University of California
Los Angeles

Stackable Layers:
An Architecture for File System Development

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

John Shelby Heidemann

1991

This document is available as Technical Report CSD-910056 of the UCLA Computer Science Department. A PostScript version is available via anonymous ftp to `ftp.cs.ucla.edu` as `pub/ficus/ucla_csd_910056.ps`. The author can be reached by electronic mail by `johnh@cs.ucla.edu`.

This document was last changed August 8, 1991.

The thesis of John Shelby Heidemann is approved.

Rajive Bagrodia

Richard R. Muntz

Wesley W. Chu, Committee Co-Chair

Gerald J. Popek, Committee Co-Chair

University of California, Los Angeles

1991

To my family—
my mother Dorothy
and my brother Ben

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possible without the support of my advisor, Dr. Gerald Popek, to whom I am most grateful. I would also like to thank the other members of the Ficus Project at UCLA. In particular I would like to thank Richard Guy for the distributed garbage collection algorithms which prompted the construction of a new file system, Tom Page for a careful reading of a draft of this thesis, and Dieter Rothmeier, Wai Mak, and Yuguang Wu for being the first to use the new interface.

I also would like to gratefully acknowledge the support of the Defense Advanced Research Projects Agency and the USENIX Association for their financial support. In the course of this research I was supported by DARPA contracts F29601-87-C-0072 and N00174-91-C-0107, and the 1990-91 USENIX Graduate Research Scholarship.

Abstract of the Thesis

Stackable Layers:
An Architecture for File System Development

by

John Shelby Heidemann
Master of Science in Computer Science
University of California, Los Angeles, 1991
Professor Gerald J. Popek, Co-Chair
Professor Wesley W. Chu, Co-Chair

This thesis proposes the *stackable layers* method of file system design. This approach constructs file systems from a number of independently developed *layers*. Each layer is bounded by a symmetric interface, syntactically identical above and below. Layers combine in *stacks*, linear or tree-shaped collections, each layer building on the functionality of those beneath it.

Stackable filing improves file system development in several ways. Stacking encourages code re-use by building upon already existing layers. Incremental improvement is possible by substitution of existing layers. The layer interface is extensible, allowing new operations to be easily added by third-parties. Each operation is carefully described, permitting existing layers to adjust automatically to the addition of new operations.

The feasibility of stackable filing is demonstrated by the development of a prototype layer interface and several file system layers. The performance of multi-layer stacks is found comparable to that of monolithic file systems. Through the re-use of existing services, we find development of new filing services with stackable layers significantly easier than development with traditional methods.

CHAPTER 1

Introduction

1.1 Introduction

Modularity is widely recognized as a necessary tool in the management of large software systems. By dividing software into small, easily managed pieces, modularity provides advantages in organization and verification throughout the software lifespan. Modularity is defined by separate software components (modules) joined by well defined interfaces.

When modular interfaces are carefully documented and published, they can also serve as an important tool for compatibility and future development. By providing a common protocol between two subsystems, such an interface allows either or both systems to be replaced without change to the other. Improved modules can therefore be independently developed and added as desired, improving the computing environment. Interfaces such as Posix.1 [IEE90] and NFS [SGK+85] are examples of interfaces widely used to provide operating system and remote filing services.

Because operating systems represent such a widely used service, the development of modular systems interfaces there is particularly important. The portability of application programs utilizing the Posix interface illustrates the utility of a successful interface. Such programs can execute on the complete range of today's hardware, from personal computers to the largest supercomputer.

One would like to see this same level of portability currently present for application programs in operating systems themselves. Large portions of an operating system are hardware independent and should run equally well on any computer. Such portability has been largely achieved, as exemplified by portable operating systems such as Unix [RT74].

What has not been achieved to the same extent is portability of major kernel subsystems. Because of the exacting nature of software, and because of the lack of modular interfaces within the operating system itself, the Unix kernel has been slow to evolve to new software technologies. While individual vendors have adopted new kernel technologies such as Streams [Rit84], new virtual memory approaches, and new file systems, such additions have only come slowly and at great expense.

Micro-kernel designs are one approach to kernel modularity. Kernels such as Mach [ABG+86] and Chorus [RAA+90] divide the operating system into two parts: a core of memory management, process control, and simple inter-process communication; and a server (or servers) supporting the remainder of the traditional operating system, including accounting, protection, file system and network services, and backwards compatibility. For the case of Mach and Unix, as a figure of merit, the core is on the order of 15% of the total operating system kernel. This intra-kernel boundary is an important structuring tool, particularly because it offers a platform on top of which third parties can offer a variety of services. But this approach does not provide a total solution, as it fails to address the modularity of the remaining 85% of the system.

File systems, a rich portion of the remaining kernel, are an active area of research. Many file system services have been proposed, including version management, user-customizable naming, fast log-structured storage, replication, and large-scale distributed filing. All of these have well developed prototypes, but appearance in commercially available systems has been both slow and piecemeal.

Adoption of these new filing services has been slow in part because file systems are large, monolithic pieces of code with limited internal modularity. Although recent approaches to file system modularity such as Sun's VFS interface [Kle86] allow easy substitution of *entire* file systems, they do little to support modularity within file systems themselves. As a result, it is not easy to replace or enhance separate portions of the file system, for example, keeping the physical disk management and installing a new directory layer.

Another problem with existing approaches to file system modularity is that they are particularly fragile in the face of change, the very capacity modularity should support. Evolution of the kernel to more efficient algorithms, and addition of new file systems have required frequent changes to the interface, resulting in incompatibility between operating system releases. Frequent change and incompatibility has largely discouraged third-party innovation, restricting introduction of new filing services to the primary operating system vendors alone. This contrasts sharply with other operating system interfaces such as the device interface and graphical user interfaces, where standard interfaces have allowed competition and rapid development of a wide array of services.

A better solution to filing service design is needed. Perhaps techniques used in the structuring of other large software systems can be applied to this context.

UNIX shell programming is one example of a successful development environment. Individual programs are easily connected by a flexible, standard interface, the pipe. Programs can be combined quickly and easily in the shell with a simple programming language. New programs are widely and independently developed by a number of vendors. These features combine to provide an excellent environment for rapid prototyping and development.

This approach to software modularity has also been applied to kernel-level subsystems. The STREAMS system is Ritchie's redesign of UNIX terminal and network processing. STREAMS' syntactically identical interface between a number of small, independent modules encourages the construction of protocol *stacks* custom built to handle the task at hand. As a result, third parties have built commercial quality layers that integrate well with other protocol modules. This modular approach allowing multiple, independent groups to contribute to communications facilities is one of the reasons UNIX is the preferred base for networking and distributed systems software in engineering and commercial use[1].

This research seeks to apply the principles of stackable layering to file system development. In particular, we envision a situation where a user's filing environment is composed of a number of independently developed layers. These layers are bounded by extensible interfaces, allowing compatible addition of services. Flexible configuration and a symmetric interface allow experimentation and composition of file system stacks. The central thesis to this work is that this layered environment will dramatically improve the file system development environment, making it easier to provide new filing services to users.

The true test of this hypothesis is the design, construction, and use of a stackably layered file system. This document describes issues involved in the design of such a file system and reports experience in the use of these principles in the development of a stackable file system and layers.

1.2 Related Work

Related material derives primarily from symmetric module design, general file system structuring, and some recent work on stackable file systems. A more detailed study and comparison to this work can be found in Chapter 7.

1.2.1 Symmetric interfaces

UNIX shell programming with pipes [RT74] is now the widest use of a symmetric interface. Pike and Kernighan describe this work for software development [PK84]; other applications are as rich as text formatting [KP84] and music processing [Lan90].

Ritchie's STREAMS paper [Rit84] is the classic application of these principles to kernel structuring. Ritchie applies symmetric layering to the terminal and network subsystems of research versions of AT&T UNIX. Such work has since been adopted by a number of UNIX flavors.

[1] In fact, commercial systems such as Novell's Netware-386 have adopted the Streams framework, presumably for similar reasons.

The *x*-kernel [HP88, HPAO89] is a new kernel designed originally to provide customized network protocols. Using a symmetric interface for all kernel services ("everything is a protocol"), great flexibility in protocol selection and combination is provided. Run-time protocol selection also allows use of the most efficient method available.

1.2.2 File system structuring

Dijkstra describes early approaches to modular operating system design [Dij67, Dij68]. Madnick and Alsop [MA69], and later Madnick and Donovan [MD74] discuss modular and layered approaches to file system design, concluding with a six-layer design. The design of Unix adopted simpler approaches, resulting in a two layer design (file system and physical devices) [Bac86].

To provide for multiple file systems, several "file system switch" mechanisms have been developed [Kle86, RKH86, KM86]. These typically found quick use in the support of network file access [SGK$^+$85, RFH$^+$86] and have since been applied to the support of other file systems [Koe87]. None of these approaches provide explicit support for stacking or extensibility, but all provide basic modularity.

NeFS describes one approach to an extensible file system interface [Sun90], focusing exclusively on remote file access. An alternative to the NFS protocol for remote access, NeFS allows remote execution of PostScript-like programs for file access.

1.2.3 Stackable file systems

Sun Microsystems applied the vnode interface to build two layer file system stacks in their loopback and translucent file systems [Hen90]. Internal to the operating system, stacking is used to support device special files.

More recently, Rosenthal has experimented with a modified vnode interface to provide dynamic file system stacking [Ros90]. This work is discussed extensively in Section 7.2.

1.3 Overview of the Thesis

Central to a layered design is the interface connecting the layers. Chapter 2 discusses the design of such an interface, and the following chapter describes the implementation 405 interface, a stackable interface developed at UCLA. A symmetrically layered file system enables new development techniques; these are discussed in Chapter 4. Chapter 5 illustrates the use of these approaches in several layers developed using the 405 interface. Chapter 6 evaluates the use of our layered file systems, both in terms of run-time costs and development effort. Related work is reviewed in Chapter 7, followed by conclusions and suggestions for future research.

CHAPTER 2

Interface Design

The ultimate success of a file system interface depends on its ability to support and encourage the growth required by future filing environments. From our experiences with stackable file design, we have identified a number of characteristics we feel critical to future development. A flexible, layered interface must be:

Extensible. The interface must be very easy to extend and evolve as requirements change, even in the face of multiple third party changes.

Stackable. Composing file systems from several layers promotes code-reuse and portability. The interface should facilitate stacking. Trees should also be possible, with both layer fan-in and fan-out.

Late bound. Stack composition should be easy and flexible. Stack definition should be possible at run-time without source code changes. New layers should be easy to add.

Fine granularity. Stacks should be configurable at a fine granularity for maximum flexibility. Each file and process should be allowed a different stack and view of stacks. Management of stack configuration should be easy.

Well defined. All characteristics of the interface should be detailed and available at run-time. Information should be sufficient, for example, for layers to move operations between address spaces, and for third party additions to blend seamlessly with existing systems.

Efficient. A file system interface is in the tight loop of computation; performance is critical. Multiple thin layers maximize code reuse, but also require very low overhead.

Rich in flow of control. Interaction of different layers in a distributed file system often requires non-traditional flow of control. The interface should support "upcalls" and a rich flow of control.

Opaque. Layers should be opaque; all access to layer details must be through well-known operations.

Unlimited. There should be no arbitrary limits on the number or kinds of layers and operations.

Each of these characteristics represent different dimensions in layer interface design. The remainder of this chapter examines each of these in turn. The following chapter considers a prototype interface, selecting one point in the design space.

Release	Vnode fields	Vnode bytes	Operation count
SunOS 2.0 (1985)	11	32	24
SunOS 4.0 (1988)	14	40	29
SVR4/no fill (1989)	11	40	37
SVR4/with fill (1989)	19	72	69
Rosenthal's prototype (1990)	6	20	39

Table 2.1: Rosenthal's evaluation of vnode interface evolution in SunOS (from [Ros90]). Fill indicates space left in SVR4 for future expansion; Rosenthal's prototype is discussed in Section 7.2.

2.1 Extensibility

An interface must be extensible, allowing addition of new operations as needed. If a software system is anticipated to have a multi-year lifespan, it can be expected that its surrounding environment will change one or more times. In addition, new features of a software system will often require additions to existing interfaces. Both of these problems are aspects of extensibility.

Examples of these problems abound in software design. Rosenthal examined the gradual evolution of the SunOS file system interface in [Ros90]. He found significant changes to the interface in every major operating system release. Table 2.1 shows his comparison of changes.

Rosenthal's example shows the reality of change in software development. In spite of the inevitability of change, a number of barriers to change exist, making evolution difficult. Often complete source code must be available to permit change; facilities of most operating systems do not support additions or extensions. On the other hand, because of market pressure and desire for new features, change must take place. One approach to minimize the disruption resulting from change is to minimize how often change is allowed. For example, one might disallow local operating system modification, using only infrequent upgrades from the workstation manufacturer. Practically speaking, widely used operating system modifications derive only from a few major vendors and research institutes.

What are the problems that result from this traditional approach to change and change management? Two sorts of problems appear: problems in development without change, and problems of adjusting to change when it eventually arrives.

Great efforts are made to avoid the effects of change. One approach to preventing change is to require that everyone use the same version of software; no change or deviation is permitted. While this approach works for small workgroups for short periods of time, it fails as scale and duration increase. The longer a configuration is frozen, the greater users' demands for new software. As the user population grows from tens of machines to hundreds or thousands, the different goals and requirements of multiple administrative domains force different software configurations.

The reason for change is nearly always support for new functionality. Accommodation of new hardware or software in an environment prohibiting change is very difficult. The more different a new device or package is (and so presumably the more interesting), the less likely it is that support will be possible without change. Consider, for example, write-once optical media. Although they offer the desirable characteristics of very large and persistent storage, there is little hope of exploiting their full potential with unmodified system software. Therefore, if change is to be disallowed, availability of new tools is also effectively prohibited.

Often, pressures to adopt new tools force their use before change can be fully accommodated. If existing interfaces must remain unchanged, the only alternative is to create an additional, parallel interface. While this approach allows adoption of new technology, it also complicates the environment. Such work needlessly duplicates existing efforts as similar goals are accomplished in different ways. In the long run, this *ad hoc* approach to evolution will likely cause difficulties in maintenance and further development.

Eventually, change must occur. In operating systems, change is typically restricted to the computer manufacturer in occasional, perhaps annual, systems software releases. While this policy of change delays problems resulting from change to an occasional event, eventually these difficulties must be faced.

Because the authority of operating system change is vested largely in the workstation manufacturer, potential for third party enhancement is greatly restricted. Unavailability of source code, incompatibility with other third party change and even vendor-supplied updates together discourage third party innovation. Finally, the methods used by manufacturers to improve services are often not available to third parties. As a result, third party modifications suffer performance penalties compared to vendor supplied improvements, further handicapping independent development.

As discussed in the introduction, support of third party software is critical to the timely development of new capabilities. These problems discourage third party development, slowing the advancement of the operating systems field as a whole.

An extensible interface addresses a number of the problems resulting from change. Controlled change is allowed, permitting the use, experimentation, and rapid adoption of new hardware and software. While change is permitted, it will be provided in a regulated, consistent way. Software must then gracefully adapt to its environment, both as a result of the presence of unexpected, new extensions, and the lack of expected support.

Explicit capacity for change will support the development of new software. To maximize the impact and availability of new services, modifications must be possible with as little disruption of existing software as possible. Ideally, a new software module could be added without source code changes to it, or any other module. Furthermore, the services offered by third party modules should not have any undue penalty for being developed independently.

Extensible operating system services have been uncommon, but user level programs supporting extensibility have proven effective. The Internet File Transfer Protocol [Bus71] and Apple Computer's HyperCard [App88] illustrate the benefits of extensibility in gradual software evolution and rapid support of new technology.

The flexible design of the Internet File Transfer Protocol, FTP, has allowed its gradual evolution with the addition of new commands (for example, [PR85]). The scope of the Internet, where hundreds of computer architectures support thousands of software revisions across hundreds of thousands of hosts, makes the concept of "one consistent software version" an impossibility. In such an environment, gradual extensibility is a requirement. FTP's capabilities for handling or rejecting new commands have allowed incremental adoption of new capabilities.

Another example of software extensibility is Claris' Hypercard for the Apple Macintosh. Hypercard provides basic hypertext services, supporting graphics and text through a scripting language, HyperTalk. HyperTalk is extensible; users can add new commands via its "XCMD" feature. This extensibility is critical to the rapid use of Hypercard to a number of applications outside its original domain, including electronic bulletin board support (with extensions to access serial ports and modems) and video-disc and CD-ROM support (extensions drive the appropriate hardware devices).

2.2 Stacking

File systems frequently implement very similar abstractions. Most file systems must coordinate disk access or file and directory allocation, for example. File system stacking is an approach to file system modularity that allows a file system to be decomposed into its component abstractions. Rather than each file system providing all user services with a monolithic implementation, separable services are placed in individual layers. With careful design, each abstraction is present in a separate layer and can then be reused in the implementation of other file systems. Improved services can also be substituted into existing stacks.

The key characteristic of a stackable layer is that it possess the same interface above and below. Each layer is implemented based on certain existing services; its interface with these services is its "below" interface. The layer then exports its "above" interface to its users. When these interfaces are syntactically identical, it becomes possible to combine these interface arbitrarily in a "stack". Given such a stack, one can construct "filter" layers which add new functionality. Because a filter layer has a symmetric interface, it could be inserted into existing stacks at any layer boundary.

To illustrate file system stacking, consider a brief example. A conventional disk file system might be provided by a stack of three layers. A base layer implements raw disk access. A middle layer provides files in a flat, fixed namespace (inode-level access), and the top layer provides hierarchical directory services. Figure 2.1 illustrates such a stack. With stackable layering, a comprehensive user-centered naming service might replace the hierarchical directory layer while still making use of the low-overhead file access layer, or a compression layer might be inserted between the directory and file layers to improve the apparent storage space.

Support for stacking is an important consideration when designing an interface. Although stacking alone is not difficult to provide, stacking with an extensible interface requires more care. Consider Figure 2.1 again. If the interface supported by the file layer were extended to provide an atomic commit service, for example, this interface should be immediately available above the directory layer. This must be possible even if source code to the directory layer is unavailable.

File system stacking is actually a special case of more general layering. In addition to simple linear stacking, the general case of file system *trees* should be possible. As an example of a file system tree, Figure 2.2 illustrates one possible implementation of disk mirroring. A mirroring layer duplicates all file system

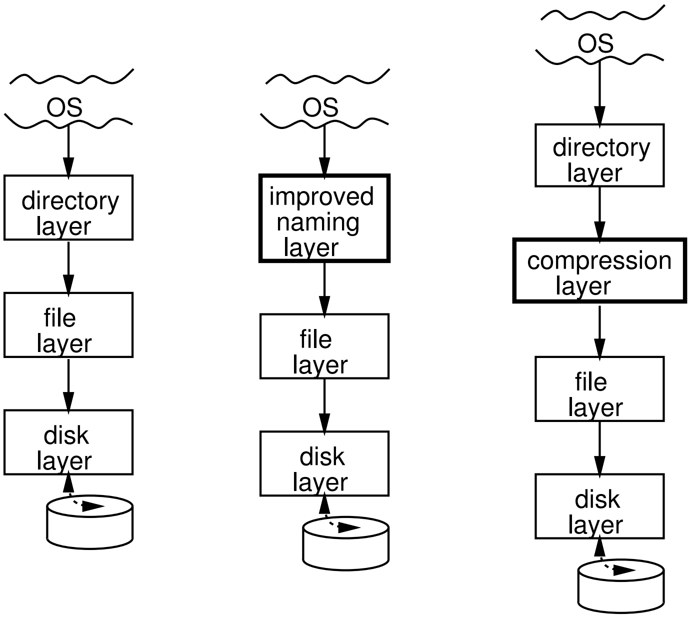


Figure 2.1: On the left is a simple file system stack. Improvements are made by layer replacement and insertion.

operations on two lower-level file systems. Such a mirroring layer could be implemented without detailed knowledge of the underlying file system type in a stackably layered environment.

File system stacking provides a unique framework for code reuse. In addition, it also offers the possibility of code substitution. When several layers provide similar semantics, one can be substituted for another. This substitution can be used to improve portability. Imagine a file compression service built over the BSD Fast File System layer. If this service were combined with the FFS, it would be portable only as an integral part of that file system. If it was instead built as an independent layer, it could easily be built to operate correctly over other file systems similar to the FFS, such as the System V file system or a vendor-customized file system.

Chapter 4 describes applications of file system layering in more detail. The uses of stacking for file system modularity and code reuse make it a key part of an improved file system development environment.

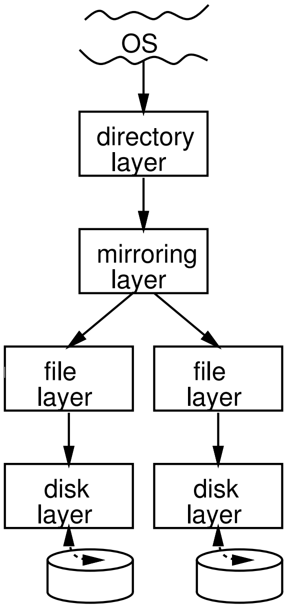


Figure 2.2: A tree of file system layers to provide disk mirroring.

2.3 Stack Configuration

File system stacking is only as useful as it is easy to use and configure. Stack configuration spans a spectrum from the very static to the very dynamic. In general, more dynamic approaches offer more possible applications with increased flexibility; they also present the potential for more overhead and management difficulties. A classic exchange of efficiency for flexibility, parallels to this spectrum exist in fields such as programming language design.

Flexibility of stack configuration varies in several aspects. First, the granularity of the object being configured in a stack may vary. Stacks may be customizable at the per-process or per-file level, on a subtree by subtree basis, or perhaps each kernel requires a particular stack.

When stacks are specified is a second aspect of configuration. In original versions of Unix, the choice of file system was hard-coded into the kernel. The vnode interface allows addition of new file systems when the kernel is configured and selection of file system type when file systems are mounted. Stacks could be defined at either of these points. File open is a natural time to specify stack contents, or one might even allow stack contents to change during file use.

Consistency in configuration is a final aspect to be considered. Are all users (or processes, process groups, or hosts) required to see the same stack configuration at all times, or are multiple independent views of stack composition permissible?

Greater configuration flexibility is important, because it allows file system stacking to be more widely used. The ability to specify stacks on a file-by-file basis would provide many of characteristics of an object oriented filing environment, for example. In practice, this flexibility must be traded off against added run-time overhead, management cost, and implementation complexity of more dynamic solutions.

The capability of running several file systems concurrently prohibits kernel-granularity file system configuration. The subtree[1] granularity offers a convenient existing approach to file system configuration. Many times, however, the flexibility of having individual files separately configured would be an advantage.

File system configuration by code modification has always been available. Much more attractive is the ability to configure stacks at run-time in the same way file systems are mounted. This allows a system administrator to customize file system configurations without halting the machine, allowing much greater freedom to experiment with different designs. More flexible still is the ability to define a stack when a particular file is opened, although this presumes that stacks are configurable on a file-by-file basis. Finally, one can imagine stacking a "measurements" layer on a stack to collect usage patterns, even while files are in use.

Configuration consistency has perhaps the clearest resolution. While it is often good to ensure a consistent appearance of service to all users, there are many times when different views of a stack are desired. File system backup, for example, might directly read the disk rather than operate at the directory level. Section 7.2.2 discusses consistency options in more detail. With the exception of completely semantics free layers, there are few cases when a program would wish its current view of an open file stack to change.

The flexibility of late, fine-granularity binding holds the promise of increased power, but also increased overhead. An implementation must trade these factors against each other. The minimum level of stackable service provides configuration on a file system basis at mount-time. Ideally, file-by-file configuration as files are opened would provide an "object oriented" environment.

[1] Subtree here refers to a subset of the tree-structured Unix file system name space. In traditional Unix terminology a subtree is referred to as a *file system* which corresponds to a disk partition.

2.4 Interface Definition

Most kernel interfaces are described only by paper documentation, relying on programmer vigilance about the types and contents of arguments. While traditional documentation is useful in the design of a new file system layer, it is insufficient if the interface connecting layers is extensible. When joined by an extensible interface, all documentation must be considered perpetually out-of-data, since new layers may add operations at any time.

Rather than restrict documentation of the layer interface and operations to the layer designer, an extensible layered interface requires that all interface details be formalized and available to the layers themselves. Meta-data about individual interface operations is necessary to allow generic layers to react to all possible operations at run-time.

By providing the specification of each operation in a machine-interpretable form, layers do not need hard-coded information about each operation. Instead, layers can often utilize meta-information to handle new operations in a generic way. For example, a network transport layer might use information about an operation's arguments to pass that operation to another machine via some RPC protocol.

Run-time description of an interface is critical in a stackable environment where extensibility is supported, and so a comprehensive interface definition will be an important part of any file system development approach utilizing these techniques.

2.5 Efficiency

Reuse of layers is enhanced when each layer encompasses few abstractions. If layer crossing overhead is at all significant, modular filing environments will either suffer serious performance penalties (relative to non-layered environments), or layers will be combined, making layer reuse more difficult. The layering strategy must be very efficient so that it does not otherwise impact the file system design.

2.6 Flow of Control

File systems are for the most part passive, responding to user actions. Performance can be improved by optimizing for this dominant case, bringing it as close to the cost of a procedure call as possible.

Occasionally, the file system should take a more active role. An example of this is raw disk I/O. Rather than wait for the physical disk read to complete, the executing process can suspend itself, allowing other processing to be done. When the read completes, an interrupt will signal the original process that it may resume.

This interrupt is a form of "upcall" [Cla85], an operation which begins at a lower level of system software and proceeds towards the user. A more general term for upcalls is non-linear flow of control[2], since one can also imagine need to call services at "parallel" levels.

Upcalls are often needed in distributed file systems. For example, to maintain cache consistency, the file server might call upon clients with invalid data pages, requesting them to purge their cache. This approach was taken by the Andrew File System. An alternative method of consistency control might have the clients communicate among themselves, making "sideways" upcalls. This method has been explored in the consistency layer described in Section 5.6. These actions do not proceed naturally down a stack of layers, but instead naturally progress up and sideways between layers.

In both cases, these upcalls would be occurring between different layers of a layered file system, and so it is natural to consider upcalls as an extension of the basic layering interface. Providing this kind of interaction within the framework of a file system interface minimizes the number of separate constructs required in development, and non-linear flow of control should be a part of a complete file system interface.

2.7 Opaqueness

A fundamental tenet of layering is information hiding. To maximize information hiding, layers should be treated as completely opaque. All access to a layer should be restricted to its provided operations. This has several implications for the design of the kernel environment.

For a file system interface to be effective, it must have complete control over the status of the file system. The remainder of the kernel should not intrude on the private contents of file system data structures, but should restrict all interaction to provided operations.

Also, the kernel should try not to second guess the file system. Many upper-level kernel optimizations make assumptions about the status of a file system

[2] Both terms will be used interchangeably here.

layer. These optimizations break when confronted with radically new layers.

These two observations are examples of a broader principle, that of underspecification[3]. To allow maximum freedom in layer use, other layers and the kernel should assume as little about layers they call as possible. By accessing the private contents of data structures directly, the kernel is assuming that these data structures and their format will not change. Optimizations based on vnode type also assume that this type information will not change. In either case, this same determination can be made by the layer itself, allowing the layer additional control over its destiny.

2.8 Summary

This chapter outlines the characteristics desirable in a stackable file system interface. The interface must be extensible to support new filing services; stackable to allow them to be constructed from existing facilities. Late binding and fine stack granularity aid the flexibility of stack configuration, and a careful stack definition allows layers to adapt to interface variations. Rich flow of control allows the interface to be applied to a wide variety of situations. These characteristics should be provided without arbitrary limitations while still supporting excellent performance. The next chapter will discuss a prototype interface designed with these issues in mind.

[3] This term was first coined by Hutchinson and Peterson in [HP91].

CHAPTER 3

Interface Implementation

The previous chapter described a number of new ideas about how file systems should be constructed. Chief among these is the idea that stackable file system design can promote file system development. To verify these ideas, we set out to experiment with file system layering by constructing and using file system stacks.

The benefits of file system modularity have resulted in the adoption of a several different file system interfaces. Several are now available commercially, including Sun Microsystems' Virtual File System interface (VFS, [Kle86]), Digital Equipment Corporation's Generic File System (GFS, [RKH86]), and AT&T's File System Switch (FSS). Each separate the implementation of the file system from the remainder of the kernel, allowing different abstractions to be accessed through a standard interface.

The initial impetus for the adoption of modular file systems was often remote file access. The VFS interface was created to support Sun's Network File System (NFS, [SGK$^+$85]), and AT&T's FSS was joined by their Remote File Sharing [RFH$^+$86] package. This modularity has also proven useful in the support of other filing services. For example, DEC's GFS was used to support several different file system formats [Koe87].

Ficus is a distributed file system supporting file replication for reliability, performance, and scale [GHM$^+$90, Guy91]. Development of Ficus begin in 1989 at UCLA, where it has served as a testbed for a number of new file system concepts. For reasons of modularity and portability, Ficus was built with a standard file system interface. Because of wide industry acceptance and availability, Sun's vnode interface was selected. To leverage existing work and to test our ideas, Ficus was structured from stackable layers. Ficus was stacked over existing file systems for storage and remote access.

As work on Ficus progressed, it quickly became apparent that long term use of the vnode interface would be impossible. There was no compatible way to add operations to this interface. Replication requires a more flexible protocol than the vnode interface provides; no clean way is available to integrate new replication operations with existing file system layers. On the other hand, the modularity and performance of the vnode interface were quite good.

Our initial work with the vnode interface encouraged us to use it as a starting point for an improved file system interface. To support stacking and extensibility, though, we found it necessary to change the interface in several ways. This chapter describes the new stackable interface we developed, the 405 interface[1]. It

[1] The 405 Freeway is a major artery in West Los Angeles. It runs from Orange County to the northern San Fernando valley, and is the primary passage for tens of thousands of people each day through the Santa Monica mountains.

begins with a brief review of the vnode interface, and then describes the significant changes made to support stacking and extensibility.

3.1 The Existing Interface

To meet the demand for several file systems within the same kernel, the file system switch was developed. Sun's vnode interface [Kle86] is a good example of this approach, separating file systems from the remainder of the kernel with an object-oriented interface[2]. Versions of the vnode interface are provided in several variants of UNIX, including SunOS, System V Release 4, and 4.3-Reno BSD. The interface has been successful in supporting a number of file systems, including the Berkeley Fast File System, the System V file system, NFS, and a variety of other file system services.

The vnode interface is a method of abstracting the details of a file system implementation from the majority of the kernel. The kernel views file access through two abstract data types. A *vnode* identifies individual files; the *vfs* describes groups of files (filesystems or disk partitions). A small set of file types is supported, including regular files, which provide an uninterpreted array of bytes for user data, and directories, which list other files. Directories include references to other directories, forming a hierarchy of files. For implementation reasons, the directory portion of this hierarchy is typically limited to a strict tree structure. Figure 3.1 shows the tree forming a typical file system namespace.

For configuration purposes, sets of files are grouped into *subtrees*[3]. One subtree, the *root subtree*, is automatically installed upon system initialization. Other subtrees are added to the file system namespace by a *mounting* process.

Mounting is the process of adding new collections of files into the global file system namespace. Figure 3.2 shows two subtrees, the root subtree, and another attached under `/usr`. Once a subtree is mounted, name translation proceeds automatically across subtree boundaries, presenting the user with an apparently seamless namespace.

[2] DEC'S GFS is quite similar to Sun's vnode interface in implementation. Both select file system operations by indirect function calls through an operations vector. The differ primarily in the set of functions offered by the interface.

Technical details about AT&T's FSS are not available outside of proprietary documentation, but selection of Sun's vnode technology for inclusion in System V Release 4 suggest that it does not offer significant technological advantage.

[3] Traditional UNIX literature uses the much overloaded term "file system" in place of subtree.

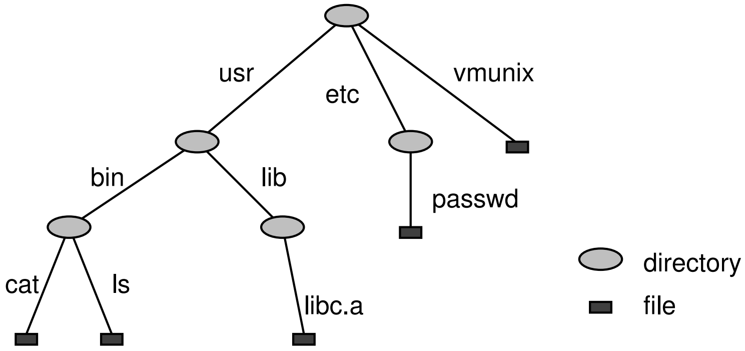


Figure 3.1: A typical file system namespace.

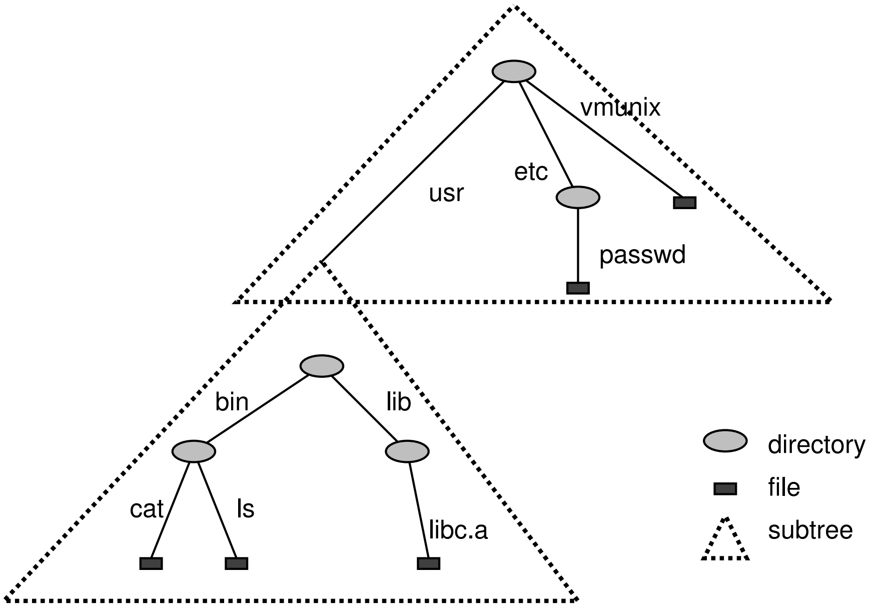


Figure 3.2: A namespace composed of two subtrees.

open
close
rdwr
ioctl
select
getattr
setattr
access
lookup
create
remove
link
rename
mkdir
rmdir
readdir
symlink
readlink
fsync
inactive
lockctl
fid
getpage
putpage
map
dump
cmp
realvp

Figure 3.3: Operations supported in SunOS 4.0.3.

All files within a subtree typically have similar characteristics. Traditional Unix disk partitions correspond one-to-one with subtrees. When NFS is employed, each collection of files from a remote machine is assigned a corresponding subtree on the local machine.

Data encapsulation requires that abstract data types be manipulated only by a restricted set of operations. Vnodes, the abstract data type for "files", each support a specific group of operations, although the exact set varies according to the particular implementation used. Vnode operations typically resemble the system call-level file system interface (open, close, read, write, link, etc.). Figure 3.3 lists the vnode operations supported in the SunOS 4.0.3 version of the vnode interface.

Different file systems typically require completely different implementations of each vnode operation. Reading a file from a local disk is quite different from fetching data from a remote file system, for example. As an abstract data type, the kernel must treat all vnodes equivalently as opaque data structures. The kernel should not be forced to select explicitly the correct implementation of a particular operation for a particular vnode type.

To allow this generic treatment of vnodes, binding of desired function to correct implementation is delayed until run time. This is implemented in standard C by associating with each vnode type an *operations vector* identifying the correct implementation of each operation for that vnode type. Operations can then be invoked on a given vnode by looking up the correct operation in this vector. In C, this is implemented as an indirect function call through the correct element of the vector. This approach is identical to that typically used to implement C++ virtual class methods [Str86].

Some file system stacking is possible with the standard vnode interface using the mount mechanism. Sun Microsystems' NFS [SGK+85], loopback, and translucent [Hen90] file systems take this approach. The private data of the mount command identifies the lower layer of the stack, the mount command creates the new upper layer and connects it into the file system name space.

3.2 Extensibility in the 405 Interface

Lack of interface extensibility is a critical problem with existing file system interfaces. Many situations arise where the file system interface must be modified to pursue new ideas. Evolution of the kernel in the face of compatibility with existing file systems requires extensibility, as does support of radically new file systems in standard kernels.

The problem of extensibility in the vnode interface stems from the conventions used to define the interface itself. It is simply assumed that all file systems in a given kernel are designed around the same set of operations. When a file system needs to add operations, it has no method to coordinating this desire with other file systems, or with the kernel as a whole. In particular, the only formal definition of the operation supported by this interface (the *vnodeops* structure definition itself) is lost at compile time and cannot be regenerated; there is no way of determining compatibility between the interface supported by the kernel and that supported by each file system.

This problem of extensibility is solved in the 405 interface by maintaining interface definition information until execution, and then dynamically constructing the interface. Each file system includes a list of all the operations it supports. At system initialization, the union of these operations is taken, yielding the list of all operations supported by this kernel. This set of operations is then used to dynamically define the structure of the operations vector by assigning each operation a unique index in this vector. These two steps construct the operations vector automatically, allowing it to adapt to changes in the set of supported operations. Given the format of the operations vector, a vector customized to each file system can then be constructed. These vectors list the implementation of each operation for that particular file system so that when operations are invoked on the corresponding vnode, the correct implementation will be selected[4].

New file systems can be added to a kernel with a simple reconfiguration. The addition of a file system has the potential to add new operations, which must be

[4] File systems may actually support several different vnode types, each with a separate (but usually related) set and implementation of operations. Directories and files, for example, are best supported with different implementations of several operations.

accommodated automatically. This fact has a profound impact of the nature of the interface. While with a traditional, fixed interface the file system designer could assume that all cases (operations) are explicitly handled, an extensible interface requires broader considerations in design because new operations can be added at any time. File systems must be able to react to new operations in a consistent manner. To handle new, "unsupported" operations, each file system supports a *default* operation. This routine will be invoked to handle all operations not otherwise provided by a file system. In a basic file system, this operation may simply return an error. More sophisticated file system layers should use the routine to pass unsupported operations to lower layers. Details of how this can be done are provided in Section 3.4.

The new structure of the operations vector also requires a new method of operation invocation. The calling sequence for new operations replaces the static offset into the operations vector of the old interface with a dynamically computed new offset. Because of careful design, these changes have very little performance impact, an important point in something that will be as frequently employed as an inter-layer interface. Section 6.1 analyses performance of the 405 interface in detail.

3.3 Stack Creation

The effort involved in developing a production-quality file system is far from small. Production file systems typically amount to about 10,000 lines of C code, and even simple file systems amount to several thousand lines of code[5]. As discussed in Chapter 2, a stackable file system design has the potential to reduce this effort by allowing reuse of existing code.

This section discusses how stacks can be formed. In the prototype interface, stacks are configured at the filesystem granularity, and constructed as required on a file-by-file basis.

3.3.1 Stack configuration

Section 3.1 described how a complete UNIX file system is built from a number of individual subtrees by mounting. Subtrees are the basic unit of file system configuration; each is either mounted making all its files accessible, or unmounted and unavailable.

[5] Measurements from SunOS 4.1: ufs, 11667; nfs, 9982; hsfs, 3665; pcfs, 3627 raw lines of code.

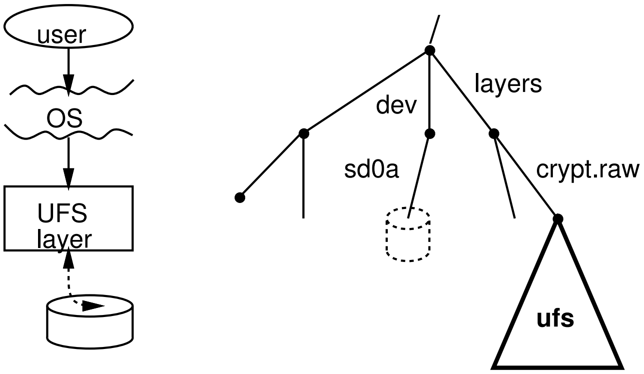


Figure 3.4: Mounting a UFS layer.

Let us review in more detail how Unix uses the mount mechanism to bind subtrees into the global name space. Figure 3.4 illustrates how one might add a new UFS disk partition. A mount system call is issued with the file system type (UFS), the identity of the lower-level object (`/dev/sd0a`, a block-special device representing the disk drivers), and the name for the new file system (`/layers/crypt.raw` in the example). Users may then access files in the new file system by accessing files from the `/layers/crypt.raw` directory; all files beneath this level are assumed to be UFS files from that disk partition.

The 405 interface overloads this mount mechanism to serve also as a layer configuration mechanism. Layers are configured on a subtree level granularity. The Unix mount command serves to create each layer of a stack, one-by-one building the layers of a multi-layer stack. Typically, stacks are built bottom up. As each layer is mounted to a name, the next higher layer uses that name to identify its "lower layer neighbor" in initialization.

To build on the example of mounting, Figure 3.5 illustrates how an encryption layer could be pushed on to an existing UFS layer. Similar to mounting normal file systems, the layer type (encryption), the name of the lower-level object (`/layers/crypt.raw`, the UFS layer), and the name for the new layer (`/usr/data`) all must be specified. Files opened through `/usr/data` will then gain the benefits of services provided by the encryption layer.

File system stacks are not necessarily linear. Stacks with multiple lower layers (trees) are also possible. File system treeing is particularly useful in the support of replication. Figure 2.2 (page 14) shows how a file system tree of layers might be used to build a disk mirroring service. When creating a tree with multiple lower layers, the names of each lower level layer must be provided to the mount

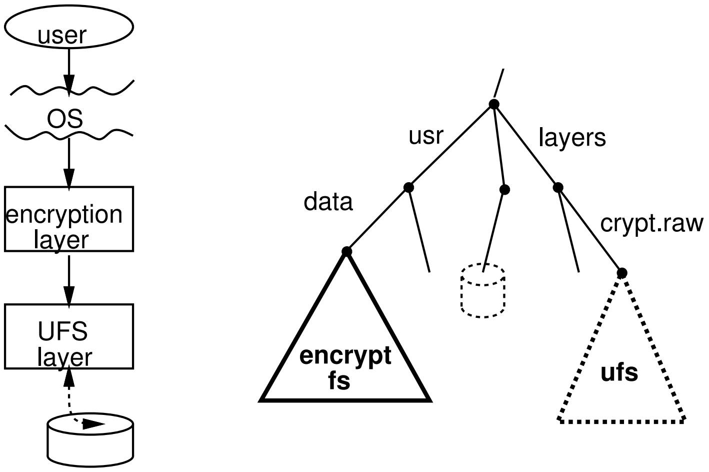


Figure 3.5: Stacking an encryption layer over the UFS.

call.

Stack construction does not always proceed from the bottom up. Sophisticated file systems may automatically create lower layers as necessary. The Ficus distributed file system takes this approach in its use of volumes. Volumes are subtrees with all mounting handled automatically by Ficus. So that all sites maintain a consistent view about the location of each volume, volume mount information is maintained on disk at the mount location. When a volume mount point is encountered during path name translation, the corresponding volume is automatically located and mounted.

3.3.2 File-level stacking

While stacks are configured at the subtree level, most user actions take place on individual files. Files are represented by vnodes, with one vnode per layer.

When a user opens a new file in a stack, a vnode is constructed to represent each layer of the stack. User actions begin in the top layer of the stack and are then forwarded down the stack as required. If an action requires creation of a new vnode (such as creating or opening a new file), then as the action proceeds down the stack, each layer will build the appropriate vnode and return its reference to the layer above. The higher layer will then store this reference in its private data. Figure 3.6 shows a two-level stack of vnodes. A file system stack with trees is handled similarly, the layer with several lower layers stores references to *each*

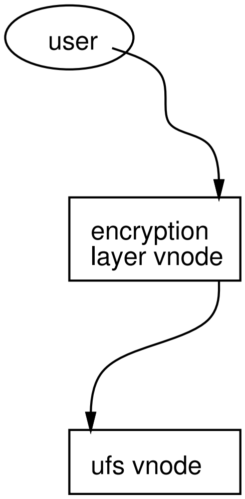


Figure 3.6: A two-level stack of vnodes.

lower-level layer in its private data.

An important point of this method of stack creation is that the only connections between layers are the vnode references, the same interface used for normal file access in the kernel. Because the same interface is used to bind layers as to access files from the rest of the kernel, no special provision need be made to perform operations between layers. The same operations used by the upper-level kernel can be used between layers to access layered file system functionality. Layers also treat all incoming operations identically. There is no need to try to distinguish calls from a higher layer from calls invoked by the user.

3.4 Stacking and Extensibility

One of the most powerful features of a stackable interface is that layers can be stacked together, each adding functionality to the whole. Often layers in the middle of a stack will pass most operations to a lower layer unchanged. For example, although an encryption layer would encrypt and decrypt all data accessed by read and write requests, it would not need to modify directory operations such as link and readdir. But since the inter-layer interface is extensible, how can an intermediate layer be prepared to forward *all* operations down? Figure 3.7 illustrates this problem. The pair of Ficus layers each add operations to support replication, but between them a standard transport layer (NFS) is placed. How can replication-specific operations be passed through NFS without requiring NFS source code changes? This section discusses methods to forward operations in an extensible environment.

One way to pass operations down to a lower level is to implement, for each operation, a routine which explicitly invokes the same operation in the next lower layer. This approach does not work when the interface is extensible, since new operations can be added at any time. This approach requires that each addition of a new operation be accompanied modification of all existing layers. This both discourages the creation of new layers and new operations, and it also make it impossible to employ unmodified third-party layers in the middle of new stacks. Since third-party layers must often be distributed as object code only (to protect development effort), such layers will often be unavailable for modification.

What is needed is a single *bypass* routine which can forward new operations to a lower level. Default routines discussed in Section 3.2 provide the capability to have a generic routine handle unknown operations. Unfortunately, existing interfaces make it impossible to implement such a routine. To handle multiple operations, a single routine must be able to handle the different numbers and kinds of arguments used by different operations. It must also be possible to identify the operation taking place. Neither of these characteristics are possible with existing interfaces where operations are implemented as standard function calls.

The 405 interface accommodates these characteristics in two ways. First, rather than passing operation arguments as parameters of the function implementing the operation, they are grouped into a structure and a pointer to this structure is passed. This allows arguments to be collectively identified by a generic pointer, and it avoids repeatedly copying arguments when passing through several layers of a file system stack.

Second, a new argument is added to each operation. This argument contains meta-information about the operation: what operation it is, the number and kinds of its arguments, and so on. This description information and the structure extend the object-oriented style provided by vnode-based file systems to the implementation of the interface itself. The original interface gave the user the ability to perform operations on a vnode without respect to its type; this modification allows a bypass layer to forward an operation to a lower level without respect to the operation involved.

These characteristics make it possible for a simple bypass routine to forward all operations to a lower layer in the 405 interface. We expect all file system layers to support such a bypass routine.

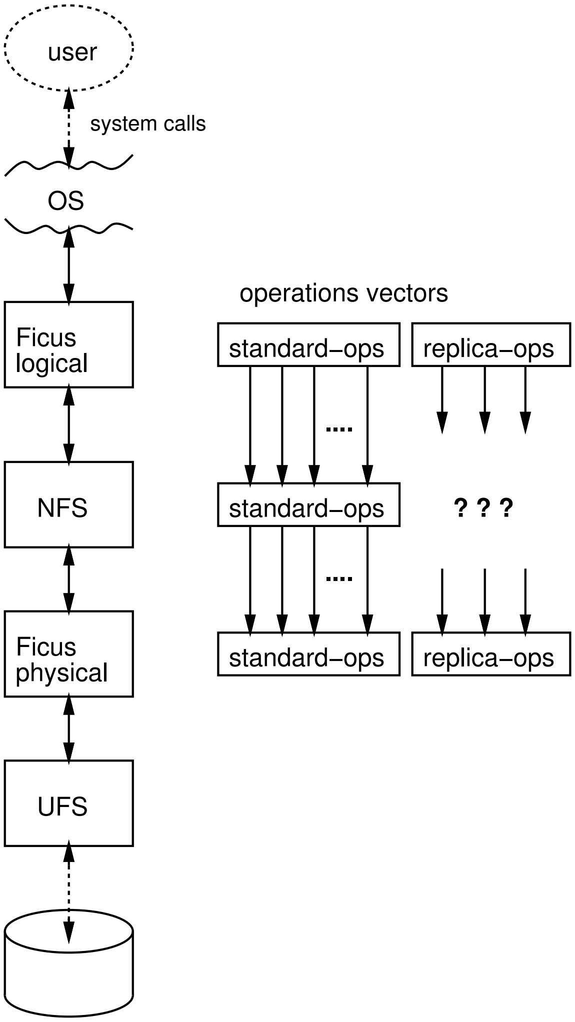


Figure 3.7: The problem of passing new operations through an old layer.

3.5 Inter-machine Operation

A *transport layer* is a stackable layer which transfers operations from one address space to another. The object-oriented flavor of this enhanced interface allows remote access to be *network transparent* to the programmer. Because vnodes for both local and remote file systems accept the same operations, the programmer may use either at any time. This transparency allows novel approaches to configuring layers as described in Sections 4.7 and 4.8.

Inter-machine operation preserving network transparency poses several problems. How can two machines transparently interact if they are configured to use different file systems and different sets of operations? What happens when two machines have different formats for basic data types such as integers and floating point numbers? What about compilers with different methods of structure padding? What about operations handling variable sized or dynamically allocated data?

For two hosts to inter-operate, it must be possible to identify each desired operation unambiguously. In the past, a well defined RPC protocol enumerating a fixed set of operations insured compatibility [Sun89]. This approach is incompatible with the principles of extensibility required by new file system development. Instead of a fixed set of operations, each operation in the 405 interface is assigned a universally unique identifier when it is defined. Inter-machine communication can then use these labels to identify known operations and reject unknown ones.

For transparency to be preserved with an extensible interface, it must be possible for transport layers to forward new operations to other address spaces, just as bypass routines forward operations to lower layers in the same address space.

Moving operations between address spaces requires that the type of each argument be known so that a network RPC protocol can marshal that operation and its arguments. Network marshaling accommodates translation between different fundamental data types, including byte order and data alignment. This information is part of the meta-data carried along with each operation, and it must be described by a formal interface definition similar to an RPC interface specification. In addition to the description of arguments and operations, each operation must be assigned a unique name for universal identification, similar to RPC protocol numbers. Thus a transport layer may be thought of as a semantics-free RPC protocol with a stylized method of marshaling and delivering arguments.

NFS provides a good prototype transport layer. It layers on top of existing local file systems, instead of implementing a monolithic networked file system, Internally, NFS uses a vnode-like RPC interface. NFS was not designed to serve as a transport layer. It is instead customized for remote file access. We have modified NFS to allow it to automatically pass previously-unknown operations. This approach quickly resulted in usable transport layer, but some complications have arisen in trying to use NFS as a semantics-free transport layer. These problems are described in Section 5.3.

In addition to the use of an NFS-like inter-address space transport layer, a more efficient transport layer operating between the user and the kernel level was conceived. Such a transport layer could very easily provide "system call" level access to the 405 interface, allowing user-level development of file system layers, and providing an interface to new file system functionality.

The desire to support a system-call-like transport layer placed one additional constraint on the interface. There are differences in the services typically offered by an RPC interface and a system call interface. In a system call interface, the kernel expects the user to provide space for all returned data. The Unix read system call, for example, requires the user to pass a pointer to a buffer for the returned data. This is necessary because the kernel cannot, in general, know how to allocate user-level buffer space itself. Because an RPC interface has complete control over both the client and the server, RPC mechanisms often allow the server to dynamically allocate space which can be returned to the client.

To allow an interface to adapt equally to the RPC and the system call transport layers, the more restrictive policy of the two must be chosen. The 405 interface therefore disallows the server side of the interface from returning dynamically allocated data. Instead, the client must provide a pre-allocated buffer. In practice, this has not been a problem, since the client can often make a good estimate of the required buffer size. If the client's first guess is wrong, information is returned so that the buffer can be re-sized correctly and the operation repeated.

3.6 Centralized Interface Definition

Several aspects of the 405 interface require precise information about the characteristics of the operation taking place. Network transparency requires a complete definition of all operation types. A bypass routine requires knowledge of each vnode argument so that it may be mapped to a lower-level vnode. This information must be provided by the designer of the file system layer using new operations.

Detailed interface information is needed at several different places throughout

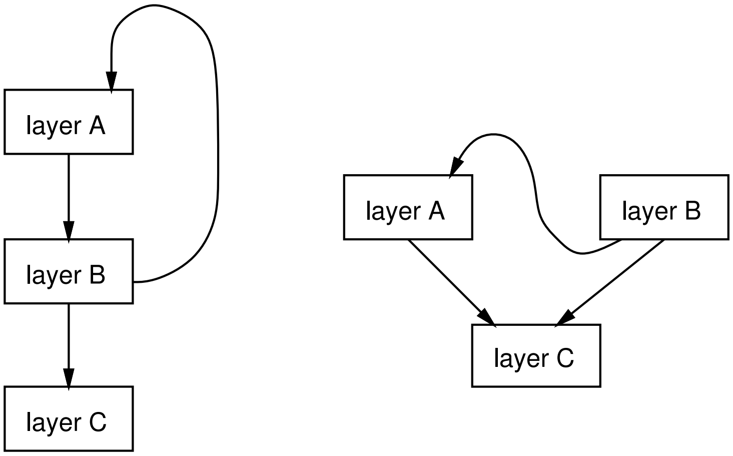


Figure 3.8: Upcalls proceeding up a stack and between parallel stacks.

the layers. Rather than require that the interface designer keep this information consistent in several different places, operation definitions are combined into an *interface definition file*. Similar to the data description language used by Sun's RPCGEN compiler [Sun87, Sun88] and also Hewlett-Packard's NIDL data description language [ZDL$^+$90], this file lists each operation and its arguments[6].

3.7 Flow of Control

Traditional layered services have had a strictly linear behavior. A user would request a service. That user's request would pass down through each layer, each returning a result to the previous and ultimately back to the user. For many reasons, layers in a distributed system frequently require a more general flow of control. *Upcalls* begin at lower layers of the system and proceed upwards to higher layers [Cla85]. More generally, operations can be invoked between two unrelated stacks. Figure 3.8 illustrates these cases.

An RPC service provides this sort of generality, typically allowing communication with any object on any machine on a network. While it would be possible to create a new RPC interface to file system facilities, the vnode interface already defines a set of services, and transport layers export this interface to other ad-

[6] Both RPCGEN and NIDL were considered for use as the interface definition language. The C code generated by RPCGEN often has additional levels of structure indirection with certain data types, making it unsuitable. A NIDL implementation is not currently available for our development environment.

dress spaces. It seems foolish to require that the services of the vnode interface be duplicated by an additional RPC protocol and level of complexity.

The chief advantage an RPC protocol offers is the ease with which an arbitrary client (layer) can establish communication with an arbitrary server (other layer, in general). Given a network address, messages can be communicated between any willing parties. This contrasts with the basic services of the 405 interface, where communication proceeds only down the stack. The problem reduces to one of naming. Services in an RPC system have a consistent, low-level namespace allowing all to be equally addressed, while vnodes are only named by "references" (pointers) which are meaningless outside the address space of their creation.

The problem of naming is made more difficult by the fact that different layers may inhabit different address spaces or even machines. The problem is further compounded by the need for general fan-in and fan-out of each layer. Although a layer knows other layers it is stacked upon (fan-out), it should not be required to make special accommodations for its clients (fan-in)—this violates information hiding.

The 405 interface partially solves this problem of naming by expanding on two services found in the current vnode interface. A facility is provided to identify any file in a filesystem by a low-level name. This *file identifier*, or fid, can be mapped to and from a vnode as desired, provided that the file system is known[7].

This level of naming provides the facilities needed to use an RPC layer as a generalized upcall mechanism. If it is desired to use an upcall between two layers, a transport layer can be placed between them. Before an upcall is desired, a fid is saved for the file upon which the upcall will be performed. When the upcall is needed, this fid is mapped to a vnode on the RPC layer. The upcall operation can then be performed as normal on the returned vnode.

Although this approach works, there are several disadvantages. Because file-ids are only unique within their file system, there is still no truly global namespace. Although mounting is not necessarily a complex operation, use of an RPC layer for upcalls requires either that either the path of upcalls be pre-specified, or that RPC layers be mounted as needed.

A concern in any environment permitting circular upcalls is one of deadlock. There are three possible positions on deadlock. It can either be detected, avoided, or not addressed. Deadlock detection is typically too expensive for a low-level

[7] The 405 interface uses a slightly different fid than the traditional vnode interface. Sun's vnode interface limits file-ids to 12 bytes of data. The 405 interface removes this limit, important because we expect layers to often add to the fid of the layer beneath them. The new fid operations also meet the restrictions of Section 3.5 regarding data allocation.

operation system, particularly when deadlock can span multiple machines or address spaces. Deadlock avoidance is easily solved by eliminating upcalls, but this denies a potentially valuable service. Rather than these approaches, the 405 interface takes the third position, leaving the problem of deadlock to the application at hand where it can be dealt with efficiently as needed.

CHAPTER 4

Layering Techniques

The previous chapters have discussed the reasons for building file systems from stackable layers and a prototype interface designed to facilitate layer development. The new interface removes many technical problems in creating stackable file system layers without an appropriate interface.

The great advantage of stackable design is its ability to allow rapid development, experimentation, and use of new file system functionality. Although advantages of layered development can be utilized with existing file systems, many new development techniques exploit the full potential of stackable development.

This chapter examines several new approaches to file system development enabled by a stackable interface. Beginning with an example to illustrate several different uses of layer design, we then examine each method in turn. The next chapter will then illustrate these techniques by layers prototyped at UCLA.

4.1 A Hypothetical Example

To consider the uses of stackable layering, let us explore a scenario in which layers may offer superior solutions to several problems.

Imagine a large University, the *Université Charmant pour Les Animaux*. At this university, Department of Cognitive Sociology has a large number of graduate students eagerly writing their dissertations on the computer system, a Unix minicomputer. Unfortunately, as at many schools, the students have been so prolific at their work that they have reached the limits of their disk space.

Utility program such as the Unix *compress(1)* program reduce file storage requirements by encoding data. The local system administrator, Thierry E. Corée, encourages the use of such tools. This method of file compression requires a conscious effort on the part of the user. Because compression becomes yet another step in getting work done, it is not consistently used and little disk space is saved.

Considering the problem, it occurred to Thierry that if compression could be added to the operating system kernel, it could be made transparent to the users. Files would be decompressed automatically, on demand. Files would then remain uncompressed (allowing rapid access to recently used files—caching, by any other name), and a daemon could run in the background compressing rarely used files.

A source code license was not available to the department for their version of Unix, so modifications could not be made directly to the file system. Furthermore, the last file system code Thierry had seen included a 17 page *namei* routine that Thierry did not understand and certainly did not want to break. Rather than directly modify the file system, Thierry decided to build a compression *layer* using the new stackable file system interface. The more he thought about it, the more attractive this approach seemed. A layer could run over several physical file systems, both the BSD Fast File System running on some of the department's computers, an older file system running on the department's PDP-11's, and even the Yaincomp file system running on their latest UNIX platform. By using layering, he could substitute one physical storage layer for another.

Design of a compression layer seemed fairly straightforward. Only a few operations would need modification. Access of a compressed file would automatically uncompress it. Thereafter, all operations would simply be passed though to the physical storage layer.

Although design of the new layer seemed straightforward, there was one design difficulty. A compression program typically creates a new (hopefully smaller) version of the file. When compression is finished, it gives the new file a file name similar to the old file and removes the uncompressed file. A typical UNIX file systems allow a file to have multiple names, or "hard links". Although there are several names for one file, all names point to the same data, so less space is taken up. Imagine compressing a file with multiple names. For each name, a separate, new, compressed version of the file will be created, probably consuming more storage rather than saving it. Thierry realized that this problem results from the way UNIX file systems are constructed. Rather than do compression at the "naming" level of the file system, where directories exist and file names are used, compression should be provided at the "file" or "inode" level. But because standard file systems were built before file system layering, they encompass too many abstractions. Physical file systems would be more useful they were broken into several layers, one providing directory services, another providing simple file access, and possible other layers.

Thierry decided to live without files with multiple names. The layer will automatically know when to uncompress files, but how can it know when to compress files? What is needed is a new operation on files, "compress". A thoughtful user could use the new operation explicitly to compress files, and a background program could run occasionally and perform this operation on rarely used files. Fortunately, the stackable file system was built with an extensible interface, so it was easy to add this new operation. The stackable interface was even available at the system call level, so the new operation was immediately available to his background compression daemon.

The department had a number of diskless workstations. How could they get access to compressed files? Thierry realized that it is easy to make the compressed files available on other machines with NFS. In effect, he was creating a three-level stack of layers: NFS on the top, the compression layer next, and finally the physical storage layer.

How could a user on a diskless workstation use the new “compress” file operation to force his or her files to be compressed? NFS would be between the user and the compression layer, and NFS does not support a compression operation (indeed, it doesn’t even support a “mknod” operation). Instead of using NFS with its fixed network protocol, Thierry used a *transport layer.* This layer is a modified version of NFS, extended to support passing any new operation over the network. When Thierry designed his new compress operation, he specified the types of each of its parameters and gave it a unique identifier. The transport layer uses this unique identifier to make sure that the other machine also knows about this new operation, and then uses the type information to send each argument “over-the-wire” between machines.

So Thierry built his new compression layer, and it worked well. Disk space was less of a concern now. When he installed this, Quoi Chen (the person doing backup dumps this month) mentioned that the backups on the 8mm paper tape reader were exceeding the size of one tape. Quoi complained that the backup program was reading the uncompressed versions of all the files, since it read through the compression layer. To solve this problem, Quoi changed the backup program to read all files directly directly from the physical storage layer, not through the compression layer. This way, the raw, compressed data was written to tape. By accessing files from different layers of the stack, both the space-saving compressed view and the standard, non-compressed view could be accessed. Users could get uncompressed data with access through the compression layer, while the backup program read the smaller, compressed data.

A new contract came to the department, dedicated to accessing sociology data stored in IMS from IBM PCs. Professor Joy A. Cartes received a grant of a dozen PCs, and wanted to be able to access her compressed files stored on the department’s main computer from the PCs running DOS. The PCs did not run Unix, though, so hope of porting the compression layer to the new operating system was slim. Fortunately, another company had developed PC-NFS, allowing PCs to access data stored on Unix computers through the standard NFS protocol. Since NFS would run stacked above the compression layer, Thierry was able to use this standard protocol to allow access to compressed files from machines with a completely different operating system.

Because of his success at conserving disk space, Thierry managed to talk his boss into loaning him a workstation to take home. He set it up, and even got NFS working over a 9600 baud modem to access his files at work from home. But he was disappointed in performance. His modem did not do data compression, and waiting a full minute to read his editor configuration file just seemed like too long. Unable to afford a new modem with compression, he realized that, with a few modifications, his compression layer could be used once again. He changed the compression layer so that it would always compress the data as it was written and decompress it as read. He also installed a switch so this compression layer could turn into a *de*compression layer, decompressing data as it is written and compressing it when it is read. Then Thierry mounted NFS in a new way. He put the decompression layer on his machine at work, and the modified compression layer on his machine at home, and NFS in-between. This way, his data was always compressed before it went through the NFS layer and over the slow modem. By having the two layers *cooperate*, a "compressed" NFS protocol was created, all without modifing NFS itself. Performance was much better; certain files (such as news) seem particularly suited to content compression.

As part of his research in "Image Compression in the Color Domain", Thierry came across a new compression algorithm suitable for use in his compression layer. He decided to replace the algorithm used in his compression layer with the new one, but since switching to the new brand CPU, Thierry's kernel debugging skills had dropped considerably. Rather than debug the new algorithm in the kernel, Thierry instead arranged that his layer could run as a user-level server, outside the kernel. The operating system would use a transport layer to access his out-of-kernel server, and he could use the system-call level vnode interface to get to the physical storage layer still running in the kernel. After changing a few makefiles and adding a "kernel-compatibility" package, Thierry was able to run the compression layer out of the kernel under his graphical debugger. The compression layer ran fine out-of-kernel for debugging, but moving all that data between address spaces had a certain cost, so when the new layer was working, Thierry recompiled his kernel and ran it more efficiently there. Because his changes were isolated to the compression layer, Thierry was confident of the integrity of the rest of the file system implementation.

4.2 Layer Composition

There are many possible ways to structure a file system into layers. While there are no all-encompassing rules for layer selection, layers can be reused most often if each implements one well-defined abstraction. Layer design is in this respect similar to design of filters in the Unix shell.

An example of the problem of layer design arose in the compression service example. Compression should be a "file" level concept, but the UFS exports only "directory" level access where files can have multiple names. Examining the UFS in more detail, we see three basic abstractions: a disk partition, file level access with fixed names (inode-level access), and a hierarchical directory service. If each of these were separated into layers, they would be useful for implementing other file systems. There are many file systems (databases, AFS, and Ficus for example) which would enjoy efficient inode-level file access without the overhead and complication of directories. To take matters even further, perhaps the access control functions of the UFS ought to be separated into a layer, allowing standard UNIX file protections to be replaced by an access-control list layer.

The Ficus replicated file system is a second example of layered file system design. Figure 4.1 shows the construction of the Ficus replicated file service. It is composed of two cooperating layers, a logical layer exporting the notion of a highly-available file, and a physical layer mapping a single replica to a standard UNIX file system. Between these layers, a transport service can be inserted to provide access to remote replicas. The physical layer is actually composed of several services: a facility to support additional file attributes, one to map low level identifiers to files, and support for replication-specific issues. One might imagine improving Ficus performance by replacing the identifier mapping facility with inode-level file access, and the extended attribute facilities seem a generally useful service. For this to be possible, the separate functions of the physical layer must be isolated in configurable layers. Section 5.4 discusses this possibility in greater detail.

4.3 Layer Substitution

One of the factors making layer composition difficult is the inclination to make layer services as general as possible. New features can often be added on to existing layers, but eventually weight of a number of options and attachments make a layer difficult to use and apply to general problems.

When several layers are designed to a similar semantic interface, one can often be substituted for another. Layer substitution can be used to promote compatibility and improve performance.

As example of several layers built to a similar interface, consider the various flavors of physical file systems. Traditional UNIX supported what is today the System V file system [RT74]. Berkeley developed the Fast File System to a very similar interface (long names and symbolic links being the primary interface modifications, [MJLF84]), and recent work in log-structured physical file

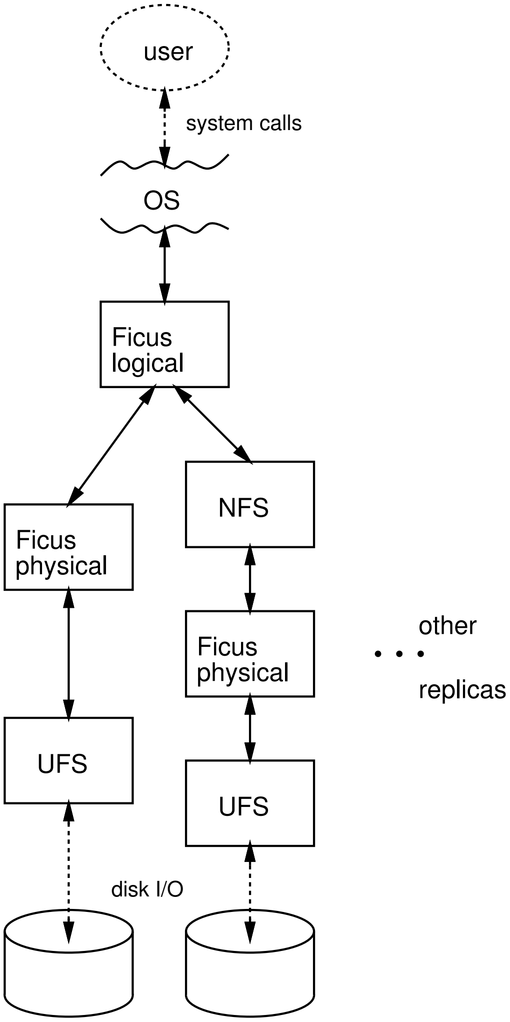


Figure 4.1: The Ficus stack of layers. The left stack provides access to a local replica. The right stack shows the addition of a transport layer to allow remote replica access.

systems appears promising [OD88]. Because of its interactions with other kernel mechanisms, porting a physical file system from one machine or Unix variant to another can be difficult. If file system layering is used, high-level file system features (such as replication, encryption, and compression) can be developed independent of the underlying physical file system. If each physical file system adheres to a similar interface, then these higher level services can run over several highly-tuned physical layers.

Layer substitution can also be used to improve performance by replacing a less efficient layer with a more specialized layer. Transport layers serve as a good example of this practice. We have developed two transport layers. The first is a version of NFS modified to bypass arbitrary operations. This layer provides the full generality of a transport layer, including operation between any two arbitrary address spaces. It also adopts a very general approach to argument passing, mandating repeated copying of arguments as they move from place to place. We have also developed a specialized transport layer, the *utok* layer. This transport layer functions only from user-space to kernel-space. Because this layer solves a much more constrained problem, it is much more efficient than the general NFS-based transport layer, requiring only one copy as arguments are moved into kernel-space.

4.4 Cooperating Layers

Layered design encourages the separation of file systems into small, reusable layers. Sometimes, services that could be reusable occur in the middle of an otherwise special purpose file system. For example, a distributed file system may consist of a client and server portion, with an RPC service in between. One can envision several possible distributed file systems offering simple stateless service, exact Unix semantics, or even file replication. All would have need of the RPC service, but such a service would be buried in the internals of each specific file system, unavailable for reuse.

Cases such as these call for *cooperating layers*. The reusable service is built as one layer, and the rest is split into two separate, cooperating layers. When the file system stack is composed, the reusable layer is placed between the others. Because it is encapsulated in a separate layer, the reusable layer is available for use in other stacks. Ficus illustrates this case, placing an optional transport layer between two cooperating layers. Data compression and decompression over a slow link as described in the example beginning this chapter is another example of cooperating layers. To compress data being moved over a slow link, a compression

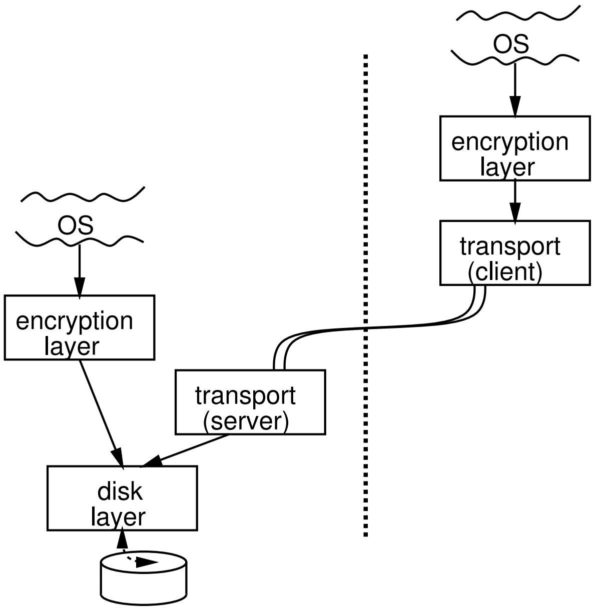


Figure 4.2: Multi-layer access for transmission of encrypted data.

and a decompression layer were wrapped around the transport layer. These layers cooperate to compress and decompress data, providing the appearance of standard, uncompressed data both above and below the pair.

4.5 Multi-layer Access

File system stacks are constructed layer by layer, each adding functionality. Users access all files through the top of the stack—each layer then has its opportunity to influence that user's action.

There are some cases when it is advantageous for user-level programs to bypass stack layers. Because each stack layer is nameable in our design, it is possible for knowledgable programs to choose to avoid upper stack layers. The example of Section 4.1 illustrated one example of why this might be done, to back up data without uncompression. An encryption layer would present a similar case; one may choose to back up the encrypted data rather than the clear-text. With encryption, it might also be advantageous to provide only encrypted data for remote access (see Figure 4.2). By requiring decryption be done at the clients machine, clear text is made not available to the network as a whole.

4.6 File System Testing and Verification

Layered file system design is a valuable tool for software verification. Production-quality software requires very careful quality control. Separation of file system services into several layers minimizes the impact of modifications of one module on others.

The layer interface serves as a tight firewall between modules. Layers do not share data structures, so layer interaction is limited to a relatively small number of well-documented operations. Changes to a monolithic file system typically must consider possible interaction with about 10,000 lines of code. Changes to a layered file system can often be isolated to one layer, minimizing the amount of affected code to about one-quarter what would otherwise be effected. The smaller size of individual layers means that testing and verification can be more tightly targeted, and therefore more quickly completed.

4.7 Compatibility with Layers

Layering allows unprecedented flexibility in the configuration of a filing environment. By allowing re-use of existing services and interface extension, layering also aids rapid development of new file systems. But rapid development has the potential to rapidly create incompatibilities.

Unix is not the only operating system in use today. Currently, few operating systems support stackable filing, and it is unlikely that many non-Unix operating systems ever will. Interoperability between machines is still required. Ideally, any machine would be able to access the resources of another supporting stackable filing, even though they use different operating systems.

Finally, as the number of machines participating in shared filing grows to campus and nationwide scale, hope of universal agreement on one software version is impossible. Many incompatibilities arise as scale grows and administrative differences and issues of autonomy become important.

There are several approaches to resolving these compatibility problems in a stackable architecture. A *compatibility layer* can reconcile incompatible software versions by mapping between differences. If two layers have similar, but not exact, views of the semantics of their shared interface, a thin layer can easily be constructed mapping incompatible operations into their equivalent. This approach has been taken several times to adapt to minor incompatibilities of existing interfaces (see Section 5.5).

There are still many computers that will not support a layered interface in the near future. The installed base of IBM PC and compatible computers approaches 18 million, and many other non-Unix machines exist. Because transport layers bridge machine and operating system boundaries, they offer a promise to support an even wider computing environment.

NFS has become widely available on a number of different operating systems, including IBM and Macintosh personal computers as well as VMS minicomputers and MVS mainframes. NFS can be placed on the top of a file system stack, making basic services available on other machines. Because NFS is not extensible, new operations will not be available, but NFS includes most standard file system capabilities.

In addition to addressing fundamental hardware and software incompatibilities, mapping services can also sometime be applied to bridge administrative differences. Most versions of Unix identify users by a small number known as a user-identifier, or *uid*. Distributed file systems often require all sites to share the same mapping of users to uids. There is rarely coordination of uid assignment between different administrative bodies, and it is difficult to re-assign uids when conflicts arise. A uid mapping layer can provide a small-scale solution to this problem by mapping between uid assignments of two administrative domains.

4.8 Out-of-kernel Development

Stackable layering is a natural complement to a micro-kernel design. Each layer can be thought of as a server, and operations are simply RPC messages between servers. In fact, new layer development usually takes this form at UCLA. Figure 4.3 shows this strategy. The NFS-based transport layer serves as the RPC interface, moving all operations from the kernel to a user-level file system server. Another transport service, the utok (user to kernel) layer, allows user-level calls on lower-level vnodes which exist inside the kernel. As a result, layers may be developed and executed as user code. Although this RPC has real cost, careful caching can provide acceptable performance for an out-of-kernel file system [SKS90].

But stackable layering offers a valuable complement to this approach. Because file system layers each interact only through the layer interface, the transport layers can be removed from this configuration without affecting a layer's implementation. The file system can then run in the kernel, avoiding all RPC overhead. Thus with stackable layering, the advantages of micro-kernel development are available when needed, but the performance overhead of RPC may be removed for production use. Advantages of both micro-kernel and integrated-

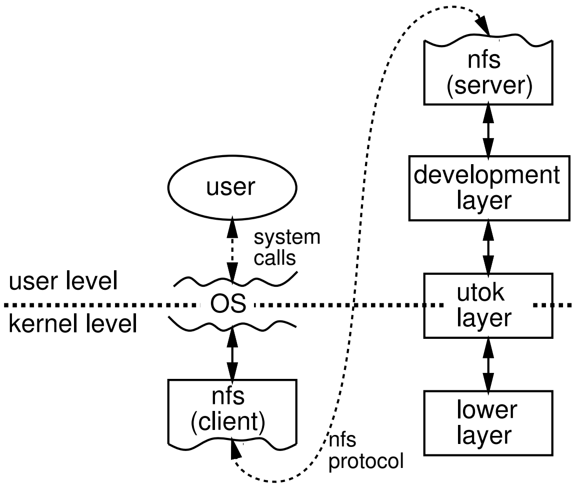


Figure 4.3: User-level layer development via transport layers.

kernel development are available.

CHAPTER 5

Example Layers

As with many new technologies, stackable layers benefit from new design approaches. The previous chapter examined some of these approaches in the abstract. This chapter instead presents a number of layers implemented at UCLA as case studies of layered file system design.

We begin with a discussion of a simple "pass through" layer. A useful variation of this "null" layer is a measurements collecting layer. This is followed by a discussion of two transport layers, layers joining two address spaces or machines. Section 5.4 examines Ficus, the most ambitious use of file system layering to date. Version mapping layers provide valuable compatibility. A cache consistency layer illustrates a service common to remote file access. Finally, ideas for several other layers are presented.

5.1 A Minimal Layer

A common first program when learning a new programming language is "Hello, World", a program which simply prints a welcome message to the output device. Such a basic program illustrates the fundamental constructs of the language. Unnecessary complexity at this level is often a sign of an inappropriate level of abstraction [Ros88].

Like a "Hello, World" program, the first layer we sought to develop under the 405 interface was the minimal layer. While the minimum program prints output, the minimal interesting layer is a "pass-through" layer. Providing no change in semantics, such a layer merely forwards all operations to a lower layer for processing. A minimal layer illustrates the support services needed for the successful operation of a layer.

The *null layer* is such a minimal layer[1]. The null layer has been an important tool in investigating the difficulty of layer development and layer overhead.

5.1.1 Null layer details

A null layer does nothing more than pass all operations to the layer it is mounted over. Although the null layer does not "do anything", it provides all the components of a real layer. These components provide three basic tasks: layer initialization, layer vnode management, and operation handling.

Layer initialization consists of interfacing with the rest of the kernel and the

[1] First implemented by Yuguang Wu under Sun's vnode interface, it has since been ported to the new interface, where it was tuned by John Heidemann.

VFS mount mechanism. Allocation and reference counting of null layer vnodes is done by null layer vnode management code. Operation handling is quite simple; the vnode *inactive* operation, *open*, *get-attributes*, and a bypass routine are implemented. Open must be provided because it is slightly irregular. Inactive is called when a null vnode is no longer needed (its reference count drops to zero); special code must then free the null layer vnode for the file. The get-attributes routine is intercepted because part of file attributes is a unique file system identifier which must be unique to the null layer. A bypass layer handles all other operations, passing them to the next lower layer.

5.1.2 Layer tuning

The null layer does not alter the semantics of the stack it is pushed upon, it simply adds another layer. Time spent executing in the null layer is therefore all layer overhead. Because no useful work is done by the null layer, it represents the "worst case" in relative overhead. More sophisticated services would spend some time accomplishing tasks for the user, providing less relative overhead. The null layer was therefore immediately useful in minimizing this overhead with performance evaluation and tuning.

Chapter 6 examines the performance of layering in detail. Here we will try to characterize the qualitative aspects of layering costs.

Initial evaluation of the layered interface revealed that the majority of overhead was occurring in vnode creation. Each time a new file was accessed, a null layer vnode data structure would be created for it. Very quickly the last reference to the file would go away, and this data structure would be immediately released. Consistent with principles of locality, this same file would often then be immediately re-accessed, unnecessarily repeating this work. In early versions of the null layer, nearly 85% of name translation overhead was spent creating vnodes.

The solution to this problem is two-fold. Most important, some null node caching is employed. Rather than destroy null nodes immediately when no longer needed, they are cached. If the same node is used again in a short period of time, it can be immediately reused without expensive re-creation. The second solution to this problem is to use less expensive memory allocation routines, trading some space for time.

As a result of careful null node management, layer overhead has been reduced from 10-20% to 1-2% of elapsed time of a mix of file system operations when stacked over a standard UNIX file system. A complete analysis of current layering costs can be found in Chapter 6.

5.1.3 Null layer use

While the null layer is useful in evaluating layer performance, it is also a valuable tool in the construction of other layers.

Construction of new layers at UCLA typically begins by copying and renaming the null layer. This cloning allows easy re-use of existing layer mechanisms. This approach has proven proven quite valuable in practice, allowing useful new layers to be created with as few as 70 new lines of code. Section 6.2 discusses the effort required for new layer development in detail.

Duplication of null layer code suggests that perhaps some layer commonality could be merged into shared subroutines. On the other hand, the success of the null layer in the development of new layers indicates that a reasonable level of abstraction has been achieved.

5.2 A Measurements Layer

Kernel measurements have often been difficult to take, traditionally requiring changes deep in the implementation of the operating system. If not done carefully, such changes can easily disturb what is being measured. Installation of these changes imply kernel re-compilation and installation, interrupting work on the target system. Furthermore, such changes can rarely be left in production code because of potential overhead. Because of these problems, file system performance analysis has been typically reserved to a few research labs and systems software houses. Even there, the effort required to take quality measurements has resulted in the publishing of only a few performance studies based upon real environments [Flo86b, Flo86a, OCH$^+$85].

Stackable layers offers an alternative to this approach. A *measurements layer* can cleanly separate instrumentation from the subject being measured. Although a layer cannot collect data on the internal functioning of other layers, it is ideal for collecting general information such as traces and usage patterns. A single measurements layer can easily be added to take measurements of any existing file system layers. Run-time stacking allows a measurement layer to be added to a machine without rebooting, and measurements can easily be configured in and out of daily use without code changes. The ease-of-use of a measurements layer raises the potential for systems administrators to examine their configuration for possible tuning.

At UCLA, Yuguang Wu has built a prototype measurements layer [Wu91]. A modification to the null layer, the measurements layer records the entry and

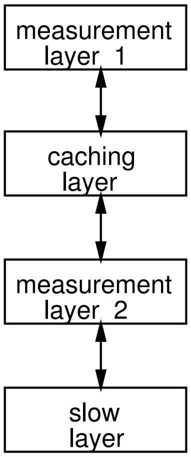


Figure 5.1: Measurement layers analyzing a caching layer.

exit times of operations, as well as interesting operation arguments. With a post-processor, the layer can examine working set size.

The measurements layer is a particularly interesting stackable layer because it is completely *semantics free.* The measurements layer should do nothing to change the semantics of the interface, it should merely record what passes through it. Most other layers alter the semantics of their stack. Because the measurements layer is completely semantics free, it is the best example of a layer which could be added anywhere in a stack while in use.

A file system stack offers an unusual environment for taking measurements. While a measurements layer cannot intrude on the private data of any individual layer, it can be inserted at several different places in a file system stack, anywhere layer interface is present. For example, Figure 5.1 shows a caching layer surrounded by two measurements layers. By taking the difference in the traffic through the two measurement layers, the effectiveness of the caching layer can be evaluated.

There are a number of interesting directions in which a measurement layer could progress. More sophisticated analysis is always possible. More interesting would be a merger of dynamic kernel loading and a measurements layer. A measurements layer could provide a framework for pluggable measurements modules which could be loaded at run-time into an active kernel. Examination of real systems with a measurement layer is also one of the most interesting possibilities.

5.3 Transport Layers

Like the null layer, the sole purpose of a transport layer is to forward all operations to the next layer down the stack. But unlike a null layer, the layers above and below a transport layer are in different address spaces. The transport layer bridges the gap, moving all operations from one domain to the other.

The duties of a null layer are mapping vnodes and file system identifiers between layers. To these, a transport layer adds the burden of transmitting the arguments and results of the operation from one address space to another.

A transport layer can be thought of as being composed of two halves, one half running in the address space of the layer above, the other in that of the layer below. In networking terminology, these would be the client and the server, respectively. Between these two halves some protocol is used to transport operations and arguments. This protocol is distinct from (but usually very similar to) the vnode interface "protocol" and varies from transport layer to transport layer.

We have implemented two transport layers. The first is a general transport layer capable of moving operations between any address spaces on machines connected by Internet network protocols. This layer uses a modified version of Sun's NFS protocol for internal communication [SGK$^+$85]. Our second transport layer is much more specialized. The *utok* layer maps user-level vnode operations to operations on kernel-level vnodes of the same machine. This layer uses one new system call and an XDR package to move data in and out of the kernel address space. Figure 4.3 (page 48) illustrates the use of both of these layers. NFS is used to access an out-of-kernel file system server, and the utok layer provides transport back in the kernel from this server.

Transport layers are an important class of stackable layers. Our experiences implementing and using these layers have presented several observations. While we found transport layers very useful and reusable, current implementations have several drawbacks.

Our first observation is that stackable layers and transport layers combine quite well. Because transport layers allow one machine to export services it provides, they can be used to bridge software or hardware compatibility problems. Transport layers also serve as an example of layer substitution. In out-of-kernel development we use the NFS-based transport layer to access the user-level server, but switch to the utok layer for more efficient return to the kernel.

By providing the 405 interface at the system-call level, the utok layer makes several important user-level programs possible. Most important is the ability to debug file system layers at user level, even though they are mounted over lower layers running in the kernel. Details of this approach are found in Section 4.8. The utok layer also makes the vnode interface available to standard user programs. Ficus makes extensive use of this facility to implement some services as user-level programs, rather than in the kernel. A suite of utility program allow examination and editing of Ficus-specific data structures, data accessible only through Ficus-specific vnode operations. Since the utok interface expands automatically with the addition of new operations, all operations are immediately available at the user-level. This provides a much more malleable environment than the traditional approach of providing new functionality with new, hand-written system calls.

Our experiences with existing physical file systems have shown that they often encompasses too many abstractions, and could be much more useful if broken into several smaller components. Interestingly, the same problem occurs in our NFS-based transport layer. NFS provides several logically independent services: transport service with an RPC protocol, statelessness for failure recovery, and a cache consistency algorithm. While the transport service provided by NFS is very useful (once extended to support additional operations), its attempts to provide stateless services has been frustrating. NFS' statelessness alters the semantics of the interface, making the view of a file system through NFS different from that provided by direct access. In a stackable environment, NFS might be better implemented as a single transport layer surrounded by layers mapping from a stateful operations to the stateless transport layer.

Finally, cache consistency in a distributed system is often a difficult issue. While NFS' approach to cache consistency is good enough for the vast majority of file system use, it can cause problems when NFS is used as a transport layer. Rather than provide cache consistency as part of a transport layer, we have been successful at separating cache consistency algorithms into a separate layer. This allows mount-time choice between inexpensive caching algorithms with poor consistency and more sophisticated algorithms provided completely transparent semantics. Section 5.6 describes our cache consistency layer in detail.

5.4 Ficus: replicated file services

Ficus is a distributed file system supporting replication with optimistic concurrency control [GHM$^+$90, Guy91]. It is also one of the most ambitious uses of stackable layering to date, using two cooperating layers in its implementation.

Ficus is an example of cooperating layers, being composed of two separate layers (logical and physical) with an optional transport layer in-between for remote access. Rather than write directly to disk, a UFS layer is utilized for all low-level storage. Figure 4.1 (page 43) shows two configurations for a Ficus stack of layers.

The upper Ficus layer presents the appearance of a single standard (non-replicated), highly available file system to the user. This Ficus logical layer selects a particular replica to serve a user's requests, coordinating access to multiple replicas. It also coordinates propagation of information about replica changes, and coordinates the reconciliation of divergent replicas, and manages location and automatic mounting of volumes as necessary.

The lower Ficus layer coordinates physical disk access, and so is known as the Ficus physical layer. It maps Ficus files to an underlying UFS file system, allowing file access by Ficus "file id" and supporting extended replication attributes and directory entries.

Ficus has proved to be an invaluable example of the use of stackable layers, driving both the design of the layer interface and stackable techniques.

Ficus' use of the stackable approach means that no code in the Ficus layers is needed to implement RPC or low-level disk management. Cooperating layers has allowed the reuse of transport services, since remote access is needed within the "middle" of Ficus, between the layers. Stacking over already existing physical storage layers allowed re-use of existing disk storage facilities.

The extensible nature of the 405 interface has proved particularly valuable in Ficus. Replication requires a number of support operations. These operations were accommodated in early versions of Ficus by overloading existing services and encoding information in unusual file names and special control files. Several styles of overloading are required to pass different kinds of information through the unsupportive interface, greatly adding to implementation effort. Support of new operations provides a clean approach to adding new operations, delegating the complicated details of encoding to automated tools.

On the other hand, the current Ficus architecture could be improved. From a layering standpoint, the Ficus physical layer encompasses too much functionality. What is desired is a simple, flat namespace of files with extensible attributes for low level storage. Because an extensible interface was unavailable when the layer was designed, all of these features are merged into the Ficus physical layer. Separating these services into a number of layers would offer more flexibility in configuration, and provide services useful in the construction of other layers. Different methods of physical storage could be used depending on requirements for compatibility and efficiency. Figure 5.2 illustrates a more modular design of the physical layer.

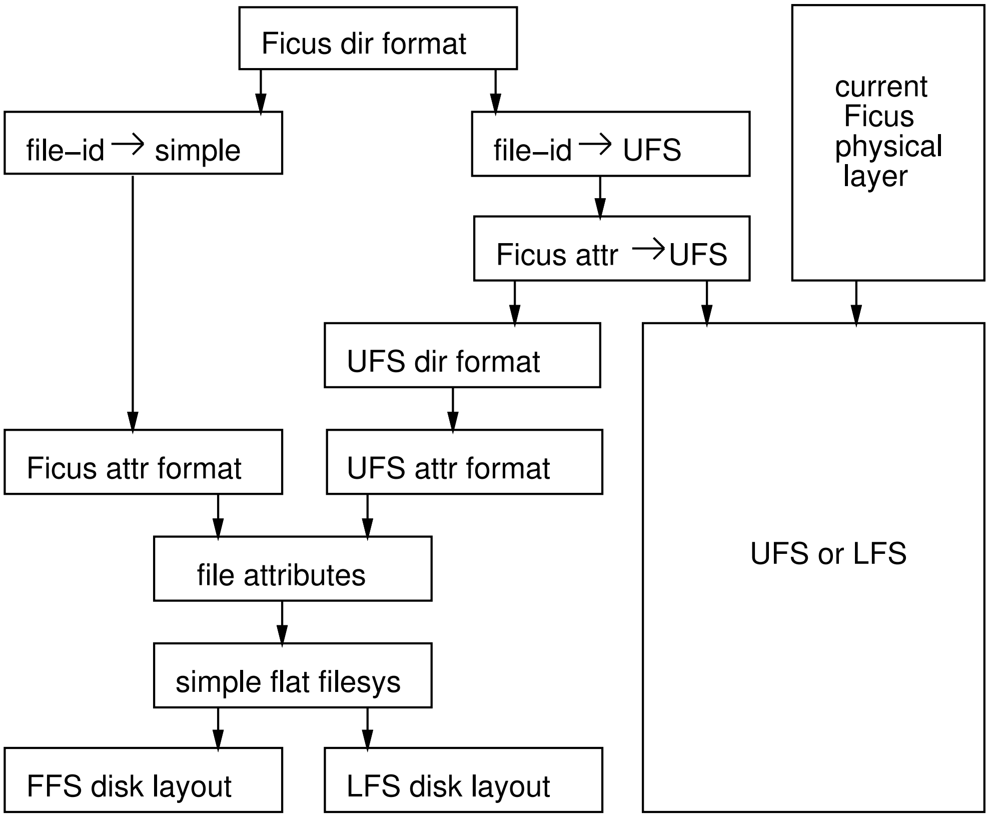


Figure 5.2: Decomposed Ficus physical and disk storage layers.

5.5 Interface Versioning Layers

An extensible interface allows incredible freedom for development. It also offers the potential for incredible incompatibility as interfaces diverge to support slightly different capabilities. Even when very similar operation semantics are provided, a difference in argument type or syntax can make similar operations unusable.

In addition to interface compatibility, the many administrative domains of the real world often results in an administrative incompatibility. Sites standardize on different methods of file storage; even user identification varies from installation to installation.

Often, incompatibilities are slight. A short patch can usually map over syntactic or slight semantic differences. File system stacking allows this mapping to be easily encapsulated in an independent layer. Such a mapping layer burdens neither the caller nor the callee with the immediate responsibility to reconcile incompatibilities, instead confining translation to a layer which can be thrown away when a common interface is established. Since changes are not made to the original layers, new compatibility layers can be swapped in easily.

Several times in our file system development we have found it necessary to employ mapping layers. The *newo* layer maps between versions of the open system call, and also provides extended-size file identifiers. The *shrinkfid* layer maps the other direction, translating large file-ids into NFS-compatible "short" file-ids. In our experiences, mapping layers can be developed quite rapidly by modifying the null layer. Each of these layers was implemented in a few days.

In addition to mapping between differing interfaces, we have also employed mapping layers to translate between different administrative domains. User identification in NFS is done by "user-id". To maintain autonomy, different administrative domains rarely coordinate allocation of these identifiers. As a result, NFS communication between two sites can result in improper file ownership and access when user-id allocation conflicts. To allow bilateral sharing between different administrative domains, a user-id mapping layer (*umap*) was developed by Tom Page [Pag91]. Developed in about a week, such a layer illustrates the power of stacking to address new problems rapidly.

5.6 A Consistency Layer

Remote filing systems are a part of the daily lives of nearly every workstation user today. Common storage of rarely used files on a centralized machine minimizes expenses both by common use of resources and simplified administration. Distributed filing services are so ubiquitous that no workstation vendor today ships a product without some remote filing capability.

Caching is critical to the performance of remote filing. Locality is so strong in filing environments, and the performance ratio between local and remote access so great that caching is a requirement of any distributed filing solution. Surprisingly, there is little agreement in the quality of caching services required. Service ranges from no explicit guarantee of cache coherence to absolute transparency in a cluster of workstations. On the other hand, perhaps lack of agreement is not surprising, given the widely varying demands of user applications, from one-time text editing to multi-user distributed databases.

Because of this wide variation in both quality and demand for service, cache coherence represents an ideal application for a modular, stackable solution. The ability to offer a number of "pluggable" coherency solutions is an attractive one, allowing the user to trade off consistency for performance as desired.

For this reason, a prototype cache consistency layer was developed. The *consistency layer* uses a simple token passing mechanism to insure cache coherence over NFS.

Figure 5.3 illustrates the structure of the consistency layer. Sitting above NFS, the layer communicates with its companion layers on other machines. Before each file operation, the layer acquires a *token* for that file. Accompanying this token is the modification time of the file. Based on this time, the consistency layer calls upon NFS to purge its cache as required.

The consistency layer employs some simple optimizations. Between operations, the token remains at the site of last use. Consecutive operations at the same site therefore require no unnecessary remote access, they simply re-use the existing token. Optimizations such as multiple read-only tokens and token regeneration could be added easily within the framework of the existing layer implementation.

In addition to the separation of coherence algorithms from the actual transport mechanism, the architecture of the consistency layer results in another novel feature. To communicate between consistency layer instances at separate machines, generalized "upcalls" are employed. The consistency layer requires the generalized flow of control of an RPC protocol, but requires only operations on files similar to file system interfaces. Rather than develop a new RPC protocol, the consistency layer uses the upcall mechanisms of the 405 interface. An RPC layer is configured to operate between pairs of consistency layers, and file iden-

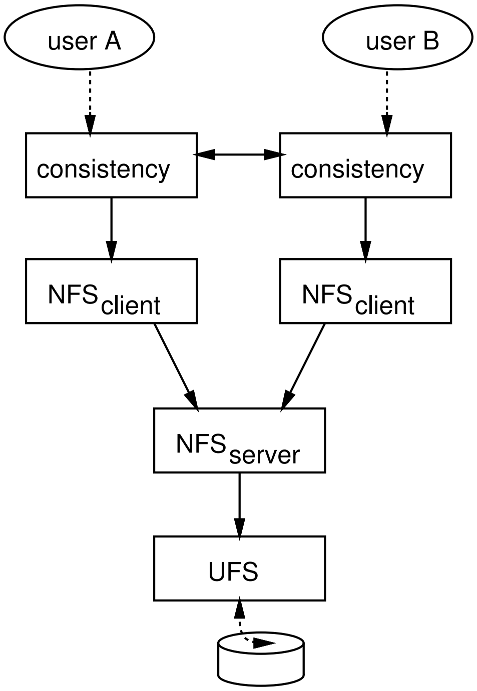


Figure 5.3: Logical configuration of a cache consistency layer.

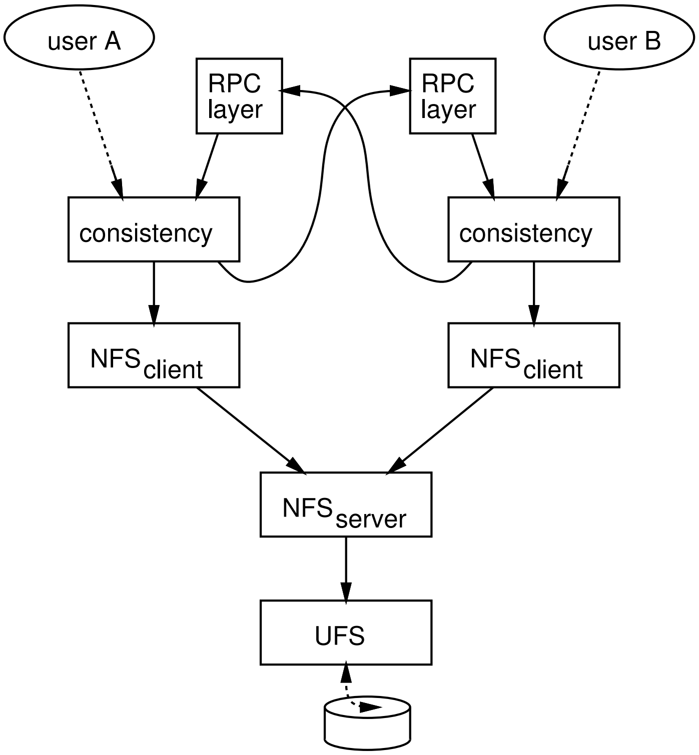


Figure 5.4: Layer configuration of the prototype cache consistency layer.

tifiers are used as low-level names for files[2]. Figure 5.4 illustrates all layers used in consistency layer operation.

The unusual figure-eight mounting pattern shown in Figure 5.4 is required for bi-directional RPC flow. It also requires a careful mounting sequence for correct operation. As such, this represents an ideal opportunity to apply on-demand "automounting" as described in [PGP+91].

The consistency layer has proven to be an exceptional demonstration of the benefits of file system layering. It cleanly separates remote access methods from the cache consistency protocol of a distributed file system, allowing either to be substituted. Its development in one week's time as a class project is a further example of the benefits of layering for rapid file system development. Such rapid development was possible only with the re-use of existing RPC and remote file

[2] Ironically, this "RPC layer" is actually the same modified NFS transport layer used for remote file access. These dual aspects of NFS are another example of how it could be split into separate layers: one to provide RPC service, one to provide caching, and another to implement a cache coherency policy.

access methods.

5.7 Other Layers

The layers described here are but a few of the many interesting file system services one might provide. This section touches on some other layers that have been built or might be built.

Several layers have been prototyped at UCLA as class projects:

file versioning This layer provides storage and creation of file version as they are edited, similar to file versioning services in VMS.

compression File compression has the potential to significantly increase effective storage [Cat90]. Compression and decompression in a layer make this savings possible without explicit user intervention.

encryption The entire UNIX file system must be accessible to administrative personnel, if only to do file system backups. An encryption layer automatically encrypts all on-disk information, resulting in truly private data.

second-class replication Laptop computers are becoming increasingly powerful and available. While their portability is attractive, currently the user must manually coordinate duplication of information between the laptop and primary computer. This layer provides support for second-class replication targeted at the laptop environment.

To date, a dozen file system layers have been implemented. Many other layers are possible:

Byzantine In software development, one must be robust to a variety of faults in other layers. A "Byzantine" layer would intentionally simulate such faults to explore what would otherwise be unusual occurrences.

Delay File systems designed for very large scale networks must assume high operation latency. Developing such software under these conditions is not always easy; it's difficult to isolate faults in a machine thousands of miles away. A delay layer could simulate the effects of random network delay distributed system performance without requiring actual physical distance.

Multi-disk partition Traditional UNIX file systems are limited to a single physical disk partition, which are limited to a single physical disk. Provided by device drivers in some versions of UNIX today, a multi-partition layer could more portably merge several physical partitions together, providing the illusion of a single, very large disk.

Volume The opposite of a multi-partition layer, a volume layer divides existing disk partitions into volumes, more manageable sub-units. Like disk partitions, volumes have resource limits, but volume reconfiguration can be a much lighter weight-operation.

Caching There are many opportunities for caching in a file system stack. Slow access from a remote machine, CD-ROM or WORM disk can be cached on magnetic media. A layered architecture allows a caching service to built from largely existing pieces.

Flat directory services Many sophisticated file systems would like a simple, highly efficient method to access "files" and are not concerned with a hierarchical directory. "Inode level" access would be a valuable service in the construction of many layers.

User-centered naming A number of recent proposals have advocated "user-centered naming" [Neu89, PHOR90]. Such a service could be naturally constructed as a naming layer relying on access to many existing file systems through lower layers.

Disk quotas The original Berkeley implementation of disk quotas was tightly integrated with the Berkeley Fast File System. No support is offered for quotas with other physical file systems, and interaction with remote filing is not always obvious. A layered quota implementation would provide a much more portable solution.

Access control lists Like disk quotas, access control lists have usually been implemented as direct modifications to the physical file system. Again, a layered implementation should prove much more portable.

Window system support Window-based file managers typically identify files by icon. To identify file types in UNIX systems, some UNIX file managers maintain a cached list mapping files to icons. Unfortunately, this cache can quickly become invalid with file system activity. A window system support layer could manage this cache, keeping it up-to-date by intercepting user changes.

5.8 Summary

This chapter has described a number of currently existing file system layers. The null layer illustrates the basic characteristics of a file system layer, and serves as the building block of a number of other, useful layers. This approach demonstrates the advantage of layered file systems to providing new services quickly, as exemplified by the measurement and software versioning layers discussed here.

File system layering is key to the implementation of Ficus' replicated filing services. Ficus makes extensive use of layering to provide a robust, large scale, filing environment in daily use today. Ficus makes particular transport layers for remote access and debugging.

The consistency layer illustrates several interesting layer characteristics. Implementing a particular cache consistency policy for a distributed file system, the consistency layer is a fine example of an optional, "value-added" service that can be selected as a user desires. The consistency layers' use of "upcalls" also illustrates a valuable facility of the 405 interface.

Taken together, the wide array of services described in this chapter illustrate the general utility of stackable layers as a method file system design.

CHAPTER 6

Evaluation

A stackable file system design offers great flexibility in configuration and development. As layers are developed, their use as software building blocks can reduce future development times. But these benefits will be for naught if layer overhead has a significant negative impact on overall performance; such an overhead might force the use of monolithic structure to secure acceptable performance. To ensure that this is not the case, a careful evaluation of layer costs was undertaken.

If stackable layering is to aid the development of new file systems, it must have not only good performance, but also a good development environment. This chapter also examines this aspect of "performance", first by comparing the development of similar file systems with and without the new interface, and then by examining the development of layers in the new system.

6.1 Interface Performance

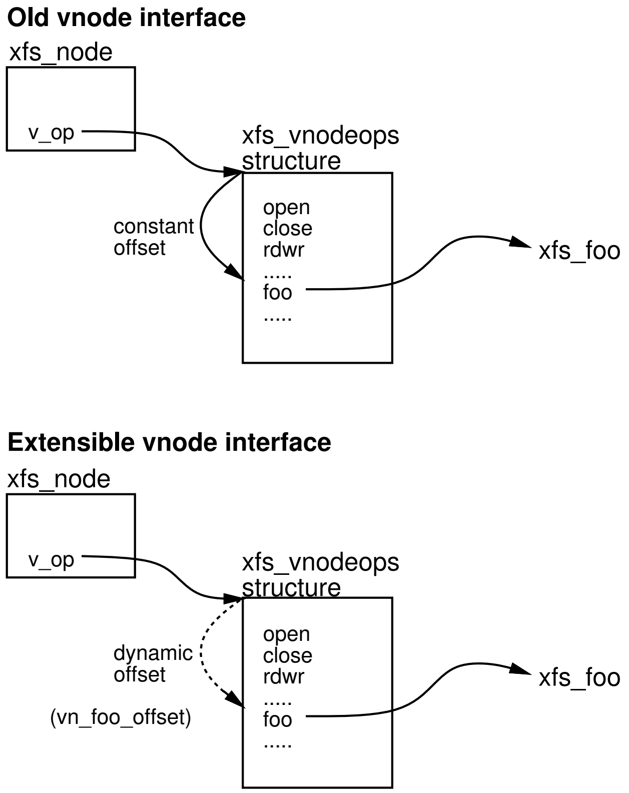
To examine the performance of the 405 interface, we consider several classes of benchmarks. First, we carefully examine the costs of particular parts of this interface with "micro-benchmarks". We then consider how the modifications of the interface affect overall system performance by comparing a kernel running the 405 interface with an unmodified kernel. To determine the cost of multiple layers with the new interface, we evaluate the performance of a file system stack composed of differing numbers of layers.

The 405 interface was implemented as a modification to SunOS 4.0.3. All timing data was collected on a Sun-3/60 with 8 Mb of RAM and two 70 Mb Maxtor XT-1085 hard disks. The measurements in Section 6.1 used the new interface throughout the new kernel, while those in Section 6.1.3 used it only within file systems.

6.1.1 Micro-benchmarks

Parts of the 405 interface are called at least once per vnode operation. To minimize the total cost of an operation, these must be carefully optimized. Here we discuss two such portions of the interface: the method for calling an operation, and the bypass routine.

To evaluate the performance of these portions of the interface, we consider the number of assembly language instructions generated in the implementation. While this statistic is only a very rough indication of true cost, it provides an



```
#define OLD_VOP_FOO(VP) ((*(VP)->v_op->vn_foo)(VP))


#define NEW_VOP_FOO(VP) ((*(VP)->v_op[vn_foo_offset])(VP))
```

Figure 6.1: Operation invocation under the vnode and 405 interfaces.

order-of-magnitude comparison[1].

We began by considering the cost of invoking an operation in the vnode and the 405 interfaces. Figure 6.1 shows the C code for calling an operation. On a Sun-3 platform, the original vnode calling sequence translates into four assembly language instructions, while the new sequence requires six instructions[2]. We view this overhead as not significant with respect to most file system operations.

[1] Factors such as machine architecture and the choice of compiler have a significant impact on these figures. Many architectures have instructions which are significantly slower than others. We claim only a rough comparison from these statistics.

[2] We found a similar ratio on SPARC-based architectures, where the old sequence required five instructions, the new eight. This calling sequence does not include cost to pass arguments to the operation.

We are also interested in the cost of the bypass routine. We envision a number of "filter" file system layers, each adding characteristics to the file system stack. File compression or local disk caching are examples of services such layers might offer. These layers pass some operations directly to the next layer down, modifying the user's actions only to uncompress a compressed file, or to bring a remote file into the local disk cache. For such layers to be practical, the bypass routine must be inexpensive. A complete bypass routine in our design amounts to about 54 assembly language instructions[3]. About one-third of these instructions are used only for uncommon argument combinations, reducing the cost of forwarding simple vnode operations to 34 instructions. Although this cost is significantly more than a simple subroutine call, it is not significant with respect to the cost of an average file system operation. To further investigate the effects of file system layering, Section 6.1.3 examines the overall performance impact of a multi-layered file system.

6.1.2 Interface performance

Encouraged by results of the previous section, we anticipated low overhead for our stackable file system. Our first goal was to compare a kernel supporting only the 405 interface with a standard kernel.

To examine overall performance, we consider two benchmarks: the modified Andrew benchmark [Ous90, HKM$^+$88] and recursive copy and remove of large subdirectory trees. In addition, we examined the effect of adding multiple layers in the new interface.

The Andrew benchmark has several phases, each of which examines different file system activities. Unfortunately, we were frustrated by two shortcomings of this benchmark. The first four phases are very brief, making accurate evaluation of these phases difficult. While the final compile phase is relatively long, on many machines compilation is compute-bound, obscuring the impact of file system performance. On the other hand, taken as a whole, this benchmark probably characterizes "normal use" better than a file-system intensive benchmark such as a recursive copy/remove.

The results from the benchmark can be seen in Table 6.1. Overhead for the first four phases averages slightly more than one percent. The very short run times for these benchmarks limit their accuracy, due to timing granularity. The compile phase shows only a slight overhead. We attribute this lower overhead to

[3] These figures were produced by the Free Software Foundation's gcc compiler. Sun's C compiler bundled with SunOS 4.0.3 produced 71 instructions.

	Vnode interface		405 interface		Percent
Phase	time	%RSD	time	%RSD	Overhead
MakeDir	3.3	16.0	3.2	14.8	−2.76
Copy	18.8	4.6	19.1	5.0	1.92
ScanDir	17.2	5.2	17.8	7.9	3.13
ReadAll	28.3	2.0	28.8	2.0	1.70
Make	327.6	0.4	328.1	0.7	0.15
Overall	395.2	0.4	396.9	0.9	0.45

Table 6.1: Modified Andrew benchmark results running on kernels using the vnode and the 405 interfaces. Time values (in seconds, accurate to one second) are the means of elapsed time from thirty sample runs; %RSD indicates the percent relative standard deviation (σ_X/μ_X); overhead is the percent overhead of the new interface. High relative standard deviations for MakeDir are a result of poor timer granularity.

the fewer number of file system operations done per unit time by this phase of the benchmark.

To exercise the interface more strenuously, we examined an additional benchmark. This benchmark employed two phases, the first doing a recursive copy and the second a recursive remove. Both phases operate on large amounts of data (a 4.8 Mb `/usr/include` directory tree) to extend the duration of the benchmark. Because we knew all overhead occurred in the kernel, we measured system time (time spent in the kernel) alone. This greatly exaggerates the impact of layering, since all overhead is in the kernel and system time is usually small compared to elapsed time (“wall clock” time, what a user actually experiences). As can be seen in Table 6.2, overhead averages about 1.5%.

6.1.3 Multiple layer performance

Since the stackable layers design philosophy advocates using several layers to implement what has traditionally been provided by a monolithic module, the cost of layer transitions must be minimal if it is to be used for serious file system implementations. To examine the overall impact of a multi-layer file system, we analyze the performance of a file system stack as the number of layers employed changes.

To perform this experiment, we began with a kernel modified to support the

	Vnode interface		405 interface		Percent
Phase	time	%RSD	time	%RSD	Overhead
Recursive Copy	51.57	1.28	52.54	1.38	1.88
Recursive Remove	25.26	2.50	25.48	2.74	0.89
Overall	76.83	0.87	78.02	1.33	1.55

Table 6.2: Recursive copy and remove benchmark results running on kernels using the vnode and 405 interfaces. Time values (in seconds, accurate to one-tenth of a second) are the means of system time from twenty sample runs; %RSD indicates the percent relative standard deviation; overhead is the percent overhead of the new interface.

405 interface within all file systems and the vnode interface throughout the rest of the kernel[4]. At the base of the stack we placed a conventional Unix file system, modified to use the 405 interface. Above this layer we mounted from zero to six null layers, each which merely forwards all operations to the next layer of the stack. Upon those file system stacks we ran the benchmarks described in the last section. This test illustrates a particularly demanding use of layers since each layer provides full layer overhead without any additional functionality.

Figure 6.2 shows the results of this study. As can be seen, performance varies nearly linearly with the number of layers used. The modified Andrew benchmark shows about 0.3% elapsed time overhead per layer. Alternate benchmarks such as the recursive copy and remove phases, also show less than 0.25% overhead per layer.

To get a better feel for the costs of layering, we also measured system time, time spent in the kernel on behalf of the process. Figure 6.3 compares recursive copy and remove system times[5]. Because all overhead is in the kernel, and the total time spent in the kernel is only one-tenth of elapsed time, comparisons of system time indicate a higher overhead: about 2% per layer for recursive copy and remove. These overheads were computed by least squares fits to the sample data, yielding good correlations of 0.9 for the system time benchmarks, and 0.7 to 0.9 for elapsed times. Slightly better performance for the case of one layer in Figure 6.3 results from a slight caching effect of the null layer over the standard UFS. Differences in benchmark overheads are the result of differences in the ratio

[4] To improve portability, we desired to modify as little of the kernel as possible. Mapping between interfaces occurs automatically when the file system is entered.

[5] The timing method employed in the modified Andrew benchmark does not include system time statistics.

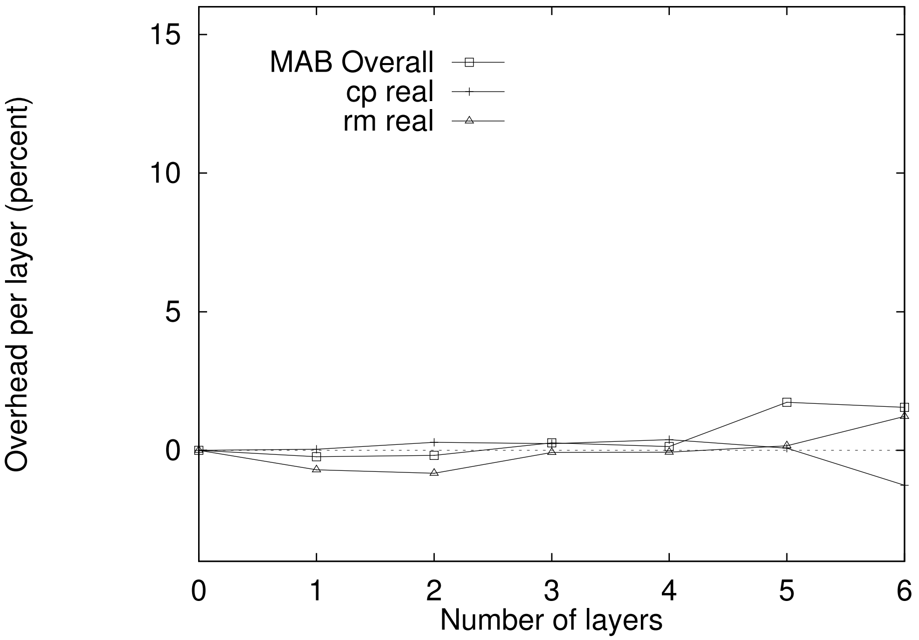


Figure 6.2: Elapsed time of recursive copy/remove and modified Andrew benchmarks as layers are added to a file system stack. Each data point is the mean of four runs.

between the number of vnode operations and benchmark length.

We draw two conclusions from these figures. First, elapsed time results indicate that under normal load usage, a layered file system architecture will be virtually undetectable. Also, system time costs imply that during heavy file system use a small overhead will be incurred when numerous layers are involved.

6.2 Layer Implementation Effort

The goal of stackable file systems and this interface is to ease the job of new file system development. Clearly, importing functionality from existing layers saves a significant amount of time in new development. Ficus, for example, borrows network transport and low-level disk storage facilities from pre-existing file systems, for great savings in implementation effort.

We address the issue of file system development effort in several different ways. First, we compare development of similar file system layers under the

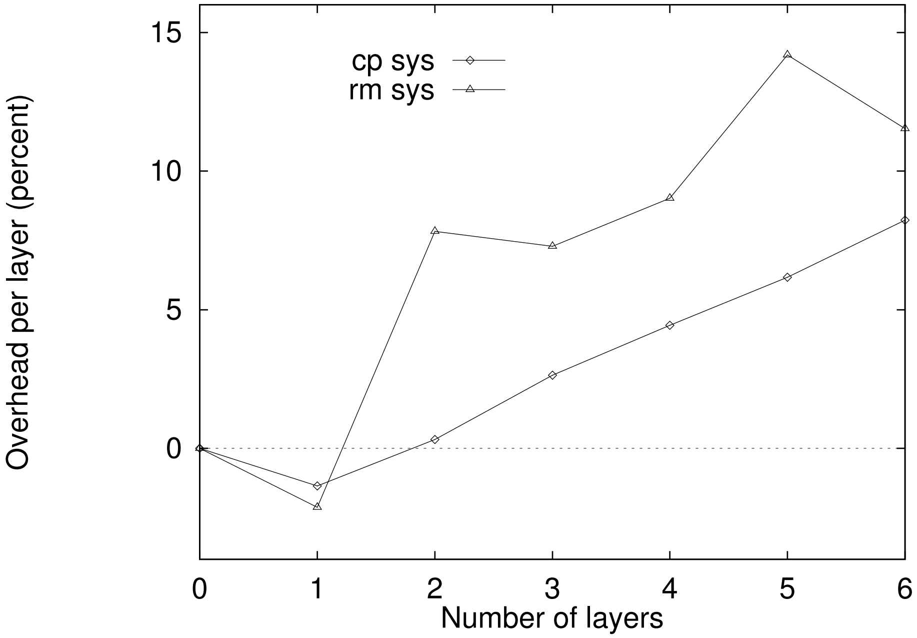


Figure 6.3: System time of recursive copy/remove and modified Andrew benchmarks as layers are added to a file system stack. Each data point is the mean of four runs.

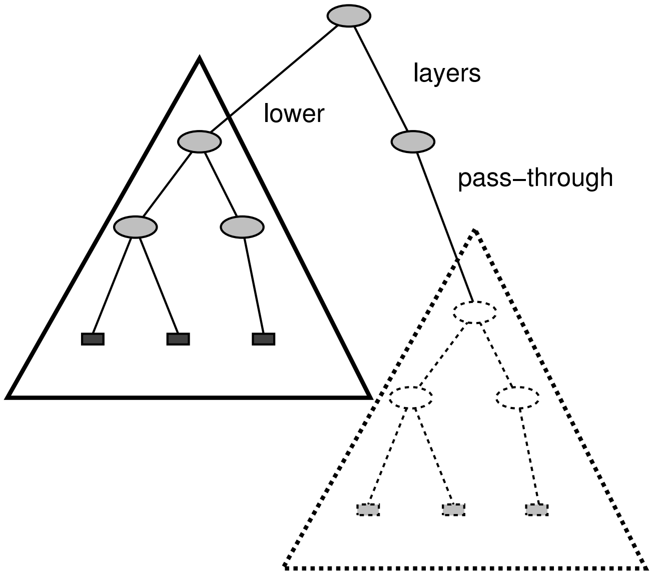


Figure 6.4: A mounted pass-through layer.

existing vnode interface with development in the 405 interface. Then we discuss observations on the development of several layers by those previously unfamiliar with file system layering. We then examine the implementation of these layers in detail to see what about the new interface improves the development cycle. Finally, we discuss the use of layering for a large scale file system project.

6.2.1 Minimum layer development

A first concern when considering development under the 405 interface was that it would prove more complicated than existing interfaces. Most other programmer interfaces do not support extensibility; would this feature make the interface significantly more difficult to use? To evaluate the complexity of the new interface, we will compare two similar file system layers, one implemented under Sun's vnode interface and the other under the 405 interface.

To simplify comparison, we chose to compare a basic "pass-through" layer. Such a layer merely aliases its lower layer. Because the new layer has a second name, the effect is to duplicate a portion of the file system namespace in two places[6]. Figure 6.4 illustrates this duplication.

[6] An unusual characteristic of these layers is that they duplicate the entire namespace over which they are mounted. Most layers choose not to exceed subtree boundaries.

module	loopback file system	merge layer	difference
node.h	10	12	+2
info.h	25	37	+12
subr.c	200	199	−1
vfsops.c	135	173	+38
vnodeops.c	373	211	−162
total	743	632	−111

node.h defines the vnode structure for that file system.
info.h provides declarations for mounting.
subr.c implements node management and other utility routines.
vfsops.c implements the file system mount protocol.
vnodeops.c provides all vnode operations.

Table 6.3: Number of lines of comment-free code needed to implement a pass-through layer or file system, and a brief description of each software module.

The loopback file system implements these features in the standard SunOS 4.0 kernel. It explicitly forwards each operation to the lower layer.

Under the 405 interface, the merge[7] layer performs a similar function. This layer takes advantage of the features of its interface, using a bypass routine and automatic interface configuration.

Table 6.3 shows the number of lines of C code needed to implement the loopback file system and the merge layer. The amount of support code needed for each implementation is very similar, as are layer configuration protocols. The merge layer implementation for vnode operations is much shorter, however, since the loopback file system requires special case code to pass each operation down while the merge layer uses a bypass routine. The services the merge layer provides are also more general, since the same implementation will automatically handle the addition of future operations.

For the example of a pass-through layer, the merge layer provides better functionality with fewer lines of code. We expect this trend to be even more marked in more sophisticated file systems, where the ability to reuse existing functionality without source code changes offers a clear savings in implementation

[7] A merge layer is a null layer which has been modified to pass over mount points in its lower file system. This is done to maintain similarity with the loopback file system.

effort.

6.2.2 Layer development experience

The best way to demonstrate the generality of a new design technique is to apply it widely. Breadth of use by different people, and for different problems is the best way to show wide applicability. To gain wider experience with stackable layers, and particularly to evaluate its use by those other than its developers, the opportunity to develop file system layers was made available to students of an operating systems class at UCLA.

The class was a ten-week graduate seminar on distributed operating systems offered Winter Quarter, 1990. Taught by Gerald Popek and Tom Page, lectures included a one-hour lecture on Ficus and a one-hour student presentation on stackable layering, in addition to a wide discussion of distributed file systems and operating systems in general.

Of the class, seven students elected to do file system layering projects. All students were graduate students enrolled at UCLA. All were proficient programmers, while kernel programming experience ranged from none to considerable. The group divided into two two-person teams and three individual projects. To aid them in their task, each team was provided with an out-of-kernel development environment and a null layer (see Section 5.1) as a framework for their development.

All projects succeeded in provided functioning prototype layers. Prototypes include a file-versioning layer, an encryption layer, a compression layer, a "laptop" layer designed to support second class replication, and a consistency layer providing NFS cache coherence via token passing. Other than the consistency layer, each was designed to stack over a standard UFS layer, providing its service as an optional enhancement.

Self-estimates of development time ranged from 40 to 60 person-hours. This figure includes time to become familiar with the development environment, as well as layer design and implementation.

Although just prototypes, each of these layers provide full layer functionality. All were usable as normal file systems by all user-level programs, seemingly no different from well established, kernel resident file systems. Each (except for consistency, which is NFS-specific) will run over several different lower layers and automatically adapts to future changes in the interface.

Since the end of the class, the encryption layer has been successfully demonstrated in the kernel, and the consistency layer has always run there. The only factor complicating moving layers into the kernel is typically library function availability, although concurrency control is also an issue.

We consider this a powerful example of the ease of development offered by layer interfaces. Previously, new file system functionality required in-kernel modification of current file systems, requiring knowledge of current, multi-thousand line file systems and unsophisticated kernel debugging tools. Instead, students in the class were able to provide significant new capabilities with knowledge only of the layer interface and programming methodology.

6.2.3 Layer development examples

Our experience with class development of layers proved very encouraging. It was possible for new layer prototypes to be developed by non-experts in a matter of weeks. We wanted to examine this matter more closely. What features of the new interface were proving most valuable? How were layered file systems being implemented? What is the structure of new file system layers?

To characterize the structure of new layers, we chose to examine the complexity of each software module making up several different layers. Table 6.4 shows the number of lines of comment-free code in each module of several class layers.

Further analysis is needed to draw useful conclusions from these figures. To better summarize the data, Table 6.5 breaks the code to new layers into several categories. “Core” represents fundamental layer routines such as node allocation, basic operations, and layer configuration. Core routines all come directly from the null layer. “Core changes” are modifications to core data structures and routines, typically additions to data structures and initialization code. “Interface changes” represent changes to the semantics of the current interface (standard operations which are overridden). “Interface additions” are lines of code to add new operations to the interface. “Layer specific” represents code necessary to support the additional functionality the new layer provides.

Several conclusions can be drawn from this data. First, it is clear that a core of software is central to each layer. The node management and layer configuration code (`subr.c` and `vfsops.c`) survive almost completely unchanged in all of these file system layers. One possible future step would be to place this code into library routines.

An important feature of this core is that it represents a basic groundwork that is guaranteed correct. Because of this, simple things really do become simple in a layered design. The umap layer does the logically trivial task of translating user-ids between different administrative domains during remote access. This task is

module	null layer	umap layer	encryption layer	compression layer	consistency layer
node.h	8	8	11	13	54
info.h	23	32	24	30	35
subr.c	100	100	110	102	104
vfsops.c	160	170	164	160	181
vnodeops.c	197	245	253	458	278
encr_key.c	—	—	265	—	—
compr_compression.c	—	—	—	84	—
compr_fixfn.c	—	—	—	89	—
compr_pkg.c	—	—	—	278	—
compression.h	—	—	—	9	—
comprfs_priv.h	—	—	—	6	—
consistent_interface.int	—	—	—	—	22
consistent_token.c	—	—	—	—	323
consistent_token.h	—	—	—	—	2
total	488	555	827	1229	999

Table 6.4: A comparison of several file system layers by lines of C code.

module	null layer	umap layer	encryption layer	compression layer	consistency layer
core	488	488	488	488	488
core changes	0	19	18	14	83
interface changes	0	48	56	261	81
interface additions	0	0	0	0	22
layer specific	0	0	265	466	325
total new code	0	67	339	780	511
total	488	555	827	1229	999
percent changes					
layer specific	—	0	78	63	67

Table 6.5: A evaluation of file system layers by lines of C code.

as trivial a layer as it should be, only 67 additional lines of code are required beyond the null-layer core.

Layered design also simplifies more complex project development. The encryption, compression, and consistency layers were all developed in about 900–1200 lines of code. Once again, a significant portion of this was an automatically generated set of core routines. Furthermore, changes to the core were minimal (as indicated by the "core changes" and "interface changes" rows of Table 6.5). For these file systems, most changes were layer specific, rather than modifications to the core itself, as shown by the "percent changes layer specific" row. The only exception is the compression layer, where a high 261 additional lines of operation handling code was required because of the implementation choice to encode information in filenames.

These layers also serve as an important example of the modularity of layered development. Because each layer relies on the services provided by existing layers, layers can often be implemented in only about 1000 lines of code. If instead they were implemented as direct changes to existing file systems, we hypothesize that there would be about 600–900 lines of modifications and additions dispersed throughout a 10,000 line file system. This traditional approach requires a much greater knowledge before development begins: 10,000 lines of code rather than a 30 operation interface. It also complicates testing and quality assurance by making all existing code suspect for errors, rather than just the code of the new layer.

Examining these layers also revealed a few weak points of the current development environment. Only one new layer added an operation, the consistency layer. There are several factors contributing to this. First, because these layers are only class prototypes, work-arounds to avoid new operations were acceptable, particularly given the one-month effective development time constraint. Instead of adding new operations, the encryption layer handled new information at mount time, and the compression layer relied on direct access to the lower-level layer and encoding information in the filename. A second reason is that installation of new operations is more difficult than it needs to be. A new operation currently requires installation of a new kernel, even in the out-of-kernel environment[8]. We plan to simplify this by supporting dynamically loaded kernel operations and layers in the near future.

Another problem we encountered is one we had met before. Many layers need

[8] For user program to access the new operation, it must pass through the kernel to the out-of-kernel server. For an operation to be transported through one address spaces to another, each intermediate layer must be aware of the operation and its arguments to handle the RPC.

to store some additional information about each file. Compression, for example, needs to record the particular compression algorithm used, and encryption needs to store the per-file key. Current layers handle this in an *ad hoc* fashion, typically transparently stealing some space from the beginning of file data storage. A more general solution is clearly needed, providing general purpose per-file attribute functionality similar to file resource forks in the Macintosh operating system [App85].

6.2.4 Large scale example

The previous section discussed our experiences in stackable development of several prototype layers. This section concludes with the the results of developing a replicated file system suitable in practice for daily use.

Ficus is a "real" system, both in terms of size and use. It is comparable in code size to other production file systems (12,000 lines for Ficus compared to 7–8,000 lines of comment-free NFS or UFS code). Ficus has seen extensive development over its two-year existence, and it is now in daily use at UCLA for its developers' home file storage.

Stacking has been a part of Ficus from its very early development. Ficus has provided both a fertile source of layered development techniques, and a proving ground for what works and what doesn't.

Particularly valuable in Ficus are the concepts of cooperating layers, an extensible transport layer, and out-of-kernel development. The concept of cooperating layers is fundamental to the Ficus architecture, and has succeeded in locating necessary portions with both the user and the data. Between the Ficus layers, the optional transport layer has provided easy access to any replica, leveraging location transparency well. Finally, the out-of-kernel debugging environment has proved invaluable, saving months of development time. The first reaction to finding a bug is often repeating the bug in the user-level Ficus version.

As a full-scale example of the use of stackable layering and the 405 interface, Ficus validates the success of these tools for file system development. Layered file systems can be robust enough for daily use, and the development process is suitable for long-term projects.

6.3 Summary

This chapter evaluated the performance of file system layering, by considering the performance of individual layers and file system stacks. It also considered how layering can improve the file system development by allowing code reuse and out-of-kernel development. To summarize the development environment, consider the comments of one of the students who developed a file system layer [Kue91]:

> For me, the really big advantage of the stackable layers was the ease of development. Combined with the ook [out-of-kernel] development, the testing cycle was vastly shorter than other kernel work I've done. I could compile, mount, debug, and unmount in the time that it would have taken to just link a kernel, and of course I had `dbx` available instead of struggling with lousy kernel debuggers.

CHAPTER 7

Related Work

Previous chapters have examined file system layering from a variety of perspectives, including the inter-layer interface, methods of using layering, and the performance of layered file systems. This chapter attempts to place stackable file systems in perspective with similar work on operating system interfaces. In particular, we examine the 405 interface and layering with respect to four existing file system interfaces.

Sun's vnode interface is an example of basic "file system switch" approaches to file system modularity. Such a design serves as the basis for our stackable file system work, and so requires careful consideration.

After examining this, we consider David Rosenthal's work with stacking vnode interfaces. Although similar in goals to our work, it differs significantly in detail.

MachObjects is an object-oriented interface targeted at a variety of kernel-level mechanisms. It addresses several of the extensibility issues raised in this work, but provides a different model for construction of new file systems.

The *x*-kernel is a completely new kernel designed around the idea of stackable protocols. Many of their experiences in building stackable network protocols relate to our experiences in building stackable file systems, and they are now seeking to apply their approaches to other kernel mechanisms.

These areas are more closely examined in the following sections.

7.1 The Vnode Interface

To separate file system implementations from the rest of the kernel, Sun Microsystems developed the virtual file system interface [Kle86]. Similar to work at DEC [RKH86] and AT&T, the vnode interface has the goal to support several file systems within the same kernel.

These approaches each concentrate on providing multiple file system types in the same kernel. To this end, they have been fairly successful. DEC's GFS has been successful in supporting several very different physical file systems [Koe87]. Sun's virtual file system has been successful in providing a number of file-system related features in the SVR4 kernel. These include the BSD Fast File System, the System V file system, XENIX semaphores, device special files, named pipes (FIFOS), STREAMS files, process control (the `/proc` file system), Sun's NFS (Network File System), and AT&T's Remote File Sharing.

This success has not come without cost, however. The constant evolution of these interfaces has caused problems in maintenance and third party support. Rosenthal [Ros90] documents this evolution well. The method of interface definition is an important difference between the vnode and the 405 interfaces.

The vnode interface has also been used to do some file system stacking. NFS, the loopback and translucent file systems can be thought of as two layer file system stacks. The methods of stack construction and vnode stacking are similar to those used in the 405 interface. Although stack construction is similar, the bypass facilities of the 405 interface makes general stacking much easier.

It is more difficult to develop new file systems under the vnode interface than the 405 interface. Early versions of Ficus were built with an unmodified vnode interface. Lack of extensibility greatly complicated this early work; the extensibility of the 405 interface made new development much easier. For detailed comparison of development and performance of the interfaces, see sections 6.2.1 and 6.1.

7.2 Rosenthal's Stackable Vnode Interface

Rosenthal of Sun Microsystems developed a stackable vnode interface similar in concept to the 405 interface [Ros90]. Although both were inspired by Ritchie's STREAMS work [Rit84], differences in focus have resulted in quite different designs and capabilities.

Differences in the two interfaces fall broadly into differences in stacking and extensibility. Rosenthal constructs stacks on a file-by-file granularity, and all users of his system are guaranteed to see identical views of a stack. Rosenthal's methods don't go as far as providing a fully extensible interface, instead providing interface versioning with a version mapping layer.

7.2.1 Stacking configuration

Rosenthal takes the approach of configuring stacks on a file-by-file basis. He adds two new vnode operations for this purpose, "push" and "pop". This fine granularity offers a great deal of flexibility. Each file in a directory can conceivably have its own stack, one supporting compression, another replication, and so on. This flexibility is better than that provided by the 405 interface and per-volume stack configuration.

While per-file stack configuration is more flexible than configuration at larger granularities, it is less clear how this level of configuration can be managed. By per-subtree configuration, the 405 interface exploits already existing configuration tools (`/etc/mtab`, `/etc/fstab`, *mount(8)*). These tools do not scale to handle the explosion of entries resulting from per-file configuration. While recent efforts have

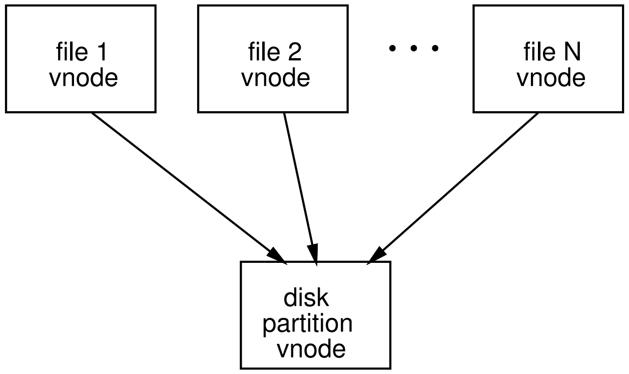


Figure 7.1: Using a vnode to represent a disk partition. The lower vnode could be a device special file, or perhaps just an ordinary file.

been made to divide file system mount information into more manageable chunks (Ficus autografting [PGP$^+$91], for example), these approaches typically require special purpose underlying file systems or file system layers. A file system with extensible file attributes could conceivably also serve to store stack construction information, but no such systems are currently available.

Another problem with the push and pop operations described in Rosenthal's paper is that they are specialized to linear stacks. "Push" stacks one vnode over another, pop removes it. It is not clear how these operations generalize to support fan-in and fan-out of vnode trees. Both fan-in and fan-out have important applications, for example, disk mirroring requires fan-out (see Figure 2.2, page 14), and using a single vnode to represent a disk partition requires fan-in (Figure 7.1).

7.2.2 Stack view consistency

Rosenthal's method of vnode stacking also provides a philosophically different interface to its clients. He adopts the principle that all clients should always see the same view of a stack. To accommodate this, user operations performed on any vnode in a stack are handled not by the requested node, but are forwarded transparently to the top of the stack. (See Figure 7.2.) This compares to the 405 interface where operations are always handled by the vnode to which they are applied and the naming system is used to direct client actions to the correct layer of the stack[1].

[1] Different stack layers have different names; user operations are presumably directed at the name for the top of the stack.

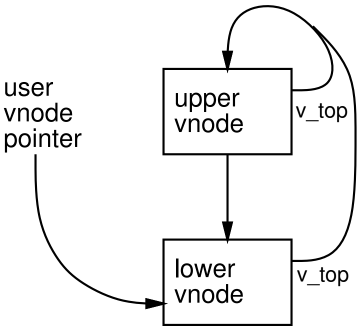


Figure 7.2: Rosenthal's stacking method forwards all user operations through the "`v_top`" pointer to the top of the stack.

This approach has two advantages over that used by the 405 interface. A minor advantage is that it is easy to have "anonymous" layers, layers without a name. Because the 405 interface merges naming and layer creation, each layer must have a name. Occasionally, dummy names must be generated for intermediate layers. On the other hand, requiring all layers to be named provides a certain regularity.

More important is that very dynamic stack manipulation is possible, since all users are always assured of seeing the same top layer. For example, one could push a measurement layer on top of an already-running stack and immediately begin collecting performance measurements. This is an attractive alternative to forcing everyone to stop using a file system, mounting the measurements layer, and then allowing everyone to continue[2]. (Which is in turn more attractive than forcing everyone to log off of the system, installing a specially instrumented kernel and then rebooting, as was required before file system layering.)

Another important use of this very dynamic stacking is handling mount points and special file types. Currently, device special files, named pipes, and other file-system name-space objects are handled by special purpose file systems. While this separates implementation of these objects from the rest of the file system, current physical file systems must have special purpose code to transparently stack these new vnodes over the underlying disk vnode. In addition, special purpose code is required to handle mount points, transitions from one subtree to another. In Rosenthal's design, this special purpose code can be replaced by vnode stacking. Since operations are always forward to the top of the vnode stack,

[2] Presumably the same name could be presented to the user for both the lower layer and the measurements layer by (for example) manipulating symbolic links.

special files can simply be pushed on the top of the stack. User operations on the physical file system will find the disk storage vnode for the special file, but all user operations on this physical-level vnode will be automatically intercepted by the special vnode on the top of the stack. Mount points will be handled similarly, operations transparently arriving at the subtree, rather than the mounted-on directory. In Figure 7.2, the lower vnode would be the mounted-on directory and the upper vnode would be the root of the mounted subtree.

Several problems arise with this form of dynamic stacking, however. Inherent with the idea that all users see the same view of a stack is the concept that no users can see different views. There are times when it is useful to have different views of a stack. Multi-layer access, described in detail in Section 4.5, is particularly useful for system maintenance tasks such as backups and debugging.

Rosenthal accomplishes dynamic stacking by forwarding all user operations to the top of the stack. However, operations between stack layers must not be forwarded, or an infinite loop would result. Therefore, two different methods must be provided to invoke each operation, or perhaps two sets of operations can be provided. "User" operations (or operations invoked in the "user" way) will be automatically forwarded, while "system" operations are not. This approach complicates the programming model.

Operations are forwarded to the top by indirecting each through a `v_top` pointer to the top vnode. When layers are pushed or popped, all `v_top` pointers in each vnode of the stack must be atomically changed. This adds to overhead and complicates implementation in symmetric multiprocessing implementations, because each stack must be protected by a readers/writer lock. In fact, stack locking overhead is a problem with current implementations of dynamic stacking [Ros90]. Since a top-of-stack pointer is not required in the 405 interface, no locking is required for the stack as a whole, and currently successful vnode locking techniques [LPLF91, CBB$^+$91, LBLM90] can be used within each layer of the stack in multiprocessor implementations.

The most significant problem with this method of dynamic stacking is that for many stacks there is no well defined notion of "top-of-stack". Stacks with fan-in have *multiple* stack tops. Encryption is one service requiring fan-in with multiple stack "views", as described in Section 4.5. It is not correct in general to send forward an operation on the lower layer to any particular upper vnode, or even to all. Furthermore, with transport layers, the correct stack top could be in another address space, making it impossible to keep a top-of-stack pointer.

Finally, it should be noted that there are very few layers which make sense to dynamically push on top of a user's stack. Nearly all file system layers change the semantics of the stack, encrypting, decompressing, or otherwise altering stack data. Pushing such a layer on an existing stack already in use makes little sense; the user's view of file contents will dramatically change. The semantics-altering stack layer should have been part of the user's stack from stack creation if its functionality is desired. The only layers which make sense to pop on and off during file use are semantics-free layers. There are few useful semantics-free layers other than measurement collecting and caching layers.

7.2.3 Interface extensibility

A final difference between Rosenthal's vnode interface and the 405 interface relate to extensibility. Rosenthal discusses the use of an versioning layer to map between different interfaces. While versioning layers work well mapping between slightly conflicting semantics or syntax, they becomes quite cumbersome as the number of interface modifications grows. Because they do not offer the full extensibility of the 405 interface, a separate layer must map between each pair of different interfaces. While this is acceptable when there are only a few interfaces, the potential of multiple third parties changing the interface implies a large number of slightly different interfaces. The number of version mapping layers grows as the square of the number of different interfaces.

An alternative is to map all interface extensions to one, common interface. But if this common interface does not support a bypass routine similar to that of 405 interface, new operations will be unable to pass through existing layers without source code changes. Requirements of source code access and change will greatly restrict layer stacking combinations without bypass capabilities.

7.3 MachObjects

MachObjects [JR89] is an object-oriented package for designing general purpose operating system interfaces. MachObjects is a macro and library package in C, running on top of the Mach operating system.

MachObjects uses object-oriented techniques to enable code reuse. Single inheritance is supported, allowing automatic reuse of routines. Multiple inheritance is supported through "delegation". An important feature is that delegation can take place across address spaces, providing a means for part of a class hierarchy to exist in one address space and the remainder in another.

MachObject cross-address space delegation is quite similar to the use of a transport layer in stackable layers. It illustrates two difficulties in the use of stackable techniques in an object-oriented framework. Simple stackable layers can easily be described in object-oriented terms. For example, a stack of an encryption layer over a standard file system can be thought of as making an encrypted sub-class of normal files. But does "remoteness" make sense as a "subclass"? If so, how can much more complicated stacks be described in object-oriented terms? Describing the out-of-kernel development platform (Figure 4.3, page 48) as subclasses seems quite difficult, for example.

An important difference between the MachObjects implementation and the stackable file system described in this document is inheritance binding time. In MachObjects, as in most object-oriented languages suitable for systems-level programming, inheritance chains are defined at compile-time. Stackable file systems delay this binding (stack creation) until mount-time. This important difference allows configuration of complex stacks with only user-level commands, while the object-oriented approach would require programming and a kernel replacement.

A final difference between MachObjects and vnode stacking is that file system stacking is targeted at a very specific kernel interface. Because of this, vnode stacking can be optimized for its expected use, while MachObjects must remain a general purpose interface. For example, all vnodes share a common set of possible operations. This would be undesirable with a general purpose interface because only a small fraction of operations would be needed by any one class.

7.4 The *x*-kernel

The *x*-kernel [HP88, HPAO89] is an operating system kernel designed to simplify network protocol implementations. Designed to provide easy configuration and efficient execution, an original goal was to provide unobtrusive customized kernels for several distributed languages. Since then, the *x*-kernel has used as a general purpose tool to explore network and RPC protocol design and configuration. Recent work has applied the *x*-kernel ideas to filing environments [PHOR90], proposing a user-customizable filing name space supporting several file systems protocols underneath.

The *x*-kernel's strengths derive from a uniform, powerful development environment. Protocols are the central *x*-kernel concept. The *x*-kernel implements all operating system services as layered protocols. Each protocol is bounded by a uniform interface, allowing substitution of protocols providing similar semantics. By supporting run-time selection of protocol stacks, the most efficient protocol applicable at a given time can be selected. Finally, the *x*-kernel emphasizes performance, providing very inexpensive transition between layers.

These x-kernel characteristics are quite similar to stackable file systems. Stackable file systems provide a uniform interface between layers. Late binding is important in stackable file systems to allow experimentation. Efficiency is also emphasized to promote separation of file systems into composable layers. Interesting parallels exist between x-kernel experiences with network protocols and our experiences with file system stacks, particularly in the difficulty of dividing existing protocols into multiple, reusable layers.

The primary difference between the x-kernel work and our stackable file system work is one of scope. The x-kernel is a complete new kernel design. It provides a complete new process facility supporting lightweight processes and a protocol-level kernel interface. Our file system work instead seeks to build on the existing Unix operating system. Another primary difference is that x-kernel research has focused primarily on network protocols, only recently addressing file systems. Our stackable file systems work instead focuses exclusively on file system design and composition.

7.5 Summary

This chapter has examined the field of file system modularity, focusing on layered protocols. The breadth of existing work indicates the importance of layered development and the problems in current development methodologies.

In the construction of the 405 interface, we have chosen to build on the existing vnode interface. We modify the interface to support third-party addition of operations, allowing multiple, compatible extensions for new services. We also provide explicit support for stacking under the interface, including the ability to identify and forward generic operations to other layers. Together, these facilities encourage independent development of filing services, allowing more rapid progress and support of services than any existing approach to file system modularity.

CHAPTER 8

Conclusions

8.1 Contributions

This research presents contributions in both the concepts and the practice of file system development. We describe the first comprehensive examination of file system construction from stackable layers. We demonstrate practicality of such a structure is demonstrated by the construction of several layers with a new, stackable interface.

This work explores the concept of stackably layered file systems in two ways. First, we identify the characteristics required of the interface joining stackable layers. Careful interface design is important because the potential of stackable layers can easily be limited by an inappropriate interface; conversely, a well designed interface can provide a number of features to make the development of new layers easy. Second, this work identifies a number of new approaches to file system design unique to the stackable environment.

The practical aspects of stackable layer design are examined by the development of a prototype interface and several file system layers. Nearly a dozen different file system layers have been developed or prototyped, several are in daily use. This broad application of layering validates its role as a general purpose tool for file system design. It also demonstrates the success of development in a layered environment. Layer construction need be no more complicated than current methods, and the ability to re-use existing code can dramatically speed development. Finally, performance analysis of these layers and their interface indicate that file system layering has almost no impact on performance as seen by the user.

8.2 Future Work

This discussion of stackable layering has focused on the design of overall layered structure and an interface capable of handling a general set of operations. A few operations are required to support general layering, but operations necessary for general file system service have not been examined. Several different sets of operations are currently in use; adoption of a standard group is important to the interchangeability of file system layers.

A number of existing file systems have been adapted to operate in a layered environment. Retaining their monolithic roots, these layers do not always provide the ideal interface for stackable use. Redesign of transport (NFS) and physical storage (UFS) layers would increase their suitability for layered use. Separation of the many concepts these layers encompass is necessary.

Interface extensibility is critical for third party development. The 405 interface provides extensibility at the operation granularity. Similar extensibility could be provided at other levels of the file system and its interface. Extensibility would be desirable *within* individual operations, allowing modifiers to be passed along with each operation. Such extensibility might be used to specify unusual variations to traditional operations, for example, extending name translation to select between multiple file versions.

Extensibility would also be valuable in the abstractions presented by traditional file systems. The UNIX file system, for example, exports the notion of a disk partition, individual files, and directory entries referring to files. In our work with layering we have found it necessary to store persistent data at each of these levels of abstraction. Standards groups have begun to address the issue of extensible file attributes, but a uniform solution to the general problem would be helpful.

Caching is a mandatory part of any file system solution. The interactions of caching in a layered system are often surprising and can be counter-intuitive. A careful examination of caching in layered systems would be interesting, particularly considering the interactions of multiple cache layers competing over shared resources such as physical memory.

Dynamic loading of kernel modules has recently become available in a number of operating systems. Application of this technology to file system layers should be relatively straightforward, and would further blur the line between kernel and user level implementation.

Stackable layering has proven a valuable tool in both terminal and network processing, and file systems. Application of this technique to other kernel interfaces is a possibility. The VFS interface for file systems and disk partitions of UNIX systems is one candidate. An intriguing approach would be to re-classify file systems as a special class of vnodes.

The ideal granularity for the specification of file system stacks is still an open question. It is not clear how best to maintain information regarding per-file stacks. On the other hand, stacking on a per-file basis even greater flexibility than typed files. Per-file stacking may be one approach to an "object-oriented file system". A closer examination of these issues would be useful.

8.3 Summary

File system development has long been an area of fruitful research. Unfortunately, application of this research has been difficult. Implementation of new ideas for filing services have been slow because new services had to be constructed from scratch. Even when completed, new services proved difficult to install and support on the variety of machines prevalent.

Stackable file system development offers an alternative. The ability to build on existing services means there is no longer a need to re-implement well understood concepts such as directory services and low-level disk access. Consistent means of adding new capabilities allows multiple third-party enhancements to cooperate instead of conflict. Improved modularity helps confine changes and focus testing. Together, these capabilities offer the potential for broader acceptance of rich new filing services.

Performance and usability are concerns with any change to basic operating system services. Through a prototype interface and layers, we have demonstrated that stackable filing can offer comparable performance to current filing designs, while offering a superior development environment.

References

[ABG$^+$86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for UNIX Development." In *USENIX Conference Proceedings*, pp. 93–113. USENIX, June 1986.

[App85] Apple Computer, Inc. *Inside Macintosh*. Addison-Wesley, Reading, Mass., 1985.

[App88] Apple Computer, Inc. *HyperCard User's Guide*. Apple Computer, Inc., Cupertino, California, 1988.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, 1986.

[Bus71] A. K. Bushan. "File Transfer Protocol." Technical Report RFC-114, Internet Request For Comments, April 1971.

[Cat90] Vince Cate. "Two Levels of Filesystem Hierarchy on One Disk." Technical Report CMU-CS-90-129, Carnegie-Mellon University, May 1990.

[CBB$^+$91] Marc Campbell, Richard Barton, Jim Browning, Dennis Cervenka, Ben Curry, Todd Davis, Tracy Edmonds, Russ Holt, John Slice, Tucker Smith, and Rich Wescott. "The Parallelization of UNIX System V Release 4.0." In *USENIX Conference Proceedings*, pp. 307–323. USENIX, January 1991.

[Cla85] David D. Clark. "The Strucutring of Systems Using Upcalls." In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 171–180. ACM, December 1985.

[Dij67] Edsgar W. Dijkstra. "The structure of the THE multiprogramming system." In *Proceedings of the Symposium on Operating Systems Principles*. ACM, October 1967.

[Dij68] Edsgar W. Dijkstra. "Complexity controlled by hierarchical ordering of function and variability." Working paper for the NATO conference on computer software engineering at Garmisch, Germany, October 1968.

[Flo86a] Rick Floyd. "Directory Reference Patterns in a UNIX Environment." Technical Report TR-179, University of Rochester, August 1986.

[Flo86b] Rick Floyd. "Short-Term File Reference Patterns in a UNIX Environment." Technical Report TR-177, University of Rochester, March 1986.

[GHM+90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. "Implementation of the Ficus Replicated File System." In *USENIX Conference Proceedings*, pp. 63–71. USENIX, June 1990.

[Guy91] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System.* Ph.D. dissertation, University of California, Los Angeles, June 1991. Also published as CSD-910018.

[Hen90] David Hendricks. "A Filesystem for Software Development." In *USENIX Conference Proceedings*, pp. 333–340. USENIX, June 1990.

[HKM+88] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[HP88] Norman C. Hutchinson and Larry L. Peterson. "Design of the *x*-Kernel." In *Proceedings of the 1988 Symposium on Commmunications Architectures and Protocols*, pp. 65–75. ACM, August 1988.

[HP91] Norman C. Hutchinson and Larry L. Peterson. "The *x*-Kernel: An Architecture for Implementing Network Protocols." *IEEE Transactions on Software Engineering*, **17**(1):64–76, January 1991.

[HPAO89] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. "RPC in the *x*-Kernel: Evaluating New Design Techniques." In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pp. 91–101. ACM, December 1989.

[IEE90] IEEE. "Standaard for Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API)." Technical Report IEEE Std. 1003.1-1990, IEEE, 1990. Also available as ISO/IEC 9945-1: 1990s.

[JR89] Daniel P. Julin and Richard F. Rashid. *MachObjects Reference Manual.* Carnegie-Mellon University, August 1989.

[Kle86] S. R. Kleiman. “Vnodes: An Architecture for Multiple File System Types in Sun Unix.” In *USENIX Conference Proceedings*, pp. 238–247. USENIX, June 1986.

[KM86] Michael J. Karels and Marshall Kirk McKusick. “Toward a Compatible Filesystem Interface.” In *Proceedings of the European Unix User’s Group*, p. 15. EUUG, September 1986.

[Koe87] Matt Koehler. “GFS Revisited or How I Lived with Four Different Local File Systems.” In *USENIX Conference Proceedings*, pp. 291–305. USENIX, June 1987.

[KP84] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice-Hall, 1984.

[Kue91] Geoff Kuenning. “Comments on CS239 Class Projects.” Personal communication, June 1991.

[Lan90] Peter S. Langston. “Unix Music Tools at Bellcore.” *Software — Pratice and Experience*, **20**(S1):47–61, June 1990.

[LBLM90] Alan Langerman, Joseph Boykin, Susan LoVerso, and Sashi Mangalat. “A Highly-Parallelized Mach-based Vnode Filesystem.” In *USENIX Conference Proceedings*, pp. 297–312. USENIX, January 1990.

[LPLF91] Susan LoVerso, Noemi Paciorek, Alan Langerman, and George Feinberg. “The OSF/1 Unix File System (UFS).” In *USENIX Conference Proceedings*, pp. 207–218. USENIX, January 1991.

[MA69] Stuart E. Madnick and Joseph W. Alsop, II. “A modular approach to file system design.” In *AFIPS Conference Proceedings Spring Joint Computer Conference*, pp. 1–13. AFIPS Press, May 1969.

[MD74] Stuart E. Madnick and John J. Donovan. *Operating Systems*. McGraw-Hill Book Company, 1974.

[MJLF84] Michael McKusick, William Joy, Samuel Leffler, and R. Fabry. “A Fast File System for UNIX.” *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[Neu89] B. Clifford Neuman. “Workstations and the Virtual System Model.” In *Proceedings of the Second Workshop on Workstation Operating Systems*, pp. 91–95. IEEE Computer Society Press, September 1989.

[OCH$^+$85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System." Technical Report UCB/CSD 85/230, UCB, 1985.

[OD88] John Ousterhout and Fred Douglis. "Beating the I/O Bottleneck: A Case for Log-Structured File Systems." Technical Report UCB/CSD 88/467, Unviversity of California, Berkeley, October 1988.

[Ous90] John K. Ousterhout. "Why Aren't Operating Systems Geting Faster As Fast as Hardware?" In *USENIX Conference Proceedings*, pp. 247–256. USENIX, June 1990.

[Pag91] Tom Page. "The Umap Layer." Personal communication, July 1991.

[PGP$^+$91] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, John S. Heidemann, Wai Mak, and Dieter Rothmeier. "Management of Replicated Volume Location Data in the Ficus Replicated File System." In *USENIX Conference Proceedings*. USENIX, June 1991.

[PHOR90] Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao. "The *x*-Kernel: A Platform for Accessing Internet Resources." *IEEE Computer*, **23**(5):23–33, May 1990.

[PK84] Rob Pike and Brian Kernighan. "Program Design in the UNIX Environment." *AT&T Bell Laboratories Technical Journal*, **63**(8):1595–1605, October 1984.

[PR85] J. B. Postel and J. K. Renolds. "File Transfer Protocol." Technical Report RFC-959, Internet Request For Comments, October 1985.

[RAA$^+$90] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. "Overview of the CHORUS Distributed Operating System." Technical Report CS/TR-90-25, Chorus systèmes, April 1990.

[RFH$^+$86] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, and Kang Yueh. "RFS Architectural Overview." In *USENIX Conference Proceedings*, pp. 248–259. USENIX, June 1986.

[Rit84] Dennis M. Ritchie. "A Stream Input-Output System." *AT&T Bell Laboratories Technical Journal*, **63**(8):1897–1910, October 1984.

[RKH86] R. Rodriguez, M. Koehler, and R. Hyde. "The Generic File System." In *USENIX Conference Proceedings*, pp. 260–269. USENIX, June 1986.

[Ros88] David Rosenthal. "A Simple X11 Client Program *or* How hard can it really be to write "Hello, World"?" In *USENIX Conference Proceedings*, pp. 229–242. USENIX, February 1988.

[Ros90] David S. H. Rosenthal. "Evolving the Vnode Interface." In *USENIX Conference Proceedings*, pp. 107–118. USENIX, June 1990.

[RT74] Dennis M. Ritchie and Ken Thompson. "The UNIX Time-sharing System." *Communications of the ACM*, **17**(7):365–375, October 1974.

[SGK$^+$85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and Implementation of the Sun Network File System." In *USENIX Conference Proceedings*, pp. 119–130. USENIX, June 1985.

[SKS90] David C. Steere, James J. Kistler, and M. Satyanarayanan. "Efficient User-Level File Cache Management on the Sun Vnode Interface." In *USENIX Conference Proceedings*, pp. 325–332. USENIX, June 1990.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Sun87] Sun Microsystems. "XDR: External Data Representation standard." Technical Report RFC-1014, Internet Request For Comments, June 1987.

[Sun88] Sun Microsystems. "RPC: Remote Procedure Call Protocol specification version 2." Technical Report RFC-1057, Internet Request For Comments, June 1988.

[Sun89] Sun Microsystems. "NFS: Network File System Protocol Specification." Technical Report RFC-1094, Internet Request For Comments, March 1989.

[Sun90] Sun Microsystems. "Network Extensible File System Protocol Specification, *draft*." Available for anonymous ftp on titan.rice.edu as public/nefs.doc.ps, February 1990.

[Wu91] Yuguang Wu. “The Measurement Layer.” Personal communication, July 1991.

[ZDL$^+$90] Lisa Zahn, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. *Network Computing Architecture*. Prentice-Hall, 1990.