to the high costs of passing messages [2].

### 2.1.6 Plan 9

Plan 9 was developed at Bell Labs in the late 1980's [54, 55, 57]. The Plan 9 approach to file-system extension is similar to that of Unix.

The Plan 9 mount system call provides a file descriptor that can be a user process or remote file server. After a successful mount, operations below the mount point are sent to the file server. Plan 9's equivalent of the vnode interface (called 9P) comprises the following operations:

**nop** The NULL ("ping") call. It could be used to synchronize a file descriptor between two entities.

**session** Initialize a connection between a client and a server. This is similar to the VFS mount operation.

**attach** Connect a user to a file server. Returns a new file descriptor for the root of the file system. Similar to the "get root" vnode operation.

**auth** Authenticate a 9P connection.

**clone** Duplicate an existing file descriptor between a user and a file server so that a new copy could be operated upon separately to provide user-specific name space.

**walk** Traverse a file server (similar to lookup).

**clwalk** Perform a clone operation followed by a walk operation. This one is an optimization of this common sequence of operations, for use with low-speed network connections.

**create** Create a new file.

**open** Prepare a file descriptor before read or write operations.

**read** Read from a file descriptor.

**write** Write to a file represented by a file descriptor.

**clunk** Close a file descriptor (without affecting the file).

**remove** Delete an existing file.

**stat** Read the attributes of a file

**wstat** Write attributes to a file.

**flush** Abort a message and discard all remaining replies to it from a server.

**error** Return an error code.

These operation messages are sent to a file server by the Plan 9 kernel in response to client requests, much the same way as user-level NFS servers behave.

Plan 9 and 9P provide little benefit over what can be done with the vnode interface and a user-level NFS server. Certainly, there is no major novelty in Plan 9 like the translation concept of HURD. Support for writing Plan 9 file servers is limited, and the functionality they can provide is not as well thought out as HURD's. HURD therefore provides a more flexible file-service–extension mechanism.

Plan 9 does not provide a library of high-level functionality and require writing code using a slightly-modified C compiler. File systems written in Plan 9 code are not portable to other systems.

#### 2.1.6.1 Inferno

Inferno is Lucent Technologies's (a.k.a. Bell Labs) successor to Plan 9. The Inferno network operating system was designed to be fully functional yet fit in a small amount of memory. It is designed to run on devices such as set-top boxes, PDAs, and other embedded systems [40].

In Inferno, everything is represented by files. Therefore, file systems are indistinguishable from other services; they are all part of the Inferno name space. Even devices appear as small directories with a few files named "data," "ctl," "status," etc. To control an entity represented by such a directory, you write strings into the "ctl" file; to get status, read the "status" file; and to write data, open the "data" file and write to it. This model is simple and powerful: operations can be done using simple open, read, write, and close sequences—all without the need for different APIs for networking, file systems, or other daemons [9].

Inferno allows name spaces to be customized by a client, server, or any application. The *mount* operation imports a remote name space onto a local point, similar to the Unix file-system mount operation. The *bind* operation makes a name space in one directory appear in another. This is similar to creating symbolic links and hard links in traditional Unix file systems, with the exception that Inferno can also unify the contents of two directories.

For Inferno to offer a new file-system functionality that might otherwise be achieved via vnode stacking, an application has to *mount* and *bind* the right name spaces, add its own names as required, and then offer them for importation (similar to exporting in Unix). All these actions can be done securely. Inferno code is written in the Limbo programming language [35], a new general-purpose language. Inferno code is not portable to other systems.

Inferno's main disadvantage is a familiar one. It is a brand new operating system, and employs a new programming language and model. Inferno is not likely to be as portable or in wide use as Unix.

### 2.1.7 Spring

Spring is an object-oriented research operating system built by Sun Microsystems Laboratories [44]. It was designed as a set of cooperating servers on top of a microkernel. Spring uses a modified Interface Definition Language (IDL) [74, 77] as outlined in the CORBA specifications [50] to define the interfaces between the different servers.

Spring includes several *generic* modules that provide services that are useful for file systems:

**Caching** A module that provides attribute caching of objects.

**Coherency** A layer that guarantees object states in different servers are identical. It is implemented at the page level, so that every object inherited from it could be coherent by default.

**I/O** A layer that supports streaming-based operations on objects similarly to the Unix `read` and `write` system calls.

**Memory Mapper** A module that provides page-based caching, sharing, and access (similar to the Unix `mmap` system call).

**Naming** A module that maintains names of objects.

**Security** A module that provides secure access and credentials verification of objects.

Spring file systems inherit from many of the above modules. The naming module provides naming of otherwise anonymous file objects, giving them persistence. The I/O layer is used when the `read` or `write` system calls are invoked. The memory pager is used when a page needs to be shared or when system calls equivalent to `mmap` are invoked. The security layer ensures that only permitted users can access files locally or remotely, and so on.

Spring file servers can reside anywhere—not just on the local machine or remotely, but also in kernel mode or in user-level. File servers can replace, overload, and augment operations they inherit from one or more file servers. This form of object-oriented composition makes file systems simpler to write.

It is easy to extend file systems in Spring. The implementation of the new file system chooses which file-system modules to inherit operations from, then changes only those that need modification. Since each file object is named, Spring stackable file systems can perform operations on a per-file basis; they can, for example, decide to alter the behavior of some files, while letting others pass through unchanged.

Spring is a research operating system used by Sun to develop new technology that could subsequently be incorporated into its commercial operating system products. As such, performance is a major concern in Spring. Performance had always been a problem in microkernel architectures due to the numerous messages that must be sent between the many servers that could be distributed over distinct machines and even wide-area networks. Spring's main solution to this problem is the abundant use of caching. Everything that can be cached is cached: pages, names, data, attributes, credentials, etc.—on both clients and servers.

Without caching, performance degradation for a file system containing a single-stack layer ranged from 23–39% in Spring, and peaked at 69–101% for a two-layer stack (for the `fstat` and `open` operations). With caching it was barely noticeable. However, even with caching extensively employed, basic file-system operations (without stacking) still took on average 2-7 *times* longer than the highly optimized SunOS 4.1.3 [36]. So while it is clear that caching helped to alleviate some overheads, many more remain.

To implement a new stackable file system in Spring, one has to write only those operations that need to be changed. The rest inherit their implementation from other file-system modules. This is a flexible and incremental development process.

The work done in the Spring project provides an interesting and flexible object-oriented programming model for file systems. Spring, however, still uses a different file-system interface and as a research operating system is not likely to become popular any time soon, if ever.

One work that resulted from Spring is the Solaris MC (Multi-Computer) File System [41]. It borrowed the object-oriented interfaces from Spring and integrated them with the existing Solaris vnode interface to provide a distributed file-system infrastructure through a special *Proxy File System* (pxfs). 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.

### 2.1.8 Windows NT

Microsoft's Windows NT 4.0 comes with a flexible driver hierarchy that includes high-level I/O managers, file-system drivers such as NTFS or FAT, and device drivers for disks and networks [49]. Windows NT allows developers to write *filter drivers*—stacked modules that can be inserted anywhere in the drivers' calling hierarchy, intercepting file-system operations. Filter drivers that are inserted above file system drivers can intercept user calls before the file system sees them; filter drivers inserted below file systems can intercept calls after the file system has seen them and before they are sent to device drivers.

The act of inserting a filter driver is called *attaching* and is similar to mounting a file system in Unix. Files in Windows NT have a special kernel handle similar to vnodes in Unix. Also, each filter driver can have a dispatch list of file-system operations, similar to vnode operation vectors in Unix.

Filter drivers can register their requests to intercept certain file system operations such as opening a file, reading or writing files, listing directories, and so on. In other words, they can selectively intercept those file-system operations they wish to see. This is more flexible than general vnode interfaces under Unix: there, all vnode operations must be intercepted and handled.

In addition, filter drivers can define *initialization* and *completion* functions to execute before and after certain file system operations execute. This allows developers the flexibility to execute arbitrary code as pre-conditions and post-conditions to file system operations.

The Windows NT I/O manager treats its various components as clients and servers. File system and I/O operations are usually encapsulated in an *I/O Request Packet* (IRP). A module such as filter driver can receive and send IRPs, modifying them as it sees fit. This ability allows for stacking-like behavior, since a list of filter drivers can register their requests to handle the IRPs for, say, reading files; each filter driver can manipulate the IRP as it sees fit, then pass it on to the next filter driver in the list.

One example of an NT filter driver is its virus signature detector. This filter driver runs above the file system, and thus tests the integrity of data before it is committed to disk through the file system. Another example of a filter driver is third-party Hierarchical Storage Management (HSM) drivers, which run between file-system drivers and disk device drivers. HSM modules determine if the data should be retrieved from a local hard-disk or, for example, a remote network-attached backup tape array.

### 2.1.9 Other Extensible File-System Efforts

The Exokernel is an extensible operating system that comes with XN, a low-level in-kernel stable storage system [33]. XN allows users to describe the on-disk data structures and the methods to implement them (along with file-system libraries called libFSes). The Exokernel requires significant porting work to each new platform, but then it can run many unmodified applications. Currently the Exokernel supports only NetBSD and FreeBSD, therefore it is not available on other systems. While the libFSes provide some high-level functionality, much work still has to be done to write a new file system.

Balzer's Mediating Connectors is system and library call wrappers for Windows NT [5]. They allow users to trap all API calls, not just file-system ones. Mediating Connectors is not easily portable to Unix platforms, and still require that developers write C/C++ code directly.

#### 2.1.9.1 Compression Support

Compression file systems are not a new idea. Windows NT supports compression in NTFS [49]. E2compr is a set of patches to Linux's Ext2 file system that add block-level compression [4]. Compression extensions to log-structured file systems resulted in halving of the storage needed while degrading performance by no more than 60% [11]. The benefit of block-level compression file systems is primarily speed. Their main disadvantage is that they are specific to one operating system and one file system, making them difficult to port to other systems and resulting in code that is hard to maintain. Our approach is more portable because we use existing stacking infrastructure, we do not change file systems or operating systems, and we run our code in the kernel to achieve good performance.

The ATTIC system demonstrated the usefulness of automatic compression of least-recently-used files [15]. It was implemented as a modified user-level NFS server. While it provided portable code, in-kernel file systems typically perform better. In addition, the ATTIC system decompresses whole files while our system decompresses only the needed data.

HURD [12] and Plan 9 [54] have an extensible file system interface and have suggested the idea of stackable compression file systems. Their primary focus was on the basic minimal extensibility infrastructure; they did not produce any working examples of size-changing file systems.

Spring [36, 45] and Ficus [29] discussed a similar idea for implementing a stackable compression file system. Both suggested a unified cache manager that can automatically map compressed and uncompressed pages to each other. Heidemann's Ficus work provided additional details on mapping cached pages of different sizes.[4] Unfortunately, no demonstration of these ideas for compression file systems was available from either of these works. In addition, no consideration was given to arbitrary SCAs and how to efficiently handle common file operations such as appends, looking up file attributes, etc.

[4] Heidemann's earlier work [31] mentioned a "prototype compression layer" built during a class project. In personal communications with the author, we were told that this prototype was implemented as a block-level compression file system, not a stackable one.

### 2.1.10 Domain Specific Languages

High-level languages have seldom been used to generate code for operating system components. FiST is the first major language to describe a large component of the operating system, the file system.

One of the most popular domain-specific languages is YACC (Yet Another Compiler Compiler) [32]. YACC has been used to produce parsers for numerous languages, many of those for specific domains. YACC, however, is most suitable for producing code that is intended to work as part of a user-level application, not inside the kernel. While FiST uses a YACC-like model for its input file, FiST was designed to produce code that runs in kernels.

Previous work in the area of operating-system–component languages include a language to describe video device drivers [76] called GAL (Graphics Adaptor Language). GAL can specify various parameters of the two main parts of a video adaptor: the frame buffer memory and the graphics controller. The controller handles access to the video memory and produces the video signal. In GAL, one can specify allowed ranges for common video adapter parameters: size of video memory, allowed resolutions and pixel depths, horizontal and vertical synchronization clock speeds, control ports, registers, I/O memory ranges, etc.

The GAL code generator produces code that can run as an X11 video server, making all the necessary calls to initialize and reset the video adapter, change modes, display graphics, and support functions that are native to the card itself (such as 2D graphics drawing commands).

While the authors claim that their framework can support different graphical windowing systems (such as Windows 95), this work was only implemented for the X11 server on Intel architectures.

## 2.2 File Systems Development Taxonomy

In this section we summarize the taxonomy of file-system development. We have covered some of these topics earlier in the chapter, while discussing the evolution of file-system development and related works. In the rest of this dissertation we will illustrate how FiST works better than the alternatives listed in each of the categories of this taxonomy.

- **Abstraction Level**: File systems can work in user level or in the kernel. If in user level, file systems usually use the NFS interface, and act as a user-level NFS server; examples include automounters such as Amd [53, 73].

  If the file system works in the kernel, it can use one of two possible interfaces:

  - **Device Level**: These file systems interact directly with device drivers, and are specific to the media they use: hard disks (UFS), CD ROMs (ISO9660), floppies (DOS), and networks (NFS).
  - **Stackable Vnode Level**: These file systems interact with other file systems through a special file-system interface called the vnode interface (described in Section 2.1.3).

- **Programming Model**: Depending on where the file system runs, different programming models may be used:

  - If the file system run in user level, the programming model is that of an NFS server, implementing the operations specified in the NFS protocol.
  - If the file system run in the kernel, it can use one of several models. The basic model is that of implementing every file-system operation defined by the vnode interface, the way most low-level file systems do.
  - Stackable file systems usually have to implement only the vnode operations that they wish to change. Other operations are automatically passed through between stacked layers. This option is similar to that of object-oriented programming models, where a subclass can use the methods of the superclass.

- **High-Level Functionality**: To ease the development of file systems, some systems come with a set of available common functions that can be used as building blocks for the new file system. These functions form a useful collection of additional, high-level operations, much the same way the C run-time library (`libc.a`) provides additional functions built using plain system calls.

- **Development Process**: The development of file systems depends on how much of a foundation is available. This affects the amount of effort involved.

  - Low-level file systems are written from scratch, most commonly in C. These file systems require a lot of effort to write because they interact with specific device drivers and special services in the operating system.
  - Using *templates*, a file-system developer uses a base file system source, copies it, and makes changes to the copy. This is often the development process with stackable file systems such as BSD-4.4's Nullfs [52] and Sun Microsystems's Lofs [71].
  - Object-oriented operating systems such as Spring [44] require the file-system programmer to define and write only code for those operations that need to be changed. Operations that do not change are automatically inherited from the parent objects and classes.

- **Performance**: The performance of the file system is often a factor relating to where it runs.

  - User-level file systems are the slowest because each exchange of information between the kernel and the file-system server causes a context switch.
  - Low-level file systems are the fastest because they run in the kernel and interact directly with device drivers. That is also because very little code—representing overhead—run between these file systems and the media they write to and read from.
  - Stackable file systems are generally fast because they run in the kernel—much faster than user-level file systems. However, they are usually slower than low-level file systems because they interact with other in-kernel file systems through a *stackable vnode interface*, an API that defines how one file system may call another.

- **Portability**: This factor is defined as the ease with which code written for one file system can be ported to another.

  - User-level code is the easiest to port, because it is written just like any other user-level C program. For example, the Amd automounter [53] has been easily ported to dozens of different Unix platforms and operating-system versions.
  - Low-level in-kernel file systems are the hardest to port because they depend on many kernel specifics and details of device-driver implementations. Rarely do these file systems get ported to other platforms; often the effort is large enough that a complete rewrite is easier than porting.
  - Stackable file systems run in the kernel and as such also depend on kernel internals. However, they do not depend on device-driver details, making them a little easier to port. If stackable file systems use a stackable vnode interface, then porting them to other platforms requires those platforms to support the same stacking API.

Table 2.1 shows a comparison of features relating to file-system development for all systems that provide some form of extensibility. The last row of Table 2.1 shows how FiST compares favorably to other systems:

- FiST's abstraction level is that of stacking, an API that many system developers are already familiar with.
- The programming model is incremental, thus requiring the least amount of work to implement new features.

| System | Abstraction Level | Programming Model | High-Level Functionality | Development Process | Performance Overhead | Portable Code |
|---|---|---|---|---|---|---|
| Native file systems | device driver, block device | device-driver operations | no | C from scratch | none | no |
| User-level file systems | NFS interface | NFS operations | no | C from scratch | very high | yes |
| Vnode file systems | Vnode interface | vnode operations | no | C from scratch | very small | no |
| Stackable File Systems | Stackable VFS | Stacking operations | no | template or incremental | small | no |
| HURD | HURD API | HURD operations | some | template | significant | no |
| Plan 9 | 9P API | 9P calls | no | from scratch | (unreported) | no |
| Inferno | files | manipulate control files | no | Limbo compiler | (unreported) | no |
| Spring | object-oriented | object-oriented | no | object-oriented | small to significant | no |
| Nullfs | stacking | stacking operations | no | template, incremental | small | no |
| Exokernel | XN API | XN API | some | development process | (unreported) | significant effort |
| Windows NT | Win32 API | Win32 file-system operations | no | C++ from scratch | (unreported) | no |
| FiST | stacking | incremental | yes | high-level language | small | yes |

Table 2.1: A comparison of the taxonomy for extensible file systems and FiST.

- We provide a high-level library of common functions useful when developing file systems.
- FiST is a high-level language that speeds up development over traditional programming languages such as C or C++.
- File systems generated from FiST incur a very small performance overhead, making them very suitable for practical use.
- Our system is portable to several systems, saving significantly on porting efforts to other platforms.

# Chapter 3

# Design Overview

The FiST system is comprised of three components: the FiST language, the FiST code generator, fistgen, and the basic stacking templates. This chapter introduces the basic design of the FiST system: where the language and the templates reside and how they fit together, what is a typical file-system development process, and what is the programming model for developing file systems using FiST. Following, Chapter 4 describes in more detail the design for the FiST language; since fistgen implements the language code generation, we describe it in detail at the end of the language description, in Section 4.5. Finally, Chapter 5 describes the third and last component, the stacking templates.

## 3.1 Layers of Abstraction

FiST is a high-level language providing a file-system abstraction at a higher level than previous file-system abstractions. Figure 3.1 shows the hierarchy for different file system abstractions:

1. The lowest-level file systems use device-driver APIs and are specific to the operating system and the device.
2. Stackable file systems offer an API that works with any file system on a given operating system, but they are specific to one operating system.
3. FiST file systems are more abstract than the previous two levels. FiST file systems are not specific to either the operating system or the file system.

<table>
<tr><td colspan="2">FiST Language</td></tr>
<tr><td>Basefs templates</td><td>Stackable (VFS) File Systems<br>(lofs, cryptfs, aclfs, unionfs, etc.)</td></tr>
<tr><td colspan="2">Low-Level File Systems (UFS, NFS, etc.)</td></tr>
</table>

Figure 3.1: FiST Structural Diagram. Stackable file systems, including Basefs, are at the VFS level, and are above low-level file systems. FiST descriptions provide a higher abstraction than that provided by the VFS. Basefs templates are the stackable file systems templates used in FiST, which we describe in Chapter 5.

At the lowest level reside file systems native to the operating system, such as disk-based and network-based file systems. They are at the lowest level because they interact directly with device drivers. They therefore depend on many specific details of the device driver and the operating system.

Low-level file systems run in the kernel and are very fast, but are difficult to develop and port to other systems. Since they interact with device drivers, these file systems have full control over the data structures and placement of data on the device.

Above native file systems are stackable file systems such as the examples in Chapters 2 and 8, as well as Basefs (which we describe in Chapter 5). These file systems provide a higher abstraction than native file systems because stackable file systems interact with other file systems only through a well-defined *virtual file system interface* (VFS) [38]. The VFS provides *virtual nodes* (vnodes), an abstraction of files across different file systems. The abstraction at this level is that of the file-system interface: stackable file systems are specific to the operating system, but not to the file system they stack on.

Stackable file systems are easier to develop than native file systems because you write code to a specific well-defined API, the stackable vnode interface. Stackable file systems reside in the kernel, and while they add a certain overhead over native file systems, they still perform very well. However, they are still difficult to port to other platforms because they must be written to specific vnode interfaces that are different from system to system. Furthermore, the stacking abstraction does not permit a stackable file system to access low-level device details such as the structure of inodes and the placement of blocks on a hard disk.

At the highest level, we define the FiST language. FiST abstracts the different vnode interfaces across *different* operating systems into a single common description language, because it is easier to write file systems this way. We found that, while vnode interfaces differ from system to system, they share many similar features. Our experience shows that similar file-system concepts exist in other non-Unix systems, and our stacking work can be generalized to include them. Therefore, we designed the FiST language to be as general as possible: we mirror existing platform-specific vnode interfaces, and extend them through the FiST language in a platform independent way. This allows us to modify vnode operations and the arguments they pass in an arbitrary way, providing great design flexibility.

There are several benefits to the FiST abstraction. You write file systems only once, and run them on many platforms; this improves portability considerably. In addition, this improves development time: not just because you write code once, but you write less code, since it is written in a high-level language. In addition, you write code incrementally—only changing that which needs to change. The FiST system takes care of everything else: proper error and return codes. FiST also provides default values and implementations for all file-system operations so the developer does not have to write them.

FiST file systems build on the ideas of stackable file-system interfaces. Therefore they share some limitations with other stackable file systems: they are not able to access or change low-level file-system data structures such as inodes, nor to control the placement of data blocks on disk devices. However, FiST is able to compensate for some of these limitations by allowing developers to extend the attributes of files; this is described in Section 4.2.

## 3.2 FiST Templates and Code Generator

The FiST system is composed of three parts: the language specification, a code generator, and stackable file-system templates. This separation is similar to that shown in Figure 3.1: the templates implement APIs that are at a lower level than the language itself.

The templates provide a fully functional null-layer stackable file system. They add stacking functionality that may be missing from some systems, provide default implementations for various operating systems, provide suitable error or return codes for all operations, handle operations that are specific to some operating systems, and also provide hooks for the FiST code generator. These hooks allow the code generator to find the right places to generate (insert, delete, or replace) some code.

The overall function of the templates is to handle many low-level system details, freeing the code generator to concentrate on FiST language issues: input file parsing and code generation. That way, all low-level details are left in the templates, and higher-level details are left to the code generator. This separation makes it easy to maintain the FiST system and port its templates to new platforms.

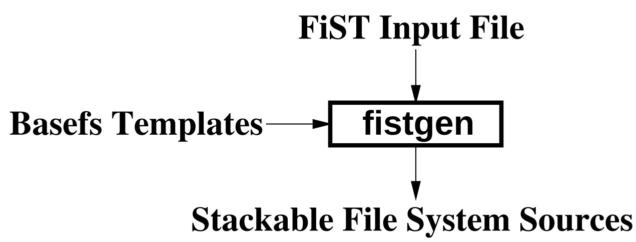



Figure 3.2: FiST Operational Diagram. Fistgen reads a FiST input file, and with the Basefs templates, produces sources for a new file system.

The overall operation of the FiST system is shown in Figure 3.2. The figure illustrates how the three parts of FiST work together: the FiST language, fistgen, and the Basefs templates. File system developers write FiST input files to implement file systems using the FiST language. *Fistgen*, the FiST language code parser and file-system code generator, reads FiST input files that describe the new file system's functionality. Fistgen then uses additional input files, the Basefs templates. These templates contain the stacking support code for each operating system and hooks to insert developer code. Fistgen combines the functionality described in the FiST input file with the Basefs templates, and produces new kernel C sources as output. The latter implement the functionality of the new file system. Developers can, for example, write simple FiST code to manipulate file data and file names. Fistgen, in turn, translates that FiST code into C code and inserts it at the right place in the templates, along with any additional support code that may be required. Developers can also turn on or off certain file-system features, and fistgen will conditionally include code that implements those features.

## 3.3 The Development Process

To illustrate the FiST development process, we contrast it with two traditional file-system–development methods: writing code from scratch and writing code using an existing stackable interface.

We use a simple example similar to Watchdogs [6]. Suppose a file-system developer wants to write a file system that will warn of any possible unauthorized access to users' files. The main idea is that only the files' owners or the root user are allowed access to those files. Any other user who might be attempting to find files that belong to another user, would normally get a "permission denied" error code. However, the system does not produce an alert when such an attempt is made. This new *snooping file system* (Snoopfs) will log these failed attempts. Logged failed attempts can be processed off-line to produce reports of failures or alerts. The one place where such a check should be made is in the lookup routine that is used to find a file in a directory.

### 3.3.1 Developing From Scratch

To implement Snoopfs from scratch, the developer has to do the following:

1. locate an operating system with available sources for any one file system
2. read and understand the code for that file system and any associated kernel code

3. write a new file system that includes the desired functionality, loosely basing the overall implementation on another file system that was already written
4. compile the sources into a new file system, possibly rebuilding a new kernel and rebooting the system
5. mount the new file system, test, and debug as needed

After completing this, the developer is left with one modified file system for one operating system. The amount of code that has to be written is in the range of thousands of lines (Table 1.1). The process has to be repeated for each new port to a new platform. In addition, changes to native file systems are unlikely to be accepted by operating system maintainers; such a file system will have to be maintained independently. Each time a change is made to the file system on which this implementation of Snoopfs was based, those changes will have to be ported to this implementation. This is true whether the new file system is implemented as a loadable kernel module or not.

### 3.3.2 Developing Using Existing Stacking

To implement Snoopfs with existing stacking file systems such as BSD-4.4's Nullfs [52], the developer has to do the following:

1. locate an operating system with available sources for one stackable file system
2. read and understand the code for that stackable file system and any associated kernel code
3. make a copy of the sources for that stackable file system, and carefully modify them to include the new functionality
4. compile the sources into a new file system, possibly rebuilding a new kernel and rebooting the system
5. mount the new file system, test, and debug as needed

After completing the above steps, the developer usually has one loadable kernel module for a new stackable file system. The amount of code that has to be read and understood is in the range of thousands of lines (Table 1.1); the amount of code that has to be written usually ranges in the hundreds of lines (see Section 9.1).

The process of writing a new stackable file system has to be repeated for each new port to a new platform. Traditionally, such stackable file systems have to be maintained independently, as operating-system maintainers resist adding new features into the code of their operating systems.

Nevertheless, it is easier to write new stackable file systems using a stackable template than from scratch.

### 3.3.3 Developing Using FiST

In contrast, the normal procedure for developing code with FiST is

1. write the code in FiST once
2. run fistgen on the input file
3. compile the produced sources into a loadable kernel module, and load it into a running system
4. mount the new file system, test, and debug as needed

Debugging code can be turned on in FiST to assist in the development of the new file system. There is no need to have kernel sources or be familiar with them and there is no need to write or port lots of code for each platform. Furthermore, the same developer can write Snoopfs using a small number of lines of FiST code:

```
%op:lookup:postcall {
if ((fistLastErr() == EPERM ||
     fistLastErr() == ENOENT) &&
    $0.owner != %uid && %uid != 0)
  fistPrintf("snoopfs detected access by uid %d,\
pid %d, to file %s\n", %uid, %pid, $name);
}
```

This short FiST example of code inserts an "if" statement after the normal call to the lookup routine. The code checks if the previous lookup call failed with one of two particular errors (who the owner of the directory is, who the effective running user is) and then decides whether to print the warning message. (This message is printed by the kernel, which in turn can be logged via `syslogd` to any log file.)

This single FiST description is portable, and can be compiled on each platform that we have ported our templates to (currently Linux, Solaris, and FreeBSD). In Chapter 4 we describe the FiST language in much greater detail.

## 3.4 The FiST Programming Model

The most basic operation of the FiST programming model is in the action that stacking operations take. Stacking works by passing a file-system operation to a file system that is stacked below, as seen in Figure 3.3. The operation executes on the lower-level file system, and then passes on to the next one below. When the operation reaches the lowest level, return values begin propagating upwards all the way to the upper-most stacked file system: status conditions, error codes, and returned computed values. In other words, stacking works by passing the same file system operation to the lower-level file system.

FiST is easier to work with than previous stacking systems. In previous systems, developers had to locate by hand all of the places where they wanted to insert their code or modify existing code. FiST, on the other hand, allows you to add or modify code more accurately. FiST file systems can insert *pre-call*, *post-call*, and *call* actions as follows:

**pre-call** Before calling the lower-level file system, you can execute arbitrary code. This can be used, for example, to perform certain security checks (such as with ACLs) before allowing the action to proceed.

**post-call** After returning from the call to lower-level file system, you can also execute arbitrary code. This can be used, for example, to decrypt a file name buffer that was returned from a lower level (encrypted) file system.

**call** You can also replace the actual call to the lower level file system with any other call. You can use this, for example, to produce a versioning file system that would rename a file instead of removing it by replacing the vnode `unlink` operation with a suitable `rename` operation.

This model is powerful and very useful for developers. Together, the above three calling forms allow developers full control over stacked operations. Developers can change any part of an operation, but they do not have to change anything by default. They need only declare their intentions to run code before or after a normal file-system operation executes, thus controlling the operation from before it begins to after it ends. Furthermore, developers can also replace the actual call to the file-system operation with another, or none, if they so choose. This programming model provides maximal programming flexibility: simple file systems are easy to create by changing code in only a small number of places, and at the same time it is possible to construct complex file systems by inserting lots of additional code in as many places as needed. We show how to control these calling conventions in FiST in Section 4.3.

To change one of these parts, you declare it with its associated code in the FiST input file. Fistgen, the FiST code generator, reads the FiST input file and the appropriate templates. It parses the templates, replacing, removing, and adding code as a result of various declarations in the FiST input file.

You can insert as much code as you would like in the *pre-call* and *post-call* parts. The FiST language (Chapter 4) allows several directives to affect the same vnode operation. In that case, additional pre-call code is inserted in front of existing pre-call code, and additional post-call code is inserted after existing post-call code. This is done in a recursive-like manner to provide proper nesting of code context, the same way that enclosing blocks of code in C programs encapsulate smaller blocks of code.

For example, if you design a file system that encrypts data and then compresses it, you want to ensure that the order of pre-call and post-call operations preserves data integrity. When writing data, it should first be encrypted and then compressed. However, when reading data, the order should be reversed: data should first be decompressed and then decrypted. The insertion rules for pre-call and post-call ensure this.

You can replace the *call* part only once, because stackable file systems have only one type of call they make. If additional calls are needed, they can be inserted in the pre-call or post-call parts. In the case of fan-out file systems, the default action for the *call* part is to call the first stacked file system (left branch in Figure 2.8). If developers want to call other branches with the same or other operations, they can insert *post-call* code as shown in the Unionfs example in Section 8.3.

## 3.5 The File-System Model

FiST-produced file systems run in the kernel, as seen in Figure 3.3. They run in the kernel to provide the best performance possible. FiST file systems mirror the vnode interface both above and below. The interface to user processes is the system-call interface. FiST does not change either the system-call interface or the vnode interface. Instead, FiST can change information passed and returned through these two interfaces. The basic motivation is to allow FiST to change everything possible, but without changing operating-system APIs. This provides the highest functionality to file-system developers that can be achieved without requiring custom kernels.

A user process generally accesses a file system by executing a system call, which traps into the kernel. The kernel VFS then translates the system call to a vnode operation, and calls the corresponding file system. If the latter is a FiST-produced file system, it may call another stacked file system below. Once the execution flow has reached the lowest file system, error codes and return values begin flowing upwards, all the way to the user process.

In FiST, we model a file system as a collection of mounts, files, and user processes, all running under one system. Several *mounts*, mounted instances of file systems, can exist at any time. A FiST-produced file system can access and manipulate various mounts and files, data associated with them, and their attributes—as well as access the functions that operate on them. Furthermore, the file system can access attributes that correspond to the run-time execution environment: the operating system and the user process currently executing.

Information (both data and control) generally flows between user processes and the mounted file system through the system-call interface. For example, file data flows between user processes and the kernel via the `read` and `write` system calls. Processes can pass specific file-system data using the `mount` system call. In addition, mounted file systems may return arbitrary (even new) error codes back to user processes.

Since a FiST-produced stackable file system is the caller of other file systems, it has a lot of control over what transpires, between it and the ones below, through the vnode interface. FiST allows access to multiple mounts and files. Each mount or file may have multiple attributes that FiST can access. Also, FiST can determine how to apply vnode functions on each file. For maximum flexibility, FiST allows the developer full control over mounts and files, their data, their attributes, and the functions that operate on them; they may be created or removed, data and attributes can be changed, and functions may be augmented, replaced, reordered, or even ignored.

Ioctls (I/O Controls) have been used as an operating-system extension mechanism as they can exchange arbitrary information between user processes and the kernel, as well as in between file-system layers, without changing interfaces. FiST allows developers to define new ioctls and define the actions to take when they are used; this can be used to create

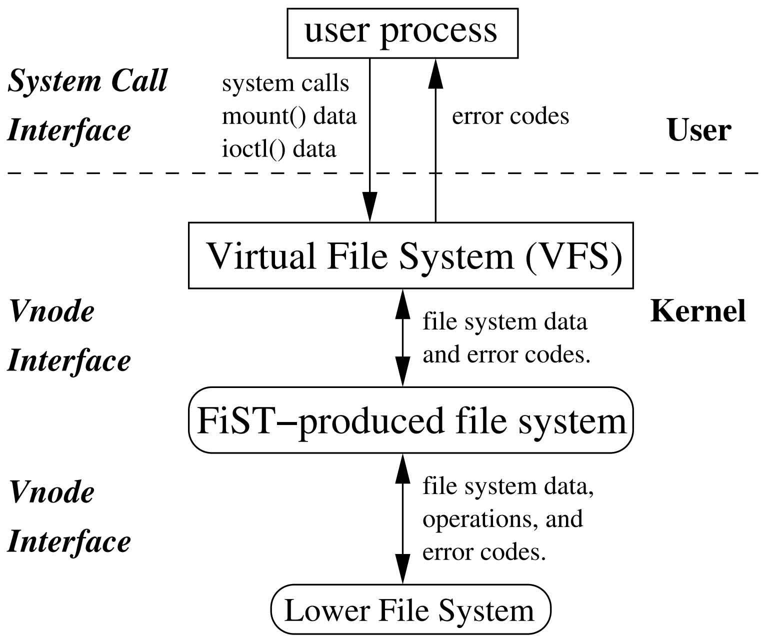



Figure 3.3: Information and execution flow in a stackable system. FiST does not change the system call or vnode interfaces, but allows for arbitrary data and control operations to flow in both directions.

application-specific file systems. FiST also provides functions for portable copying of ioctl data between user and kernel spaces. For example, our encryption file system (Section 8.1) uses an ioctl to set cipher keys.

# Chapter 4

# The FiST Language

The FiST language is the first of the three main components of the FiST system, as listed in Section 3.2. In this chapter we discuss the structure of the FiST input file and the various parts of the FiST syntax: primitives, declarations, and rules. We also discuss the second main component of the FiST system, fistgen—the FiST language code generator, which combines the language with stacking templates to produce a new file system. We discuss this third main component of the FiST system—the stacking templates—in Chapter 5.

## 4.1 Overview of the FiST Input File

The FiST language is a high-level language that uses file-system features common to several operating systems. It provides file-system–specific language constructs for simplifying file-system development. In addition, FiST language constructs can be used in conjunction with additional C code to offer the full flexibility of a system programming language familiar to file-system developers. The ability to integrate C and FiST code is reflected in the general structure of FiST input files. Figure 4.1 shows the four main sections of a FiST input file.

| | |
|---|---|
| 1 | `%{`<br>*C Declarations*<br>`%}` |
| 2 | *FiST Declarations*<br>`%%` |
| 3 | *FiST Rules*<br>`%%` |
| 4 | *Additional C Code* |

Figure 4.1: FiST Grammar Outline. FiST input file's format resembles that of YACC. It has four sections with similar meaning for each corresponding section.

The FiST grammar was modeled after YACC [32] input files, because (1) YACC is familiar to programmers and (2) the purpose and meaning of each of its four sections in YACC (also delimited by "%%") matches the purpose and meaning of the four different subdivisions of desired file system code: raw included header declarations, declarations that globally affect the produced code, actions to perform when matching vnode operations, and additional code.

**C Declarations** (enclosed in "{% %}") are used to include additional C headers, define macros or typedefs, list forward function prototypes, etc. These declarations are used throughout the rest of the code, just as in YACC input files.

**FiST Declarations** define global file-system properties that affect the overall semantics of the produced code and how a mounted file system will behave. These properties are useful because they allow developers to make common global changes in a simple manner. In this section we declare if the file system will be read-only or not, whether or not to include debugging code, if fan-in is allowed or not, and what level (if any) of fan-out is used.

FiST Declarations can also define special data structures used by the rest of the code for this file system. We can define mount-time data that can be passed with the mount(2) system call. A versioning file system, for example, can be passed a number indicating the maximum number of versions to allow per file. FiST can also define new error codes that can be returned to user processes, in order for the latter to understand additional modes of failure. For example, an encryption file system can return a new error code indicating that the cipher key in use has expired. Doing so, however, may require updating of potentially many legacy programs to understand the new error codes; otherwise, such programs could fail unexpectedly.

This section is also similar in meaning to YACC's second section—declarations that globally affect the overall behavior of the code produced.

**FiST Rules** define actions that generally determine the behavior for individual files. A FiST rule is a piece of code that executes for a selected set of vnode operations, for one operation, or even a portion of a vnode operation. Rules allow developers to control the behavior of one or more file-system functions in a portable manner. The FiST rules section is the primary section, where most of the actions for the produced code are written. In this section, for example, we can choose to change the behavior of `unlink` to rename the target file, so it might be restored later. We separate the declarations and rules sections for programming ease: developers know that global declarations go in the former, and actions that affect vnode operations go in the latter.

This section is only loosely related to YACC. In YACC, the rules section performs an arbitrary action when a sequence of tokens matches. In FiST, each rule includes arbitrary code to run when the operation listed in the rule "matches" the current operation that the file system is executing.

**Additional C Code** includes additional C functions that might be referenced by code in the rest of the file system, also as in a YACC input file's last section. We separated this section from the rules section for code modularity: FiST rules are actions to take for a given vnode function, while the additional C code may contain arbitrary code that could be called from anywhere. This section provides a flexible extension mechanism for FiST-based file systems. Code in this section may use any basic FiST primitives (discussed in Section 4.2) which are helpful in writing portable code. We also allow developers to write code that takes advantage of system-specific features; this flexibility, however, may result in non-portable code.

The sections of the FiST input file relate to the programming model as follows. The FiST Declarations section defines data structures used by the FiST Rules section. The latter section is where per-vnode actions are defined: pre-call, call, and post-call as described in Section 3.4. It is in this section that you declare what actions to take for each vnode operation, sets of vnode operations, or portions of vnode operations.

The FiST input file also relates to the file-system model described in Section 3.5. In the last three sections of the input file, you can freely refer to mounts, files, and their attributes. We describe this syntax shortly below. The FiST Declarations section is where you define new ioctls; you use these new ioctls in the FiST Rules and Additional C Code sections. Also, you declare fan-in and fan-out in the FiST Declarations section, and use refer to multiple objects accordingly in the rest of the FiST input file.

The remainder of this chapter discusses the FiST language primitives, the various participants in a file system (such as files, mounts, and processes), their attributes and how to extend them and store them persistently, and how to control the execution flow in a file system. The examples in Chapter 8 further illustrate the FiST language.

## 4.2 FiST Syntax

FiST syntax allows referencing mounted file systems and files, accessing attributes, and calling FiST functions. Mount references begin with `$vfs`, while file references use a shorter "$" syntax because we expect them to appear often in FiST code. References may be followed by a name or number that distinguishes among multiple instances (e.g., `$1`, `$2`, etc.) This is especially useful when fan-out is used (Figure 2.8). Attributes of mounts and files are specified by appending a dot and the attribute name to the reference (e.g., `$vfs.blocksize`, `$1.name`, `$2.owner`, etc.) The scope of these references is the current vnode function in which they are executing.

There is only one instance of a running operating system. Similarly, there is only one process context that the file system has to be concerned with. Therefore FiST need only refer to these attributes. These read-only attributes are summarized in Table 4.1. The scope of all read-only "%" attributes is global.

| **Global** | **Meaning** |
|---|---|
| %blocksize | native disk block size |
| %gid | effective group ID |
| %pagesize | native page size |
| %pid | process ID |
| %time | current time (seconds since epoch) |
| %uid | effective user ID |

Table 4.1: Global read-only FiST variables provide information about the current state of processes and users.

FiST code can call FiST functions from anywhere in the file system. Some of these functions are shown in Table 4.2. The scope of FiST functions is global in the mounted file system. These functions form a comprehensive library of portable routines useful in writing file systems. The names of these functions begin with "fist" and they have the following features:

- FiST functions can take a variable number of arguments. For example, the lookup function can be given an optional fifth argument to use as the effective UID to perform the lookup with.
- FiST functions can omit some arguments where suitable defaults exist. For example, the unlink function's first argument is the directory in which to delete the file. If the first argument is omitted, FiST will assume that that the directory is the current directory.
- FiST functions can use different types for each argument. For example, the rename function can be given an object reference or a string name of the file to rename.
- FiST functions can be nested and may return any single value.

Each mount and file has attributes associated with it. FiST recognizes common attributes of mounted file systems and files that are defined by the system, such as the name, owner, last modification time, or protection modes. FiST also allows developers to define new attributes and optionally store them persistently. Attributes are accessed by appending the name of the attribute to the mount or file reference, with a single dot in between, much the same way that C dereferences structure field names. For example, the native block size of a mounted file system is accessed as `$vfs.blocksize` and the name of a file is `$0.name`.

FiST allows users to create new file attributes. For example, an ACL file system may wish to add timed access to certain files. The following FiST Declaration defines the new file attributes in such a file system:

```
per_vnode {
    int     user;       /* extra user */
```

| Function | Meaning |
|---|---|
| fistPrintf | print messages |
| fistStrEq | string comparison |
| fistMemCpy | buffer copying similar |
| fistLastErr | get the last error code |
| fistSetErr | set the return error code |
| fistReturnErr | return an error code immediately |
| fistSetIoctlData | set ioctl value to pass to a user process |
| fistGetIoctlData | get ioctl value from a user process |
| fistSetFileData | write arbitrary data to a file |
| fistGetFileData | read arbitrary data from a file |
| fistLookup | find a file in a directory |
| fistReaddir | read a directory |
| fistSkipName | hide a name of a file in a directory |
| fistOp | execute an arbitrary vnode operation |

Table 4.2: A sample of FiST functions. The full list appears in Appendix A.

```
    int     group;    /* extra group */
    time_t  expire;   /* access expiration time */
};
```

With the above definition in place, a FiST file system may refer to the additional user and group who are allowed to access the file as `$0.user` and `$0.group`, respectively. The expiration time is accessed as `$0.expire`.

The `per_vnode` declaration defines new attributes for files, but those attributes are only kept in memory. FiST also provides different methods to define, store, and access additional attributes persistently. This way, a file-system developer has the flexibility of deciding if new attributes need only remain in memory or should be saved more permanently.

For example, an encrypting file system may want to store an encryption key, cipher ID, and Initialization Vector (IV) for each file. This can be declared in FiST Declared section using:

```
fileformat SECDAT {
    char  key[16];    /* cipher key */
    int   cipher;     /* cipher ID */
    char  iv[16];     /* initialization vector */
};
```

Two FiST functions exist for handling file formats: fistSetFileData and fistGetFileData. These two routines can store persistently and retrieve (respectively) additional file system and file attributes, as well as any other arbitrary data. For example, to save the cipher ID in a file called `.key`, you can put the following code as part of an ioctl to process the key:

```
int cid;
/* set cipher ID */
fistSetFileData(".key", SECDAT, cipher, cid);
```

The above FiST function will produce kernel code to open the file named "`.key`" and write the value of the "cid" variable into the "cipher" field of the "SECDAT" file format, as if the latter had been a data structure stored in the "`.key`" file.

Finally, the mechanism for adding new attributes to mounts is similar. For files, the declaration is `per_vnode` whereas, for mounts, it is `per_vfs`. The routines fistSetFileData and fistGetFileData can be used to access any arbitrary persistent data, for both mounts and files.

## 4.3 Rules for Controlling Execution and Information Flow

In the previous sections we considered how FiST can control the flow of information between the various layers. In this section we describe how FiST can control the execution flow of various operations using FiST rules.

FiST does not change the interfaces that call it, because such changes will not be portable across operating systems and may require changing many user applications. FiST therefore only exchanges information with applications using existing APIs (e.g., ioctls) and those specific applications can then affect change.

The most control FiST file systems have is over the file system (vnode) operations that execute in a normal stackable setting. Figure 4.2 highlights what a typical stackable vnode operation does: (1) find the vnode of the lower-level mount, and (2) repeat the same operation on the lower vnode.

```
int fsname_getattr(vnode_t *vp, args...)
{
  int error;
  vnode_t *lower_vp = get_lower(vp);

  /* (1) pre-call code goes here */
  /* (2) call same operation on lower file system */
  error = VOP_GETATTR(lower_vp, args...);
  /* (3) post-call code goes here */
  return error;
}
```

Figure 4.2: A skeleton of typical kernel C code for stackable vnode functions. FiST can control all three sections of every vnode function: pre-call, post-call, and the call itself.

In this example, a vnode function receives a pointer to the vnode on which to apply the operation, and other arguments. First, the function finds the corresponding vnode at the lower-level mount. Next, the function actually calls the lower-level mounted file system through a standard `VOP_*` macro that applies the same operation, but on the file system corresponding to the type of the lower vnode. The macro uses the lower-level vnode, and the rest of the arguments unchanged. Finally, the function returns to the caller the status code which the lower level mount passed to the function.

There are three key parts in any stackable function that FiST can control: the code that may run before calling the lower-level mount (pre-call), the actual call to the lower-level mount, and the code that may run afterwards (post-call). FiST can insert arbitrary code in the pre-call and post-call sections, as well as replace the call part itself with any function you describe.

By default, the pre-call and post-call sections are empty, and the call section contains code to pass the operation to the lower-level file system. These defaults produce a file system that stacks on another but does not change behavior, and was designed so developers do not have to worry about the basic stacking behavior—only about their changes.

For example, a useful pre-call code in an encryption file system would be to verify the validity of cipher keys. A replication file system may insert post-call code to repeat the same vnode operation on other replicas. A versioning file system could replace the actual call to remove a file with a call to rename it; an example FiST code for the latter might be:

```
%op:unlink:call {
  fistRename($name, fistStrAdd($name, ".unrm"));
}
```

The general form for a FiST rule is:

$$\%callset : optype : part \; \{code\} \tag{4.1}$$

The format for each FiST rule was designed so that a single statement can identify the exact location or locations where code should be inserted, and the actual *code* to insert. The identification section of each FiST rule contains three components, separated by colons. These three components were designed such that each one continually refines the selection of the location to insert code. The first component, *callset*, selects operations based on their type (all, reading, or writing operations). The second component, *optype*, further refines the selection to a single vnode operation or a set of operations based on the data they manipulate. The last component, *part*, further refines the selection to a portion of a single vnode operation (pre-call, call, or post-call).

| **Call Sets** | |
|---|---|
| op | to refer to a single operation |
| ops | to refer to all operations |
| readops | to refer to non state changing operations |
| writeops | to refer to state changing operations |
| **Operation Types** | |
| all | all operations |
| data | operations that manipulate file data |
| name | operations that manipulate file names |
| The rest of the operation types specify one of the following vnode operations: create, getattr, l/stat, link, lookup, mkdir, read, readdir, readlink, rename, rmdir, setattr, statfs, symlink, unlink, and write. | |
| **Call Part** | |
| precall | part before calling the lower file system |
| call | the actual call to the lower file system |
| postcall | part after calling the lower file system |
| *ioctl* | name of a newly defined ioctl |

Table 4.3: Possible values in FiST rules' callsets, optypes, and parts include all individual file-system operations, sets of those, as well as parts of those operations.

Table 4.3 summarizes the possible values that a FiST rule can have. *Callset* defines a collection of operations to operate on. This allows developers to pick one or more operations based on their functionality—for example, all those operations that modify the file-system state (writeops). This is useful because file-system developers developing a new file system often have to modify a subset of file system operations that work similarly in a consistent manner. For example, a read-only file system can insert special code for all of the `writeops` operations to disable the operation.

*Optype* further defines the call set to a subset of operations based on the data that they operate on or to a single operation. This is useful because file-system developers often want to modify a subset of file system operations that act on similar data. For example, a file system that wants to prevent a certain file name from being accessed can do so in all of the file-system operations whose *optype* is `name`.

*Part* defines the part of the call that the following code refers to: pre-call, call, post-call, or the name of a newly defined I/O control (ioctl) system function. This allows developers the flexibility to insert or change any action that a single file-system operation may execute, as we described in Section 3.4.

Finally, *code* contains any C code enclosed in braces. This allows developers the power to include any code they wish, and therefore lets them create new functionality in FiST.

## 4.4 Filter Declarations and Filter Functions

FiST file systems can perform arbitrary manipulations of the data they exchange between layers. The most useful and at the same time most complex data manipulations in a stackable file system involve file data and file names. To manipulate them consistently without FiST or Basefs, developers must make careful changes in many places. For example, file data is manipulated in read, write, and all of the MMAP functions; file names also appear in many places: lookup, create, unlink, readdir, mkdir, etc.

While it would be possible to manipulate file data and file names in FiST using FiST rules, it would take too long to write and it increases the chances of human errors: developers would have to manipulate all related operations consistently, ensure that data pages and other objects are locked and unlocked correctly, insert and remove data pages from the page cache at the right times, update object reference counts, update error codes, update page mode bits, and more. Therefore, for manipulating file data and file names consistently, FiST provides a better mechanism that requires simple changes in one place only.

FiST simplifies the task of manipulating file data or file names using two types of *filters*. A filter is a function similar to a Unix shell filter such as `sed` or `sort`: it takes some input, and produce possibly modified output.

If developers declare `filter data` in their FiST files, fistgen looks for two data coding functions in the Additional C Code section of the FiST File: `encode_data` and `decode_data`. These functions take an input data page, and an allocated output page of the same size. Developers are expected to implement these coding functions in the Additional C Code section of the FiST file. The two functions must fill in the output page by encoding or decoding it appropriately and return a success or failure status code. Our encryption file system uses a data filter to encrypt and decrypt data (Section 8.1).

A variation of this declaration is `filter sca`. In addition to the encoding of data pages, this also includes code to support size-changing file systems, as we discuss in Chapter 6. This is a separate declaration because the size-changing file-system support includes substantially more code and carries a performance overhead that is not necessary for all file systems that manipulate data pages.

With the FiST declaration `filter name`, fistgen inserts code and calls to encode or decode strings representing file names. The file name coding functions (`encode_name` and `decode_name`) take an input file name string and its length. They must allocate a new string and encode or decode the file name appropriately. Finally, the coding functions return the number of bytes in the newly allocated string, or a negative error code. Fistgen inserts code at the caller's level to free the memory allocated by file name coding functions.

Using FiST filters, developers can easily produce file systems that perform complex manipulations of data or names exchanged between file-system layers.

## 4.5 Fistgen: The FiST Language Code Generator

Fistgen is the FiST language code generator. Fistgen reads in an input FiST file, and using the correct Basefs templates, produces all the files necessary to build a new file system described in the FiST input file. These output files include C file system source files, headers, sources for user-level utilities, and a Makefile to compile them on the given platform.

Fistgen implements a subset of the C language parser and a subset of the C preprocessor. It handles conditional macros (such as #ifdef and #endif). It recognizes the beginning of functions after the first set of declarations and the ending of functions. It parses FiST tags inserted in Basefs (explained in the next section) used to mark special places in the templates. Finally, fistgen handles FiST variables (beginning with $ or %) and FiST functions (such as fistLookup) and their arguments.

After parsing an input file, fistgen builds internal data structures and symbol tables for all the keywords it must handle. Fistgen then reads the templates, and generates output files for each file in the template directory. For each such file, fistgen inserts needed code, excludes unused code, or replaces existing code. In particular, fistgen conditionally includes large portions of code that support FiST filters: code to manipulate file data or file names. It also produces several new files (including comments) useful in the compilation for the new file system: a header file for common definitions, and two source files containing auxiliary code.

The code generated by fistgen may contain automatically generated functions that are necessary to support proper FiST function semantics. Each FiST function is replaced with one true C function—rather than a macro, inlined code, a block of code statements, or any feature that may not be portable across operating systems and compilers. While it might have been possible to use other mechanisms such as C macros to handle some of the FiST language, it would have resulted in unmaintainable and unreadable code. One of the advantages of the FiST system is that it produces highly readable code. Developers can even edit that code and add more features by hand, if they so choose.

Fistgen also produces real C functions for specialized FiST syntax that cannot be trivially handled in C. For example, the fistGetIoctlData function takes arguments that represent names of data structures and names of fields within. A C function cannot pass such arguments; C++ templates would be needed, but we opted against C++ to avoid requiring developers to know another language, because modern Unix kernels are still written in C, and to avoid interoperability problems between C++ produced code and C produced code in a running kernel. Preprocessor macros can handle data structure names and names of fields, but they do not have exact or portable C function semantics. To solve this problem, fistgen replaces calls to functions such as fistGetIoctlData with automatically generated specially named C functions that hard-code the names of the data structures and fields to manipulate. Fistgen generates these functions only if needed and only once.

# Chapter 5

# Stackable Templates

Basefs is the template file system used in FiST, and is the third and last of the three main components of the FiST system (the other two being the FiST language and the fistgen code generator). In this chapter we discuss the most important issues in designing the stacking templates and with an emphasis on how we achieved the most important goal of the templates—file system portability across different operating systems.

We begin by introducing the templates. Then we describe the two most important aspects that file systems manipulate—file data and file names—and the APIs that the templates use to manipulate these. The next two sections detail the design of the templates with respect to file data and file names. Finally, we discuss error handling in our templates.

## 5.1 Overview of the Basefs Templates

Basefs provides basic stacking functionality without changing other file systems or the kernel. This functionality is useful because it improves portability of the system. To achieve this functionality, the kernel must support three features:

1. In each of the VFS data structures, Basefs requires a field to store pointers to data structures at the layer below. This is needed to link data structures in one layer to the corresponding data structures in the layer immediately below. This linkage of upper to lower objects is what enables stacking: code in one layer can follow a pointer and then call code in a lower layer, passing it a corresponding object, as seen in Figure 4.2.
2. New file systems should be able to call VFS functions. This is needed so that newly added kernel code can call functions that already exist in the rest of the kernel. For dynamically-linked modules, a kernel file system may have to be partially linked and refer to symbols that can be resolved only after the file system is loaded into the kernel.
3. The kernel should export all symbols that may be needed by a new file system module. This is needed so that new in-kernel file system code is allowed to execute functions that exist in the rest of the kernel.

There are two key points that allow FiST to support portable stacking. First, we were able to abstract seemingly different vnode interfaces and find a common set of functionality that is useful to users and that all of the operating systems can use. This common functionality formed the basis for our templates. Second, we were able to fill in any missing functionality or one deemed useful to developers, by adding code to the templates; this was done without changing the core operating system code. In contrast, all past stacking works attempted to make significant changes to operating systems and file systems, and created new interfaces that were incompatible with interfaces of other systems or even previous interfaces on the same operating system.

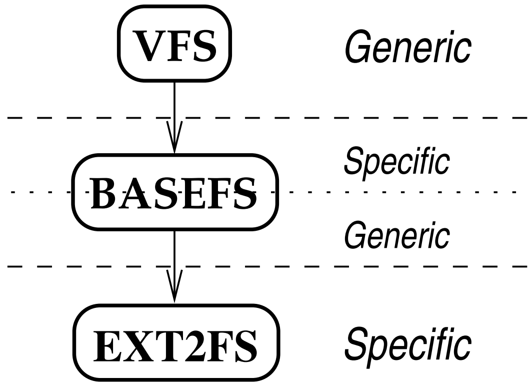



Figure 5.1: Where Basefs fits inside the kernel: it behaves as both an upper virtual file system and as a lower-level file system.

Basefs handles many of the internal details of operating systems, thus freeing developers from dealing with kernel specifics. Basefs provides a stacking layer that is independent from the layers above and below it, as shown in Figure 5.1. Basefs appears to the upper VFS as a lower-level file system. Basefs also appears to file systems below it as a VFS. Initially, Basefs simply calls the same vnode operation on the lower level file system.

Basefs performs data reading and writing on whole pages. This simplifies mixing regular reads and writes with memory-mapped operations, and gives developers a single paged-based interface to work with. Additional code can in included in Basefs that supports file systems that change data size, such as for compression. See Chapter 6.

To improve performance, Basefs copies and caches data pages in its layer; Basefs will also cache pages of the layers below it, in case the lower-level file system does not do so directly (on some operating systems, the VFS is responsible for inserting pages into the cache, not the actual file system). Basefs saves memory by caching at the lower layer only if file data is manipulated and fan-in was used; these are the usual conditions that require caching at each layer.

Basefs includes only the minimal code that it needs to function as a null-layer stackable file system. Substantial portions of extra code are included conditionally. Code to manipulate file data and file names, as well as debugging code is not included in Basefs by default. This code is included only if the file system needs it. By including only code that is necessary we generate output code that is more readable than code with multi-nested `#ifdef`/`#endif` pairs. Conditionally including this code typically results in improved performance compared to system that include all functionality and turn on or off what they need at run time. We report our performance in Section 9.3. Matching or exceeding the performance of other layered file systems was one of the design goals for FiST.

Basefs adds support for fan-out file systems natively. This code is also included conditionally, because it is more complex than single-stack file systems, adds more performance overhead, and consumes more memory. To improve performance, developers must define the maximum fan-out level used in their FiST input file. This allows fistgen to generate faster-running code. That code refers to objects of the file systems mounted immediately below using by dereferencing a single pointer from a fixed-size array of pointers. Compiling fixed offset references avoids having to compute them at run time and speeds up fan-out processing.

Basefs includes (conditionally compiled) support for many other features. This added support can be thought of as a library of common functions: opening, reading or writing, and then closing arbitrary files; storing extended attributes persistently; user-level utilities to mount and unmount file systems, as well as manipulate ioctls; inspecting and modifying file attributes, and more. See Appendix A for a full listing of these functions.

Finally, Basefs includes special *tags* that help fistgen locate the proper places to insert certain code. Inserting code at the beginning (pre-call) or the end (post-call) of functions is simple, but in some cases the code to add has to go elsewhere. For example, handling newly defined ioctls is done (in the `basefs_ioctl` vnode function) at the end of a C "switch" statement, right before the "default:" case.

In the rest of this chapter we detail some of the more complex design issues of the Basefs templates. In particular, we discuss the code that gets included conditionally to manipulate file data and file names. This code, while complex, allows the manipulation of fixed-size data pages. In the next chapter, Chapter 6, we discuss in much greater detail the additional template support for SCAs—file systems that can dynamically change the size of data they manipulate.

## 5.2 Manipulating Files

There are three parts of a file system that developers wish to manipulate most often: file data, file names, and file attributes. It is therefore important that the FiST system be able to handle these common cases efficiently and flexibly. 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 complex to use not just because many functions use them, but also because the directory reading function, `readdir`, is a restartable function: it processes only part of the total data in a directory, and thus needs to record the offset where it should continue to process data when the function is invoked again.

When you declare in your FiST input file that your file system wants to manipulate file data pages and file data names, fistgen includes additional code into the Basefs template. This code exports four functions that developers can use. These four functions address the manipulation of file data and file names:

1. **encode_data** takes a buffer whose size is the same as the native 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 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 as 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 FiST can implement most of the desired functionality of their file system in a few places and not have to worry about the rest.

## 5.3 Encoding and Decoding File Data Pages

File systems are designed to manipulate whole pages. This improves performance and simplifies the interaction with the VM system and the page cache. Our templates therefore also manipulate files one page at a time. This section discusses reading, writing, and appending to files in whole page units, the interfacing with the VM system, and page caching issues.

### 5.3.1 Paged Reading and Writing

We perform reading and writing on whole blocks of a size matching the native page size. Whenever a read for a range of bytes is requested, we compute the extended range of bytes up to the next page boundary, and apply the operation to the lower file system using the extended range. Upon successful completion, the exact number of bytes requested are returned to the caller of the vnode operation.

Writing a range of bytes is more complicated than reading. Within one page, bytes may depend on previous bytes (e.g., encryption), so we have to read and decode parts of pages before writing other parts of them.

Throughout the rest of this section we will refer to the upper vnode as $V$, and to the lower vnode as $V'$; $P$ and $P'$ refer to memory mapped pages at these two levels, respectively. The example [1] depicted in Figure 5.2 shows what happens when a process asks to write bytes of an existing file from byte 9000 until byte 25000. Let us assume that the file in question has a total of 4 pages (32768) worth of bytes in it.

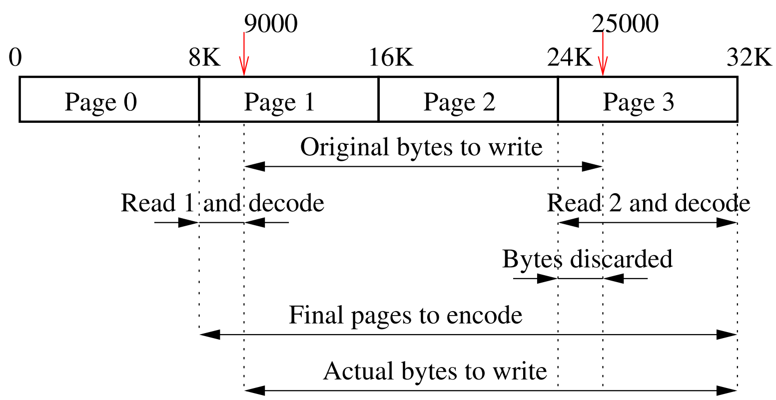



Figure 5.2: Writing Bytes in Basefs is done in terms of whole pages

1. Compute the extended page boundary for the write range as 8192–32767 and allocate three empty pages. (Page 0 of $V$ is untouched.)

2. Read bytes 8192–8999 (page 1) from $V'$, decode them, and place them in the first allocated page. We do not need to read or decode the bytes from 9000 onwards in page 1 because they will be overwritten by the data we wish to write anyway.

3. Skip intermediate data pages that will be overwritten by the overall write operation (page 2).

4. Read bytes 24576–32767 (page 3) from $V'$, decode them, and place them in the third allocated page. This time we read and decode the whole page because we need the last 32767$-$25000=7767 bytes and these bytes depend on the first 8192$-$7767=426 bytes of that page.

5. Copy the bytes that were passed to us into the appropriate offset in the three allocated pages.

6. Finally, we encode the three data pages and call the write operation on $V'$ for the same starting offset (9000). This time we write the bytes all the way to the last byte of the last page processed (byte 32767), to ensure validity of the data past file offset 25000.

[1] The example is simplified because it does not take into account sparse files or appending to files.

#### 5.3.1.1 Appending to Files

When files are opened for appending only, the VFS does not provide the vnode `write` function the real size of the file and where writing begins. If the size of the file before an append is not an exact multiple of the page size, data corruption may occur, since we will not begin a new encoding sequence on a page boundary. We therefore need to handle appended files also on page boundaries.

We solve this problem by detecting when a file is opened with an append flag on, turn off that flag before the open operation is passed on to $V'$, and replace it with flags that indicate to $V'$ that the file was opened for normal reading and writing. We save the initial flags of the opened file, so that other operations on $V$ could tell that the file was originally opened for appending. Whenever we write bytes to a file that was opened in append-only mode, we first find its size, and add that to the file offsets of the write request. In essence we convert append requests to regular write requests starting at the end of the file.

### 5.3.2 Memory Mapping

To support MMAP operations and executable binary files, we have to support memory-mapping vnode functions. As per Section 5.3.3, Basefs caches decoded pages, while the lower file system keeps cached encoded pages.

When a page fault occurs, the kernel calls the vnode operation `getpage`. This function retrieves one or more pages from a file and is designed to call repeatedly a function that retrieves a single page—`getapage`. The `getapage` function works as follows:

1. Check if the page is cached; if so, return it.
2. If the page is not cached, create a new page $P$.
3. Find $V'$ from $V$ and call the `getpage` operation on $V'$, returning page $P'$.
4. Copy the (encoded) data from $P'$ to $P$.
5. Map $P$ into kernel virtual memory and decode the bytes by calling `basefs_decode`.
6. Unmap $P$ from kernel VM, insert it into $V$'s cache, and return it.

The design of `putpage` was similar to `getpage`. In practice we also had to carefully handle two additional details, to avoid deadlocks and data corruption. First, pages contain several types of locks, and these locks must be held and released in the right order and at the right time. Second, the MMU keeps mode bits indicating status of pages in hardware, especially the referenced and modified bits. We had to update and synchronize the hardware version of these bits with their software version kept in the pages' flags. For a file system to have to know and handle all of these low-level details blurs the distinction between the file system and the VM system.

### 5.3.3 Interaction Between Caches

When Basefs is used on top of a native file system, both layers cache their pages; this improves performance. However, cache incoherency could result if pages at different layers are modified independently. A mechanism for keeping caches synchronized through a centralized cache manager was proposed by Heidemann [29]. Unfortunately, that solution involved modifying the rest of the operating system and other file systems.

Basefs performs its own caching, and does not explicitly touch the caches of lower layers. This keeps Basefs simpler and more independent of other file systems. Also, since pages can be served off of their respective layers, performance is improved. We decided that the higher a layer is, the more authoritative it is: when writing to disk, cached pages for the same file in Basefs overwrite their counterparts at the layers below. This policy correlates with the most common case of cache access, through the uppermost layer. Finally, note that a user process can access cached pages of a lower level file system only if it was mounted as a regular mount (Figure 2.9), meaning that fan-in was turned on; this is unavoidable and could lead to cache incoherency. If, however, Basefs is overlay mounted (fan-in turned off), user processes could not access lower level files directly, and cache incoherency for those files is less likely to occur.

## 5.4 Encoding and Decoding File Names

Encoding and decoding file names is done similarly to data pages. With file names, we do not have the added complexity of memory-mapped operations; most operations that manipulate file names are simple, such as mkdir, unlink, and symlink. One exception is the directory reading operation and we therefore describe how we handle this complex operation here.

Readdir is implemented in the kernel as a restartable function. A user process calls the readdir C library call, which is translated into repeated calls to the `getdents(2)` system call, passing it a buffer of a given size. The kernel fills the buffer with as many directory entries as will fit in the caller's buffer. If the directory was not read completely, the kernel sets a special EOF flag to false. As long as the flag is false, the C library function calls `getdents(2)` again.

The important issue with respect to directory reading is how to continue reading the directory from the offset where the previous read finished. This is accomplished by recording the last position and ensuring that it is returned to us upon the next invocation. Assume that a readdir vnode operation is called on vnode $V$ for $N$ bytes worth of directory data. We designed the functionality of our stacked `readdir` as follows:

1. Call the same vnode operation on $V'$ and read back $N$ bytes.
2. Create a new temporary buffer of a size that is as large as $N$.
3. Loop over the bytes read from $V'$, breaking them into individual records representing one directory entry at a time (`struct dirent`). For each such, we call `decode_filename` to find the original file name. We construct a new directory entry record containing the decoded file name and add the new record to the allocated temporary buffer.
4. Record the offset to read from on the next call to `readdir`; this is the position past the last file name we just read and decoded. This offset is stored in one of the fields of the `struct uio` (representing data movement between user and kernel space) that is returned to the caller. A new structure is passed to us upon the next invocation of readdir with the offset field untouched. This is how we are able to restart the call from the correct offset.
5. Return the temporary buffer to the caller of the vnode operation. If there is more data to read from $V'$, then we set the EOF flag to false before returning from this function.

The caller of `readdir` asks to read at most $N$ bytes. When we decode or encode file names, the result can be a longer or shorter file name. We ensure that we fill in the user buffer with no more `struct dirent` entries than could fit (but fewer is acceptable). Regardless of how many directory entries were read and processed, we set the file offset of the directory being read such that the next invocation of the `readdir` vnode operation will resume reading file names from exactly where it left off the last time.

## 5.5 Error Codes

FiST creates stackable, modular file systems that are designed to stand alone. That is, they are symmetric above and below them: they do not know which file system may be above or below them. This is an important feature of the modularity of stackable layers. With stacking, each layer handles its own error codes, but they must be propagated so the right error code reaches user processes.

By default, FiST file systems return the error codes they receive from the lower-level file system to the caller. If the file system has to perform multiple operations, each of which may succeed or fail, it returns the first failure code and aborts the remaining operations. This way you can compose a long stack of file systems, and the operations in the stack continue to propagate downward as long as they are successful. At the same time, any serious failure is propagated upwards in the stack until it reaches the user process that made the initial system call.

Because each stackable file system is independent, it does not automatically know what irreversible or non-idempotent actions another stackable file system has made. Therefore, such a stackable file system cannot automatically undo the actions of another. It is possible for several file systems to cooperate together and agree on the semantics of (possibly new) error codes such that by the time an error is propagated back to a user, all intermediate actions could be reversed. Such programming style, however, is more complex and begins to violate the nice independence between stacking layers because now file systems do have to know about the internal behavior of others.

The case for fan-out is similar. If your file system performs one or more operations on several file systems that it is stacked on, and some error occurs, it is up to the FiST file-system developer to determine the correct error behavior of such partial failures. FiST's default error behavior for fan-out file systems is to return any error as soon as it occurs. FiST does not impose any other error behavior because the correct behavior depends on the applicability of the file system that the developer is writing. For example, in a replicated writable file system, it is reasonable to assume that any error should be reported immediately and that intermediate changes be undone; that way the various replicas can remain in sync. In a Union file system (Section 8.3), on the other hand, an error in one file system need not be fatal: the file system can try the next one in the joined list of file systems, until one succeeds.

Much of the complexity of intermediate error semantics in stackable file systems can be alleviated with in-kernel transactions, but that is beyond the scope of this dissertation. See Section 10.1 for more discussion on future directions.

# Chapter 6

# Support for Size-Changing File Systems

Basefs provides basic stacking functionality without changing other file systems or the kernel. Basefs allows developers to define data encoding functions that apply to whole pages of file data, making it easy to produce, for example, a limited subset of encryption file systems. Like other stacking systems, however, Basefs as described in Chapter 5 did not support encoding data pages such that the result is of a different size. In this chapter we discuss the design for our additions to Basefs that support size-changing file systems such as compression. We describe a new algorithm for handling size-changing file systems in an flexible and efficient manner that provides good performance.

## 6.1 Size-Changing Algorithms

*Size-changing algorithms* (SCAs) are those that take as input a stream of data bits and produce output of a different number of bits. These SCAs share one quality in common: they are generally intended to work on whole streams of input data, from the beginning to the end of the stream. Some of the applications of such algorithms fall into several possible categories:

**Compression:** Algorithms that reduce the overall data size to save on storage space or transmission bandwidths. In this case we consider non-lossy compression.

**Encoding:** Algorithms that encode the data such that it has a better chance of being transferred, often via email, to their intended recipients. For example, Uuencode is an algorithm that uses only the simplest printable ASCII characters and no more than 72 characters per line. This is to ensure that uuencoded email or binaries can traverse the Internet even if they go through legacy email or networking systems that may not support the full ASCII set of characters or text lines longer than 72 characters.

In this category we also consider transformations to support internationalization of text, as well as Unicoding.

**Encryption:** These are algorithms that transform the data so it is more difficult to decode it without an authorization—a decryption key. Encryption algorithms work in different modes. The simplest mode is Cipher Feedback (CFB) mode [67]. This mode does not change the size of the data. As such, it is not the strongest algorithm because it provides potential attackers one more piece of information: the original input data size.

Stronger encryption modes include Cipher Block Chaining (CBC), a mode that typically increases the size of the output [67]. This algorithm is often stronger because it does not reveal what the original input data size was. Furthermore, by encoding the input into more output bits, the input data can become more *randomized*, increasing the brute-force search space.

Note that the above are just some of the possible SCAs. Nevertheless, these SCAs share one quality in common: they are intended to work on whole streams of input data, from the beginning to the end of the stream. For example, when you encrypt or decompress a data file, you will often do so on the complete file. Even if you wish to access just a portion of the file, you still have to encode or decode all of it until you reach the portion of interest.

This quality of these algorithms makes them suitable for email systems, for example, where a whole message needs to be encoded. These algorithms are more difficult to work with in a random access environment such as a file system. In a file system, users perform various actions such as reading or writing small portions of files in the middle of those files, or appending data to files. If you use SCAs with a file system, you may have to encode or decode whole files from beginning to end—before you can access the data you wish.

For example, consider a large compressed text file. You wish to insert a line of text, say, several lines before the end of the original file. From looking at the compressed data, you do not know where to insert the data, and how it should be compressed: SCAs do not provide that information. You will have to decompress the entire file, seek to the position of interest, insert your text line, and then compress the entire new file. This is a very expensive operation, especially if performed repeatedly. Next, we begin to describe our solution to this problem, and how it was designed to perform well.

## 6.2 Overview of Support for Size-Changing Stackable File Systems

Size-changing algorithms (SCAs) may change data offsets arbitrarily: either shrinking or enlarging data. A file encoded with an SCA will have offsets that do not correspond to the same offsets in the decoded file. In a stacking environment, the lower-level file system contains the encoded files, while the decoded files are accessed via the upper layer. To find where specific data resides in the lower layer, an efficient mapping is needed that can tell where the starting offset of the encoded data is for a given offset in the original file.

Unfortunately, general-purpose SCAs have never been implemented in stackable file systems. Several works we describe in Chapter 2.1.9.1 suggested the idea of stackable compression file systems, but only one showed promise—Heidemann's unified cache manager [29]. In that work, Heidemann showed how his cache manager could map uncompressed cached data pages to compressed pages. However, a prototype was not made available and no further details were given. Furthermore, none of the past stacking works showed up to support general-purpose SCAs in any file system framework. The problem we set out to solve was how to support general-purpose SCAs in a way that is easy to use, performs well, and is available for many file systems.

We propose an efficient mapping for SCA stackable file systems based on an *index file*. The index file is a separate file containing meta data that serves as a fast index into an encoded file. Many file systems separate data and meta data: this is done for efficiency and reliability. Meta-data is considered more important and so it gets cached, stored, and updated differently than regular data. The index file is separate from the encoded file data for the same reasons. The index file stores meta-data information identifying offsets in an associated encoded file. For efficiency reasons, we can read the generally smaller index file quickly, and hence find out the exact offsets in the larger data file to read from. This improves performance. For reliability reasons, we designed the index file so it could be recovered from the data file in case the index file is lost or damaged. This, in fact, offers some improvement over usual Unix file systems: if their meta data (inodes, super-blocks, directory data blocks) is lost, it rarely can be recovered. The alternative to our design would have been to include the index data into the main encoded file, but this would have (1) hurt performance as we would have had to perform many seeks in the file to locate the data needed, and (2) complicated the rest of the design and implementation considerably.

As shown in Figure 6.1, the basic idea is that an upper level file system can efficiently access the decoded version of an encoded file in a lower-level file system by using the meta-data information in an associated index file that resides on the lower-level file system.

The cost of SCAs can be high. Therefore it is important to ensure that we minimize the number of times we invoke

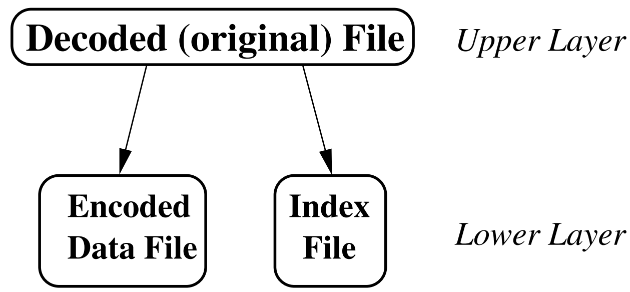



Figure 6.1: Each original data file is encoded into a lower data file. Additional meta-data index information is stored in an index file. Both the index file and the encoded data files reside in the lower-level file system.

these algorithms and the number of bytes they have to process each time. The way data chunks are stored and accessed can affect this performance, depending on the types and frequencies of file operations. File accesses follow several patterns:

- The most popular file-system operation is stat(), which results in a file lookup. Lookups account for 40–50% of all file-system operations [47, 61].
- Most files are read, not written. The ratio of reads to writes is often 4–6 [47, 61]. For example, compilers and editors read in many header and configuration files, but only write out a handful of files.
- Files that are written are often written from beginning to end. Compilers, user tools such as “cp”, and editors such as emacs write whole files in this way. Furthermore, the unit of writing is usually set to match the system page size. We have verified this by running a set of common edit and build tools on Linux and recording the write start offsets, the size of write buffers, and the current size of the file.
- Files that are not written from beginning to end are often appended to. The number of appended bytes is often small. This is true for various log files that reside in /var/log, such as Web server access logs.
- Very few files are written in the middle. This happens in two cases. First, when the GNU `ld` creates large binaries, it writes a sparse file of the target size and then seeks and writes the rest of the file in a non-sequential manner. To estimate the frequency of writes in the middle, we instrumented a null-layer file system with a few counters. We then measured the number and type of writes for our large compile benchmark (Section 9.4.3.1). We counted 9193 writes, of which 58 (0.6%) were writes before the end of a file.

  Second, data-base files are also written in the middle. We surveyed our own site’s file servers and workstations (several hundred hosts totaling over 1TB of storage) and found that these files represented less than 0.015% of all storage. Of those, only 2.4% were modified in the past 30 days, and only 3% were larger than 100MB.
- All other operations (together) account for a small fraction of file operations [47, 61].

Given the above access patterns, we designed our system to optimize performance for the more common and important cases, while not harming performance unduly when the seldom-executed cases occur. We discuss how we support these operations in Section 6.3.1.

Throughout the rest of this chapter, we will use the following three terms. An *original file* is the complete un-encoded file that the user accessing our stackable file system sees; the *data file* is the SCA-encoded representation of the original file, which encodes whole pages of the original file; the *index file* maps offsets of encoded pages between their locations in the original file and the data file.

## 6.3 The Index File

Our system encodes and decodes whole pages, which fits well with file system operations. The index table assumes this and stores offsets of encoded pages as they appear in the encoded file.

To illustrate how this works, consider an example of a file in a compression file system as shown in Figure 6.2. The figure shows the mapping of offsets between the upper (original) file, and the lower (encoded) data file. To find out the bytes in page 2 of the original file, we read the data bytes 3000–7200 in the encoded data file, decode them, and return to the VFS that data in page 2.

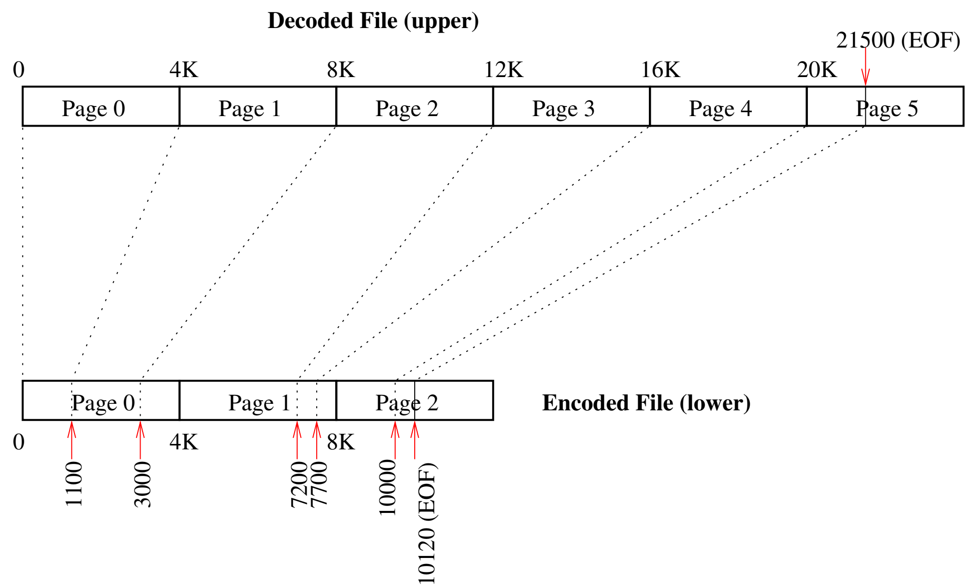



Figure 6.2: An example of a 32-bit file system that shrinks data size (compression). Each upper page is represented by an encoded lower "chunk." The mapping of offsets is shown in Table 6.1.

To find out which encoded bytes we need to read from the lower file, we consult the index file, shown in Table 6.1. The index file tells us that the original file has 6 pages, that its original size is 21500 bytes, and then it lists the ending offsets of the encoded data for each upper page. Finding the lower offset for the upper page 2 is a simple linear dereferencing of the data in the index file; we do not have to search the index file linearly. Note that our design of the index file supports both 32-bit and 64-bit file systems, but the examples we provide here are for 32-bit file systems.

The index information is stored in a separate small file, thus using up one more inode. We measured the effect that the consumption of an additional inode would have on typical file systems in our environment. We found that disk data block usage is often 6–8 times greater than inode utilization on disk-based file systems, leaving plenty of free inodes to use.

For a given data file $F$, we create an index file called $F$`.idx`. We decided to store the index table in a separate file for three reasons:

1. The index file is small. We store one word (4 bytes on a 32-bit file system) for each data page (usually 4096 bytes). On average, the index table size is about 1024 times smaller than the original data file.
2. The index contains meta data—original file size and page offsets—which are more logically stored outside the data itself, as is the case with many file systems. That allows us to manage the data file and the index file separately.

| **Word** (32/64 bits) | **Representing** | **Regular IDX File** | **With Fast Tail** (ft) |
|---|---|---|---|
| 1 (12 bits) | flags | ls=0,ft=0,... | ls=0,ft=1,... |
| 1 (20–52 bits) | # pages | 6 | 5 |
| 2 | orig. file size | 21500 | 21500 |
| 3 | page 0 | 1100 | 1100 |
| 4 | page 1 | 3000 | 3000 |
| 5 | page 2 | 7200 | 7200 |
| 6 | page 3 | 7700 | 7700 |
| 7 | page 4 | 10000 | 10000 |
| 8 | page 5 | 10120 | |

Table 6.1: Format of the index file for Figures 6.2 and 6.3. Fast tails are described in Section 6.3.1.1. The first word encodes both flags and the number of pages in the index file. The “ls” (large size) flag is the very first bit in the index file and indicates if the index file encodes a 32-bit (0) or 64-bit (1) file system.

3. Since the index file is relatively small, we can read it completely into kernel memory and manipulate it there. To improve performance, we write the final modified index table only after the original file is closed and all of its data flushed to stable media. (Section 6.3.2.2 discusses how to recover from a lost index file.)

We read the index file into memory as soon as the main file is open. That way we have fast access to the index data in memory. The index information for all pages is stored linearly, and each index entry typically takes 4 bytes. That way we can compute the needed index information very simply, and find it from the index table using a single dereference into an array of 4-byte words (integers). We write any modified index information out after the main file is closed and its data flushed to stable media.

The index file starts with a word that encodes two things: flags and the number of pages in the corresponding original data file. We reserve the lower 12 bits for special flags such as whether the index file encodes a file in a 32-bit or a 64-bit file system, whether fast tails were encoded in this file (see Section 6.3.1.1), etc. The very first bit of these flags, and therefore the first bit in the index file, determines if the file encoded is part of a 32-bit or a 64-bit file system. This way, just by reading the first bit we can determine how to interpret the rest of the index file: 4 bytes to encode page offsets on 32-bit file systems or 8 bytes to encode page offsets on 64-bit file systems.

We use the remaining 20 bits (on a 32-bit file system) for the number of pages because $2^{20}$ 4KB pages (the typical page size on i386 and SPARCv8 systems) would give us the exact maximum file size we can encode in 4 bytes on a 32-bit file system, as explained next; similarly $2^{52}$ 4KB pages is the exact maximum file size on a 64-bit file system.

The index file also contains the original file’s size (second word). We store this information in the index file so that commands like “ls –l” and others using stat(2) would work correctly. That is, if a process looks at the size of the file through the upper-level file system, it would get the original number of bytes and blocks. The original file’s size can be computed from the starting offset of the last data chunk in the encoded file, but it would require decoding the last (possibly incomplete) chunk (bytes 10000–10120 in the encoded file in Figure 6.2) which can be an expensive operation depending on the SCA. Storing the original file size in the index file is a speed optimization that only consumes one more word—in a physical data block that most likely was already allocated.

### 6.3.1 File Operations

As we discussed in Section 6.2, we designed our system to optimize performance for the most common file operations. In this section we detail those optimizations. Note that most of this design relates to performance optimizations, while a small part (Section 6.3.1.3) addresses correctness.

To handle file lookups fast, we store the original file's size in the index table. The index file is usually 1024 times smaller than the original file. Due to locality in the creation of the index file, we assume that its name will be found in the same directory block as the original file name, and that the inode for the index file will be found in the same inode block as the encoded data file. Therefore reading the index file requires reading one additional inode and often only one data block. After the index file is read into memory, returning the file size is done by copying the information from the index table into the "size" field in the current inode structure. All other attributes of the original file come from the inode of the actual encoded file. Once we read the index table into memory, we allow the system to cache its data for as long as possible. That way, subsequent lookups will find files' attributes in the attribute cache.

Since most file systems are structured and implemented internally for access and caching of whole pages (usually 4KB or 8KB), we decided to encode the original data file in whole pages. In this way we improved performance because our encoding unit is the same as that used by the paging system, and especially the page cache (see Section 9.4). This also helped simplify our code because interfacing with the VFS and the page cache was more natural. For file reads, the cost of reading in a data page is fixed: a fixed offset lookup into the index table gives us the offsets of encoded data on the lower-level data file; we read this encoded sequence of bytes, decode it into exactly one page, and return that decoded page to the user.

Since our stackable system is page-based, it was easier for us to write whole files, especially if the write unit was one page size. In the case of whole file writes, we simply encode each page size unit, add it to the lower level encoded file, and add one more entry to the index table. We discuss the cases of file appends and writes in the middle in Sections 6.3.1.1 and 6.3.1.2, respectively.

We did not have to design anything special for handling all other file operations. We simply treat the index file at the same time we manipulate the corresponding encoded data file. An index file is created only for regular files; we do not have to worry about symbolic links, because the VFS will only call our file system to open a regular file. When a file is hard-linked, we also hard-link the index file, using the name of the new link with a the ".idx" extension added. When a file is removed from a directory or renamed, we apply the same operation to the corresponding index file. We can do so in the context of the same file-system operation, because the directory in which the operation occurs is already locked.

#### 6.3.1.1 Fast Tails

One common action on files is to append to them. Often, a small number of bytes is appended to an existing file. Encoding algorithms such as compression and encryption are more efficient when they encode larger chunks of data. Therefore it is better to encode a larger number of bytes together. Our design calls for encoding whole pages whenever possible. Table 6.1 and Figure 6.2 show that only the last page in the original file may be incomplete, and that incomplete page gets encoded too. If we append, say, 10 more bytes to the original (upper) file of Figure 6.2, we have to keep it and the index file consistent: we must read the 1020 bytes from 20480 (20K) until 21500, decode them, add the 10 new bytes, encode the new 1030 sequence of bytes, and write it out in place of the older 1020 bytes in the lower file. We also have to update the index table for two things: the total size of the original file is now 21510, and word number 8 in the index file may be in a different location than 10120 (depending on the encoding algorithm, it may be greater, smaller, or even the same).

The number of times that we need to read, decode, append, and re-encode a chunk of bytes for each append increases as the number of bytes to append grows smaller and the number of encoded bytes grows closer to one full page. In the worst case, this method yields a complexity of $O(n^2)$ in the number of bytes that have to be decoded and encoded, multiplied by the cost of the encoding and decoding of the SCA. To solve this problem, we added a *fast tails* run-time mount option that allows for up to a page size worth of unencoded data to be added to an otherwise encoded data file. This is shown in the example in Figure 6.3.

In this example, the last full page that was encoded is page 4. Its data bytes end on the encoded data file at offset 10000 (page 2). The last page of the original upper file contains 1020 bytes (21500 less 20K). So we store these 1020 bytes directly at the end of the encoded file, after offset 10000. To aid in computing the size of the fast tail, we add two more

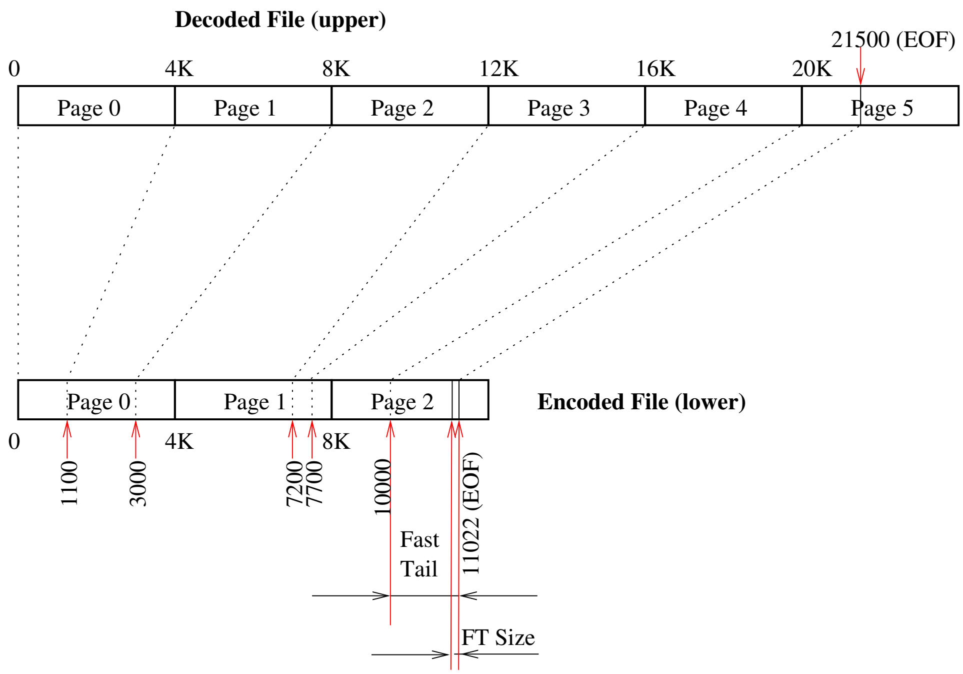



Figure 6.3: Fast tails. A file system similar to Figure 6.2, only here we store up to one page full of un-encoded raw data. When enough raw data is collected to fill a whole fast-tail page, that page is encoded.

bytes to the end of the file past the fast tail itself, listing the length of the fast tail. (Two bytes is enough to list this length, since typical page sizes are less than $2^{16}$ bytes long.) The final size of the encoded file is now 11022 bytes long.

With fast tails, the index file does not record the offset of the last tail, as can be seen from the right-most column of Table 6.1. The index file, however, does record in its flags field (first 12 bits of the first word) that a fast tail is in use. We put that flag in the index table to speed up the computations that depend on the presence of fast tails. We put the length of the fast tail in the encoded data file to aid in reconstruction of a potentially lost index file, as described in Section 6.3.2.2.

When fast tails are in use, appending a small number of bytes to an existing file does not require data encoding or decoding, which can speed up the append operation considerably. When the size of the fast tail exceeds one page, we then encode the first page worth of bytes, and start a new fast tail.

Fast tails, however, may not be desirable all the time exactly because they store unencoded bytes in the encoded file. If the SCA used is an encryption one, it is insecure to expose plaintext bytes at the end of the ciphertext file. For this reason, fast tails is a run-time global mount option that affects the whole file system mounted with it. The option is global because typically users wish to change the overall behavior of the file system with respect to this feature, not on a per-file basis.

#### 6.3.1.2 Write in the Middle

User processes can write any number of bytes in the middle of an existing file. A key point with our system is that *whole* pages are encoded and stored in a lower level file as individual encoded chunks. A new set of bytes written in the middle of the file may encode to a different number of bytes in the lower level file. If the number of new encoded bytes is greater than the old number, we have to shift the remaining encoded file outward to make room for the new bytes. If the number of bytes is smaller, we have to shift the remaining encoded file inward to cover unused space. In addition, we have to adjust the index table for each encoded data chunk which was shifted.

To improve performance, we shift data pages in memory and keep them in the cache as long as possible. That way, subsequent write-in-the-middle operations that may result in additional inward or outward shifts will only have to manipulate data pages already cached and in memory. Of course, any data page shifted is marked as dirty, and we let the paging system flush it to disk when it sees fit.

Note that data that is shifted in the lower-level file does not have to be re-encoded. This is because that data still represents the actual encoded chunks that decode into their respective pages in the upper file. The only thing remaining is to change the end offsets for each shifted encoded chunk in the index file.

We examined several performance optimization alternatives that would have encoded the information about inward or outward shifts in the index table, and even possibly some of the shifted data. We rejected these ideas for several reasons: (1) they would have complicated the code considerably, (2) they would have made recovery of an index file very difficult, and (3) they would have resulted in fragmented data files that would have required a defragmentation procedure. Since the number of writes in the middle we measured was so small (0.6% of all writes), we do consider our simplified design to be a good cost vs. performance balance. Note that even with our simplified solution, our file systems work perfectly correctly. Section 9.4 shows the benchmarks we ran to test writes in the middle and proves that our solution produces very good performance overall.

#### 6.3.1.3 Truncate

One interesting design issue we faced was with the truncate(2) system call. Although this call occurs less than 0.02% of the time [47, 61], we still had to ensure that it behaved correctly. Truncate can be used to shrink a file as well as enlarge it, potentially making it sparse with new "holes." We had four cases to deal with:

1. Truncating on a page boundary. In this case, we truncate the encoded file exactly after the end of the chunk that now represents the last page of the upper file. We update the index table accordingly: it has fewer pages in it.
2. Truncating in the middle of an existing page. In this case, which results in a partial page, we read and decode the whole page, and re-encode the bytes within representing the part before the truncation point. We update the index table accordingly: it now has fewer pages in it.
3. Truncating in the middle of a fast tail. In that case we just truncate the lower file where the fast tail is actually located. We then update the size of the fast tail at its end, and update the index file to indicate the (now) smaller size of the original file.
4. Truncating past the end of the file is akin to extending the size of the file and possibly creating zero-filled holes. We read and re-encode any partially filled page or fast tail that used to be at the end of the file before the truncation; we have to do that because that page now contains a mix of non-zero data and zeroed data. We encode all subsequent zero-filled pages. This is important for some applications such as encryption, where every bit of data—zeros or otherwise—must be encrypted.

### 6.3.2 Additional Benefits of the Index File

So far we have concentrated on performance concerns in our system, since they are an important part of our design. However, the index file design provides two additional benefits:

1. **Low Resource Usage**: without harming performance, our system uses little disk space for storing the index file. The index file is a small fraction of the size of the original file.
2. **Consistency**: the index file can be recovered completely, since it represents important meta data that is stored separately.

#### 6.3.2.1 Low Resource Usage

We designed our system to use few additional resources over what would be consumed normally. However, when considering resource consumption, we gave a higher priority to performance concerns.

The index file was designed to be small, as seen in Table 6.1. It usually includes four bytes for the size of the full original file, four bytes indicating the number of page entries (including flags), and then that many index entries, four bytes each. For each page of 4096 bytes we store 4 bytes in the index file. This results in a reduction size factor of over 1000 between the size of the original file and the index file. Specifically, an index file that is exactly 4096 bytes long (one disk block on an EXT2 file system formatted with 4KB blocks) can describe an original file size of 1022 pages, or 4,186,112 bytes (almost 4MB).

By keeping the index file small, we ensure that the contents of most index files can be read and stored in memory in under one page, and can then be manipulated in fast memory. Since we create index files along with the encoded data files, we benefit from locality: the directory data block and inode blocks for the two files are already likely to be in memory, and the physical data blocks for the two files are likely to reside in close proximity to each other on the physical media.

The size of the index file is less important for SCAs which increase the data size, such as unicoding, uuencoding, and most forms of encryption. The more the SCA increases the data size, the less significant the size of the index file becomes. Even in the case of SCAs that decreased data size, such as compression, the size of the index file may not be as important given the savings already gained from compression.

To save resources even further, we efficiently support zero-length files. A zero-length original data file is represented by a zero-length index file. When the encoded file exists but the index file does not, it indicates that the index file was lost, and can be recovered as described in Section 6.3.2.2.

#### 6.3.2.2 Index File Consistency

With the introduction of a separate index file to store the index table, we now have to maintain two files consistently.

Normally, when a write or create operation occurs on a file, the directory of that file is locked. We keep the directory locked also when the we update the index file, so that both the encoded data file and the index file are guaranteed to be written correctly.

We assume that encoded data files and index files will not become corrupt internally due to media failures. This situation is no worse than normal file systems where a random data corruption may not be possible to fix. However, we do concern ourselves with two potential problems: partially written or lost index files.

An index file could be partially written if the file system is full or the user ran out of quota. In the case where we were unable to write the complete index file, we simply remove it and print a warning message on the console. The absence of the index file on subsequent file accesses will trigger an in-kernel mechanism to recover the index file.

An index file could be lost if it was removed intentionally, say, after a partial write. It could be lost unintentionally by a user who removed it directly from the lower file system. If the index file is lost or does not exist, we can no longer easily tell where encoded bytes were stored. In the worst case, without an index file, we have to decode the complete file to locate any arbitrary byte within. However, since the cost of decoding a complete file and regenerating and index table are nearly identical (see Section 9.4.6), we chose to regenerate the index table immediately if it does not exist, and then proceed as usual as the index file now exists.

We verify the validity of the index file when we use the index table. We check that all index entries are monotonically increasing, that it has the correct number of entries, file size matches the last entry, flags used are known, etc. The index file is regenerated if an inconsistency is detected. This helps our system to survive certain meta-data corruptions that could occur as a result of software bugs or direct editing of the index file.

We designed our system so that the index file can be recovered reliably in all cases. Four important pieces of information are needed to recover an index file given an encoded data file. These four are available in the kernel to the running file system:

1. the SCA used,
2. the page size of the system on which the encoded data file was created,
3. whether the file system used is 32-bit or 64-bit, and
4. whether fast tails were used.

To recover an index file we read an input encoded data file and decode the bytes until we fill out one whole page of output data. We rely on the fact that the original data file was encoded in units of page size. The offset of the input data where we finished decoding onto one full page becomes the first entry in the index table. We continue reading input bytes and produce more full pages and more index table entries. If fast tails were used, then we read the size of the fast tail from the last two bytes of the encoded file, and do not try to decode it (since it was written un-encoded).

If fast tails were not used and we reached the end of the input file, that last chunk of bytes may not decode to a whole output page. In that case, we know that was the end of the original file, and we mark the last page in the index table as a partial page. While we are decoding pages, we sum up the number of decoded bytes and fast tails, if any. The total is the original size of the data file, which we record in the index table. We now have all the information necessary to write the correct index file and we do so.

## 6.4 Summary

SCAs are useful tools in the manipulations of data files, but were not designed for use with file systems. If used trivially with file systems, SCAs could perform so poorly as to outweigh their benefits. We have designed a system that allows SCAs to be used with file systems efficiently. Using an index file, we encode special meta data separately from the encoded data file itself. Our system uses the index file to speed up common file operations many-fold. We evaluate our system in Section 9.4.

# Chapter 7

# Implementation

We implemented the FiST system for Solaris, Linux, and FreeBSD because these three operating systems span the most popular modern Unix platforms and their internal designs are sufficiently different from each other. This forced us to understand the generic problems in addition to the system-specific problems. Also, we had access to kernel sources for all three platforms, which proved valuable during the development of our templates. Finally, all three platforms support loadable kernel modules, which sped up the development and debugging process. Loadable kernel modules are a convenience in implementing FiST; they are not required. In this chapter we discuss some of the important aspects of our implementation of the FiST system.

## 7.1 Templates

In order to achieve portability at the FiST language level, the templates had to export a fixed API to fistgen, the FiST language code generator. This API includes the four encoding and decoding functions (Section 5.2). Also needed were hooks for fistgen to insert certain code (Section 3.2), and finally, the ability to link objects between layers.

Linking objects between layers is the key to stacking. A stackable file system follows these links, often C pointers, to find the corresponding objects in the layer below. Then the stackable file system calls the layer below but using the lower object, as can be seen in Figure 4.2.

### 7.1.1 Stacking

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 7.1.

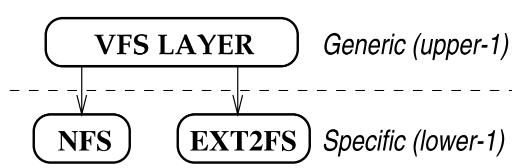



Figure 7.1: Normal file-system boundaries. Most kernels implement a common set of file-system functionality in a generic component called the VFS. Actual file systems are called from the VFS, and may also call functions in the VFS.

When a stackable file system such as Basefs is added to the kernel, these boundaries are obscured, as seen in Figure 7.2.

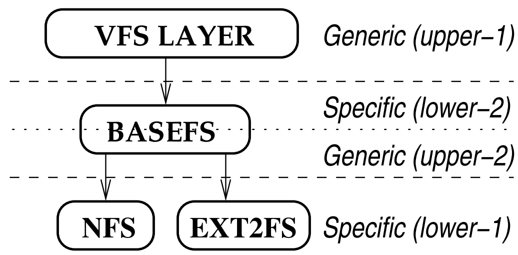



Figure 7.2: File-system boundaries with Basefs. Basefs must appear to the VFS as a lower-level file system. At the same time Basefs must treat lower-level file systems as if it is the VFS that is calling the lower-level file system

Basefs assumes a dual responsibility: it must appear to the 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 implementation of Basefs. The file system boundary as depicted in Figure 7.1 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 Basefs in its interaction with them.

The implementation of the templates for Solaris proceeded as described in Chapter 5, where we gave details using Solaris terminology. In the next two sections we describe the differences in the implementations of the templates between Solaris and FreeBSD, and Solaris and Linux, respectively.

### 7.1.2 FreeBSD

FreeBSD 3.$x$ is based on BSD-4.4Lite. We chose it as the second port because it represents another major segment of Unix operating systems. FreeBSD's vnode interface is similar to Solaris's and the port was straightforward. FreeBSD's version of the loopback file system is called *nullfs* [52], which is a template for writing stackable file systems. Unfortunately, ever since the merging of the VM and Buffer Cache in FreeBSD 3.0, stackable file systems stopped working because of the inability of the VFS to correctly map data pages of stackable file systems to their on-disk locations. To solve this, we worked around two deficiencies in nullfs. First, writing large files resulted in some data pages getting zero-filled on disk; this forced us to perform all writes synchronously. Second, memory mapping through nullfs panicked the kernel, so we implemented MMAP functions ourselves. We implemented `getpages` and `putpages` using `read` and `write`, respectively, because calling the lower-level's page functions resulted in a UFS pager error.

Thanks in part to the efforts of this work, the FreeBSD community has begun fixing these problems in FreeBSD 5.1, which is currently under development. When FreeBSD 5 is released with all of the necessary fixes, we would be able to port our existing FreeBSD Basefs templates to version 5.$x$ without much effort, since the FreeBSD community chose to change very little in the VFS's APIs to accommodate the needed fixes. Nevertheless, no changes to the FiST language or fistgen would be needed, just an update to the FreeBSD templates (largely to remove our workarounds).

### 7.1.3 Linux

Linux supports many more different file systems than either Solaris or FreeBSD. Because different file systems require different services from the rest of the kernel, the Linux VFS is more complex than Solaris and FreeBSD. This complexity, however, results in more flexibility for file system designers, as the VFS offloads much of the functionality traditionally implemented by file-system developers. In the rest of this section we describe the operations and the data structures that make up the Linux VFS, and outline where they differ from Solaris and FreeBSD.

#### 7.1.3.1 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, reads its fields from the mounted file system. Note that in Solaris and FreeBSD, all vnode operations are mandatory and must be implemented by all file systems.

**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.

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

#### 7.1.3.2 Data Structures

When we began our work in Linux, we used Linux version 2.0. We have since used all versions of Linux up to 2.4. In the process, we updated our templates to accommodate changes to the Linux VFS.

There are three primary data structures that are used in all Linux virtual file systems. We used these first in our Linux 2.0 port.

**super_block:** represents an instance of a mounted file system (also known as `struct vfs` in BSD).

**inode:** represents a file object in memory (also known as `struct vnode` in BSD).

**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 directory entry (dentry).

Later, Linux 2.1 and 2.2 added two more data structures that we support:

**dentry:** represents an inode that is cached in the Directory Cache (dcache) and also includes its name. A native form of this data structure existed in 2.0, but we did not have to use it. This structure was 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.

**vm_area_struct:** represents custom per-process virtual memory manager page-fault handlers. Multiple such page-fault handlers can exist for different pages of the same file.

More recently, Linux 2.3 and 2.4 added two more data structures which we also support:

**vfsmount:** is to a super_block what a dentry is to an inode, a higher-level abstraction. The vfsmount data structure contains fields, data, and operations that are common to all super_block data structures. The latter contain file-system–specific data and operations. With the vfsmount data structure, for example, a single mount point can contain a list of physical file systems mounted at that point, opening the door to device-level file-system features such as unification and fail-over.

**address_space:** is a data structure that contains paging operations related to vm_area_struct. One address space can contain a list of vm_area_struct structures (custom page-fault handlers). This data structure contains some operations that used to be in other data structures, but also newer operations intended to support a transaction-like interface to page data synchronization.

The key point that enables stacking is that each of the major data structures used in the file system contains a field into which file system specific data can be stored. Basefs uses that private field to store several pieces of information, especially a pointer to the corresponding lower-level file system's object.

When we began our work in Linux, the vm_area_struct data structure was missing this field. One of the contributions we have made to Linux in the past few years is the addition of such a field to this data structure and the associated code that uses it. Many of our contributions are now part of the mainline Linux kernel.

Figure 7.3 shows the connections between some objects in Basefs and the 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 Basefs is called, it finds the corresponding lower-level's object from the current one, and repeats the same operation on the lower object.

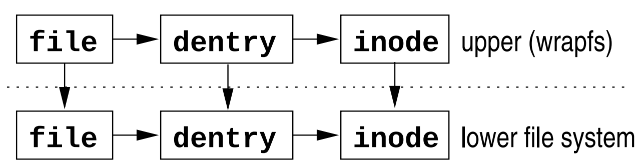



Figure 7.3: Connections between Basefs and the stacked-on file system. Each data structure that has multiple references to it, also contains an increased reference count.

Figure 7.3 also suggests one additional complication that Basefs 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 Basefs 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 Basefs as independent from other layers as possible. The horizontal arrows in Figure 7.3 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 Basefs's objects pointing to invalid objects.

## 7.2 Size-Changing Algorithms

Our size-changing extensions are based on our original Basefs templates. The idea of templates fits with this work quite well: all of the complexity of the handling of the index file can be hidden from developer's eyes. We changed the templates to support SCAs without changing the encoding and decoding routines' prototypes: developers now may return arbitrary length buffers rather than being required to fill in exactly one output page.

Our SCA work is done entirely in the templates. The only change we introduced in the FiST language is to add another high-level directive that tells the code generator to include SCA support. We decided to make this code optional because it adds overhead and is not needed for file systems that do not change data size. Using FiST, it is possible to write stackable file systems that do not pay the overhead of SCA support if they do not require changing the size of data.

Currently, SCA support is available for Linux 2.3 only. We concentrated on a single port in order to explore and design all of the algorithmic issues related to the index file, before embarking on SCA support for other platforms. Our primary goal in extending our Basefs templates to support SCA was to prove that size-changing stackable file systems can be designed to perform well. When we port our SCA support to the other templates, we would still be able to describe an SCA file system once in the FiST language. From this single description, however, we would then produce a number of working file-system modules with size-changing support.

## 7.3 Fistgen

The remainder of this section describes the implementation of fistgen. Fistgen translates FiST code into C code which implements the file system described in the FiST input file. The code can be compiled as a dynamically loadable kernel module or statically linked with a kernel. In this section we describe the implementation of key features of FiST that span its full range of capabilities.

We implemented read-only execution environment variables (Section 4.2) such as `%uid` by looking for them in one of the fields from `struct cred` in Solaris or `struct ucred` in FreeBSD. The VFS passes these structures to vnode functions. The Linux VFS simplifies access to credentials by reading that information from the disk inode and into the in-memory vnode structure, `struct inode`. So on Linux we find UID and other credentials by referencing a field directly in the inode which the VFS passes to us.

Most of the vnode attributes listed Section 4.2 are simple to find. On Linux they are part of the main vnode structure. On Solaris and FreeBSD, however, we first perform a `VOP_GETATTR` vnode operation to find them, and then return the appropriate field from the structure that the getattr function fills.

The vnode attribute "name" was more complex to implement, because most kernels do not store file names after the initial name lookup routine translates the name to a vnode. On Linux, implementing the vnode name attribute was simple, because it is part of a standard directory entry structure, `dentry`. On Solaris and FreeBSD, however, we added code to the lookup vnode function that stores the initial file name in the private data of the vnode. That way we can access it as any other vnode attribute, or any other per-vnode attribute added using the `per_vnode` declaration.

We implemented all other fields defined using the `per_vfs` FiST declaration in a similar fashion.

The FiST declarations described in Chapter 4 affect the overall behavior of the generated file system. We implemented the read-only access mode by replacing the call part of every file-system function that modifies state (such as unlink and mkdir) to return the error code "read-only file system." We implemented the fan-in mount style by excluding code that uses the mounted directory's vnode also as the mount point.

The only difficult part of implementing the `ioctl` declaration and its associated functions, fistGetIoctlData and fistSetIoctlData (Section 3.5 and Table 4.2), was finding how to copy data between user space and kernel space. Solaris and FreeBSD use the routines `copyin` and `copyout`; Linux 2.3 uses `copy_from_user` and `copy_to_user`.

The last complex feature we implemented was the `fileformat` FiST declaration and the functions used with it: fistGetFileData and fistSetFileData (Section 4.2). Consider this small code excerpt:

```
fileformat fmt { data structure;}
fistGetFileData(file, fmt, field, out);
```

First, we generate a C data structure named *fmt*. To implement fistGetFileData, we open *file*, read as many bytes from it as the size of the data structure, map these bytes onto a temporary variable of the same data structure type, copy the desired *field* within that data structure into *out*, close the file, and finally return a error/success status value from the function. To improve performance, if fileformat related functions are called several times inside a vnode function, we keep the file they refer to open until the last call that uses it.

Fistgen itself (excluding the templates) is highly portable, and can be compiled on any Unix system. The total number of source lines for fistgen is 4813. Fistgen can process each 1KB of template data in under 0.25 seconds (measured on the same platform used in Section 9.3).

# Chapter 8

# File Systems Developed Using FiST

This chapter describes the design and implementation of several file systems we wrote using FiST. We generally progress from those with a simple FiST design to those with a more complex design. Each file system's design introduces a few more FiST features.

1. **Snoopfs**: is a file system that detects simple unauthorized attempts to access files. We described this file system in detail in Section 3.3.
2. **Cryptfs**: is an encryption file system.
3. **Aclfs**: adds simple access control lists.
4. **Unionfs**: joins the contents of two file systems.

Since we also improved our templates by adding support for SCAs, we are including a few examples of file systems built using this special support. The most significant FiST input file requirement for file systems that change size is the addition of a single declaration in the FiST Declarations section: `filter sca`.

1. **Copyfs**: a baseline file system that copies the data without changing it or its size.
2. **Uuencodefs**: a file system that increases data sizes.
3. **Gzipfs**: a compression file system which generally shrinks data sizes.

These file systems are experimental and intended to illustrate the kinds of file systems that can be written using FiST. We illustrate and discuss only the more important parts of these file-system examples—those that depict key features of FiST. Whenever possible, we mention potential enhancements to our examples. We hope to convince readers of the flexibility and simplicity of writing new file systems using FiST. An additional file system example, Snoopfs, was described earlier in Section 3.3.

## 8.1 Cryptfs

Cryptfs is a strong encryption file system. It uses the Blowfish [68] encryption algorithm in Cipher Feedback (CFB) mode [67]. This algorithm does not change the data size of the input. We used one fixed Initialization Vector (IV) and one 128-bit key per mounted instance of Cryptfs. Cryptfs encrypts both file data and file names. After encrypting file names, Cryptfs also uuencodes them to avoid characters that are illegal in file names. Additional design and important details are available elsewhere [83].

The FiST implementation of Cryptfs shows three additional features: file data encoding, ioctl calls, and per-VFS data. Cryptfs's FiST code uses all four sections of a FiST file. Some of the more important code for Cryptfs is:

```
%{
#include <blowfish.h>
%}
filter data;
filter name;
ioctl:fromuser SETKEY {char ukey[16];};
per_vfs {char key[16];};
%%
%op:ioctl:SETKEY {
  char temp_buf[16];
  if (fistGetIoctlData(SETKEY, ukey, temp_buf)<0)
    fistSetErr(EFAULT);
  else
    BF_set_key(&$vfs.key, 16, temp_buf);
}
%%
unsigned char global_iv[8] = {
  0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10 };
int cryptfs_encode_data(const page_t *in,
                        page_t *out)
{
  int n = 0;    /* blowfish variables */
  unsigned char iv[8];

  fistMemCpy(iv, global_iv, 8);
  BF_cfb64_encrypt(in, out, %pagesize,
                   &($vfs.key), iv, &n,
                   BF_ENCRYPT);
  return %pagesize;
}
...
```

The above example omits the call to decode data and the calls to encode and decode file names because they are similar in behavior to data encoding. For the full code to Cryptfs, refer to Appendix B.1.

Cryptfs defines an ioctl named SETKEY, used to set 128-bit encryption keys. We wrote a simple user-level tool which prompts the user for a passphrase and sends an MD5-hash of it to the kernel using this ioctl. When the SETKEY ioctl is called, Cryptfs stores the (cipher) key in the private VFS data field "key", to be used later.

There are several possible extensions to Cryptfs:

- Storing per-file or per-directory keys in auxiliary files that would otherwise remain hidden from users' view, much the same as Aclfs does (Section 8.2).
- Using several types of encryption algorithms.
- Defining mount flags to select among them.

## 8.2 Aclfs

Aclfs allows an additional UID and GID to share access to a directory as if they were the owner and group of that directory. Aclfs shows three additional features of FiST: disallowing fan-in (more secure), using special purpose auxiliary files, and hiding files from users' view. The FiST code for Aclfs uses the FiST Declarations and FiST Rules sections:

```
fanin no;
ioctl:fromuser SETACL {int u; int g;};
fileformat ACLDATA {int us; int gr;};
%%
%op:ioctl:SETACL {
  if ($0.owner == %uid) {
    int u2, g2;
    if (fistGetIoctlData(SETACL, u, &u2) < 0 ||
        fistGetIoctlData(SETACL, g, &g2) < 0)
      fistSetErr(EFAULT);
    else {
      fistSetFileData(".acl", ACLDATA, us, u2);
      fistSetFileData(".acl", ACLDATA, gr, g2);
    }
  } else
    fistSetErr(EPERM);
}
%op:lookup:postcall {
  int u2, g2;
  if (fistLastErr() == EPERM
      &&
      fistGetFileData(".acl", ACLDATA, us, u2)>=0
      &&
      fistGetFileData(".acl", ACLDATA, gr, g2)>=0
      &&
      (%uid == u2 || %gid == g2))
        fistLookup($dir:1, $name, $1,
                   $dir:1.owner, $dir:1.group);
}
%op:lookup:precall {
  if (fistStrEq($name, ".acl") &&
      $dir.owner != %uid)
    fistReturnErr(ENOENT);
}
%op:readdir:call {
  if (fistStrEq($name, ".acl"))
    fistSkipName($name);
}
```

When looking up a file in a directory, Aclfs first performs the normal access checks (in `lookup`). We insert postcall code after the normal lookup that checks if access to the file was denied and if an additional file named `.acl` exists in that directory. We then read one UID and GID from the `.acl` file. If the effective UID and GID of the current process match those listed in the `.acl` file, we repeat the lookup operation on the originally looked-up file, but using the ownership and group credentials of the *actual* owner of the directory. We must use the owner's credentials, or the lower file system will deny our request.

The `.acl` file itself is modifiable only by the directory's owner. We accomplish this by using a special ioctl. Finally, we hide `.acl` files from anyone but their owner. We insert code in the beginning of lookup that returns the error "no such file" if anyone other than the directory's owner attempted to lookup the ACL file. To complete the hiding of ACL files, we skip listing `.acl` files when reading directories.

Aclfs shows the full set of arguments to the fistLookup routine. In order, the five arguments are: the directory to lookup in, the name to lookup, the vnode to store the newly looked up entry, and the credentials to perform the lookup with (UID and GID, respectively).

There are several possible extensions to this implementation of Aclfs. Instead of using the UID and GID listed in the `.acl` file, it can contain an arbitrarily long list of user and group IDs to allow access to. The `.acl` file may also include sets of permissions to deny access from, perhaps using negative integers to distinguish them from access permissions. The granularity of Aclfs's control can be made finer, say, on a per-file basis; for each file $F$, access permissions can be read from a file `.`$F$`.acl`, if one exists.

## 8.3 Unionfs

Unionfs joins the contents of two file systems similar to the union mounts in BSD-4.4 [52] and Plan 9 [54]. The two lower file systems can be considered two branches of a stackable file system tree. Unionfs shows how to merge the contents of directories in FiST, and how to define behavior on a set of file-system operations. The FiST code for Unionfs uses the FiST Declarations and FiST Rules sections:

```
fanout 2;
%%
%op:lookup:postcall {
  if (fistLastErr() == ENOENT)
    fistSetErr(fistLookup($dir:2, $name));
}
%op:readdir:postcall {
  fistSetErr(fistReaddir($dir:2, NODUPS));
}
%delops:all:postcall {
  fistSetErr(fistOp($2));
}
%writeops:all:call {
  fistSetErr(fistOp($1));
}
```

Normal lookup will try the first lower file-system branch ($1). We add code to lookup in the second branch ($2) if the first lookup did not find the file. If a file exists in both lower file systems, Unionfs will use the one from the first branch. Normal directory reading is augmented to include the contents of the second branch, but setting a flag to eliminate duplicates; that way files that exist in both lower file systems are listed only once. Since files may exist in both branches, they must be removed (unlink, rmdir, and rename) from all branches. Finally we declare that all writing operations should perform their respective operations only on the first branch; this means that new files are created in the first branch where they will be found first by subsequent lookups.

There are several other issues involving file-system semantics and especially concerning error propagation and partial failures, but these are beyond the scope of this dissertation. Extensions to our Unionfs include larger fan-outs, masking the existence of a file in $2 if it was removed from $1, and ioctls or mount options to decide the order of lookups and writing operations on the individual file-system branches.

## 8.4 Copyfs

Copyfs this file system simply copies its input bytes to its output, without changing data sizes. We wrote this simple file system to serve as a base file system to compare to gzipfs and uuencodefs. Copyfs exercises all of the index management algorithms and other SCA support without the costs of encoding or decoding pages.

The full FiST code for Copyfs is listed in Appendix B.4. This FiST file shows how simple it is to declare an SCA file system and how to write code that simply copies its input to its output. Most of the code is in the encoding and decoding functions. A brief excerpt of the code is shown below:

```
filter sca;
filter data;
%%
%%
int
copyfs_encode_data(char *hidden_pages_data,
                   char *page_data,
                   unsigned to,
                   inode_t *inode,
                   vfs_t *vfs,
                   void **opaque)
{
  fistMemCpy(hidden_pages_data, page_data, to);
  return(to);
}
```

Here, we declare the file system to manipulate data pages and further to turn on SCA support. Our data-pages–encoding function simply copies the input data pages to their output and returns the number of bytes encoded. The decoding function proceeds similarly to the encoding function.

## 8.5 UUencodefs

Uuencodefs this file system is intended to illustrate an algorithm that increased the data size. This simple algorithm converts every 3-byte sequence into a 4-byte sequence. Uuencode produces 4 bytes that can have at most 64 values each, starting at the ASCII character for space ($20_h$). We chose this algorithm over encryption algorithms that run in Electronic Codebook mode (ECB) or Cipher Block Chaining mode because they do not increase the data size by much [67]. With uuencodefs we were able to increase the data size of the output by one-third.

The full FiST code for UUencodefs is listed in Appendix B.3. This FiST file also shows how to declare an SCA file system, but also how to write code that expands the input stream size into its output buffer. An excerpt of the code is shown below:

```
filter sca;
filter data;
%%
%%
int
uuencodefs_encode_data(char *hidden_pages_data,
                       char *page_data,
                       int *need_to_call,
                       unsigned to,
                       inode_t *inode,
                       vfs_t *vfs,
                       void **opaque)
{
    int in_bytes_left;
    int out_bytes_left = PAGE_CACHE_SIZE;
    int startpt;
    unsigned char A, B, C;
    int bytes_written = 0;

    startpt = (int)*opaque;
    in_bytes_left = to - startpt;

    while ((in_bytes_left > 0) && (out_bytes_left >= 4)) {
        A = page_data[startpt];

        switch(in_bytes_left) {
        case 1:
            B = 0;
            C = 0;
            in_bytes_left--;
            startpt += 1;
            break;
        case 2:
            B = page_data[startpt + 1];
            C = 0;
            startpt += 2;
            in_bytes_left -= 2;
            break;
        default:
            B = page_data[startpt + 1];
            C = page_data[startpt + 2];
            startpt += 3;
            in_bytes_left -= 3;
            break;
        }
```

```
        hidden_pages_data[bytes_written] = 0x20 + (( A >> 2 ) & 0x3F);
        out_bytes_left--; bytes_written++;
        hidden_pages_data[bytes_written] = 0x20 +
            ((( A << 4 ) | ((B >> 4) & 0xF)) & 0x3F);
        out_bytes_left--; bytes_written++;
        hidden_pages_data[bytes_written] = 0x20 +
            ((( B << 2 ) | ((C >> 6) & 0x3)) & 0x3F);
        out_bytes_left--; bytes_written++;
        hidden_pages_data[bytes_written] = 0x20 + ((C) & 0x3F);
        out_bytes_left--; bytes_written++;
    }

    if (in_bytes_left > 0)
        *opaque = (void *)startpt;
    else
        *need_to_call = 0;

    return bytes_written;
}
```

Here, we declare the file system to manipulate data pages and further to turn on SCA support. Our data-pages–encoding function spends most of its time performing the uuencode function: converting every 3-byte sequence of characters to four bytes. The function returns the number of bytes it actually produced. Since this function expands data size, it may not have enough space left in the output buffer (`page_data`) to process all of the input data. If that is the case, then it sets an opaque pointer to the byte index within the input data where it stopped encoding. The caller in the main SCA code will recognize this and will use the remaining bytes in the next invocation of the encode function.

## 8.6 Gzipfs

Gzipfs is a compression file system using the Deflate algorithm [18] from the zlib-1.1.3 package [20, 24], the same algorithm used by GNU zip (gzip)) [19, 23]. This file system is intended to demonstrate an algorithm that (usually) reduces data size.

The full FiST code for Gzipfs is listed in Appendix B.2. This FiST file again shows how simple it is to declare an SCA file system. Here, however, we show how to write code that compresses its input into its output buffer. A brief excerpt of the code is shown below:

```
filter sca;
filter data;
%%
%%
int
gzipfs_encode_data(char *hidden_pages_data,
                   char *page_data,
                   int *need_to_call,
                   unsigned to,
```

```
                   inode_t *inode,
                   vfs_t *vfs,
                   void **opaque)
{
    z_stream *zptr;
    int rc, err;

    if (*opaque != NULL) {
        zptr = (z_stream *) *opaque;
        zptr->next_out = hidden_pages_data;
        zptr->avail_out = PAGE_CACHE_SIZE;
        zptr->total_out = 0;
    } else {
        zptr = kmalloc(sizeof(z_stream), GFP_KERNEL);
        if (zptr == NULL) {
            err = -ENOMEM;
            goto out;
        }
        zptr->zalloc = (alloc_func)0;
        zptr->zfree = (free_func)0;
        zptr->opaque = (voidpf)0;
        zptr->next_in = page_data;
        zptr->avail_in = to;
        zptr->next_out = hidden_pages_data;
        zptr->avail_out = PAGE_CACHE_SIZE;

        /*
         * First arg is a stream object
         * Second arg is compression level (0-9)
         */
        rc = deflateInit(zptr, GZIPFS_DEFLATE_LEVEL);
        if (rc != Z_OK ) {
            printk("inflateInit error %d: Abort",rc);
            /* This is bad. Lack of memory is the usual cause */
            err = -ENOMEM;
            goto out;
        }

        while ((zptr->avail_out > 0) &&
               (zptr->avail_in >  0)) { /* While we're not finished */
            rc = deflate(zptr, Z_FULL_FLUSH); /* Do a deflate */
            if ((rc != Z_OK) && (rc != Z_STREAM_END)) {
                printk("Compression error! rc=%d",rc);
                err = -EIO;
                goto out;
```

```
                }
            }
        }

        rc = deflate(zptr, Z_FINISH);

        if (rc == Z_STREAM_END) {
            deflateEnd(zptr);
            kfree(zptr);
            *need_to_call = 0;
        } else if (rc == Z_BUF_ERROR || rc == Z_OK) {
            *opaque = zptr;
        } else
            printk("encode_buffers error: rc=%d", rc);

        err = zptr->total_out;      /* Return encoded bytes */

    out:
        return(err);
}
```

Similarly to Uuencodefs, we declare the use of SCA support. The encode function is more complex for two reasons. First, it has to handle the more complex API to the zlib function when calling the Deflate algorithm's functions. This is seen in the calls to the `deflate()` and `deflatInit()` functions with various zlib-specific arguments. Second, compression typically results in smaller data streams, but could also result in larger ones (when trying to compress already-compressed data). So this encoding function must handle the cases when the output buffer may not be large enough, as well as when the input buffer was completely consumed and unused space may be left in the output buffer.

# Chapter 9

# Evaluation

FiST is a new high-level programming language. As such, we are interested in typical evaluation of languages: how much code one has to write in the high-level language to create realistic applications, and how much development time one saves by using the new high-level language. To evaluate these two criteria, we compare the process of writing several file systems using FiST, using other stacking mechanisms, and writing them from scratch in a low-level language (C). We show that code sizes and development times are improved significantly.

The primary focus of the FiST work was not performance, but rather the simplicity that the language provides and the time savings it affords to developers. Nevertheless, performance is still an important criteria. We ran large scale realistic benchmarks as well as micro-benchmarks on all of the file systems we developed using FiST: Snoopfs, Cryptfs, Aclfs, Unionfs, Wrapfs, Copyfs, Uuencodefs, and Gzipfs. We show that the basic performance overhead of FiST file systems is very low.

Since our example file systems are based on our templates, we also compared these file systems to the base stacking templates. This allowed us to identify the differences and overheads that each component or layer brings.

FiST is ported to three different platforms: Linux 2.3, Solaris 2.6, and FreeBSD 3.3. We evaluate each of the three criteria (code size, development time, and performance), for each file system we developed, and on each of the three platforms.

Later in this chapter, we evaluate separately the performance of our SCAs in detail. This support was added to FiST specifically to handle size-changing file systems in a manner that performs well. Therefore, most of our evaluation of SCAs centers around the performance overheads that they add, illustrated through many experiments. We show that these performance overheads are small.

## 9.1 Code Size

Code size is one measure of the development effort necessary for a file system. To demonstrate the savings in code size achieved using FiST, we compare the number of lines of code that need to be written to implement the four example file systems in FiST versus three other implementation approaches: writing C code using a stand-alone version of Basefs, writing C code using Wrapfs, and writing the file systems from scratch as kernel modules using C. In particular, we first wrote all four of the example file systems from scratch before writing them using FiST. For these example file systems, the C code generated from FiST was identical in size (modulo white-spaces and comments) to the hand-written code. We chose to include results for both Basefs and Wrapfs because the latter was released last year, and includes code that makes writing some file systems easier with Wrapfs than Basefs directly.

When counting lines of code, we excluded comments, empty lines, and %% separators. For Cryptfs we excluded

627 lines of C code of the Blowfish encryption algorithm, since we did not write it. When counting lines of code for implementing the example file systems using the Basefs and Wrapfs stackable templates, we exclude code that is part of the templates and only count code that is specific to the given example file system. We then averaged the code sizes for the three platforms we implemented the file systems on: Linux 2.3, Solaris 2.6, and FreeBSD 3.3. These results are shown in Figure 9.1. For reference, we include the code sizes of Basefs and Wrapfs and also show the number of lines of code required to implement Wrapfs in FiST and Basefs.

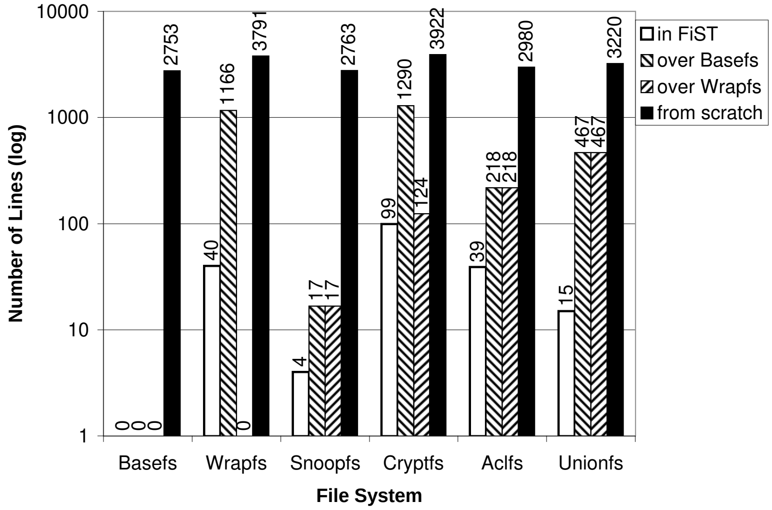



Figure 9.1: Average code size for various file systems when written in FiST, written given the Basefs or Wrapfs templates, and written from scratch in C.

Figure 9.1 shows large reductions in code size when comparing FiST versus code hand-written from scratch—generally writing tens of lines instead of thousands. We also include results for the two templates. Size reductions for the four example file systems range from a factor of 40 to 691, with an average of 255. We focus though on the comparison of FiST versus stackable template systems. As Wrapfs represents the most conservative comparison, the figure shows for each file system the additional number of lines of code written using Wrapfs. The smallest average code size reduction in using FiST versus Wrapfs or Basefs across all four file systems ranges from a factor of 1.3 to 31.1; the average reduction rate is 10.5.

Figure 9.1 suggests two size reduction classes. First, moderate (5–6 times) savings are achieved for Snoopfs, Cryptfs, and Aclfs. The reason for this is that some lines of FiST code for these file systems produce ten or more lines of C code, while others result in almost a one-to-one translation in terms of number of lines.

Second, the largest savings appeared for Unionfs, a factor of 28–33 times. The reason for this is that fan-out file systems produce C code that affects all vnode operations; each vnode operation must handle more than one lower vnode. This additional code was not part of the original Wrapfs implementation, and it is not used unless fan-outs of two or more are defined (to save memory and improve performance). If we exclude the code to handle fan-outs, Unionfs's added C code is still over 100 lines producing savings of a factor of 7–10. FreeBSD's Unionfs is 4863 lines long, which is 50% larger than our Unionfs (3232 lines). FreeBSD's Unionfs is 2221 lines longer than their Nullfs, while ours is only 481 lines longer than our Basefs.[1]

Figure 9.1 shows the average code sizes over all three platforms. The savings gained by FiST are multiplied with each port. If we sum up the savings for the above three platforms, we reach reduction factors ranging from 4 to over 100

[1]Unfortunately, the stacking infrastructure in FreeBSD is currently broken, so we were unable to compare the performance of our stacking to FreeBSD's.

times when comparing FiST to code written using the templates. This aggregated reduction factor exceeds 750 times when comparing FiST to C code written from scratch. The more ports of Basefs exist, the better these cumulative savings would be.

## 9.2 Development Time

Estimating the time to develop kernel software is very difficult. Developers' experience can affect this time significantly, and this time is generally reduced with each port. In this section we report our own personal experiences given these file-system examples and the three platforms we worked with; these figures do not represent a controlled study. Figure 9.2 shows the number of days we spent developing various file systems and porting them to three different platforms.

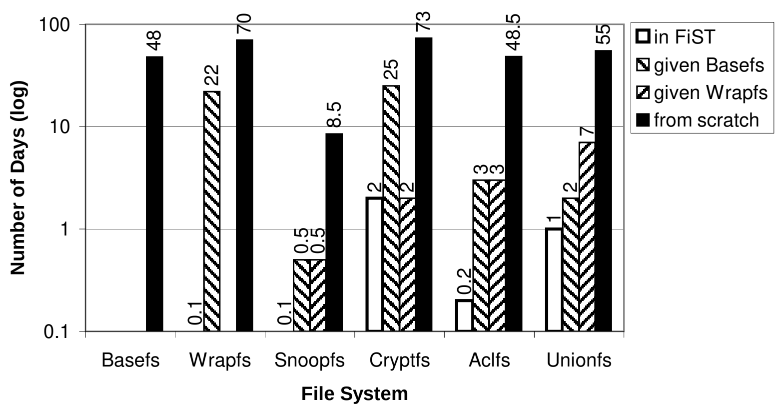



Figure 9.2: Average estimated reduction in development time

We estimated the incremental time spent designing, developing, and debugging each file system, assuming 8 hour work days, and using our source commit logs and change logs. We estimated the time it took us to develop Wrapfs, Basefs, and the example file systems. Then we measured the time it took us to develop each of these file systems using the FiST language.

For most file systems, incremental time savings are a factor of 5–15 because hand writing C code for each platform can be time consuming, while FiST provides this as part of the base templates and the additional library code that comes with Basefs. For Cryptfs, however, there are no time savings per platform, because the vast majority of the code for Cryptfs is in implementing the four encoding and decoding functions, which are implemented in C code in the Additional C Code section of the FiST file; the rest of the support for Cryptfs is already in Wrapfs.

The average per platform reduction in development time across the four file systems is a factor of seven in using FiST versus the Wrapfs templates. If we assume that development time correlates directly to productivity, we can corroborate our results with Brooks's report that high-level languages are responsible for at least a factor of five in improved productivity [10].

An additional metric of productivity is comparing the number of lines of C code developed for each man-day, given the templates. The average number of lines of code we wrote per man-day was 80. One user of our Wrapfs templates had used them to create a new migration file system called mfs[2]. The average number of lines of code he wrote per man-day was 68. The difference between his rate of productivity and ours is only 20%, which can be explained because we are more experienced in writing file systems than he is.

[2] http://www-internal.alphanet.ch/~schaefer/mfs.html

The most obvious savings in development time come when taking into account multiple platforms. Then it is clearer that each additional platform increases the savings factor of FiST versus other methods by yet more.

## 9.3 Performance

To evaluate the performance of file systems written using FiST, we tested each of the example file systems by mounting it on top of a disk-based native file system and running benchmarks in the mounted file system. We conducted measurements for Linux 2.3, Solaris 2.6, and FreeBSD 3.3. The native file systems used were EXT2, UFS, and FFS, respectively. We measured the performance of our file systems by building a large package: am-utils-6.0, which contains about 50,000 lines of C code in several dozen small files and builds eight binaries; the build process contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file-system operations. Each benchmark was run once to warm up the cache for executables, libraries, and header files which are *outside* the tested file system; the results of this first test were discarded. Afterwards, we took 10 new measurements and averaged them. In between each test, we unmounted the tested file system and the one below it, and then remounted them; this ensured that we started each test on a cold cache for that file system. The standard deviations for our measurements were less than 2% of the mean. We ran all tests on the same machine: a P5/90, with 64MB RAM and a Quantum Fireball 4.35GB IDE hard disk.

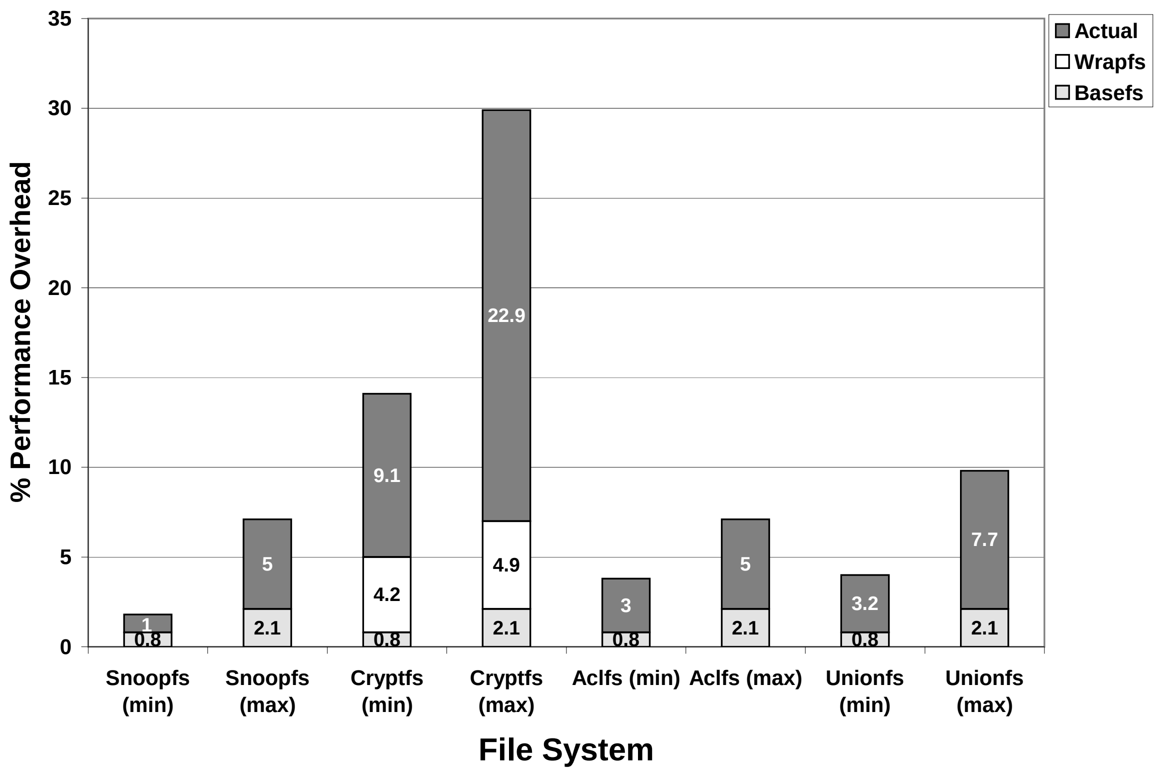



Figure 9.3: Performance overhead of various file systems for the large compile benchmark, across three operating systems.

Figure 9.3 shows the performance overhead of each file system compared to the one it was based on. The intent of these figures is two-fold: (1) to show that the basic stacking overhead is small, and (2) to show the performance benefits of conditionally including code for manipulating file names and file data in Basefs. Wrapfs refers to Basefs with code for manipulating file names and file data.

The most important performance metric is the basic overhead imposed by our templates. The overhead of Basefs over the file systems it mounts on is just 0.8–2.1%. This minimum overhead is below the 3–10% degradation previously reported for null-layer stacking [31, 69]. In addition, the overhead of the example file systems due to new file-system functionality is greater than the basic stacking overhead imposed by our templates in all cases, even for very simple file systems. With regard to performance, developers who extend file-system functionality using FiST primarily need to be concerned with the performance cost of new file-system functionality as opposed to the cost of the FiST stacking infrastructure. For instance, the overhead of Cryptfs is the largest of all the file systems shown due to the cost of the Blowfish cipher. Note that the performance of individual file systems can vary greatly depending on the operating system in question.

Figure 9.3 also shows the benefits of having FiST customize the generated file-system infrastructure based on the file system functionality required. It shows that the overhead of the added code for manipulating file names and file data is 4.2–4.9% over Basefs: this includes copying and caching data pages and file names. This added overhead is not incurred in Basefs unless the file systems derived from it require file data or file name manipulations. While Cryptfs requires Wrapfs functionality, Snoopfs, Aclfs, and Unionfs do not. Compared to a stackable file system such as Wrapfs, FiST's ability to conditionally include file system infrastructure code saves an additional 4–5% of performance overhead for Snoopfs, Aclfs, and Unionfs.

We also performed several micro-benchmarks which included a series of recursive copies (cp –r), recursive removals (rm –rf), recursive find, and "find-grep" (find /mnt –print | xargs grep *pattern*) using the same file set used for the large compile. The focus of this dissertation is not on performance, but on savings in code size and development time. Since these micro-benchmarks confirmed our previous good results, we do not repeat them here [84]; instead, we list other micro-benchmarks next, in Section 9.3.1.

Finally, since we did not change the VFS, and all of our stacking work is in the templates, there is no overhead on the rest of the system; performance of native file systems (NFS, FFS, etc.) is unaffected when our stacking is not used.

### 9.3.1 Micro-Benchmarks

We ran a set of benchmarks intended to illustrate the differences in performance between user-level and kernel-level stackable file systems. We chose to run these tests on our Cryptfs file system, because we could compare it to CFS, a user-level encryption file system based on a user-level NFS server [8]. In addition, we focused on the specific differences in performance on different operating systems, and the performance of the most common operations: reading and writing small and large files.

For most of our tests, we included figures for a native disk-based file system because disk hardware performance can be a significant factor. This number should be considered the base to which other file systems compare to. Since Cryptfs is a stackable file system and is based on Wrapfs, we also included figures for Wrapfs and for Basefs, to be used as a base for evaluating the cost of stacking. When using Basefs, Wrapfs, or Cryptfs, we mounted them over a local disk-based file system. CFS is based on NFS, so we 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.

Since CFS is implemented as a user-level NFS file server, we expected that CFS would run slower due to the number of additional context switches that must take place when a user-level file server is called by the kernel to satisfy a user process request, and due to NFS V.2 protocol overheads such as synchronous writing.

For this set of micro-benchmarks, we concentrated on the x86 platform since it was common to all ports. We ran tests that represent common operations in file systems: opening files and reading or writing them. In the first test we wrote 1024 different new files of 8KB size each. The second test wrote 8 new files of 1MB size each. Then we read one 8KB file 1024 times, and one 1MB file 8 times. The intent of these tests was that the total amount of data read and written would be the same. Finally we included measurements for reading a directory with 1024 entries repeatedly for 100 times; while that is a less popular operation, cryptographic file systems encrypt file names and thus can significantly affect the performance of reading a directory. All times reported are elapsed, in seconds, and measured on an otherwise quiet system.

To avoid repetition, we report full results for Linux only, as seen in Table 9.1. For Solaris and FreeBSD, we only included figures for the file systems relevant to comparing Cryptfs to CFS; these are reported in Tables 9.2 and 9.3, respectively.

| File System | Writes | | Reads | | 1024× readdir |
|---|---|---|---|---|---|
| | 1024× 8KB | 8× 1MB | 1024× 8KB | 8× 1MB | |
| ext2fs | 3.33 | 3.06 | 0.17 | 0.34 | 1.49 |
| basefs | 3.40 | 3.35 | 0.19 | 0.34 | 1.51 |
| wrapfs | 3.48 | 3.58 | 0.26 | 0.34 | 1.57 |
| cryptfs | 9.27 | 8.33 | 0.32 | 0.34 | 3.18 |
| nfs | 26.85 | 17.67 | 0.47 | 3.17 | 16.27 |
| cfs | 101.90 | 50.84 | 0.89 | 8.77 | 118.35 |

Table 9.1: Linux x86 Times for Repeated Calls (Sec)

Concentrating on Linux (Table 9.1) first, we see that Basefs adds a small overhead over the native disk-based file system, and wrapfs adds another overhead due to performing data copies. The difference between Cryptfs and Wrapfs is that of encryption only. Writing files is 6–12 times faster in Cryptfs than in CFS. The main reasons for this are the additional context switches that must take place in user-level file servers, and that NFS V.2 writes are synchronous. When reading files, caching and memory sizes come into play more than the file system in question. That is why the difference in file reading performance for all file systems is not as significant as when writing files.

Reading a directory with 1024 files one hundred times is 10–37 times faster in Cryptfs than in CFS, mostly due to context switches. When Cryptfs is mounted on top of ext2fs, it slows performance of these measured operations 2–3 times. But since these are fast to begin with, users hardly notice the difference; in practice overall slowness is smaller, as reported in Figure 9.3.

| File System | Writes | | Reads | | 1024× readdir |
|---|---|---|---|---|---|
| | 1024× 8KB | 8× 1MB | 1024× 8KB | 8× 1MB | |
| ufs | 4.88 | 3.98 | 0.48 | 0.38 | 0.52 |
| cryptfs | 63.95 | 11.10 | 10.72 | 7.23 | 7.14 |
| nfs | 54.17 | 18.82 | 1.69 | 0.38 | 0.28 |
| cfs | 140.78 | 140.98 | 27.68 | 24.57 | 18.02 |

Table 9.2: Solaris x86 Times for Repeated Calls (Sec)

| File System | Writes | | Reads | | 1024× readdir |
|---|---|---|---|---|---|
| | 1024× 8KB | 8× 1MB | 1024× 8KB | 8× 1MB | |
| ffs | 12.55 | 6.04 | 1.00 | 1.01 | 0.15 |
| cryptfs | 56.59 | 22.55 | 1.04 | 1.05 | 0.29 |
| nfs | 55.69 | 21.63 | 1.31 | 1.09 | 0.33 |
| cfs | 99.34 | 31.80 | 2.09 | 4.80 | 0.87 |

Table 9.3: FreeBSD x86 Times for Repeated Calls (Sec)

Native file systems in Linux perform their operations asynchronously, while Solaris and FreeBSD do so synchronously. That is why the performance improvement of Cryptfs over CFS for Solaris and FreeBSD is smaller; when vnode operations that perform writing are passed from Cryptfs to the lower-level file system, they must be completed before returning to the caller. For the operations measured, Cryptfs improves performance by anywhere from 1.5 to 2 times, with the exception of writing large files on Solaris, where performance is improved by more than an order of magnitude.

## 9.4 Size-Changing File Systems

To evaluate fast indexing in a real world operating system environment, we built several SCA stackable file systems based on fast indexing. We then conducted extensive measurements in Linux comparing them against non-SCA file systems on a variety of file system workloads. In this section we discuss the experiments we performed on these systems to (1) show overall performance on general-purpose file system workloads, (2) determine the performance of individual common file operations and related optimizations, and (3) compare the efficiency of SCAs in stackable file systems to equivalent user-level tools. Section 9.4.1 describes the SCA file systems we built and our experimental design. Section 9.4.2 describes the file system workloads we used for our measurements. Sections 9.4.3 to 9.4.6 present our experimental results.

### 9.4.1 Experimental Design

We ran our experiments on five file systems. We built three SCA file systems and compared their performance to two non-SCA file systems. The three SCA file systems we built were:

1. **Copyfs**: this file system simply copies its input bytes to its output without changing data sizes. Copyfs exercises all of the index-management algorithms and other SCA support without the cost of encoding or decoding pages.
2. **Uuencodefs**: this is a file system that stores files in uuencoded format and uudecodes files when they are read. It is intended to illustrate an algorithm that increases the data size. This simple algorithm converts every 3-byte sequence into a 4-byte sequence. Uuencode produces 4 bytes that can have at most 64 values each, starting at the ASCII character for space ($20_h$). We chose this algorithm because it is simple and yet increases data size significantly (by one third).
3. **Gzipfs**: this is a compression file system using the Deflate algorithm [18] from the zlib-1.1.3 package [24]. This algorithm is used by GNU zip (`gzip`) [23]. This file system is intended to demonstrate an algorithm that (usually) reduces data size.

The two non-SCA file systems we used were Ext2fs, the native disk-based file system most commonly used in Linux, and Wrapfs, a stackable null-layer file system we trivially generated using FiST [86]. Ext2fs provides a measure of base file system performance without any stacking or SCA overhead. Wrapfs simply copies the data of files between layers but does not include SCA support. By comparing Wrapfs to Ext2fs, we can measure the overhead of stacking and copying data without fast indexing and without changing its content or size. Copyfs copies data like Wrapfs but uses all of the SCA support. By comparing Copyfs to Wrapfs, we can measure the overhead of basic SCA support. By comparing Uuencodefs to Copyfs, we can measure the overhead of an SCA algorithm incorporated into the file system that increases data size. Similarly, by comparing Gzipfs to Copyfs, we can measure the overhead of a compression file system that reduces data size.

One of the primary optimizations in this work is fast tails as described in Section 6.3.1.1. For all of the SCA file systems, we ran all of our tests first without fail-tails support enabled and then with it. We reported results for both whenever fast tails made a difference.

All experiments were conducted on four equivalent 433Mhz Intel Celeron machines with 128MB of RAM and a Quantum Fireball lct10 9.8GB IDE disk drive. We installed a Linux 2.3.99-pre3 kernel on each machine. Each of the four stackable file systems we tested was mounted on top of an Ext2 file system. For each benchmark, we only read, wrote, or compiled the test files in the file system being tested. All other user utilities, compilers, headers, and libraries resided outside the tested file system.

Unless otherwise noted, all tests were run with a cold cache. To ensure that we used a cold cache for each test, we unmounted all file systems which participated in the given test after the test completed and mounted the file systems again before running the next iteration of the test. We verified that unmounting a file system indeed flushes and discards all possible cached information about that file system. In one benchmark we report the warm cache performance, to show the effectiveness of our code's interaction with the page and attribute caches.

We ran all of our experiments 10 times on an otherwise quiet system. We measured the standard deviations in our experiments and found them to be small, less than 1% for most micro-benchmarks described in Section 9.4.2. We report deviations which exceeded 1% with their relevant benchmarks.

### 9.4.2 File System Benchmarks

We measured the performance of the five file systems on a variety of file system workloads. For our workloads, we used five file system benchmarks: two general-purpose benchmarks for measuring overall file system performance, and three micro-benchmarks for measuring the performance of common file operations that may be impacted by fast indexing. We also used the micro-benchmarks to compare the efficiency of SCAs in stackable file systems to equivalent user-level tools.

#### 9.4.2.1 General-Purpose Benchmarks

**Am-utils**: The first benchmark we used to measure overall file system performance was am-utils (The Berkeley Automounter) [3]. This benchmark configures and compiles the large am-utils software package inside a given file system. We used am-utils-6.0.4: it contains over 50,000 lines of C code in 960 files. The build process begins by running several hundred small configuration tests intended to detect system features. It then builds a shared library, about ten binaries, four scripts, and documentation: a total of 265 additional files. Overall this benchmark contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations such as unlink, mkdir, and symlink. During the linking phase, several large binaries are linked by GNU `ld`.

The am-utils benchmark is the only test that we also ran with a warm cache. Our stackable file systems cache decoded and encoded pages whenever possible, to improve performance. While normal file system benchmarks are done using a cold cache, we also felt that there is value in showing what effect our caching has on performance. This is because user level SCA tools rarely benefit from page caching, while file systems are designed to perform better with warm caches; this is what users will experience in practice.

**Bonnie**: The second benchmark we used to measure overall file system performance was Bonnie [17], a file system test that intensely exercises file data reading and writing, both sequential and random. [3] Bonnie is a less general benchmark than am-utils. Bonnie has three phases. First, it creates a file of a given size by writing it one character at a time, then one block at a time, and then it rewrites the same file 1024 bytes at a time. Second, Bonnie writes the file one character at a time, then a block at a time; this can be used to exercise the file system cache, since cached pages have to be invalidated as they get overwritten. Third, Bonnie forks 3 processes that each perform 4000 random `lseeks` in the file, and read one block; in 10% of those seeks, Bonnie also writes the block with random data. This last phase exercises the file system quite intensively, and especially the code that performs writes in the middle of files.

For our experiments, we ran Bonnie using files of increasing sizes, from 1MB and doubling in size up to 128MB. The last size is important because it matched the available memory on our systems. Running Bonnie on a file that large is

[3] We also tested using a third benchmark, the Modified Andrew Benchmark (MAB). MAB consists of five phase: making directories, copying files, recursive listing, recursive scanning of files, and compilation. MAB was designed at a time when hardware was much slower and resources scarce. On our hardware, MAB completed in under 10 seconds of elapsed time, with little variance among different tests. We therefore opted not to use this simple compile benchmark.

important, especially in a stackable setting where pages are cached in both layers, because the page cache should not be able to hold the complete file in memory.

#### 9.4.2.2 Micro-Benchmarks

**File-copy**: The first micro-benchmark we used was designed to measure file system performance on typical bulk file writes. This benchmark copies files of different sizes into the file system being tested. Each file is copied just once. Because file system performance can be affected by the size of the file, we exponentially varied the sizes of the files we ran these tests on—from 0 bytes all the way to 32MB files.

**File-append**: The second micro-benchmark we used was designed to measure file system performance on file appends. It was useful for evaluating the effectiveness of our fast tails code. This benchmark read in large files of different types and used their bytes to append to a newly created file. New files are created by appending to them a fixed but growing number of bytes. The benchmark appended bytes in three different sizes: 10 bytes representing a relatively small append; 100 bytes representing a typical size for a log entry on a Web server or syslog daemon; and 1000 bytes, representing a relatively large append unit. We did not not try to append more than 4KB because that is the boundary where fast appended bytes get encoded. Because file system performance can be affected by the size of the file, we exponentially varied the sizes of the files we ran these tests on—from 0 bytes all the way to 32MB files.

Compression algorithms such as used in Gzipfs behave differently based on the input they are given. To account for this in evaluating the append performance of Gzipfs, we ran the file-append benchmark on four types of data files, ranging from easy to compress to difficult to compress:

1. A file containing the character "`a`" repeatedly should compress really well.
2. A file containing English text, actually written by users, collected from our Usenet News server. We expected this file to compress well.
3. A file containing a concatenation of many different binaries we located on the same host system, such as those found in `/usr/bin` and `/usr/X11R6/bin`. This file should be more difficult to compress because it contains fewer patterns useful for compression algorithms.
4. A file containing previously compressed data. We took this data from Microsoft NT's Service Pack 6 (`sp6i386.exe`) which is a self-unarchiving large compressed executable. We expect this file to be difficult to compress.

**File-attributes**: The third micro-benchmark we used was designed to measure file system performance in getting file attributes. This benchmark performs a recursive listing (`ls -lRF`) on a freshly unpacked and built am-utils benchmark file set, consisting of 1225 files. With our SCA support, the size of the original file is now stored in the index file, not in the inode of the encoded data file. Finding this size requires reading an additional inode of the index file and then reading its data. This micro-benchmark measures the additional overhead that results from also having to read the index file.

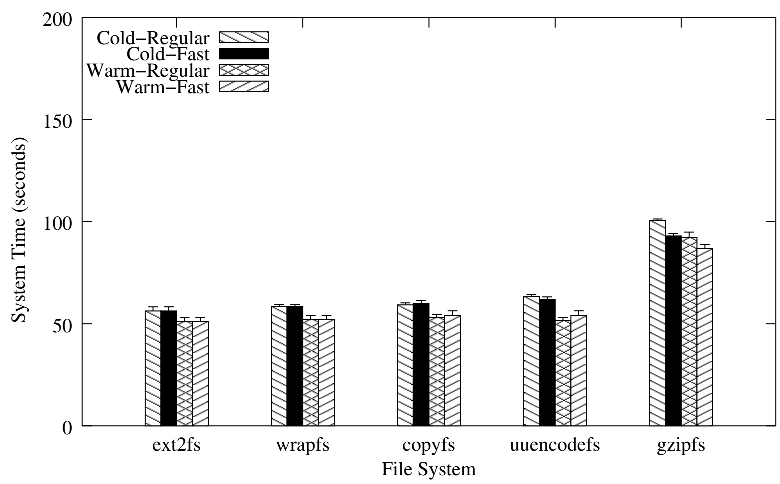
#### 9.4.2.3 File System vs. User-Level Tool Benchmarks

To compare the SCAs in our stackable file systems versus user-level tools, we used the file-copy micro-benchmark to compare the performance of the two stackable file systems with real SCAs, Gzipfs and Uuencodefs, against their equivalent user-level tools, `gzip` [23] and `uuencode`, respectively. In particular, the same Deflate algorithm and compression level (9) was used for both Gzipfs and `gzip`. In comparing Gzipfs and `gzip`, we measured both the compression time and the resulting space savings. Because the performance of compression algorithms depends on the type of input, we compared Gzipfs to `gzip` using the file-copy micro-benchmark on all four of the different file types discussed in Section 9.4.2.2.

### 9.4.3 General-Purpose Benchmark Results

#### 9.4.3.1 Am-Utils

Figure 9.4 summarizes the results of the am-utils benchmark. We report both system and elapsed times. The top part of Figure 9.4 shows system times spent by this benchmark. This is useful to isolate the total effect on the CPU alone, since SCA-based file systems change data size and thus change the amount of disk I/O performed. Wrapfs adds 14.4% overhead over Ext2, because of the need to copy data pages between layers. Copyfs adds only 1.3% overhead over Wrapfs; this shows that our index file handling is fast. Compared to Copyfs, Uuencodefs adds 7% overhead and Gzipfs adds 69.9%. These are the costs of the respective SCAs in use and are unavoidable—whether running in the kernel or user-level.



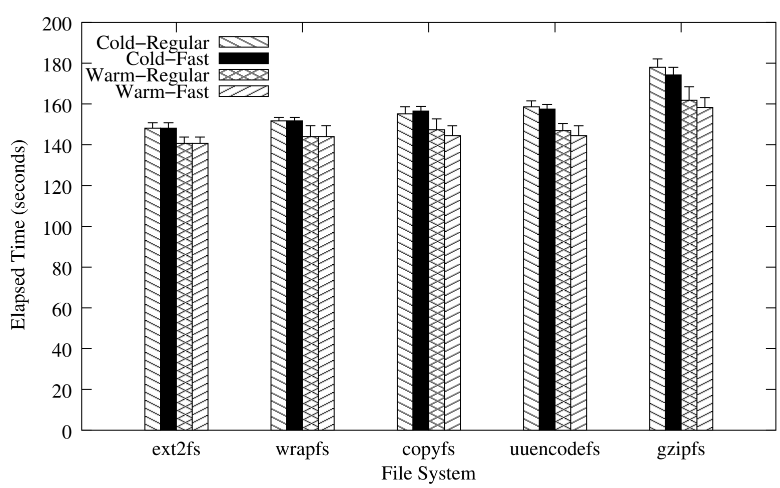




Figure 9.4: The Am-utils large-compile benchmark. Elapsed times shown on top and system times shown on bottom. The standard deviations for this benchmark were less than 3% of the mean.

The total size of an unencoded build of am-utils is 22.9MB; a Uuencoded build is one-third larger; Gzipfs reduces this size by a factor of 2.66 to 8.6MB. So while Uuencodefs increases disk I/O, it does not translate to a lot of additional system time because the Uuencode algorithm is trivial. Gzipfs, while decreasing disk I/O, however, is a costlier algorithm than Uuencode. That's why Gzipfs's system time overhead is greater overall than Uuencodefs's. The additional disk I/O performed by Copyfs is small and relative to the size of the index file.

The bottom part of Figure 9.4 shows elapsed times for this benchmark. These figures are the closest to what users will see in practice. Elapsed times factor in increased CPU times the more expensive the SCA is, as well as changes in I/O that a given file system performs: I/O for index file, increased I/O for Uuencodefs, and decreased I/O for Gzipfs.

On average, the cost of data copying without size-changing (Wrapfs compared to Ext2fs) is an additional 2.4%. SCA support (Copyfs over Wrapfs) adds another 2.3% overhead. The Uuencode algorithm is simple and adds only 2.2% additional overhead over Copyfs. Gzipfs, however, uses a more expensive algorithm (Deflate) [18], and it adds 14.7% overhead over Copyfs. Note that the elapsed-time overhead for Gzipfs is smaller than its CPU overhead (almost 70%) because whereas the Deflate algorithm is expensive, Gzipfs is able to win back some of that overhead by its I/O savings.

Using a warm cache improves performance by 5–10%. Using fast tails improves performance by at most 2%. The code that is enabled by fast tails must check, for each read or write operation, if we are at the end of the file, if a fast tail already exists, and if a fast tail is large enough that it should be encoded and a new fast tail started. This code has a small overhead of its own. For file systems that do not need fast tails (e.g., Copyfs), fast tails add an overhead of 1%. We determined that fast tails is an option best used for expensive SCAs where many small appends are occurring, a conclusion demonstrated more visibly in Section 9.4.4.2.

#### 9.4.3.2 Bonnie

Figure 9.5 shows the results of running Bonnie on the five file systems. Since Bonnie exercises data reading and writing heavily, we expect it to be affected by the SCA in use. This is confirmed in Figure 9.5. Over all runs in this benchmark, Wrapfs has an average overhead of 20% above Ext2fs, ranging from 2–73% for the given files. Copyfs only adds an additional 8% average overhead over Wrapfs. Uuencodefs adds an overhead over Copyfs that ranges from 5% to 73% for large files. Gzipfs, with its expensive SCA, adds an overhead over Copyfs that ranges from 22% to 418% on the large 128MB test file.

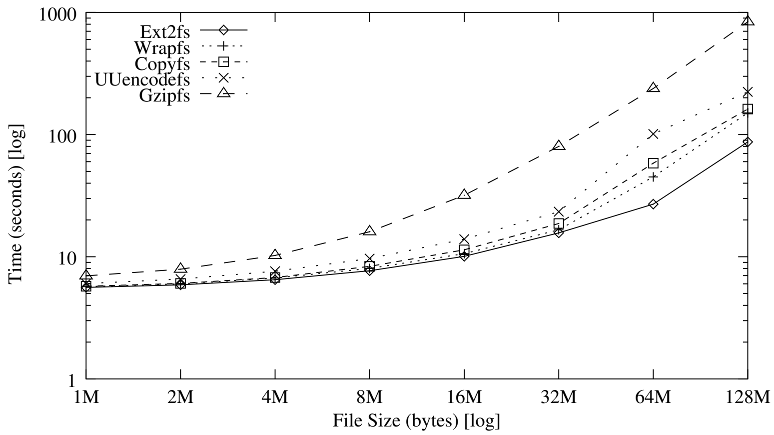
Figure 9.5 exhibits overhead spikes for 64MB files. Our test machines had 128MB of memory. Our stackable system caches two pages for each page of a file: one encoded page and one decoded page, effectively doubling the memory requirements. The 64MB files are the smallest test files that are large enough for the system to run out of memory. Linux keeps data pages cached for as long as possible. When it runs out of memory, Linux executes an expensive scan of the entire page cache and other in-kernel caches, purging as many memory objects as it can, possibly to disk. The overhead spikes in this figure occur at that time.

Bonnie shows that an expensive algorithm such as compression, coupled with many writes in the middle of large files, can degrade performance by as much as a factor of 5–6. In Section 10.1 we describe certain optimizations that we are exploring for this particular problem.

### 9.4.4 Micro-Benchmark Results

#### 9.4.4.1 File-Copy

Figure 9.6 shows the results of running the file-copy benchmark on the different file systems. Wrapfs adds an average overhead of 16.4% over Ext2fs, which goes to 60% for a file size of 32MB; this is the overhead of data page copying. Copyfs adds an average overhead of 23.7% over Wrapfs; this is the overhead of updating and writing the index file as well as having to make temporary data copies (explained in Section 6.3.1) to support writes in the middle of files. The Uuencode algorithm adds an additional average overhead of 43.2% over Copyfs, and as much as 153% overhead for the large 32MB file. The linear overheads of Copyfs increase with the file's size due to the extra page copies that Copyfs must make, as



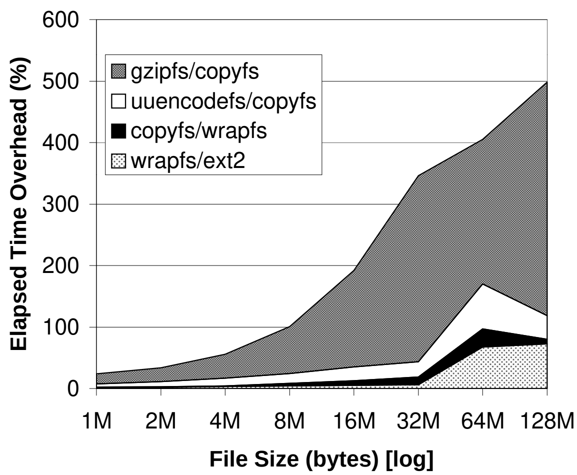




Figure 9.5: The Bonnie benchmark performs many repeated reads and writes on one file as well as numerous random seeks and writes in three concurrent processes. We show the total cumulative overhead of each file system. Note that the overhead bands for Gzipfs and Uuencodefs are each relative to Copyfs. We report the results for files 1MB and larger, where the overheads are more visible.

explained in Section 6.3.1. For all copies over 4KB, fast-tails makes no difference at all. Below 4KB, it only improves performance by 1.6% for Uuencodefs. The reason for this is that this benchmark copies files only once, whereas fast-tails is intended to work better in situations with multiple small appends.

#### 9.4.4.2 File-Append

Figure 9.7 shows the results of running the file-append benchmark on the different file systems. The figure shows the two emerging trends in effectiveness of the fast tails code. First, the more expensive the algorithm, the more helpful fast tails become. This can be seen in the right column of plots. Second, the smaller the number of bytes appended to the file is, the more savings fast tails provide, because the SCA is called fewer times. This can be seen as the trend from the bottom plots (1000 byte appends) to the top plots (10 byte appends). The upper rightmost plot clearly clusters together the benchmarks

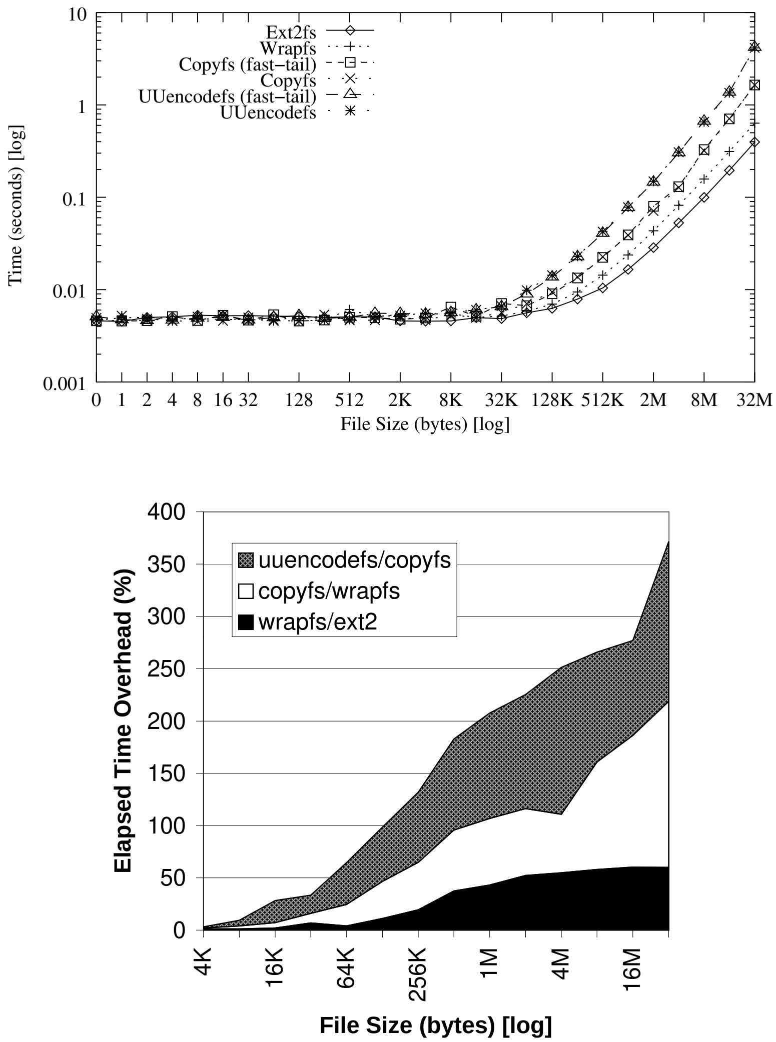



Figure 9.6: Copying files into a tested file system. As expected, Uuencodefs is costlier that Copyfs, Wrapfs, and Ext2fs. Fast-tails do not make a difference in this test, since we are not appending multiple times.

performed with fast tails support on and those benchmarks conducted without fast tails support.

Not surprisingly, there is little savings from fast tail support for Copyfs, no matter what the append size is. Uuencodefs is a simple algorithm that does not consume too much CPU cycles. That is why savings for using fast tails in Uuencodefs range from 22% for 1000-byte appends to a factor of 2.2 performance improvement for 10-byte appends. Gzipfs, using an expensive SCA, shows significant savings: from a minimum performance improvement factor of 3 for 1000-byte appends to as much as a factor of 77 speedup (both for moderately sized files).

#### 9.4.4.3 File-Attributes

Figure 9.8 shows the results of running the file-attributes benchmark on the different file systems. Wrapfs add an overhead of 35% to the GETATTR file system operation because it has to copy the attributes from one inode data structure into

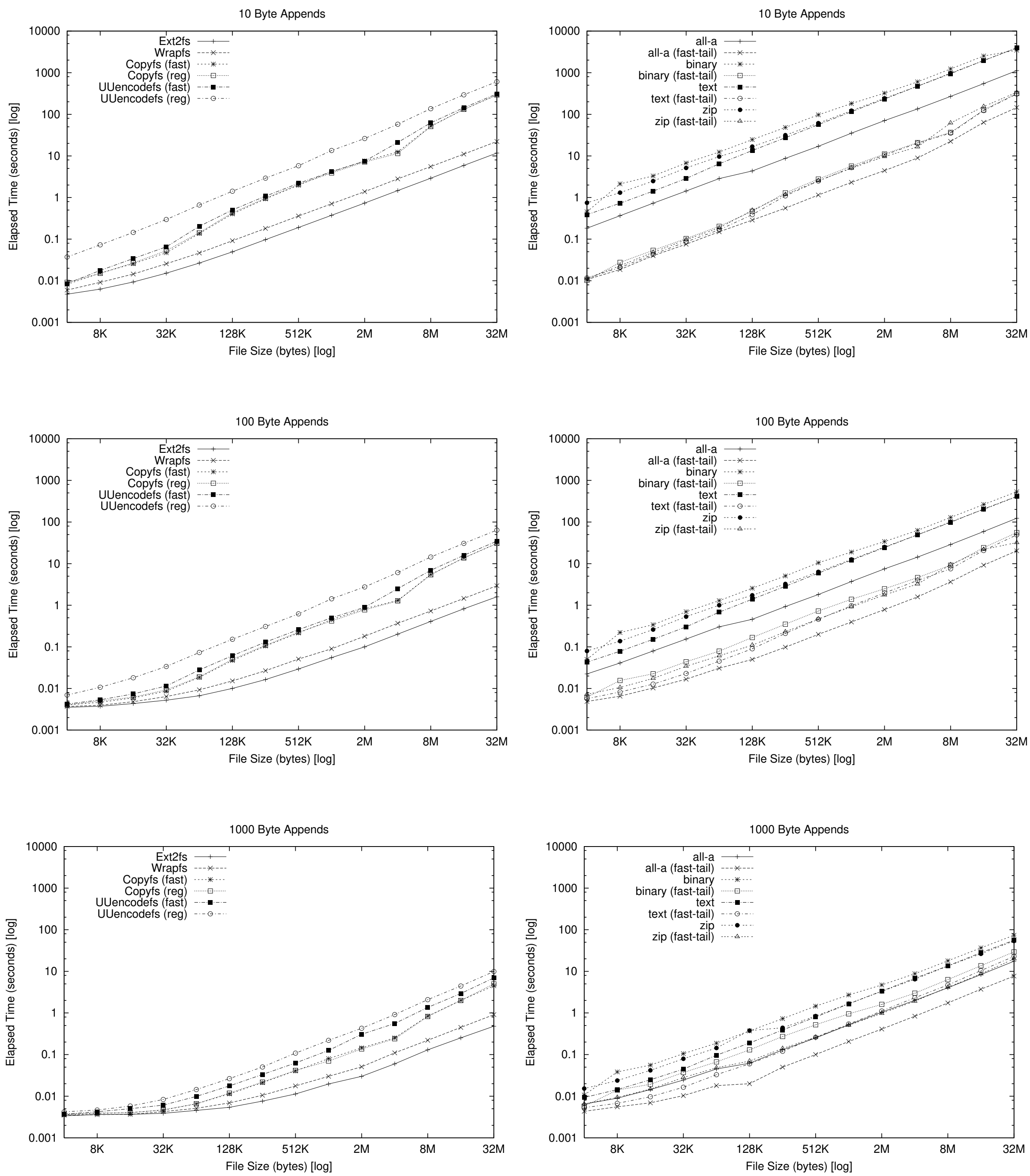



Figure 9.7: Appending to files. The left column of plots shows appends for Uuencodefs and Copyfs. The right column shows them for Gzipfs, which uses a more expensive algorithm; we ran Gzipfs on four different file types. The three rows of two plots each show, from top to bottom, appends of increasing sizes: 10, 100, and 1000 bytes, respectively. The more expensive the SCA is, and the smaller the number of bytes appended is, the more effective fast tails become; this can be seen as the trend from lower leftmost plot to the upper rightmost plot. The standard deviation for these plots did not exceed 9% of the mean.

another. SCA-based file systems add the most significant overhead, a factor of 2.6–2.9 over Wrapfs; that is because Copyfs, Uuencodefs, and Gzipfs include stackable SCA support, managing the index file in memory and on disk. The differences between the three SCA file systems in Figure 9.8 are small and within the error margin.

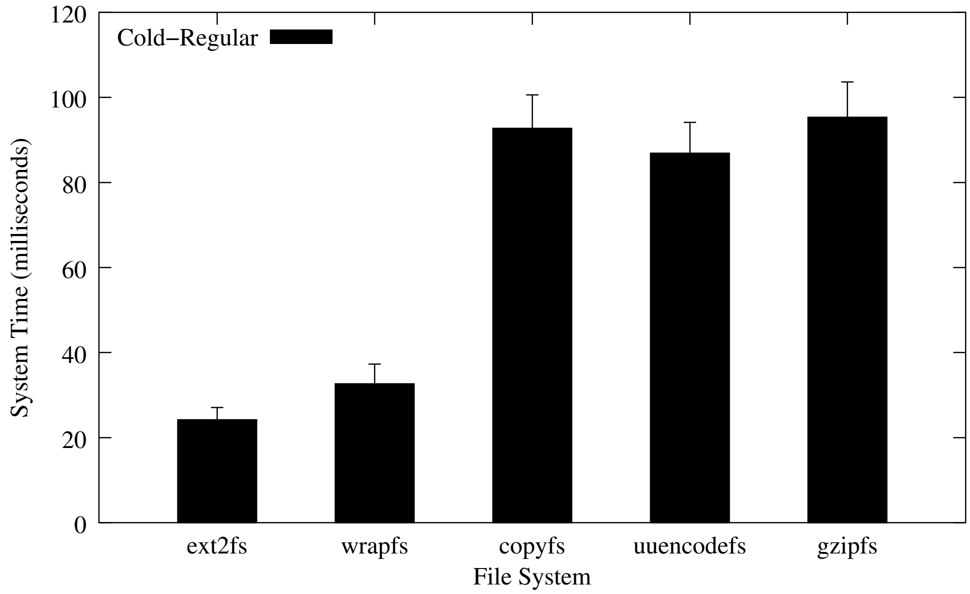



Figure 9.8: System times for retrieving file attributes using `lstat(2)` (cold cache)

While the GETATTR file operation is a popular one, it is still fast because the additional inode for the small index file is likely to be in the locality of the data file. Note that Figure 9.8 shows cold cache results, whereas most operating systems cache attributes once they are retrieved. Our measured speedup of cached vs. uncached attributes shows an improvement factor of 12–21. Finally, in a typical workload, bulk data reads and writes are likely to dominate any other file system operation such as GETATTR.

### 9.4.5 File System vs. User-Level Tool Results

Figure 9.9 shows the results of comparing Gzipfs against `gzip` using the file-copy benchmark. The reason Gzipfs is faster than `gzip` is primarily due to running in the kernel and reducing the number of context switches and kernel/user data copies.

As expected, the speedup for all files up to one page size is about the same, 43.3–53.3% on average; that is because the savings in context switches are almost constant. More interesting is what happens for files greater than 4KB. This depends on two factors: the number of pages that are copied and the type of data being compressed.

The Deflate compression algorithm is dynamic; it will scan ahead and back in the input data to try to compress more of it. Deflate will stop compressing if it thinks that it cannot do better. We see that for binary and text files, Gzipfs is 3–4 times faster than `gzip` for large files; this speedup is significant because these types of data compress well and thus more pages are manipulated at any given time by Deflate. For previously compressed data, we see that the savings is reduced to about double; that is because Deflate realizes that these bits do not compress easily and it stops trying to compress sooner (fewer pages are scanned forward). Interestingly, for the all-`a` file, the savings average only 12%. That is because the Deflate algorithm is quite efficient with that type of data: it does not need to scan the input backward and it continues to scan forward for longer. However, these forward-scanned pages are looked at few times, minimizing the

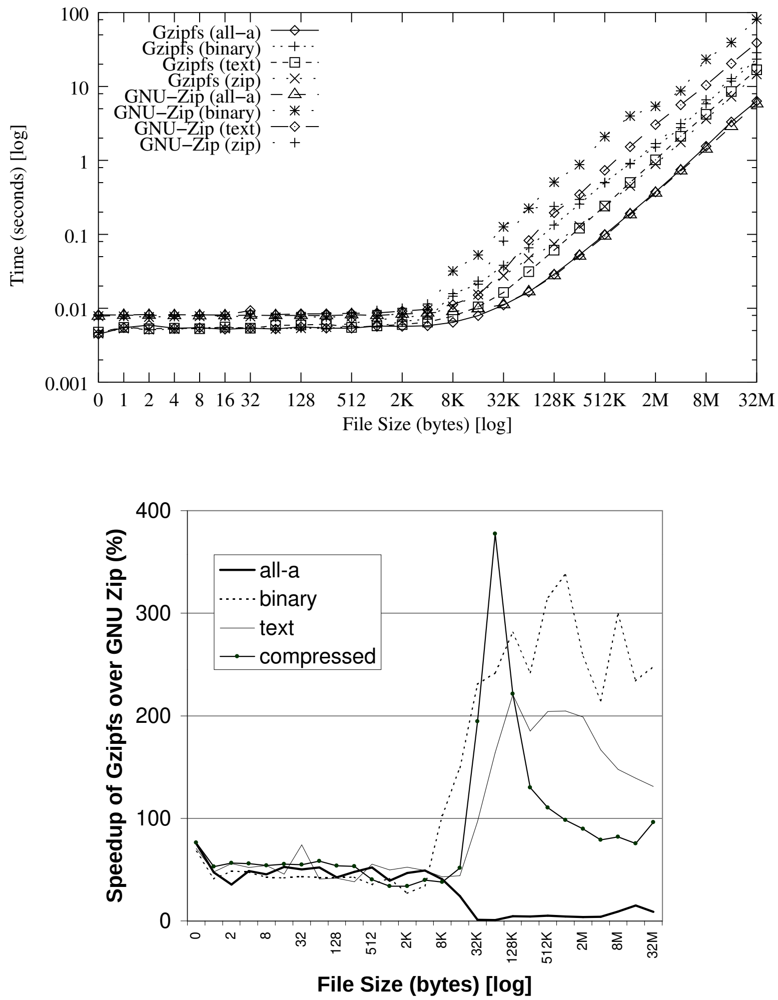



Figure 9.9: Comparing file copying into Gzipfs (kernel) and using `gzip` (user-level) for various file types and sizes. Here, a 100% speedup implies twice as fast.

number of data pages that `gzip` must copy between the user and the kernel. Finally, the plots in Figure 9.9 are not smooth because most of the input data is not uniform and thus it takes Deflate a different amount of effort to compress different bytes sequences.

One additional benchmark of note is the space savings for Gzipfs as compared to the user level `gzip` tool. The Deflate algorithm used in both works best when it is given as much input data to work with at once. GNU zip looks ahead at 64KB of data, while Gzipfs currently limits itself to 4KB (one page). For this reason, `gzip` achieves on average better compression ratios: as little as 4% better for compressing previously compressed data, to 56% for compressing the all-`a` file.

We also compared the performance of Uuencodefs to the user level `uuencode` utility. We found the performance savings to be comparable to those with Gzipfs compared to `gzip`.

### 9.4.6 Additional Tests

We measured the time it takes to recover an index file and found it to be statistically indifferent from the cost of reading the whole file. This is expected because to recover the index file we have to decode the complete data file.

Finally, we checked the in-kernel memory consumption. As expected, the total number of pages cached in the page cache is the sum of the encoded and decoded files' sizes (in pages). This is because in the worst case, when all pages are warm and in the cache, the operating system may cache all encoded and decoded pages. For Copyfs, this means doubling the number of pages cached; for Gzipfs, fewer pages than double are cached because the encoded file size is smaller than the original file; for Uuencodefs, 2.33 times the number of original data pages are cached because the algorithm increased the data size by one-third. In practice, we did not find the memory consumption in stacking file systems on modern systems to be onerous [86].

# Chapter 10

# Conclusion

The main contribution of this work is the FiST language which can describe stackable file systems. This is the first time a high-level language has been used to describe stackable file systems. From a single FiST description we generate code for different platforms. We achieved this portability because FiST uses an API that combines common features from several vnode interfaces. FiST saves its developers from dealing with many kernel internals, and lets developers concentrate on the core issues of the file system they are developing. FiST reduces the learning curve involved in writing file systems, by enabling non-experts to write file systems more easily.

The most significant savings FiST offers is in reduced development and porting time. The average time it took us to develop a stackable file system using FiST was about seven times faster than when we wrote the code using Basefs. We showed how FiST descriptions are more concise than hand-written C code: 5–8 times smaller for average stackable file systems, and as much as 33 times smaller for more complex ones. FiST generates file system modules that run in the kernel, thus benefitting from increased performance over user-level file servers. The minimum overhead imposed by our stacking infrastructure is 1–2%. As shown in Section 9.3, we have met our goal of meeting or exceeding the performance of other stackable file systems

FiST can be ported to other Unix platforms in 1–3 weeks, assuming the developers have access to kernel sources. The benefits of FiST are increased each time it is ported to a new platform: existing file systems described with FiST can be used on the new platform without modification.

The other contributions of this work include:

- The creation of stacking templates for different platforms, enabling for the first time for stackable file systems to run the same way on multiple operating systems.
- Support for arbitrary size-changing file systems such as compression and encryption—for both file systems that enlarge or shrink data sizes.
- The creation of a collection of useful file system code that can be shared between different file systems. This code defines a library of common functions for use by FiST developers.
- Providing a set of useful example file systems that can be used by future developers in various settings: academic, research, and commercial.
- Achieving stacking functionality without formal support and with no changes to *any* kernel code for either Solaris or FreeBSD.
- Contributing small amounts of code to Linux that allow it to implement stackable file systems. Our submitted changes have since been integrated into the Linux kernel by its maintainers. As of the Linux kernel version 2.3.99-pre6, no kernel changes to Linux are required.

## 10.1 Future Work

Our short-term goals include template work and FiST language work. These are intended to make the development of all types of future file systems easier than it is nowadays, hopefully as easy as user-level software.

### 10.1.1 Templates

We plan to port our system to Windows NT. NT has a different file system interface than Unix's vnode interface, but includes all of the same concepts (as we described in Section 2.1.8). NT's I/O subsystem defines its file-system interface. NT *Filter Drivers* are optional software modules that can be inserted above or below existing file systems [49]. Their task is to intercept and possibly extend file-system functionality, one file-system operation at a time. Filter drivers are able to execute initialization and completion functions which are very similar to our pre-call and post-call code. Therefore, it is quite possible to emulate file-system stacking under NT. We estimate that porting Basefs to NT will take 2–3 months, not 1–3 weeks as we predict for Unix ports.

VFS transactions are an important addition to stackable file systems (and file systems in general) because they allow for graceful recovery from partial failures of multi-component file systems (such as with fan-out). Transactions can also be used to provide additional tools to file system developers: journalling and logging.

### 10.1.2 FiST Language

We are exploring layer collapsing in FiST: a method to generate one file system that merges the functionality from several FiST descriptions, thus saving the per-layer stacking overheads. For example, an upper layer could call an operation directly in a lower layer several layers below, if fistgen determines that the intermediate layers for that operation simply pass it through unchanged. Another possibility of saving per-layer overheads is to allow each layer to set up callback functions for other layers, thus ensuring that each layer calls other layers in the the most direct way.

We are investigating ideas for describing low-level media-based file systems as well as distributed file systems. To produce low-level file systems, the language would have to describe the media type, its physical and geometrical properties, the layout of data objects, meta data, and data structures on the media, and the algorithms to used to manipulate the various objects. To produce distributed file systems, the language would have to include mechanisms to transfer data objects and their attributes across a network to multiple hosts, as well as synchronization, locking, and recovery mechanisms such as those available in Coda [37, 48, 66].

### 10.1.3 Operating Systems

Our longer term plans are to take the file-system ideas created in this work and apply them to other components of operating systems. For example, networking layers exhibit similar levels of modularity and interfacing between modules. While this modularity and the interfaces are common across operating systems, their implementations vary greatly [79].

In a similar fashion, device drivers have different in-kernel implementations, yet also have enough common features and implementations across systems [76, 27]. Networking and device drivers appear ideal candidates for generalization: the creation of a high-level language to describe their functionality, and using implementation mechanisms such as templates and libraries of common functions to offload the implementation burdens from developers.

We envision the exploration of generalization techniques for all other components of the operating system. Eventually, we intend to unify these techniques into a single system that can be used to rapidly prototype new operating systems with all of their complexity. Such a system will be comprised of a small kernel engine that exports many APIs, a library of many common functions that can be used as needed, and a language or set of languages to describe the general functionality of the operating system and its components.

# Bibliography

[1] V. Abrosimov, F. Armand, and M. I. Ortega. A Distributed Consistency Server for the CHORUS System. *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)* (Newport Beach, CA), pages 129–48, 26-27 March 1992.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the Summer USENIX Technical Conference*, pages 93–112, Summer 1986.

[3] Am-utils (4.4BSD Automounter Utilities). Am-utils version 6.0.4 User Manual. February 2000. http://www.am-utils.org.

[4] L. Ayers. E2compr: Transparent File Compression for Linux. *Linux Gazette*, Issue 18, June 1997. http://www.linuxgazette.com/issue18/e2compr.html.

[5] R. Balzer and N. Goldman. Mediating Connectors. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, pages 72–7, June 1999.

[6] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. *Proceedings of the Winter USENIX Technical Conference*, pages 267–75, Winter 1988.

[7] J. L. Bertoni. Understanding Solaris Filesystems and Paging. Tech Report TR-98-55. Sun Microsystems Research, November 1998. http://research.sun.com/research/techrep/1998/abstract-55.html.

[8] M. Blaze. A Cryptographic File System for Unix. *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, November 1993.

[9] S. R. Breitstein. Inferno Namespaces. Online White-Paper. Lucent Technologies, 11 March 1997. http://www.inferno.lucent.com/namespace.html.

[10] F. Brooks. “No Silver Bullet” Refired. In *The Mythical Man-Month, Anniversary Ed.*, pages 205–26. Addison-Wesley, 1995.

[11] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 2–9.

[12] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU’s Bulletin*. Free Software Foundation, 1994. http://www.gnu.org/software/hurd/hurd.html.

[13] B. Callaghan and T. Lyon. The Automounter. *Proceedings of the Winter USENIX Technical Conference* (San Diego, CA), pages 43–51, Winter 1989.

[14] V. Cate. Alex – a global file system. *Proceedings of the USENIX File Systems Workshop* (Ann Arbor, MI), pages 1–11, 21-22 May 1992.

[15] V. Cate and T. Gross. Combining the concepts of compression and caching for a two-level filesystem. *Fourth ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA), pages 200-211.

[16] P. Chen, C. Aycock, W. Ng, G. Rajamani, and R. Sivaramakrishnan. Rio: Storing Files Reliably in Memory. Technical Report CSE-TR250-95. University of Michigan, July 1995.

[17] R. Coker. The Bonnie Home Page. http://www.textuality.com/bonnie.

[18] P. Deutsch. Deflate 1.3 Specification. RFC 1051. Network Working Group, May 1996.

[19] P. Deutsch and J. L. Gailly. Gzip 4.3 Specification. RFC 1052. Network Working Group, May 1996.

[20] P. Deutsch and J. L. Gailly. Zlib 3.3 Specification. RFC 1050. Network Working Group, May 1996.

[21] J. Fitzhardinge. Userfs – user process filesystem. Unpublished software package documentation. Softway Pty, Ltd., August 1994. ftp://tsx-11.mit.edu/pub/linux/ALPHA/userfs.

[22] A. Forin and G. Malan. An MS-DOS Filesystem for UNIX. *Proceedings of the Winter USENIX Technical Conference* (San Francisco, CA), pages 337–54, Winter 1994.

[23] J. L. Gailly. GNU zip. http://www.gnu.org/software/gzip/gzip.html.

[24] J. L. Gailly and M. Adler. The zlib Home Page. http://www.cdrom.com/pub/infozip/zlib/.

[25] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared Libraries in SunOS. *Proceedings of the Summer USENIX Technical Conference* (Phoenix, AZ), pages 131–45, Summer 1987.

[26] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual Memory Architecture in SunOS. *Proceedings of the Summer USENIX Technical Conference* (Phoenix, AZ), pages 81–94, Summer 1987.

[27] S. Goel and D. Duchamp. Linux Device Driver Emulation in Mach. *Proceedings of the Annual USENIX Technical Conference*, pages 65–73, January 1996.

[28] 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. *Proceedings of the Summer USENIX Technical Conference*, pages 63–71, Summer 1990.

[29] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Proceedings of Fifteenth ACM Symposium on Operating Systems Principles*. ACM SIGOPS, December 1995.

[30] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Tech-report CSD-910007. UCLA, 1991.

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

[32] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler*, UNIX Programmer's Manual Volume 2 — Supplementary Documents. Bell Laboratories, Murray Hill, New Jersey, July 1978.

[33] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. *Proceedings of 16th ACM Symposium on Operating Systems Principles*, pages 52–65, October 1997.

[34] P. H. Kao, B. Gates, B. Thompson, and D. McCluskey. Support of the ISO-9660/HSG CD-ROM File System in HP-UX. *Proceedings of the Summer USENIX Technical Conference* (Baltimore, MD), pages 189–202, Summer 1989.

[35] B. W. Kernighan. A Descent into Limbo. Online White-Paper. Lucent Technologies, 12 July 1996. http://www.inferno.lucent.com/inferno/limbotut.html.

[36] Y. A. Khalidi and M. N. Nelson. Extensible File Systems in Spring. *Proceedings of the 14th Symposium on Operating Systems Principles* (Asheville, North Carolina). ACM, December 1993.

[37] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar Conference Center, Pacific Grove, U.S.), volume 25, number 5, pages 213–25. ACM Press, October 1991.

[38] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the Summer USENIX Technical Conference*, pages 238–47, Summer 1986.

[39] T. Lord. Subject: SNFS – Scheme-extensible NFS. Unpublished USENET article. emf.net, 30 October 1996. ftp://emf.net/users/lord/systas-1.1.tar.gz.

[40] Lucent. Inferno: la Commedia Interattiva. Online White-Paper. Lucent Technologies, 13 March 1997. http://inferno.lucent.com/inferno/infernosum.html.

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

[42] 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.

[43] R. G. Minnich. The AutoCacher: A File Cache Which Operates at the NFS Level. *Proceedings of the Winter USENIX Technical Conference* (San Diego, CA), pages 77–83, Winter 1993.

[44] 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, February 1994.

[45] J. G. Mitchell, J. J. Gibbons, 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*, February 1994.

[46] J. Moran and B. Lyon. The Restore-o-Mounter: The File Motel Revisited. *Proceedings of the Summer USENIX Technical Conference* (Cincinnati, OH), pages 45–58, Summer 1993.

[47] L. Mummert and M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. Report CMU-CS-94-213. Carnegie Mellon University, Pittsburgh, U.S., 1994.

[48] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. *Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Association for Computing Machinery SIGOPS, 3–6 December 1995.

[49] R. Nagar. Filter Drivers. In *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, September 1997.

[50] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, OMG document number 91.12.1, Rev. 1.1, December 1991.

[51] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. *Proceedings of the Summer USENIX Technical Conference*, pages 137–52, June 1994.

[52] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 25–33, December 1995.

[53] J. S. Pendry and N. Williams. Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha. March 1991.

[54] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Proceedings of Summer UKUUG Conference*, pages 1–9, July 1990.

[55] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9, a distributed system. *Proceedings of Spring EurOpen Conference*, pages 43–50, May 1991.

[56] PLC. StackFS: The Stackable File System Architecture. Online White-Paper. Programmed Logic Corporation, 1996. http://www.plc.com/st-wp.html.

[57] D. Presotto and P. Winterbottom. The Organization of Networks in Plan 9. *Proceedings of the Winter USENIX Technical Conference* (San Diego, CA), pages 271–80, Winter 1993.

[58] J. Rees, P. H. Levine, N. Mishkin, and P. J. Leach. An Extensible I/O System. *Proceedings of the Summer USENIX Technical Conference* (Atlanta, GA), pages 114–25, Summer 1986.

[59] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, **63**(8):1897–910, October 1984.

[60] R. Rodriguez, M. Koehler, and R. Hyde. The generic file system. *Proceedings of the Summer USENIX Technical Conference* (Atlanta, GA, June 1986), pages 260–9, June 1986.

[61] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. *Proceedings of the Annual USENIX Technical Conference*, June 2000.

[62] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 1–15. Association for Computing Machinery SIGOPS, 13 October 1991.

[63] D. S. H. Rosenthal. Requirements for a "Stacking" Vnode/VFS Interface. UI document SD-01-02-N014. UNIX International, 1992.

[64] D. S. H. Rosenthal. Evolving the Vnode Interface. *Proceedings of the Summer USENIX Technical Conference*, pages 107–18, Summer 1990.

[65] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *Proceedings of the Summer USENIX Technical Conference*, pages 119–30, Summer 1985.

[66] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, **39**:447–59, 1990.

[67] B. Schneier. Algorithm Types and Modes. In *Applied Cryptography, Second Edition*, pages 189–97. John Wiley & Sons, October 1995.

[68] B. Schneier. Blowfish. In *Applied Cryptography, Second Edition*, pages 336–9. John Wiley & Sons, October 1995.

[69] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *Proceedings of the Summer USENIX Technical Conference*, pages 161–74, June 1993.

[70] SMCC. swap(1M). SunOS 5.5 Reference Manual, Section 1M. Sun Microsystems, Incorporated, 2 March 1994.

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

[72] SMCC. crash(1M). SunOS 5.5 Reference Manual, Section 1M. Sun Microsystems, Incorporated, 25 January 1995.

[73] J. N. Stewart. AMD – The Berkeley Automounter, Part 1. *;login:*, **18**(3):19, May/June 1993.

[74] D. L. Stone and J. R. Nestor. IDL: background and status. *SIGPLAN Notices*, **22**(11):5–9, November 1987.

[75] A. S. Tanenbaum. The MS-DOS File System. In *Modern Operating Systems*, pages 340–2. Prentice Hall, 1992.

[76] S. Thibault, R. Marlet, and C. Consel. A Domain Specific Language for Video Device Drivers: From Design to Implementation. *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 11–26, October 1997.

[77] W. B. Warren, J. Kickenson, and R. Snodgrass. A tutorial introduction to using IDL. *SIGPLAN Notices*, **22**(11):18–34, November 1987.

[78] N. Webber. Operating System Support for Portable Filesystem Extensions. *Proceedings of the Winter USENIX Technical Conference*, pages 219–25, January 1993.

[79] G. Wright and R. Stevens. In *TCP/IP Illustrated, Volume 2: The implementation*. Addison-Wesley, January 1995.

[80] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems. *To appear in Proceedings of the Annual USENIX Technical Conference*, June 2001.

[81] E. Zadok and I. Bădulescu. Usenetfs: A Stackable File System for Large Article Directories. Technical Report CUCS-022-98. 1998.

[82] E. Zadok and I. Bădulescu. A Stackable File System Interface for Linux. *LinuxExpo Conference Proceedings*, May 1999.

[83] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 1998.

[84] E. Zadok, I. Bădulescu, and A. Shender. Extending File Systems Using Stackable Templates. *Proceedings of the Annual USENIX Technical Conference*, June 1999.

[85] E. Zadok and A. Dupuy. HLFSD: Delivering Email to Your $HOME. *Proceedings of the Seventh USENIX Systems Administration Conference (LISA VII)* (Monterey, CA), pages 243–54, 1-5 November 1993.

[86] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. *Proceedings of the Annual USENIX Technical Conference*, June 2000.

Software, documentation, a copy of this document, and additional papers are available from http://www.cs.columbia.edu/~ezk/research/fist/.

# Appendix A

# FiST Language Specification

In Chapter 4 we described the design of the FiST language, its major components, and most of its syntax. There, we aimed to explain the design decisions of the language. In this appendix we list the full syntax and what each feature is used for. We do not explain here why each feature exists. This appendix is intended as a reference for FiST users.

## A.1 Input File

The FiST input file has four sections. Each section is optional. If not specified, the meaning of each section remains unchanged from the behavior of the Basefs templates—a null layer stackable file system.

**C Declarations:** Lists C code that is included verbatim in the generated sources. This usually includes C headers, CPP macros, and forward function definitions. This section must be enclosed in a pair of `%{` and `%}`.

**FiST Declarations:** Includes directives that change the overall behavior of the generated file system. FiST parses and expands primitives and functions in this section. This section must be separated from the next section by a `%%`.

**FiST Rules:** Defines code actions to execute for a given operation or sets of operations. FiST parses and expands primitives and functions in this section. This section must be separated from the next section by a `%%`.

**Additional C Code:** Includes raw C code that is included almost verbatim. FiST also parses and expands primitives and functions in this section. In this section developers can insert raw C code that is written portably using FiST primitives. That way fistgen can generate portable code. Fistgen, however, does not validate the C code that is written in this or any other section for portability or correctness. This section must be separated from the previous section by a `%%`.

In the last three sections, blank lines are generally ignored. C style comments are copied verbatim to the output. C++ style comments are used as FiST language comments: their text is ignored.

## A.2 Primitives

FiST primitives include variables and their attributes. There are global read-only variables, global file-system variables, and file-system variables local to each file system operation. These primitives may appear anywhere in the last three sections of the FiST input file.

### A.2.1 Global Read-Only Variables

These variables represent operating-system state that cannot be changed by the FiST developer, but may change from call to call. Such variables begin with a "`%`":

**%blocksize:** native disk block size

**%gid:** effective group ID of calling process

**%pagesize:** native page size

**%pid:** process ID of calling process

**%time:** current time (seconds since epoch)

**%uid:** effective user ID of calling process

### A.2.2 File-System Variables and Their Attributes

File-system variables are references to a whole file system and to its attributes. The general syntax for such a reference is:

$$\$\texttt{vfs} : N.attribute \tag{A.1}$$

The context of file-system variables is the mounted file-system instance currently running. The values of these variables is not likely to change while the same file system is mounted.

File-system variables begin with a `$`. There is currently only one such variable: `$vfs`. The variable's name may be followed by a colon and a non-negative integer $N$ that describes the stacking branch to refer this VFS object to:

**$vfs:0** refers to the VFS object of this file system and is synonymous to `$vfs`.

**$vfs:1** refers to the first file system on the lower level. If no fan-out is used, then this refers to the only lower-level file system VFS object.

**$vfs:2** refers to the second file system on the lower level, assuming a fan-out of 2 or more was used.

**$vfs:**$N$ refers to the $N$-th file system on the lower level, assuming a fan-out of $N$ or more was used.

To refer to an attribute of a specific VFS object, append a dot and the attribute name to it. The list of allowed attributes are:

**bitsize:** the bit-size of the file system (32 or 64)

**blocksize:** the block size of the file system

**fstype:** the name of the file system being defined

*user-defined***:** any other pre-defined attribute name as specified in Section A.3.2.2

For example, `$vfs:2.blocksize` refers to the block size of the second lower branch of a stackable file system with a fan-out of two or more.

### A.2.3 File Variables and Their Attributes

File variables are references to individual files and to their attributes. The general syntax for such a reference is:

$$\$name : N.attribute \tag{A.2}$$

The context of file variables is the file-system function currently executing. The values of these variables change from each invocation of even the same function on the same mounted file system, since file objects correspond to user processes making system calls for different files.

Each file-system function has a reference to at least one file. Some functions, however, refer to more than one. Therefore, there are several possible names for these file references:

**$this:** refers to the primary file object of the function. This is synonymous to `$0`

**$dir:** refers to the vnode of the directory object for operations that use a directory object. For example, the remove file operation specifies a file to remove and a directory to remove the file from.

**$from:** refers to the source file in a rename operation that renames a file from a given name to another.

**$to:** refers to the target file in a rename operation that renames a file from a given name to another.

**$fromdir:** refers to the source directory in a rename operation that renames a file within a given directory to another directory.

**$todir:** refers to the target directory in a rename operation that renames a file within a given directory to another directory.

File variables' names may be followed by a colon and a non-negative integer $N$ that describes the stacking-branch number of this file object.

**$this:0:** refers to the file object of this file and is synonymous to `$0` and to `$this`.

**$dir:1:** refers to the first lower-level directory. If no fan-out is used, then this refers to the only lower-level directory.

**$from:2:** refers to the source file in the second lower-level file system, for a rename operation, assuming a fan-out of 2 or more was used.

**$todir:$N$:** refers to the target directory in a rename operation, of the $N$-th lower-level file system, assuming a fan-out of $N$ or more was used.

To refer to an attribute of a specific file object, append a dot and the attribute name to it. The list of allowed attributes is:

**ext:** file's extension (string component after the last dot)

**name:** full name of the file

**symlinkval:** string value of the target of a symbolic link, defaults to NULL for non-symlinks

**type:** the type of file (directory, socket, block/character device, symlink, etc.)

**atime:** access time, same as for the stat(2) system call

**blocks:** number of blocks, same as for the stat(2) system call

**ctime:** creation (or last chmod) time, same as for the stat(2) system call

**group:** group owner, same as for the stat(2) system call

**mode:** file access mode bits, same as for the stat(2) system call

**mtime:** last modification time, same as for the stat(2) system call

**nlink:** number of links, same as for the stat(2) system call

**size:** file size in bytes, same as for the stat(2) system call

**owner:** user who owns the file, same as for the stat(2) system call

*user-defined***:** any other pre-defined attribute name as specified in Section A.3.2.3

For example, `$1.name` refers to the name of the first lower file of a stackable file system with a fan-out of one or more.

## A.3 FiST Declarations

FiST declarations affect the overall behavior of the produced code. A declaration has at least one word and ends with a semi-colon. The words of declarations with multiple words are separated by whitespace.

Unless specified otherwise, FiST declarations may only appear once.

### A.3.1 Simple Declarations

The following declarations are simple in that they pick a value out of two or more allowed values.

**accessmode (readonly | writeonly | readwrite):** defines if the file system is read-only, write-only, or both (the default). A read-only file system, for example, will automatically turn off support for all file-system state-changing operations such as mkdir and unlink.

**debug (on | off):** turn on/off debugging (off by default). If on, debugging support is compiled in and the level of debugging can be set between 1 and 18 by the user-level tool `fist_ioctl`. For example, running `fist_ioctl 18` turns on the most verbose debugging level. Debugging output is printed by the kernel on the console.

**filter (data | name | sca):** Turn on filter support for fix-sized data pages (data), for file names (name), or for size-changing algorithms (sca). All three filters may be defined, but no more than one per line.

Turning on the data or SCA filters requires the developer to write two functions in the Additional C Code section:

1. **encode_data(inpage, inlen, outpage, outlen)**: a function to encode a data page. The function takes an input page and input length, and must fill in the output page and the output length integer with the number of bytes encoded. The function must return an integer status/error code: a 0 indicates success and a negative number indicates the error number (complies with standard errno values listed in /usr/include/sys/errno.h).
2. **decode_data(inpage, inlen, outpage, outlen)**: a function to decode a data page, otherwise behaves the same as `encode_data`.

Turning on the name filter requires the developer to write two functions in the Additional C Code section:

1. **encode_name(inname, inlen, outname, outlen)**: a function to encode a file name. The function takes an input name and input length, and must fill in the output name and the output length integer with the number of bytes encoded. The function must also allocate the output name string using fistMalloc (described below). The function must return an integer status/error code: a 0 indicates success and a negative number indicates the error number (complies with standard errno values listed in /usr/include/sys/errno.h).
2. **decode_name(inname, inlen, outname, outlen)**: a function to decode a file name, otherwise behaves the same as `encode_name`.

**mntstyle (regular | overlay):** defines the file system's mount style. Regular (the default) leaves the mounted directory exposed and available for direct access. Overlay mounts hide the mounted directory with the mount point.

**errorcode** $ARG$**:** defines a new error code. This declaration may be used multiple times to define additional error codes. Error code names must not conflict with system-defined or previously defined error codes. Newly defined error codes may be used anywhere in the last two sections of the FiST input file.

**fsname** $ARG$**:** set the name of the file system being defined. If not specified, it defaults to the name of the FiST input file name.

**mntflag** $ARG$**:** defines additional mount(2) flags to allow user processes mounting this file system to pass to the kernel. This declaration may be used multiple times to define additional mount flags. Mount flag names must not conflict with system-defined or previously defined ones. Newly defined mount flags may be used anywhere in the last two sections of the FiST input file.

**fanout** $N$**:** define the fan-out level of the file system. Defaults to 1 (no fan-out).

**fanin (yes | no):** allow ("yes") or disallow fan-in (allowed by default). If fan-in is disallowed, the file system will overlay itself on top of the mounted directory, thus hiding the directory mounted on.

### A.3.2 Complex Declarations

Complex FiST declarations are those that take a *Basic Data Type* (BDT) as an argument. A BDT is a C data structure that includes only simple data types in each field of the data structure: not pointers, typedefs, or other structures. Allowed types include shorts, integers, longs, floats, doubles, characters, signed and unsigned values, and fixed size arrays thereof.

A BDT is therefore a unique list of C type and variable declarations, delimited by semicolons, and enclosed entirely by a pair of curly braces. For example:

```
{
    int   cipher;
    char  key[16];
}
```

#### A.3.2.1 Additional Mount Data

Additional mount data may be passed from a user-level process performing a mount(2). This data is passed only once during the mount. Typical uses for this include data and flags that dynamically affect the overall behavior of the file system.

This declaration may be defined only once. The syntax for defining additional mount data is as follows:

$$\texttt{mntdata}\ BDT \tag{A.3}$$

For example:

```
mntdata {
    int     zip_level;
    time_t  expire;
};
```

#### A.3.2.2 New File-System Attributes

You may define additional attributes for file-system objects as explained in Section A.2.2. The fields of the BDT automatically become new attribute names. New attribute names may not conflict with existing ones.

This declaration may be defined only once.

The syntax for defining additional file-system attributes is as follows:

$$\texttt{pervfs}\ BDT \tag{A.4}$$

For example, if you define:

```
pervfs {
    int     max_vers;
    char    extension[4];
};
```

then you can refer to a new VFS attribute `$vfs.max_vers` anywhere in the last two sections of the FiST input file.

#### A.3.2.3 New File Attributes

You may define additional attributes for file objects as explained in Section A.2.3. The fields of the BDT automatically become new attribute names. New attribute names may not conflict with existing ones.

This declaration may be defined only once.

The syntax for defining additional file attributes is as follows:

$$\texttt{pervnode}\ BDT \tag{A.5}$$

For example, if you define:

```
pervnode {
    int     cipher;
    char    key[128];
};
```

then you can refer to a new file attribute `$0.key` (say, 1024-bit per-file encryption key) anywhere in the last two sections of the FiST input file.

#### A.3.2.4 Persistent Attributes

The `pervfs` and `pervnode` declarations above define volatile attributes: their values remain in memory until the file system is unmounted. If you wish to store attributes or any other data persistently, use the file format declaration.

The declaration defines a data structure that can be formatted on top of a file: the bits of the data structure are serialized onto a file. This declaration is used in conjunction with two FiST functions: fistSetFileData and fistGetFileData, described below in Section A.4.2.

The syntax for defining additional file formats is as follows:

$$\texttt{fileformat}\ NAME\ BDT \tag{A.6}$$

Here, $NAME$ is a unique name for the file format, followed by a $BDT$. You may define multiple file formats, but each must have a unique name.

#### A.3.2.5 New I/O Controls

This declaration defines a new I/O control—ioctl, and an optional data structure that can be used with that data structure. This declaration is used in conjunction with two FiST functions: fistSetIoctlData and fistGetIoctlData, described below in Section A.4.3.

The syntax for defining additional ioctls is as follows:

$$\texttt{ioctl}[:(\texttt{fromuser}|\texttt{touser}|\texttt{both}|\texttt{none})]\ NAME\ \ BDT \tag{A.7}$$

The ioctl declaration may be optionally followed by a colon and one of four values:

**fromuser:** the ioctl can only copy data from a user process to the kernel

**touser:** the ioctl can only copy data from the kernel to a user process

**both:** the ioctl can exchange data between the kernel and a user process bidirectionally (default)

**none:** the ioctl exchanges no data

In the definition, $NAME$ is a unique name for the ioctl, followed by a $BDT$. You may define multiple ioctls, but each must have a unique name.

### A.3.3 Makefile Support

These declarations help fistgen to produce the proper Makefile for compiling the new file system.

**mod_src** $FILE$ **...:** declares a list of additional files that must be compiled and linked with the loadable kernel modules for this file system. This is useful for example to list the C sources for your cipher of choice, if defining an encryption file system.

**mod_hdr** $FILE$ **...:** declares a list of additional files that the kernel module must depend on when compiling the loadable kernel module for this file system. This is useful for example to list the C headers for your compression algorithm of choice, if defining a compression file system.

**user_src** $FILE$ **...:** declares a list of additional files, each of which represents a stand-alone user-level program. These programs get compiled in addition to the file-system kernel-loadable module. This declaration can be used to list additional utilities that developers write. For example, our compression file system uses this to define a utility that can recover an index file from a compressed data file (see Section 6.3.2.2).

**add_mk** $FILE$ **...:** defines the names of files defining additional custom Makefile rules that the developer wants to include in the master Makefile used to build the file system. This can be used as a flexible extension mechanism to add any arbitrary Makefile rules to process.

## A.4 FiST Functions

FiST functions are functions like other C functions: they have a name, take a number of arguments, may return one value back, may be nested, and their return values may be assigned to other variables.

FiST functions may be called anywhere in the last two sections of the FiST input file.

FiST functions are very easy to add to the FiST language, and they form a collection library of common auxiliary file operations. Recall also that developers may write and refer to any regular C functions in the Additional C Section of the FiST input file.

### A.4.1 Basic Functions

These functions are simple. Their arguments are similar to other user-level (C library) functions. They are FiST functions because their usage on different operating systems is different and does not match the same usage as their user-level equivalents.

**fistMemCpy:** copies one buffer to another, same as memcpy(3).

**fistMalloc:** allocates kernel memory, same as malloc(3).

**fistFree:** frees kernel memory, same as free(3).

**fistStrEq:** compares two strings. Returns 1 (TRUE) if the strings are equal, and 0 otherwise.

**fistStrAdd($A$, $B$):** appends string $B$ to string $A$, same as strcat(3).

**fistPrintf:** print a formatted string, same as printf(3).

**fistSetErr($E$):** set the current error status to $E$. If not changed, that error is returned back from the file-system function to its caller.

**fistLastErr:** this function returns the last error that was explicitly set by fistSetErr or occurred as a result of calling the lower level file system or any other function that may have failed. If the last such action did not fail, this function returns 0.

**fistReturnErr($E$):** immediately returns from this function with the error code $E$. If $E$ is omitted, returns the last error (or 0 if there was none).

### A.4.2 File Format Functions

The two functions used in conjunction with the `fileformat` declaration (described in Section A.3.2.4) are:

$$\texttt{fistSetFileData}(FILE, FMTNAME, FIELD, IN) \tag{A.8}$$

and

$$\texttt{fistGetFileData}(FILE, FMTNAME, FIELD, OUT) \tag{A.9}$$

For both, $FILE$ is a name or reference to a file, $FMTNAME$ is the name of a format declared in `fileformat`, and $FIELD$ is a name of a field in the BDT of the `fileformat`.

The fistSetFileData function reads a variable's value from $IN$, and writes it to a file $FILE$ formatted with $FMTNAME$. It writes it to the field named $FIELD$ of that file. The fistGetFileData function reads a field from a given file formatted with a given file format, and stores that value into the variable specified in $OUT$. Both functions return 0 if successful, and an error code otherwise.

Both $IN$ and $OUT$ are typically pointers to other objects. See an example using this in Section 4.2.

### A.4.3 Ioctl Functions

The two functions used in conjunction with the `ioctl` declaration (described in Section A.3.2.5) are:

$$\texttt{fistSetIoctlData}(IOCTL, FIELD, IN) \tag{A.10}$$

and

$$\texttt{fistGetIoctlData}(IOCTL, FIELD, OUT) \tag{A.11}$$

For both, $IOCTL$ is the name of an Ioctl as defined by the `ioctl` declaration, and $FIELD$ is a name of a field in the BDT of the `ioctl`.

The fistSetIoctlData function reads a variable's value from $IN$, and writes it out to the $FIELD$ of the $IOCTL$. The fistGetIoctlData reads a value from a field of an ioctl and stores it in a variable named in $OUT$. Both functions return 0 if successful, and an error code otherwise.

See an example using this in Appendix B.1.

### A.4.4 File-System–Stacking Functions

These functions mirror many vnode and system-call functions, allowing FiST developers to write code that calls other basic file-system functions such as creating new files, deleting them, listing directory contents, and more.

**fistLookup(**$DIR$**,** $NAME$**,** $OUT$**,** $USER$**,** $GROUP$**):** looks up a file in a directory $DIR$, using the credentials of $USER$ and $GROUP$, and places the resulting new file/vnode reference in $OUT$. If not specified, $DIR$ defaults to `$dir`, $OUT$ defaults to `$1`, and $USER$ and $GROUP$ default to the credentials of the currently executing process. When arguments at the end of a function are omitted, their delimiting commas can also be omitted. When arguments not at the end of a function are omitted, their delimiting commas must remain.

**fistReaddir(**$DIR$**,** $N$**,** $OUT$**,** $FLAGS$**):** reads $N$ names from a directory $DIR$ and stores them in $OUT$. If $FLAGS$ includes `NODUPS`, will automatically ignore file names that have been seen already in the context of this vnode function.

**fistSkipName(**$NAME$**):** do not list the file name $NAME$ in the context of this function. Only applicable when we are in the readdir vnode function.

**fistRename(**$A$**,** $B$**):** rename file $A$ to file $B$. This is one of several functions that use the same arguments as their system-call counterparts: unlink, mkdir, rmdir, etc.

**fistCopyFile(**$A$**,** $B$**):** copies file $A$ to file $B$. Both files may be referenced by their name or by a vnode reference as described in Section A.2.3.

**fistOp(...):** calls the same vnode function that we are executing. Arguments may be replaced depending on the function being called. This is a way to make a simple stacking call to a lower-level file system.

## A.5 FiST Rules

The third section of the FiST input file specifies any number of rules: code actions that can execute for any number or portions of file system functions. The format for a rule is:

$$\%callset : optype : part \; \{code\} \tag{A.12}$$

The code portion enclosed in braces is the action that gets executed for a given file-system function. The first parts—callset, optype, and part—define to which functions or portions thereof to apply the code.

### A.5.1 Call Sets

The call sets of a FiST rule (callset) picks one or more file system functions based on their overall operation:

**op:** pick a single operation as defined in *optype*

**ops:** select all operations

**readops:** select all operations that do not change state on disk

**writeops:** selects all operations that do change state on disk

### A.5.2 Operation Types

The operation type field (optype) further refines the selection of functions to one function or a set of them based on the type of data they manipulate:

**all:** all functions

**data:** all functions that manipulate data pages (read, write, getpage, putpage, etc.)

**name:** all functions that manipulate file names (open, lookup, rmdir, unlink, rename, readdir, etc.)

*single***:** any single operation out of the following: create, getattr, l/stat link, lookup, mkdir, read, readdir, readlink, rename, rmdir, setattr, statfs, symlink, unlink, and write.

### A.5.3 Call Part

The call part makes the final refinement of where to place the code:

**precall:** insert the code before calling the lower-level stackable file-system operation (as depicted in Figure 4.2.

**call:** replace the actual call to the lower-level file system with this code.

**postcall:** insert the code after calling the lower-level stackable file-system operation.

*ioctl***:** apply the code as the result of calling the specific ioctl as described in Section A.3.2.5.

# Appendix B

# Extended Code Samples

In this appendix we include the full FiST or C code for examples depicted in this dissertation. The full code given here is for reference. Additional code is available from http://www.cs.columbia.edu/~ezk/research/fist/.

## B.1 FiST Code for Cryptfs

The following is the full FiST code for Cryptfs, described in Section 8.1. We do not include the generic code for the Blowfish cipher because we have used those verbatim. The FiST file instructs fistgen to compile and link in the blowfish sources using the declarations `mod_src` and `mod_hdr`.

```
%{
extern unsigned char global_iv[8];
#include <blowfish.h>
%}
debug on;
filter data;
filter name;
mod_src bf_cfb64.c bf_enc.c bf_skey.c;
mod_hdr bf_locl.h bf_pi.h blowfish.h;
user_src fist_setkey.c fist_getiv.c;

ioctl:fromuser SETKEY {
  char ukey[16];
};
ioctl:touser GETIV {
  char outiv[8];
};
pervfs {
  BF_KEY key;
};
%%
%op:ioctl:SETKEY {
```

```
  char temp_buf[16];

  if (fistGetIoctlData(SETKEY, ukey, temp_buf) < 0)
    fistSetErr(EFAULT);
  else
    BF_set_key(&$vfs.key, 16, temp_buf);
}
%op:ioctl:GETIV {
  if (fistSetIoctlData(GETIV, outiv, global_iv) < 0)
    fistSetErr(EFAULT);
}
%%

unsigned char global_iv[8] = {
  0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10
};


int
cryptfs_encode_block(const char *from, char *to, int len,
                     const vnode_t *this_vnode, const vfs_t *this_vfs)
{
  int n = 0;                    /* internal blowfish variables */
  unsigned char iv[8];

  fistMemCpy(iv, global_iv, 8);
  BF_cfb64_encrypt((char *)from, to, %pagesize,
                   &($vfs.key), iv, &n,
                   BF_ENCRYPT);
  return %pagesize;
}


int
cryptfs_decode_block(const char *from, char *to, int len,
                     const vnode_t *this_vnode, const vfs_t *this_vfs)
{
  int n = 0;                    /* internal blowfish variables */
  unsigned char iv[8];

  fistMemCpy(iv, global_iv, 8);
  BF_cfb64_encrypt((char *) from, to, %pagesize,
                   &($vfs.key), iv, &n,
                   BF_DECRYPT);
  return %pagesize;
```

```
}


int
cryptfs_encode_filename(const char *name,
                        int length,
                        char **encoded_name,
                        int skip_dots,
                        const vnode_t *this_vnode,
                        const vfs_t *this_vfs)
{
    char *crypted_name;
    const char *ptr;
    int rounded_length, encoded_length, n, i, j;
    unsigned char iv[8];
    short csum;
    void *key = &($vfs.key);

    fist_dprint(8, "ENCODEFILENAME: cleartext filename \"%s\"\n", name);

    if ((skip_dots && (name[0] == '.' &&
                       (length == 1 ||
                        (name[1] == '.' && length == 2))))) {
        encoded_length = length + 1;
        *encoded_name = fistMalloc(encoded_length);
        fistMemCpy(*encoded_name, name, length);
        (*encoded_name)[length] = '\0';
        goto out;
    }
    for (csum = 0, i = 0, ptr = name; i < length; ptr++, i++)
        csum += *ptr;
    /*
     * rounded_length is an multiple of 3 rounded-up length
     * the encode algorithm processes 3 source bytes at a time
     * so we have to make sure we don't read past the memory
     * we have allocated
     *
     * it uses length + 3 to provide 2 bytes for the checksum
     * and one byte for the length
     */
    rounded_length = (((length + 3) + 2) / 3) * 3;
    crypted_name = fistMalloc(rounded_length);

    fistMemCpy(iv, global_iv, 8);
    n = 0;
```

```
    *(short *) crypted_name = csum;
    crypted_name[2] = length;
    BF_cfb64_encrypt((char *) name, crypted_name + 3,
                     length, (BF_KEY *) key, iv, &n,
                     BF_ENCRYPT);
    /*
     * clear the last few unused bytes
     * so that we get consistent results from encode
     */
    for (i = length + 3; i < rounded_length; i++)
        crypted_name[i] = 0;

    encoded_length = (((length + 3) + 2) / 3) * 4 + 1;
    *encoded_name = fistMalloc(encoded_length);

    for (i = 0, j = 0; i < rounded_length; i += 3, j += 4) {
        (*encoded_name)[j] = 48 + ((crypted_name[i] >> 2) & 63);
        (*encoded_name)[j + 1] = 48 + (((crypted_name[i] << 4) & 48) |
                                       ((crypted_name[i + 1] >> 4) & 15));
        (*encoded_name)[j + 2] = 48 + (((crypted_name[i + 1] << 2) & 60) |
                                       ((crypted_name[i + 2] >> 6) & 3));
        (*encoded_name)[j + 3] = 48 + (crypted_name[i + 2] & 63);
    }
    (*encoded_name)[j] = '\0';

    fistFree(crypted_name, rounded_length);
 out:
    fist_dprint(8, "ENCODEFILENAME: encoded filename \"%s\"\n", *encoded_name);
    return encoded_length;
}


int
cryptfs_decode_filename(const char *name,
                        int length,
                        char **decrypted_name,
                        int skip_dots,
                        const vnode_t *this_vnode,
                        const vfs_t *this_vfs)
{
    int n, i, j, saved_length, saved_csum, csum;
    int uudecoded_length, error = 0;
    unsigned char iv[8];
    char *uudecoded_name;
    void *key = &($vfs.key);
```

```
if ((skip_dots && (name[0] == '.' &&
                   (length == 1 ||
                    (name[1] == '.' && length == 2))))) {
    *decrypted_name = fistMalloc(length);
    for (i = 0; i < length; i++)
        (*decrypted_name)[i] = name[i];
    error = length;
    goto out;
}
if (key == NULL) {
    error = -EACCES;
    goto out;
}
uudecoded_length = ((length + 3) / 4) * 3;
uudecoded_name = fistMalloc(uudecoded_length);

for (i = 0, j = 0; i < length; i += 4, j += 3) {
    uudecoded_name[j] = ((name[i] - 48) <<2) | ((name[i + 1] - 48) >>4);
    uudecoded_name[j + 1] = (((name[i + 1] - 48) <<4) & 240) |
                             ((name[i + 2] - 48) >>2);
    uudecoded_name[j + 2] = (((name[i + 2] - 48) <<6) & 192) |
                             ((name[i + 3] - 48) &63);
}
saved_csum = *(short *) uudecoded_name;
saved_length = uudecoded_name[2];
if (saved_length > uudecoded_length) {
    fist_dprint(7, "Problems with the length - too big: %d", saved_length);
    error = -EACCES;
    goto out_free;
}
*decrypted_name = (char *) fistMalloc(saved_length);
fistMemCpy(iv, global_iv, 8);
n = 0;
BF_cfb64_encrypt(uudecoded_name + 3, *decrypted_name,
                 saved_length, (BF_KEY *) key, iv, &n,
                 BF_DECRYPT);
for (csum = 0, i = 0; i < saved_length; i++)
    csum += (*decrypted_name)[i];
if (csum != saved_csum) {
    fist_dprint(7, "Checksum error\n");
    fistFree(*decrypted_name, saved_length);
    error = -EACCES;
    goto out_free;
}
```

```
    error = saved_length;
 out_free:
    fistFree(uudecoded_name, uudecoded_length);
 out:
    return error;
}
```

## B.2 FiST Code for Gzipfs

Even a complex stackable file system such as Gzipfs, described in Section 8.6, involves mostly using the correct calls from the zlib compression library [20, 24]. Since we have used zlib unchanged, we are not including the code for it here.

```
%{
#define MY_ZCALLOC              /* Define our own alloc/free functions */
#define GZIPFS_DEFLATE_LEVEL 9

#include "zlib.h"
#include "fist.h"
%}
debug on;
filter sca;
filter data;
mod_src inflate.c zutil.c infblock.c deflate.c trees.c infutil.c inftrees.c \
        infcodes.c adler32.c inffast.c;
mod_hdr zutil.h infblock.h deflate.h zlib.h zconf.h trees.h infutil.h \
        inftrees.h infcodes.h inffast.h inffixed.h;

add_mk gzipfs.mk;

%%

%%

void *zcalloc (void *opaque, unsigned items, unsigned size)
{
    void *out;
    int cnt = size * items;

    print_entry_location();
    out = (void *)fistMalloc(cnt);
    print_exit_pointer(out);
    return(out);
}

void zcfree (void *opaque, void *ptr)
```

```
{
    print_entry_location();
    kfree(ptr);
    print_exit_location();
}

int
gzipfs_encode_buffers(char *hidden_pages_data, /* A PAGE_SIZE buffer
                                                   (already allocated)
                                                   passed to us to fill in
                                                */
                      char *page_data, /* The data we are to encode */
                      int *need_to_call, /* Call us again? */
                      unsigned to, /* how far into page_data to encode */
                      inode_t *inode, /* The inode in question */
                      vfs_t *vfs, /* vfs_t, unused */
                      void **opaque)    /* Opaque data */
     /* encodes the data in page_data into hidden_pages_data. Returns
        -errno for error, and the size of hidden_pages_data for success */
{
    z_stream *zptr;
    int rc, err;

    print_entry_location();

    if (*opaque != NULL) {
        zptr = (z_stream *) *opaque;
        zptr->next_out = hidden_pages_data;
        zptr->avail_out = PAGE_CACHE_SIZE;
        zptr->total_out = 0;
    } else {
        zptr = kmalloc(sizeof(z_stream), GFP_KERNEL);
        if (zptr == NULL) {
            err = -ENOMEM;
            goto out;
        }
        zptr->zalloc = (alloc_func)0;   /* Custom memory allocator called with
                                           opaque */
        zptr->zfree = (free_func)0;     /* Custom memory free-er called with
                                           opaque as an argument */
        zptr->opaque = (voidpf)0;       /* Opaque argument to zalloc/zfree */

        zptr->next_in = page_data;      /* + *(from);*/
        zptr->avail_in = to;            /* to - *(from) */
        zptr->next_out = hidden_pages_data;
```

```
        zptr->avail_out = PAGE_CACHE_SIZE;

        /*
         * First arg is a stream object
         * Second arg is compression level (0-9)
         */
        rc = deflateInit(zptr, GZIPFS_DEFLATE_LEVEL);
        if (rc != Z_OK ) {
            printk("inflateInit error %d: Abort\n",rc);
            /* This is bad. Lack of memory is the usual cause */
            err = -ENOMEM;
            goto out;
        }

        while ((zptr->avail_out > 0) &&
               (zptr->avail_in >  0)) { /* While we're not finished */
            rc = deflate(zptr, Z_FULL_FLUSH); /* Do a deflate */
            if ((rc != Z_OK) && (rc != Z_STREAM_END)) {
                printk("Compression error! rc=%d\n",rc);
                err = -EIO;
                goto out;
            }
        }
    }

    rc = deflate(zptr, Z_FINISH);

    if (rc == Z_STREAM_END) {
        deflateEnd(zptr);
        kfree(zptr);
        *need_to_call = 0;
    } else if (rc == Z_BUF_ERROR || rc == Z_OK) {
        *opaque = zptr;
    } else
        printk("encode_buffers error: rc=%d\n", rc);

    err = zptr->total_out;      /* Return encoded bytes */

 out:
    print_exit_status(err);
    return(err);
}


int
```

```
gzipfs_decode_buffers(int num_hidden_pages, /* The number of pages in
                                               hidden_pages_data */
                      char **hidden_pages_data, /* An array of pages
                                                   containing encoded
                                                   data */
                      char *page_data, /* A pre-allocated PAGE_SIZE
                                          buffer to write the decoded
                                          data to */
                      vnode_t *vnode,  /* The vnode of the file in
                                          question, unused */
                      vfs_t *vfs,      /* vfs_t, unused */
                      void *opaque)      /* opaque data filled in by
                                            wrapfs_Sca_count_pages,
                                            containing an int, the
                                            starting offset within the
                                            first page of hidden_pages_data
                                            of the encoded page we want
                                         */
     /* Returns -errno for error, or the number of bytes decoded on success
        (Should usually return PAGE_SIZE bytes) */
{
    z_stream stream;            /* Decompression stream obj */
    int i, offset, rc;
    int err = 0;

    print_entry_location();

    stream.zalloc = (alloc_func)0; /* Custom memory allocator called with
                                      stream.opaque */
    stream.zfree = (free_func)0; /* Custom memory free-er called with
                                    stream.opaque as an argument */
    stream.opaque = (voidpf)0;  /* Opaque argument to zalloc/zfree */

    offset = (int)opaque;       /* This is filled in with the starting
                                   offset in wrapfs_count_hidden_pages */

    stream.next_in = hidden_pages_data[0] + offset; /* start offset bytes
                                                       into page 0. */
    stream.avail_in = PAGE_CACHE_SIZE - offset;
    stream.next_out = page_data; /* Set the output buffer to what we were
                                    passed */
    stream.avail_out = PAGE_CACHE_SIZE;

    fist_dprint(6, "next_in = 0x%x\n", stream.next_in);
    fist_dprint(6, "avail_in = 0x%x\n", stream.avail_in);
```

```
fist_dprint(6, "next_out = 0x%x\n", stream.next_out);
fist_dprint(6, "avail_out = 0x%x\n", stream.avail_out);

rc = inflateInit(&stream);  /* Initialize the decompression stream */
if (rc != Z_OK) {
    printk("inflateInit error %d: Abort\n",rc);
    /* This is bad. Lack of memory is the usual cause */
    err = -ENOMEM;
    goto out;
}

fist_dprint(8, "Page 0: %02x%02x%02x%02x\n",
            (u8)hidden_pages_data[0][0],
            (u8)hidden_pages_data[0][1],
            (u8)hidden_pages_data[0][2],
            (u8)hidden_pages_data[0][3]);
rc = inflate(&stream, Z_NO_FLUSH); /* Inflate some data */

/* If there was more than one page in hidden_pages_data, we go here */
for (i = 1; i < num_hidden_pages; i++) { /* Step over each encoded page */
    fist_dprint(6, "Crossed a page boundary (%d)\n", rc);
    fist_dprint(8, "Page %d: %02x%02x%02x%02x\n", i,
                (u8)hidden_pages_data[i][0],
                (u8)hidden_pages_data[i][1],
                (u8)hidden_pages_data[i][2],
                (u8)hidden_pages_data[i][3]);
    if (rc == Z_STREAM_END) { /* We've finished early? Bug? */
        printk("gzipfs: Premature end of compressed data!\n");
        err = stream.total_out;
        goto out;
    } else if (rc == Z_OK) { /* Normal case, successful inflation,
                               need more input data */
        stream.next_in = hidden_pages_data[i];
        stream.avail_in = PAGE_CACHE_SIZE; /* This is a lie for the last
                                              page, but zlib is smart
                                              enough to figure it out. */

        fist_dprint(6, "***next_in = 0x%x\n", stream.next_in);
        fist_dprint(6, "***avail_in = 0x%x\n", stream.avail_in);
        fist_dprint(6, "***next_out = 0x%x\n", stream.next_out);
        fist_dprint(6, "***avail_out = 0x%x\n", stream.avail_out);

        rc = inflate(&stream, Z_FULL_FLUSH); /* Inflate some data */

        /* If, after this inflate, there is no space left, and
```

```
               we are not finished, bomb out */
            if (rc == Z_BUF_ERROR) {
                printk("Data contains more than %ld bytes\n",stream.total_out);
                printk("%ld written out, %d left to decode %ld decoded(%d)\n",
                       stream.total_out, stream.avail_in,stream.total_in,rc);
                err = -EIO;
                goto out;
            }
        } else  { /* Error condition! */
            printk("zlib error %d\n", rc);
            err = -EIO;
            goto out;
        }
    }

    err = stream.total_out;     /* Return the number of bytes we decoded */
    /* We finished out input data without getting an end-of-stream */
    if (rc != Z_STREAM_END && stream.avail_in > 0)
        printk("Finished loop without reaching end of chunk(%d)\n",rc);

 out:
    inflateEnd(&stream);        /* Close the stream object */

    print_exit_status(err);
    return(err);
}

/*
 * Local variables:
 * c-basic-offset: 4
 * End:
 */
```

## B.3 FiST Code for UUencodefs

Below we include the full FiST code to UUencodefs, which we described in Section 8.5.

```
%{
#include "fist.h"
%}
debug off;
filter sca;
filter data;
add_mk uuencodefs.mk;
%%
```

```
%%

int
uuencodefs_encode_buffers(char *hidden_pages_data, /* A PAGE_SIZE buffer
      (already allocated)
      passed to us to fill in
   */
  char *page_data, /* The data we are to encode */
  int *need_to_call, /* Call us again? */
  unsigned to, /* from + no. bytes to write */
  inode_t *inode, /* The inode in question */
  vfs_t *vfs, /* vfs_t ??? unused */
  void **opaque) /* Opaque data */
     /* encodes the data in page_data into hidden_pages_data. Returns
-errno for error, and the size of hidden_pages_data for success */
{
    int in_bytes_left;
    int out_bytes_left = PAGE_CACHE_SIZE;
    int startpt;
    unsigned char A, B, C;
    int bytes_written = 0;

    print_entry_location();

    startpt = (int)*opaque;
    in_bytes_left = to - startpt;

    while ((in_bytes_left > 0) && (out_bytes_left >= 4)) {
ASSERT(startpt < PAGE_CACHE_SIZE);

A = page_data[startpt];

switch(in_bytes_left) {
case 1:
    B = 0;
    C = 0;
    in_bytes_left--;
    startpt += 1;
    break;
case 2:
    B = page_data[startpt + 1];
    C = 0;
    startpt += 2;
    in_bytes_left -= 2;
```

```
    break;
default:
    B = page_data[startpt + 1];
    C = page_data[startpt + 2];
    startpt += 3;
    in_bytes_left -= 3;
    break;
}

hidden_pages_data[bytes_written] = 0x20 + (( A >> 2 ) & 0x3F);
out_bytes_left--; bytes_written++;

ASSERT(bytes_written < PAGE_CACHE_SIZE);

hidden_pages_data[bytes_written] = 0x20 +
    ((( A << 4 ) | ((B >> 4) & 0xF)) & 0x3F);
out_bytes_left--; bytes_written++;

ASSERT(bytes_written < PAGE_CACHE_SIZE);

hidden_pages_data[bytes_written] = 0x20 +
    ((( B << 2 ) | ((C >> 6) & 0x3)) & 0x3F);
out_bytes_left--; bytes_written++;

ASSERT(bytes_written < PAGE_CACHE_SIZE);

hidden_pages_data[bytes_written] = 0x20 + ((C) & 0x3F);
out_bytes_left--; bytes_written++;

ASSERT(bytes_written <= PAGE_CACHE_SIZE);
    }

    if (in_bytes_left > 0)
*opaque = (void *)startpt;
    else
*need_to_call = 0;

    print_exit_status(bytes_written);
    return bytes_written;
}

int
uuencodefs_decode_buffers(int num_hidden_pages, /* The number of pages in
   hidden_pages_data */
  char **hidden_pages_data, /* An array of pages
```

```
       containing encoded
       data */
  char *page_data, /* A pre-allocated PAGE_SIZE
      buffer to write the decoded
      data to */
  inode_t *inode,  /* The inode of the file in
      question */
  vfs_t *vfs,      /* vfs_t, unused */
  void *opaque)    /* opaque data filled in by
      wrapfs_Sca_count_pages,
      containing an int, the
      starting offset within the
      first page of hidden_pages_data
      of the encoded page we want
   */
     /* Returns -errno for error, or the number of bytes decoded on success
(Should usually return PAGE_SIZE bytes) */
{
    int i;
    int startpt = (int)opaque;
    int bytes_left = PAGE_CACHE_SIZE;
    unsigned char *ptr;
    unsigned char A,B,C,D;
    int outcnt = 0;

    print_entry_location();
    for (i = 0; i < num_hidden_pages; i++) { /* Step through each page */
ptr = hidden_pages_data[i] + startpt;
bytes_left = PAGE_CACHE_SIZE - startpt;
while(bytes_left >= 4) {
    A = ptr[0] - 0x20;
    B = ptr[1] - 0x20;
    C = ptr[2] - 0x20;
    D = ptr[3] - 0x20;

    switch (PAGE_CACHE_SIZE - outcnt) {
    case 0:
goto out;
    case 1:
page_data[outcnt] = (A<<2) | (B>>4);
outcnt += 1;
goto out;
    case 2:
page_data[outcnt] = (A<<2) | (B>>4);
page_data[outcnt+1] = (B<<4) | (C>>2);
```

```
outcnt += 2;
goto out;
    default:
page_data[outcnt] = (A<<2) | (B>>4);
page_data[outcnt+1] = (B<<4) | (C>>2);
page_data[outcnt+2] = (C<<6) | D;
    }

    ptr+=4;
    outcnt += 3;
    bytes_left -= 4;
}
startpt = 0;
    }

 out:
    print_exit_status(outcnt);
    return(outcnt);
}

/*
 * Local variables:
 * c-basic-offset: 4
 * End:
 */
```

## B.4 FiST Code for Copyfs

To illustrate the simplicity in the FiST language of using size-changing file systems, we include the full FiST code for Copyfs 8.4.

```
%{
#include "fist.h"
%}
debug off;
filter sca;
filter data;
add_mk copyfs.mk;
%%

%%

int
copyfs_encode_buffers(char *hidden_pages_data, /* A PAGE_SIZE buffer
                                                   (already allocated)
```

```
                                                passed to us to fill in
                                             */
                      char *page_data, /* The data we are to encode */
                      int *need_to_call,
                      unsigned to, /* how much to encode in page_data */
                      inode_t *inode, /* The inode in question */
                      vfs_t *vfs, /* vfs object */
                      void **opaque)    /* Opaque data */
     /* encodes the data in page_data into hidden_pages_data. Returns
        -errno for error, and the size of hidden_pages_data for success */
{
  print_entry_location();
  fistMemCpy(hidden_pages_data, page_data, to);
  *need_to_call = 0;
  print_exit_status(to);
  return(to);
}


int
copyfs_decode_buffers(int num_hidden_pages, /* The number of pages in
                                               hidden_pages_data */
                      char **hidden_pages_data, /* An array of pages
                                                   containing encoded
                                                   data */
                      char *page_data, /* A pre-allocated PAGE_SIZE
                                          buffer to write the decoded
                                          data to */
                      inode_t *inode,  /* The inode of the file in
                                          question */
                      vfs_t *vfs,      /* vfs object */
                      void *opaque)      /* opaque data filled in by
                                            wrapfs_Sca_count_pages,
                                            containing an int, the
                                            starting offset within the
                                            first page of hidden_pages_data
                                            of the encoded page we want
                                         */
     /* Returns -errno for error, or the number of bytes decoded on success
        (Should usually return PAGE_SIZE bytes) */
{
  int err = 0, tmp = 0;

  if (num_hidden_pages != 1) {
    printk("copyfs: Too many pages! (%d)\n",num_hidden_pages);
```

```
    err = -EIO;
    goto out;
  }

  tmp = (int) opaque;
  err = PAGE_CACHE_SIZE - tmp;
  fistMemCpy(page_data,&(hidden_pages_data[0][tmp]), err);

 out:
  print_exit_status(err);
  return(err);
}
```

# Appendix C

# Vnode Interface Tutorial

This section provides a simple introduction to the vnode interface. The information herein is gathered from pivotal papers on the subject [38, 64] and from system C header Files—specifically `<sys/vfs.h>` and `<sys/vnode.h>`. The vnode interface described herein is based on the Solaris $2.x$ operating system. Key differences between Solaris and FreeBSD or Linux are described in Section C.6.

The two important data structures used in the vnode interface are `struct vfs` and `struct vnode`, depicted in Figures C.1 and C.5, respectively.

## C.1 `struct vfs`

An instance of the `vfs` structure exists in a running kernel for each mounted file system. All of these instances are chained together in a singly linked list. The head of the list is a global variable called `root_vp`, which contains the vfs for the root device. The field `vfs_next` links one vfs structure to the following one in the list.

```
typedef struct vfs {
  struct vfs      *vfs_next;            /* next VFS in VFS list */
  struct vfsops   *vfs_op;              /* operations on VFS */
  struct vnode    *vfs_vnodecovered;    /* vnode mounted on */
  u_long          vfs_flag;             /* flags */
  u_long          vfs_bsize;            /* native block size */
  int             vfs_fstype;           /* file system type index */
  fsid_t          vfs_fsid;             /* file system id */
  caddr_t         vfs_data;             /* private data */
  dev_t           vfs_dev;              /* device of mounted VFS */
  u_long          vfs_bcount;           /* I/O count (accounting) */
  u_short         vfs_nsubmounts;       /* immediate sub-mount count */
  struct vfs      *vfs_list;            /* sync list pointer */
  struct vfs      *vfs_hash;            /* hash list pointer */
  kmutex_t        vfs_reflock;          /* mount/unmount/sync lock */
} vfs_t;
```

Figure C.1: Solaris 2.x VFS Interface

The fields relevant to this proposal are as follows:

- `vfs_next` is a pointer to the next vfs in the linked list.
- `vfs_op` is a pointer to a function-pointer table. That is, this `vfs_op` can hold pointers to UFS functions, NFS, PCFS, HSFS, etc. For example, if the vnode interface calls the function to mount the file system, it will call whatever subfield of `struct vfsops` (See Section C.2) is designated for the mount function. That is how the transition from the vnode level to a file-system–specific level is made.
- `vfs_vnodecovered` is the vnode on which this file system is mounted (the mount point).
- `vfs_flag` contains bit flags for characteristics such as whether this file system is mounted read-only, if the setuid/setgid bits should be turned off when exec-ing a new process, if sub-mounts are allowed, etc.
- `vfs_data` is a pointer to opaque data specific to this vfs and the type of file system this one is. For an NFS vfs, this would be a pointer to `struct mntinfo` (located in `<nfs/nfs_clnt.h>`)—a large NFS-specific structure containing such information as the NFS mount options, NFS read and write sizes, host name, attribute cache limits, whether the remote server is down or not, and more.
- `vfs_reflock` is a mutual exclusion variable used by locking functions that need to change values of certain fields in the vfs structure.

## C.2 `struct vfsops`

The vfs operations structure (`struct vfsops`, seen in Figure C.2) is constant for each type of file system. For every instance of a file system, the vfs field `vfs_op` is set to the pointer of the operations vector of the underlying file system.

```
typedef struct vfsops {
  int     (*vfs_mount)();
  int     (*vfs_unmount)();
  int     (*vfs_root)();
  int     (*vfs_statvfs)();
  int     (*vfs_sync)();
  int     (*vfs_vget)();
  int     (*vfs_mountroot)();
  int     (*vfs_swapvp)();
} vfsops_t;
```

Figure C.2: Solaris 2.x VFS Operations Interface

Each field of the structure is assigned a pointer to a function that implements a particular operation for the file system in question:

- `vfs_mount` is the function to mount a file system on a particular vnode. It is responsible for initializing data structures, and filling in the vfs structure with all the relevant information (such as the `vfs_data` field).
- `vfs_unmount` is the function to release this file system, or unmount it. It is the one, for example, responsible for detecting that a file system has still opened resources that cannot be released, and for returning an *errno* code that results in the user process getting a “device busy” error.

- `vfs_root` will return the root vnode of this file system. Each file system has a root vnode from which traversal to all other vnodes in the file system is enabled. This vnode usually is hand crafted (via `kernel_malloc`) and not created as part of the standard ways of creating new vnodes (i.e. `vn_lookup`).
- `vfs_statvfs` is used by programs such `df` to return the resource usage status of this file system (number of used/free blocks/inodes).
- `vfs_sync` is called successively in every file system when the `sync(2)` system call is invoked, to flush in-memory buffers onto persistent media.
- `vfs_vget` turns a unique file identifier *fid* for a vnode into the vnode representing this file. This call works in conjunction with the vnode operation `vop_fid`, described in Appendix section C.4.
- `vfs_mountroot` is used to mount this file system as the root (first) file system on this host. It is different from `vfs_mount` because it is the first one, and therefore many resources such as `root_vp` do not yet exist. This function has to manually create and initialize all of these resources.
- `vfs_swapvp` returns a vnode specific to a particular device onto which the system can swap. It is used for example when adding a file as a virtual swap device via the `swap -a` command [70].

The VFS operations get invoked transparently via macros that dereference the operations vector's field for that operation, and pass along the vfs and the arguments it needs. Each VFS operation has a macro associated with it, located in `<sys/vfs.h>`. Figure C.3 shows the definitions for these macros.

```
#define VFS_MOUNT(vfsp, mvp, uap, cr) (*(vfsp)->vfs_op->vfs_mount)(vfsp, mvp, uap, cr)
#define VFS_UNMOUNT(vfsp, cr)         (*(vfsp)->vfs_op->vfs_unmount)(vfsp, cr)
#define VFS_ROOT(vfsp, vpp)           (*(vfsp)->vfs_op->vfs_root)(vfsp, vpp)
#define VFS_STATVFS(vfsp, sp)         (*(vfsp)->vfs_op->vfs_statvfs)(vfsp, sp)
#define VFS_SYNC(vfsp, flag, cr)      (*(vfsp)->vfs_op->vfs_sync)(vfsp, flag, cr)
#define VFS_VGET(vfsp, vpp, fidp)     (*(vfsp)->vfs_op->vfs_vget)(vfsp, vpp, fidp)
#define VFS_MOUNTROOT(vfsp, init)     (*(vfsp)->vfs_op->vfs_mountroot)(vfsp, init)
#define VFS_SWAPVP(vfsp, vpp, nm)     (*(vfsp)->vfs_op->vfs_swapvp)(vfsp, vpp, nm)
```

Figure C.3: VFS Macros

When any piece of file-system code, that has a handle on a vfs, wants to call a vfs operation on that vfs, they simply dereference the macro, as depicted in Figure C.4.

```
int foo(const vfs_t *vfsp, vnode_t **vpp)
{
  int error;

  error = VFS_ROOT(vfsp, vpp);
  if (error)
    return (error);
}
```

Figure C.4: VFS Macros Usage Example

# C.3 `struct vnode`

An instance of `struct vnode` (Figure C.5) exists in a running system for every opened (in-use) file, directory, symbolic-link, hard-link, block or character device, a socket, a Unix pipe, etc.

```
typedef struct vnode {
  kmutex_t         v_lock;                  /* protects vnode fields */
  u_short          v_flag;                  /* vnode flags (see below) */
  u_long           v_count;                 /* reference count */
  struct vfs       *v_vfsmountedhere;       /* ptr to vfs mounted here */
  struct vnodeops  *v_op;                   /* vnode operations */
  struct vfs       *v_vfsp;                 /* ptr to containing VFS */
  struct stdata    *v_stream;               /* associated stream */
  struct page      *v_pages;                /* vnode pages list */
  enum vtype       v_type;                  /* vnode type */
  dev_t            v_rdev;                  /* device (VCHR, VBLK) */
  caddr_t          v_data;                  /* private data for fs */
  struct filock    *v_filocks;              /* ptr to filock list */
  kcondvar_t       v_cv;                    /* synchronize locking */
} vnode_t;
```

Figure C.5: Solaris 2.x Vnode Interface

Structure fields relevant to our work are:

- `v_lock` is a mutual exclusion variable used by locking functions that need to perform changes to values of certain fields in the vnode structure.
- `v_flag` contains bit flags for characteristics such as whether this vnode is the root of its file system, if it has a shared or exclusive lock, whether pages should be cached, if it is a swap device, etc.
- `v_count` is incremented each time a new process opens the same vnode.
- `v_vfsmountedhere`, if non-null, contains a pointer to the vfs that is mounted on this vnode. This vnode thus is a directory that is a mount point for a mounted file system.
- `v_op` is a pointer to a function-pointer table. That is, this `v_op` can hold pointers to UFS functions, NFS, PCFS, HSFS, etc. For example, if the vnode interface calls the function to open a file, it will call whatever subfield of `struct vnodeops` (See Section C.4) is designated for the open function. That is how the transition from the vnode level to a file-system–specific level is made.
- `v_vfsp` is a pointer to the vfs that this vnode belongs to. If the value of the field `v_vfsmountedhere` is non-null, it is also said that `v_vfsp` is the parent file system of the one mounted here.
- `v_type` is used to distinguish between a regular file, a directory, a symbolic link, a block/character device, a socket, a Unix pipe (fifo), etc.
- `v_data` is a pointer to opaque data specific to this vnode. For an NFS vfs, this might be a pointer to `struct rnode` (located in `<nfs/rnode.h>`)—a remote file system-specific structure containing such information as the file-handle, owner, user credentials, file size (from the client's view), and more.

## C.4 struct vnodeops

An instance of the vnode operations structure (`struct vnodeops`, listed in Figure C.6) exists for each different type of file system. For each vnode, the vnode field `v_op` is set to the pointer of the operations vector of the underlying file system.

```
typedef struct vnodeops {
  int      (*vop_open)();
  int      (*vop_close)();
  int      (*vop_read)();
  int      (*vop_write)();
  int      (*vop_ioctl)();
  int      (*vop_setfl)();
  int      (*vop_getattr)();
  int      (*vop_setattr)();
  int      (*vop_access)();
  int      (*vop_lookup)();
  int      (*vop_create)();
  int      (*vop_remove)();
  int      (*vop_link)();
  int      (*vop_rename)();
  int      (*vop_mkdir)();
  int      (*vop_rmdir)();
  int      (*vop_readdir)();
  int      (*vop_symlink)();
  int      (*vop_readlink)();
  int      (*vop_fsync)();
  void     (*vop_inactive)();
  int      (*vop_fid)();
  void     (*vop_rwlock)();
  void     (*vop_rwunlock)();
  int      (*vop_seek)();
  int      (*vop_cmp)();
  int      (*vop_frlock)();
  int      (*vop_space)();
  int      (*vop_realvp)();
  int      (*vop_getpage)();
  int      (*vop_putpage)();
  int      (*vop_map)();
  int      (*vop_addmap)();
  int      (*vop_delmap)();
  int      (*vop_poll)();
  int      (*vop_dump)();
  int      (*vop_pathconf)();
  int      (*vop_pageio)();
  int      (*vop_dumpctl)();
  void     (*vop_dispose)();
  int      (*vop_setsecattr)();
  int      (*vop_getsecattr)();
} vnodeops_t;
```

Figure C.6: Solaris 2.x Vnode Operations Interface

Each field of the structure is assigned a pointer to a function that implements a particular operation on the file system in question:

- `vop_open` opens the requested file and returns a new vnode for it.
- `vop_close` closes a file.
- `vop_read` reads data from the opened file.
- `vop_write` writes data to the file.
- `vop_ioctl` performs miscellaneous I/O control operations on the file, such as setting non-blocking I/O access.
- `vop_setfl` is used to set arbitrary file flags.
- `vop_getattr` gets the attributes of a file, such as the mode bits, user and group ownership, etc.
- `vop_setattr` sets the attributes of a file.
- `vop_access` checks to see if a particular user, given the user's credentials, is allowed to access a file.
- `vop_lookup` looks up a directory for a file name. If found, a new vnode is returned.

- `vop_create` creates a new file.
- `vop_remove` removes a file from the file system.
- `vop_link` makes a hard-link to an existing file.
- `vop_rename` renames a file.
- `vop_mkdir` makes a new directory.
- `vop_rmdir` removes an existing directory.
- `vop_readdir` reads a directory for entries within.
- `vop_symlink` creates a symbolic-link to a file.
- `vop_readlink` reads the value of a symbolic link, that is, what the link points to.
- `vop_fsync` writes out all cached information for a file.
- `vop_inactive` signifies to the vnode layer that this file is no longer in use, that all its references had been released, and that it can now we deallocated.
- `vop_fid` returns a unique file identifier *fid* for a vnode. This call works in conjunction with the vfs operation `vfs_vget` described in Appendix section C.2.
- `vop_rwlock` locks a file before attempting to read from or write to it.
- `vop_rwunlock` unlocks a file after having read from or wrote to it.
- `vop_seek` sets the read/write head to a particular point within a file, so the next read/write call can work from that location in the file.
- `vop_cmp` compares two vnodes and returns true/false.
- `vop_frlock` perform file and record locking on a file.
- `vop_space` frees any storage space associated with this file.
- `vop_realvp` for certain file systems, returns the "real" vnode. This is useful in stackable vnodes, where a higher layer may request the real/hidden vnode underneath, so it can operate on it.
- `vop_getpage` reads a page of a memory-mapped file.
- `vop_putpage` writes to a page of a memory-mapped file.
- `vop_map` maps a file into memory. See [25, 26] for more details.
- `vop_addmap` adds more pages to a memory-mapped file.
- `vop_delmap` removes some pages from a memory-mapped file.
- `vop_poll` polls for events on the file. This is mostly useful when the vnode is of type "socket" or "fifo," and replaces the older `vop_select` vnode operation. This operation is often used to implement the `select(2)` system call.
- `vop_dump` dumps the state of the kernel (memory buffers, tables, variables, registers, etc.) to a given vnode, usually a swap device. This is used as the last action performed when a kernel panics and needs to save state for post-mortem recovery by tools such as `crash` [72].
- `vop_pathconf` supports the POSIX path configuration standard. This call returns various configurable file or directory variables.
- `vop_pageio` performs I/O directly on mapped pages of a file.

- `vop_dumpctl` works in conjunction with `vop_dump`. It is used to prepare a file system before a dump operation by storing data structures that might otherwise get corrupted shortly after a panic had occurred, and deallocates these private dump data structures after a successful dump.
- `vop_dispose` removes a mapped page from memory.
- `vop_setsecattr` is used to set Access Control Lists (ACLs) on a file.
- `vop_getsecattr` is used to retrieve the ACLs of a file.

Vnode operations get invoked transparently via macros that dereference the operations vector's field for that operation, and pass along the vnode and the arguments it needs. Each vnode operation has a macro associated with it, located in `<sys/vnode.h>`. Figure C.7 shows as an example, the definitions for some of these calls.

```
#define VOP_OPEN(vpp, mode, cr)         (*(*(vpp))->v_op->vop_open)(vpp, mode, cr)
#define VOP_CLOSE(vp, f, c, o, cr)      (*(vp)->v_op->vop_close)(vp, f, c, o, cr)
#define VOP_READ(vp, uiop, iof, cr)     (*(vp)->v_op->vop_read)(vp, uiop, iof, cr)
#define VOP_MKDIR(dp, p, vap, vpp, cr)  (*(dp)->v_op->vop_mkdir)(dp, p, vap, vpp, cr)
#define VOP_GETATTR(vp, vap, f, cr)     (*(vp)->v_op->vop_getattr)(vp, vap, f, cr)
#define VOP_LOOKUP(vp, cp, vpp, pnp, f, rdir, cr) \
                            (*(vp)->v_op->vop_lookup)(vp, cp, vpp, pnp, f, rdir, cr)
#define VOP_CREATE(dvp, p, vap, ex, mode, vpp, cr) \
                          (*(dvp)->v_op->vop_create)(dvp, p, vap, ex, mode, vpp, cr)
```

Figure C.7: Some Vnode Macros

When any piece of file-system code, that has a handle on a vnode, wants to call a vnode operation on it, it simply dereferences the macro, as depicted in Figure C.8.

```
int foo(vnode_t *dp, char *name,
        vattr_t *vap, vnode_t **vpp, cred_t *cr)
{
  int error;

  error = VOP_MKDIR(dp, name, vap, vpp, cr);
  if (error)
    return (error);
}
```

Figure C.8: Vnode Macros Usage Example

## C.5 How It All Fits

To see how it all fits in, the following example depicts what happens when a remote (NFS) file system is mounted onto a local (UFS) file system, and the sequence of operations that a user-level process goes through to satisfy a simple read of a file on the mounted file system.

## C.5.1 Mounting

Consider first the two file systems X and Y, depicted in Figure C.9. In this figure, the numbers near the node names represent the file/inode/vnode numbers of that file or directory within that particular file system. For example "X5" refers to the vnode of the directory `/usr/local` on file system X.

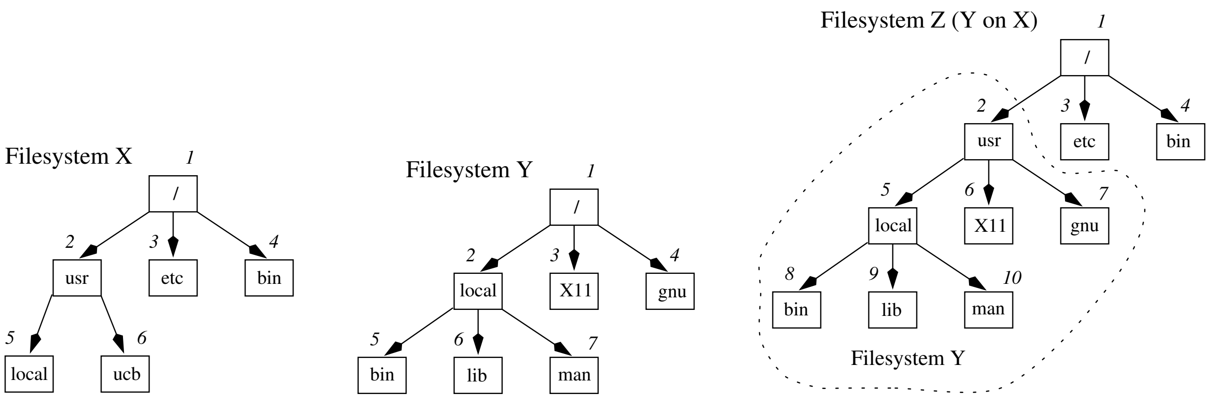



Figure C.9: File-System Z as Y mounted on X

Let's also assume that X is a UFS (local) file system, and that Y is the `/usr` file system available on a remote file server named "titan." We wish to perform the following NFS mount action: `mount titan:/usr /usr`.

The in-kernel actions that proceed, assuming that all export and mount permissions are successful, are the following:

1. A new vfs is created and is passed on to `nfs_mount`.
2. `nfs_mount` fills in the new vfs structure with the vfs operations structure for NFS, and sets the `v_vfsmountedhere` of the vnode X2 to this new vfs.
3. `nfs_mount` also creates a new vnode to serve as the root vnode of the Y file system as mounted on X. It stores this vnode in the `v_data` field of the new vfs structure.

## C.5.2 Path Traversal

Figure C.9 also shows the new structure of file system X, after Y had been mounted, as file system Z.

The sequence of in-kernel operations to, say, read the file `/usr/local/bin/tex` would be as follows:

1. The system call `read()` is executed. It begins by looking up the file.
2. The generic lookup function performs a `VOP_LOOKUP(rootvp, "usr")`. It tries to look for the next component in the path, starting from the current lookup directory (root vnode).
3. The lookup function is translated into `ufs_lookup`. The vnode X2 is found. Note that X2 is *not* the same vnode as Z2! X2 is hidden, while Z2 overshadows it.
4. The lookup function now notices that X2's `v_vfsmountedhere` field is non-null, so it knows that X2 is a mount point. It calls the `VOP_ROOT` function on the vfs that is "mounted here," that translates to `nfs_lookup`. This function returns the root vnode of the Y file system as it is mounted on X. This root vnode is X2. The "magic" part that happens at this point is that the lookup routine now resumes its path traversal but on the *mounted* file system.

5. An `nfs_lookup` is performed on the Z2 vnode for the component `"local"`, that will return the vnode Z5.
6. An NFS lookup is performed on vnode Z5 for the component `"bin"`, that will return the vnode Z8.
7. An NFS lookup is performed on vnode Z8 for the component `"tex"`, that will return the vnode for the file.
8. The lookup is complete and returns the newly found vnode for component `"tex"` to the `read()` system call.
9. The generic read function performs a `VOP_READ` on the newly found vnode. Since that vnode is an NFS one, the read is translated into `nfs_read`.
10. Actual reading of the file `/usr/local/bin/tex` begins in earnest.

## C.6 FreeBSD and Linux Vnode Interfaces

The FreeBSD vnode interface is very similar to Solaris's. It also has two main data structures. The per-file data structure is also called `struct vnode` and the per-file system data structure is called `struct mount`. FreeBSD has nearly identical file and file-system methods with very similar names: getattr, read, write, getpage, putpage, etc.

The Linux vnode interface, on the other hand, is rather different than both Solaris and FreeBSD. This is because it was written from scratch and designed to support dozens of different file systems. The Linux VFS is therefore more flexible but also more complex. Primarily, Linux has more data structures in the VFS, as we described in Section 7.1.3.2: `super_block` representing per-file system information such as its size; `inode` representing file data that is on disk such as the file's owner; `file` representing information about opened files such as the flags the file was opened with; `dentry` representing information about directory entries such as their name and directory caching information; and more. Although the internals of the Linux VFS are different, it does contain the similar methods that apply on file and file-system objects: mkdir, unlink, symlink, statfs, open, close, read, write, write_page, and so on.