Exhibit 21

On Incremental File System Development

EREZ ZADOK, RAKESH IYER, NIKOLAI JOUKOV, GOPALAN SIVATHANU, AND CHARLES P. WRIGHT

Developing file systems from scratch is difficult and error prone. Layered, or stackable, file systems are a powerful technique to incrementally extend the functionality of existing file systems on commodity OSes at runtime. In this paper, we analyze the evolution of layering from historical models to what is found in four different present day commodity OSes: Solaris, FreeBSD, Linux, and Microsoft Windows. We classify layered file systems into five types based on their functionality and identify the requirements that each class imposes on the OS. We then present five major design issues that we encountered during our experience of developing over twenty layered file systems on four OSes. We discuss how we have addressed each of these issues on current OSes, and present insights into useful OS and VFS features that would provide future developers more versatile solutions for incremental file system development.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.4.3 [**Operating Systems**]: File Systems Management—*File organization*; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design

Additional Key Words and Phrases: Layered File Systems, Stackable File Systems, VFS, Vnode, I/O Manager, IRP, Extensibility

1. INTRODUCTION

Data management is a fundamental facility provided by the operating system (OS). File systems are tasked with the bulk of data management, including storing data on disk (or over the network) and naming (i.e., translating a user-visible name such as `/usr/src` into an on-disk object). File systems are complex, and it is difficult to enhance them. Furthermore, OS vendors are reluctant to make major changes to a file system, because file system bugs have the potential to corrupt all data on a machine. Because file system development is so difficult, extending file system functionality in an incremental manner is valuable. Incremental development also makes it possible for a third-party software developer to release file system improvements, without developing a whole file system from scratch.

Originally, file systems were thoroughly integrated into the OS, and system calls directly invoked file system methods. This architecture made it difficult to add multiple file systems. The introduction of a *virtual node* or *vnode* provided a layer of abstraction that separates the core of the OS from file systems [Kleiman 1986]. Each file is represented in

memory by a vnode. A vnode has an operations vector that defines several operations that the OS can call, thereby allowing the OS to add and remove types of file systems at runtime. Most current OSes use something similar to the vnode interface, and the number of file systems supported by the OS has grown accordingly. For example, Linux 2.6 supports over 30 file systems and many more are maintained outside of the official kernel tree.

Clearly defining the interface between the OS and file systems makes interposition possible. A *layered*, or *stackable*, file system creates a vnode with its own operations vector to be interposed on another vnode. Each time one of the layered file system's operations is invoked, the layered file system maps its own vnode to a lower-level vnode, and then calls the lower-level vnode's operation. To add functionality, the layered file system can perform additional operations before or after the lower-level operation (e.g., encrypting data before a `write` or decrypting data after a `read`). The key advantage of layered file systems is that they can change the functionality of a commodity OS at runtime so hard-to-develop lower-level file systems do not need to be changed. This is important, because OS developers often resist change, especially to file systems where bugs can cause data loss.

Rosenthal was among the first to propose layering as a method of extending file systems [Rosenthal 1990; 1992]. To enable layering, Rosenthal radically changed the VFS internals of SunOS. Each public vnode field was converted into a method; and all knowledge of vnode types (e.g., directory vs. regular file) was removed from the core OS. Researchers at UCLA independently developed another layering infrastructure [Heidemann and Popek 1991; 1994] that placed an emphasis on light-weight layers and extensibility. The original pioneers of layering envisioned creating building blocks that could be composed together to create more sophisticated and rich file systems. For example, the directory-name lookup cache (DNLC) could simply be implemented as a file system layer, which returns results on a cache hit, but passes operations down on a miss [Skinner and Wong 1993].

Layering has not commonly been used to create and compose building-block file systems, but instead has been widely used to add functionality rapidly and portably to existing file systems. Many applications of layered file system are features that could be implemented as part of the VFS (e.g., unification), but for practical reasons it is easier to develop them as layered file systems. Several OSes have been designed to support layered file systems, including Solaris, FreeBSD, and Windows. Several layered file systems are available for Linux, even though it was not originally designed to support them. Many users use layered file systems unknowingly as part of Antivirus solutions [Symantec 2004; Miretskiy et al. 2004], and Windows XP's system restore feature [Harder 2001]. On Unix, a null-layer file system is used to provide support for accessing one directory through multiple paths. When the layer additionally modifies the data, useful new functionality like encryption [Corner and Noble 2002; Halcrow 2004] or compression [Zadok et al. 2001] can be added. Another class of layered file systems, called *fan out*, operates directly on top of several lower-level file systems. For example, unification file systems merge the contents of several directories [Pendry and McKusick 1995; Wright et al. 2006]. Fanout file systems can also be used for replication, load-balancing, failover, snapshotting, and caching.

The authors of this paper have over fifteen years of combined experience developing layered file systems on four OSes: Solaris, FreeBSD, Linux, and Windows. We have developed more than twenty layered file systems that provide encryption, compression, versioning, tracing, antivirus, unification, snapshotting, replication, checksumming, and more.

The rest of this paper is organized as follows. In Section 2 we survey alternative techniques to enhance file system functionality. In Section 3 we describe four models of layered file system development. We then proceed to describe five broad classes of layered file systems in Section 4. In Section 5 we describe five general problems and their solutions that are useful for all types of layered file systems. We conclude in Section 6 with guiding principles for future OS and layered file system developers.

2. RELATED WORK

In this section we describe alternatives to achieve the extensibility offered by layered file systems. We discuss four classes of related works based on the level at which extensibility is achieved: in hardware, in the device driver, at the system-call level, or in user-level programs. We have a detailed discussion of layered file system infrastructures in Section 3.

Hardware level. Slice [Anderson et al. 2000] is a storage system architecture for high speed networks with network-attached block storage. Slice interposes a piece of code called a *switching filter* in the network hardware to route packets among a group of servers. Slice appears to the upper level as a single block-oriented storage device. High-level control information (e.g., files) is unavailable to interposition code at the hardware level, and therefore cannot perform optimizations for specific devices.

Semantically-Smart Disk Systems (SDSs) [Sivathanu et al. 2003] attempt to provide file-system–like functionality without modifying the file system. Knowledge of a specific file system is embedded into the storage device, and the device provides additional functionality that would traditionally be implemented in the file system. Such systems are relatively easy to deploy, because they do not require modifications to existing file system code. Layered file systems share a similar goal in terms of reusing and leveraging existing infrastructures. Unlike a layered file system, an SDS is closely tied to the format of the file system running on top of it, so porting SDSs to new file systems is difficult.

Device-driver level. Software RAID and Logical Volume Managers (LVMs) introduce another layer of abstraction between the storage hardware and the file system. They provide additional features such as increased reliability and performance, while appearing to the file system as a simple disk, which makes them easy to deploy in existing infrastructure. For example, on Linux a Cryptoloop devices uses a loopback block driver to encrypt data stored on a disk or in a file. A new file system is then created within the Cryptoloop device. Any file system can run on top of a block device-driver extension. However, block device extensions cannot exploit the control information (e.g., names) that is available at the file system level.

System-call level. SLIC [Ghormley et al. 1998] is a protected extensibility system for OSes that uses interposition techniques to enable the addition of a large class of untrusted extensions to existing code. Several OS extensions can be implemented using SLIC such as encryption file systems and a protected environment for binary execution. The Interposition Agents toolkit [Jones 1993], developed by Microsoft Research, allows a user's code to be written in terms of high-level objects provided by this interface. The toolkit was designed to ease interposing code between the clients and the instances of the system interface to facilitate adding new functionality like file reference tracing, customizable file system views, etc. to existing systems. Similarly, Mediating Connectors [Balzer and Goldman 1999] is a system call (and library call) wrapper mechanism for Windows NT that

allows users to trap API calls.

System call interposition techniques rely on the communication channel between user-space and the kernel, and hence cannot handle operations that bypass that channel (e.g., `mmap` operations and their associated page faults). Also, interposing at the system call level results in overhead for all system calls even if only a subset of kernel components (e.g., the file system) need to be interposed.

User level. Gray-box Information and Control Layers (ICL) [Arpaci-Dusseau and Arpaci-Dusseau 2001] extend the functionality of OSes by acquiring information about their internal state. ICLs provide OS-like functionality without modifying existing OS code. Dust [Burnett et al. 2002] is a direct application of ICLs that uses gray-box knowledge of the OS's buffer cache management policy to optimize application performance. For example, if a Web server first services Web pages that are believed to be in the OS buffer cache, then both average response time and throughput can be improved.

Blaze's CFS is a cryptographic file system that is implemented as a user-level NFS server [Blaze 1993]. The OS's unmodified NFS client mounts the NFS server over the loopback network interface. The SFS toolkit [Maziéres 2001] aims to simplify Unix file system extensibility by allowing development of file systems at the user level. Using the toolkit, one can implement a simple user-level NFS server and redirect local file system operations into the user level implementation. The popularity of the SFS toolkit demonstrates that developers have observed the complexity of modifying existing time-tested file systems. SiRiUS [Goh et al. 2003], a file system for securing remote untrusted storage, and Dabek's CFS [Dabek et al. 2001], a wide area cooperative file system, were built using the SFS toolkit.

Filesystem in Userspace [Szeredi 2005], or FUSE, is a hybrid approach that consists of two parts: (1) a standard kernel-level file system which passes calls to a user-level demon, and (2) a library to easily develop file-system–specific FUSE demons. Developing new file systems with FUSE is relatively simple because the user-level demon can issue normal system calls (e.g., `read`) to service a VFS call (e.g., `vfs_read`). The main two disadvantages of a FUSE file system are that (1) performance is limited by crossing the user-kernel boundary, and (2) the file system can only use FUSE's API, which closely matches the VFS API, whereas kernel file systems may access a richer kernel API (e.g., for process and memory management).

Sun designed and developed Spring as an object-oriented microkernel OS. Spring's architecture allowed various components, including file systems, to be transparently extended with user libraries [Khalidi and Nelson 1993]. Spring's design was radically different from current commodity OSes. As it was research prototype, it was not deployed in real systems. K42 [Appavoo et al. 2002] is a new OS under development at IBM which incorporates innovative mechanisms and modern programming technologies. Various system functionalities can be extended at the user level through libraries by providing a microkernel-like interface. The Exokernel architecture [Engler et al. 1995] implements minimal components in the kernel and allows user-level applications to customize OS functionality using library OSes.

In general, user-level extensions are easy to implement, but their performance is not as good as kernel extensions because the former involve data copies between the user level and the kernel level, as well as additional context switches.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.