To appear, ACM Transactions on Computer Systems.

File System Development with Stackable Layers*

John S. Heidemann   Gerald J. Popek

Department of Computer Science
University of California, Los Angeles

Technical Report CSD-930019

July 1993

Abstract

Filing services have experienced a number of innovations in recent years, but many of these promising ideas have failed to enter into broad use. One reason is that current filing environments present several barriers to new development. For example, file systems today typically stand alone instead of building on the work of others, and support of new filing services often requires changes which invalidate existing work.

Stackable file system design addresses these issues in several ways. Complex filing services are constructed from layer "building blocks", each of which may be provided by independent parties. There are no syntactic constraints to layer order, and layers can occupy different address spaces, allowing very flexible layer configuration. Independent layer evolution and development is supported by an extensible interface bounding each layer.

This paper discusses stackable layering in detail and presents design techniques it enables. We describe an implementation providing these facilities that exhibits very high performance. By lowering barriers to new filing design, stackable layering offers the potential of broad third party file system development not feasible today.

*This work was sponsored by the Defense Advanced Research Projects Agency under contracts F29601-87-C-0072 and N00174-91-C-0107. Also, John Heidemann was sponsored by a USENIX scholarship for the 1990-91 academic year, and Gerald Popek is affiliated with Locus Computing Corporation.

The authors can be reached at 3680 Boelter Hall, UCLA, Los Angeles, CA, 90024, or by electronic mail to `johnh@cs.ucla.edu` or `popek@cs.ucla.edu`.

Last revised 7 July 1993.

1 Introduction

File systems represent one of the most important aspects of operating system services. Traditionally, the file system has been tightly integrated with the operating system proper. As a result, the evolution of filing services has been relatively slow. For example, the primary file system in UNIX[1] systems today, called UFS, is basically the Berkeley Fast File System introduced almost a decade ago. The situation for filing in proprietary operating systems is similar. The MVS catalog system, for example, has seen little architectural change in over a decade and a half. This state of affairs exists despite the fact that there are numerous improvements that are well known and have been constructed in one context or another. By contrast, applications software in many areas has evolved much more rapidly, giving far more benefit to users. This stifling of innovation and inhibition of evolution has kept a variety of benefits from users and has caused application developers such as database builders to provide their own filing services inside their applications software at considerable expense and without much generality.

There are several reasons why this unappealing situation persists. First, it is very difficult for anyone other than a major system vendor to introduce a file system service or incremental improvement into an operating system. There is no well defined interface to employ. Despite the fact that in a few systems, like UNIX, there is a coarse grain interface (the Virtual File System, or VFS) by which an *entire* file system can be installed, in practice this fact has not worked well. VFS is inflexible in addressing the range of issues, so most vendors have extended it in incompatible ways. Further, any real file system is a complex service that requires successful solu-

[1] UNIX is a trademark of UNIX System Laboratories.

tion to a variety of functional requirements, and is thus a daunting effort when viewed in its entirety. Since filing is such a key part of an operating system, with so much dependent on it, excellent performance is critical. Also, the impact of an error in the filing system's logic can be quite devastating, as it is usually responsible for all of the persistent storage in the computer system.

In addition, the file service must interact on an intimate basis with other core operating system services. For example, there is considerable motivation to arrange for the file service and the operating system's virtual memory manager to share management of pages; perhaps with a single buffer pool.

In the face of these observations, it is not surprising then that a file system, once operational and deployed, is not changed for a long time.

Despite these problems, a great deal of benefit would accrue if it were possible to add file system services to an existing system in a very simple manner, analogous to the way that an additional service is obtained at the user level merely by adding another application. That is, the set of user services are not provided by a monolithic program, but by a large number of individually developed packages, many of which today can exchange data with one another in a straightforward way. In the personal computer arena, a user may select the services he wishes, acquire appropriate "shrink-wrap software" at a local store, and install it himself. This situation has provided an explosion of services at a far faster rate, for example, than on mainframe systems, where the process is traditionally considerably more cumbersome.

We believe that a similar situation would benefit filing services a great deal. Suppose it were just as straightforward to construct independently, or obtain and install packages for such services as:

- fast physical storage management
- extended directory services
- compression and decompression
- automatic encryption and decryption
- cache performance enhancement
- remote access services
- selective file replication
- undo and undelete
- transactions

with assurance that they would work together effectively. Then, we contend, the available services would be much richer, particular solutions would be much more widely available, and evolution would be far more rapid. In addition, the wide variety of expensive and confusing ad hoc solutions employed today would not occur[2].

The goal of the research reported in this paper is to contribute towards making it as easy to add a function to a file system in practice as it is to install an application on a personal computer.

To do so, it is necessary to provide an environment in which file system fragments can be easily and effectively assembled, and for which the result makes no compromises in functionality or performance. The environment must be extensible so that its useful lifetime spans generations of computer systems. Continued successful operation of modules built in the past must be assured. Multiple third party additions must be able to coexist in this environment, some of which may provide unforseen services.

The impact of such an environment can be considerable. For example, many recognize the utility of microkernels such as Mach or Chorus. Such systems, however, address the structure of about 15% of a typical operating system, compared to the filing environment, which often represents 40% of the operating system.

This paper describes an interface and framework to support stackable file system development. This framework allows new layers to build on the functionality of existing services while allowing all parties to gradually grow and evolve the interface and set of services. Unique characteristics of this framework include the ability of layers in different address spaces to interact and for layers to react gracefully in the face of change.

Actual experience with software environments is an important test of their utility. For this reason much of the body of this paper is accompanied by examples drawn from the UNIX system where these experiences were obtained. Our work draws on earlier research with stream protocols [10] file system modularity [7], and object-oriented design (see Section 6.2), as well as fragments of ideas in a variety of other referenced work. The experiences and application of these principles to a complex interface controlling access to persistent data differentiates this work from that which has come before.

1.1 Organization of the paper

This paper first examines the evolution of file system development. We then consider the essential characteristics of a stackable file system and discuss how direct support for stacking can enable new approaches to file system design. An implementation of this framework is next examined, focusing on the elements of filing unique to stacking. We follow with an evaluation of this implementation, examining both performance and usability.

[2] Consider for example the grouping facility in MS-Windows. It is in effect another directory service, on top of and independent of the underling MS-DOS directory system. It provides services that would not have been easy to incorporate into the underlying file system in an upward compatible fashion.

Finally, we consider similar areas of operating systems research to place this work in context.

2 File System Design

Many new filing services have been suggested in recent years. The introduction presented a certainly incomplete list of nine such services, each of which exists today in some form. This section compares alternative implementation approaches for new services, examining characteristics of each that hinder or simplify development.

2.1 Traditional Design

A first approach to providing new filing functionality might be to modify an existing file system. There are several widely available file systems, any of which can be modified to add new features. While beginning with an existing implementation can speed development, "standard" services have frequently evolved significantly since their initial implementation. Supporting new capabilities across a dozen different platforms may well mean a dozen separate sets of modifications, each nearly as difficult as the previous. Furthermore, it is often difficult to localize changes; verification of the modified file system will require examination of its entire implementation, not just the modifications. Finally, although file systems may be widely used, source code for a particular system may be expensive, difficult, or impossible to acquire. Even when source code is available, it can be expected to change frequently as vendors update their system.

Standard filing interfaces such as the vnode interface [7] and NFS [13] address some of these issues. By defining an "air-tight" boundary for change such interfaces avoid modification of existing services, preventing introduction of bugs outside new activity. Yet, by employing existing file systems for data storage, basic filing services do not need to be re-implemented. NFS also allows file systems to be developed at the user-level, simplifying development and reducing the impact of error.

Use of standard interfaces introduces implementation problems of their own. The interface is either evolving or static. If, like the vnode interface, it evolves to provide new services and functionality, compatibility problems are introduced. A change to the vnode interface requires corresponding changes to each existing filing service, today.

Compatibility problems can be avoided by keeping the interface static, as with NFS. While this approach improves portability, it becomes difficult to provide new services cleanly. If protocol change is not allowed then new services must either be overloaded on existing facilities or employ a parallel interface. Finally, NFS-like RPC interfaces to user-level services burden the concept of modularity with particular choices of process execution and protection. A naive approach can easily interpose repeated data copies between the hardware and the user. Although performance can be improved by in-kernel caching [14], portability is reduced and the result is not fully satisfactory. Finally, even with careful caching, the cost of layer crossings is often orders of magnitude greater than subroutine calls, thus discouraging the use of layering for structuring and re-use.

2.2 Stackable Design

Stackable file systems construct complex filing services from a number of independently developed layers, each potentially available only in binary form. New services are provided as separate layers; the layer division provides a clear boundary for modifications. Errors can be then isolated to the current layer or an invalid implementation of the inter-layer interface by another layer. Figures 1 through 5 (discussed further in the following sections) illustrate how stacking can be used to provide services in a variety of environments. Stacking is actually somewhat of a misnomer, since non-linear "stacks" such as those of Figures 3 and 4 should be common; we retain the term for historic reasons.

Layers are joined by a *symmetric* interface; syntactically identical above and below. Because of this symmetry there are no syntactic restrictions in the configuration of layers. New layers can be easily added or removed from a file system stack, much as Streams modules can be configured onto a network interface. Such filing configuration is simple to do; no kernel changes are required, allowing easy experimentation with stack behavior.

Because new services often employ or export additional functionality, the inter-layer interface is *extensible*. Any layer can add new operations; existing layers adapt automatically to support these operations. If a layer encounters an operation it does not recognize, the operation can be forwarded to a lower layer in the stack for processing. Since new file systems can support operations not conceived of by the original operating system developer, unconventional file services can now be supported as easily as standard file systems.

The goals of interface symmetry and extensibility may initially appear incompatible. Configuration is most flexible when all interfaces are literally syntactically identical—all layers can take advantage of the same mechanisms. But extensibility implies that layers may be semantically distinct, restricting how layers can be combined. Semantic constraints do limit layer configurations, but sections 4.2 and 4.4 discuss how layers may provide reasonable default semantics in the face of extensibility.

Address space independence is another important characteristic of stackable file system development. Layers often execute in a single protection domain, but there is considerable advantage to making layers also able to run transparently in different address spaces. Whether or not different layers exist in the same address space should require no changes to the layer and should not affect stack behavior. New layers can then be developed at user-level and put into the kernel for maximum performance.

A *transport layer* bridges the gap between address spaces, transferring all operations and results back and forth. Like the inter-layer interface, a transport layer must be extensible, to support new operations automatically. In this way, distributed filing implementations fit smoothly into the same framework, and are afforded the same advantages. More importantly, the transport layer isolates the "distributed" component of the stacking and its corresponding performance impact. Stack calls between layers in a single address space, the statistically dominate case, can then operate in an extremely rapid manner, avoiding distributed protocols such as argument marshaling. The interface has been developed to enable absolutely minimum crossing cost locally, while still maintaining the structure necessary for address space independence.

This stacking model of layers joined by a symmetric but extensible filing interface, constrained by the requirements of address space independence, the binary-only environment of kernels, and the overriding demand for absolutely minimal performance impact, represents the primary contribution of this work. The next section discusses approaches to file system design that are enabled or facilitated by the stacking model.

3 Stackable Layering Techniques

This section examines in detail a number of different file system development techniques enabled or simplified by stackable layering.

3.1 Layer Composition

One goal of layered file system design is the construction of complex filing services from a number of simple, independently developed layers. If file systems are to be constructed from multiple layers, one must decide how services should be decomposed to make individual components most reusable. Our experience shows that layers are most easily reusable and composable when each encompasses a single abstraction. This experience parallels those encountered in designing composable network protocols in the x-kernel [5] and tool development with the UNIX shells [9].

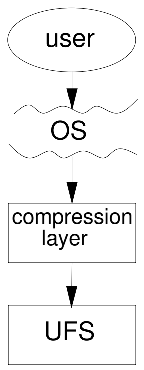

Figure 1: A compression service stacked over a UNIX file system.

As an example of this problem in the context of file system layering, consider the stack presented in Figure 1. A compression layer is stacked over a standard UNIX file system (UFS); the UFS handles file services while the compression layer periodically compresses rarely used files.

A compression service provided above the UNIX directory abstraction has difficulty efficiently handling files with multiple names (hard links). This is because the UFS was not designed as a stackable layer; it encompasses several separate abstractions. Examining the UFS in more detail, we see at least three basic abstractions: a disk partition, arbitrary length files referenced by fixed names (inode-level access), and a hierarchical directory service. Instead of a single layer, the "UFS service" should be composed of a stack of directory, file, and disk layers. In this architecture the compression layer could be configured directly above the file layer. Multiply-named files would no longer be a problem because multiple names would be provided by a higher-level layer. One could also imagine re-using the directory service over other low level storage implementations. Stacks of this type are show in Figure 2.

3.2 Layer Substitution

Figure 2 also demonstrates layer substitution. Because the log structured file system and the UFS are semantically similar, the compression layer can stack equally

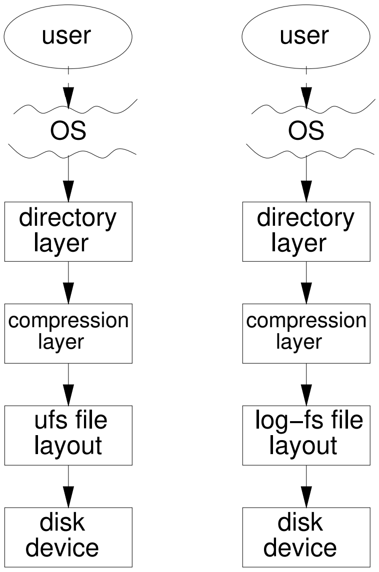

Figure 2: A compression layer configured with a modular physical storage service. Each stack also uses a different file storage layer (UFS and log structured layout).

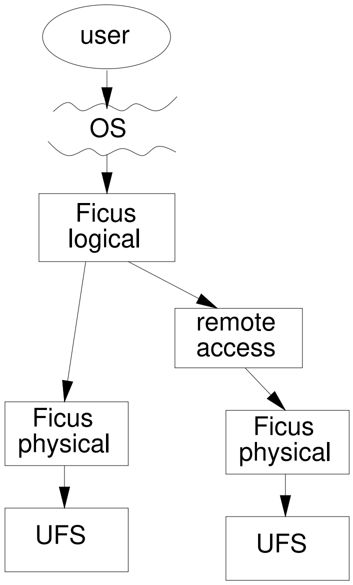

Figure 3: Cooperating Ficus layers. Fan-out allows the logical layer to identify several replicas, while a remote access layer is inserted between cooperating Ficus layers as necessary.

well over either. Substitution of one for the other is possible, allowing selection of low level storage to be independent of higher-level services. This ability to have "plug-compatible" layers not only supports higher-level services across a variety of vendor customized storage facilities, but it also supports the evolution and replacement of the lower layers as desired.

3.3 Non-linear Stacking

File system stacks are frequently linear; all access proceeds from the top down through each layer. However, there are also times when non-linear stacks are desirable.

Fan-out occurs when a layer references "out" to multiple layers beneath it. Figure 3 illustrates how this structure is used to provide replication in Ficus [2].

Fan-in allows multiple clients access to a particular layer. If each stack layer is separately named, it is possible for knowledgeable programs to choose to avoid upper stack layers. For example, one would prefer to back up compressed or encrypted data without uncompressing or decrypting it to conserve space and preserve privacy (Figure 4). This could easily be done by direct access to the underlying storage layer. Network access of encrypted data could also be provided by direct access to the underlying encrypted storage, avoiding clear-text transfer over the network.

3.4 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 a remote access service in-between. One can envision several possible distributed file systems offering simple stateless service, exact UNIX semantics, or even file replication. Each might build its particular semantics on top of an "RPC" remote access service, but if remote access is buried in the internals of each specific file system, it will be unavailable for reuse.

Cases such as these call for *cooperating layers*. A "semantics-free" remote access service is provided as a reusable layer, and the remainder is split into two separate, cooperating layers. When the file system stack is composed, the reusable layer is placed between the others. Because the reusable portion is encapsulated as a separate layer, it is available for use in other stacks. For example, a new secure remote filing service could be built by configuring encryption/decryption layers around the basic transport service.

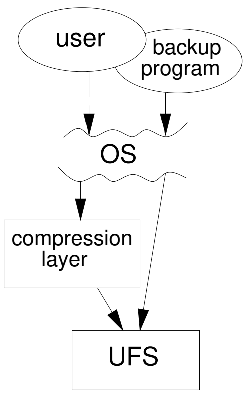

Figure 4: Access through the compression layer provides users transparently uncompressed data. Fan-in allows a backup program to directly access the compressed version.

An example of the use of cooperating layers in the Ficus replicated file system [2] is shown in Figure 3. The Ficus logical and physical layers correspond roughly to a client and server of a replicated service. A remote access layer is placed between them when necessary.

3.5 Compatibility with Layers

The flexibility stacking provides promotes rapid interface and layer evolution. Unfortunately, rapid change often rapidly results in incompatibility. Interface change and incompatibility today often prevent the use of existing filing abstractions [15]. A goal of our design is to provide approaches to cope with interface change in a binary-only environment.

File system interface evolution takes a number of forms. Third parties wish to extend interfaces to provide new services. Operating system vendors must change interfaces to evolve the operating system, but usually also wish to maintain backwards compatibility. Stackable layering provides a number of approaches to address the problems of interface evolution.

Extensibility of the file system interface is the primary tool to address compatibility. Any party can add operations to the interface; such additions need not invalidating existing services. Third party development is facilitated, gradual operating system evolution becomes possible, and the useful lifetime of a filing layer is greatly increased, protecting the investment in its construction.

Layer substitution (see Section 3.2) is another approach to address simple incompatibilities. Substitution of semantically similar layers allows easy adaption to differences in environment. For example, a low-level storage format tied to particular hardware can be replaced by an alternate base layer on other machines.

Resolution of more significant problems may employ a *compatibility layer*. If two layers have similar but not identical views of the semantics of their shared interface, a thin layer can easily be constructed to map between incompatibilities. This facility could be used by third parties to map a single service to several similar platforms, or by an operating system vendor to provide backwards compatibility after significant changes.

A still more significant barrier is posed by different operating systems. Although direct portability of layers between operating systems with radically different system services and operation sets is difficult, limited access to remote services may be possible. Transport layers can bridge machine and operating system boundaries, extending many of the benefits of stackable layering to a non-stacking computing environment. NFS can be thought of as a widely used transport layer, available on platforms ranging from personal computers to mainframes. Although standard NFS provides only core filing services, imparts restrictions, and is not extensible, it is still quite useful in this limited role. Section 5.3 describes how this approach is used to make Ficus replication available on PCs.

3.6 User-level Development

One advantage of micro-kernel design is the ability to move large portions of the operating system outside of the kernel. Stackable layering fits naturally with this approach. 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 5). A transport layer (such as NFS) serves as the RPC interface, moving all operations from the kernel to a user-level file system server. Another transport service (the "u-to-k layer") allows user-level calls on vnodes that exist inside the kernel. With this framework layers may be developed and executed as user code. Although inter-address space RPC has real cost, caching may provide reasonable performance for an out-of-kernel file system [14] in some cases, particularly if other characteristics of the filing service have inherently high latency (for example, hierarchical storage management).

Nevertheless, many filing services will find the cost of frequent RPCs overly expensive. Stackable layering offers valuable flexibility in this case. 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. An appro-

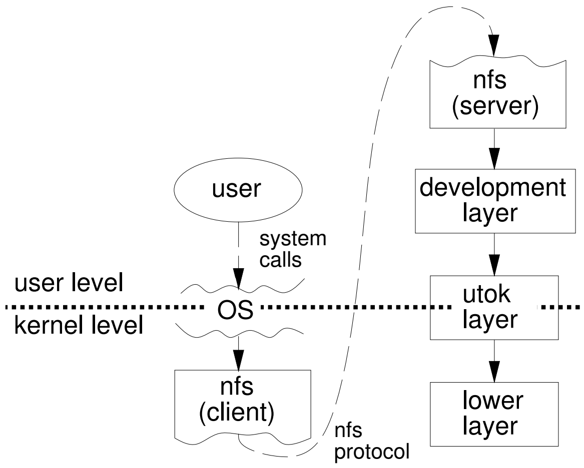

Figure 5: User-level layer development via transport layers.

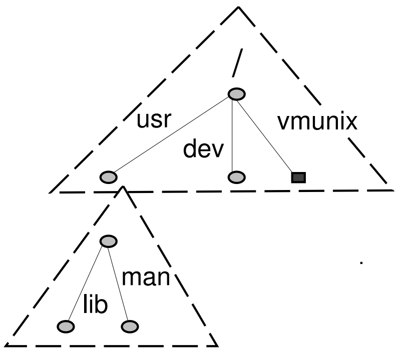

Figure 6: A namespace composed of two subtrees.

priately constructed layer can then run in the kernel, avoiding all RPC overhead. Layers can be moved in and out of the kernel (or between different user-level servers) as usage requires. By separating the concepts of modularity from address space protection, stackable layering permits the advantages of micro-kernel development and the efficiency of an integrated execution environment.

4 Implementation

The UCLA stackable layers interface and its environment are the results of our efforts to tailor file system development to the stackable model. Sun's vnode interface is extended to provide extensibility, stacking, and address-space independence. We describe this implementation here, beginning with a summary of the vnode interface and then examining important differences in our stackable interface.

4.1 Existing File System Interfaces

Sun's vnode interface is a good example of several "file system switches" developed for the UNIX operating system [7, 11]. All have the same goal, to support multiple file system types in the same operating system. The vnode interface has been quite successful in this respect, providing dozens of different filing services in several versions of UNIX.

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. 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.

The other major data structure is the *vfs*, representing groups of files. For configuration purposes, sets of files are grouped into *subtrees* (traditionally referred to as file systems or disk partitions), each corresponding to one vfs. Subtrees are added to the file system namespace by *mounting*.

Mounting is the process of adding new collections of files into the global file system namespace. Figure 6 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.

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. Each subtree is allowed a completely separate implementation.

Data encapsulation requires that these abstract data types for files and subtrees be manipulated only by a restricted set of operations. The operations supported by vnodes, the abstract data type for "files", vary according to implementation (see [7] and [6] for semantics of typical operations).

To allow this generic treatment of vnodes, binding of desired function to correct implementation is delayed until kernel initialization. This is implemented 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 (this mechanism is analogous to typical implementations of C++ virtual class method invocation).

Limited file system stacking is possible with the standard vnode interface using the mount mechanism. Sun Microsystems' NFS [13], loopback, and translucent [3] file systems take this approach. Information associated with the mount command identifies the existing stack layer and where the new layer should be attached into the filing name space.

4.2 Extensibility in the UCLA Interface

Accommodation of interface evolution is a critical problem with existing interfaces. Incompatible change and the lock-step release problem [15] are serious concerns of developers today. The ability to add to the set of filing services without disrupting existing practices is a requirement of diverse third party filing development and would greatly ease vendor evolution of existing systems.

The vnode interface allows that the association of an operation with its implementation to be delayed until run-time by fixing the formal definition of all permissible operations before kernel compilation. This convention prohibits the addition of new operations at kernel link time or during execution, since file systems have no method of insuring interface compatibility after change.

The UCLA interface addresses this problem of extensibility by maintaining all interface definition information until execution begins, and then dynamically constructing the interface. Each file system provides a list of all the operations it supports. At kernel 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 define the global operations vector dynamically, adapting it to arbitrary additions[3]. Vectors customized to each file system are then constructed, caching information sufficient to permit very rapid operation invocation. During operation these vectors select the correct implementation of each operation for a given vnode. Thus, each file system may include new operations, and new file systems can be added to a kernel with a simple reconfiguration.

New operations may be added by any layer. Because the interface does not define a fixed set of operations, a new layer must expect "unsupported" operations and accommodate them consistently. The UCLA interface requires a *default* routine which will be invoked for all operations not otherwise provided by a file system. File systems may simply return an "unsupported operation" error code, but we expect most layers to pass unknown operations to a lower layer for processing.

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. These changes have very little performance impact, an important consideration for a service that will be as frequently employed as an inter-layer interface. Section 5.1 discusses performance of stackable layering in detail.

4.3 Stack Creation

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

4.3.1 Stack configuration

Section 4.1 described how a 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. We employ this same mechanism for layer construction.

Fundamentally, the UNIX mount mechanism has two purposes: it creates a new "subtree object" of the requested type, and it attaches this object into the file system name-space for later use. Frequently, creation of subtrees uses other objects in the file system. An example of this is shown in Figure 7 where a new UFS is instantiated from a disk device (`/layer/ufs/crypt.raw` from `/dev/dsk0g`).

Configuration of layers requires the same basic steps of layer creation and naming, so we employ the same mount mechanism for layer construction[4]. Layers are built at the subtree granularity; a mount command creating each layer of a stack. Typically, stacks are built bottom up. After a layer is mounted to a name, the next higher layer's mount command uses that name to identify its "lower layer neighbor" in initialization. Figure 8

[3]For simplicity we ignore here the problem of adding new operations at run-time. This "fully dynamic" addition of operations can be supported with an extension to the approach described here, although this extension is not part of current implementations.

[4]Although mount is typically used today to provide "expensive" services, the mechanism is not inherently costly. Mount constructs an object and gives it a name; when object initialization is inexpensive, so is the corresponding mount.

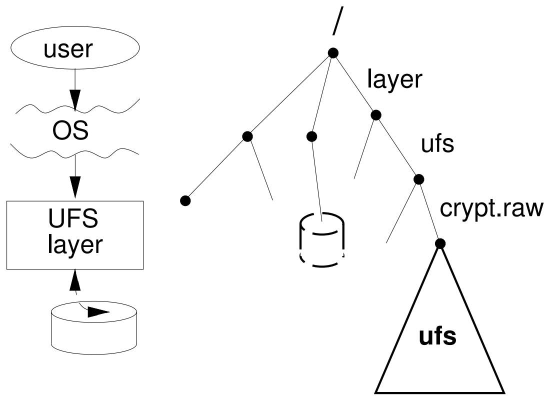

Figure 7: Mounting a UFS layer. The new layer is instantiated at `/layer/ufs/crypt.raw` from a disk device `/dev/dsk0g`.

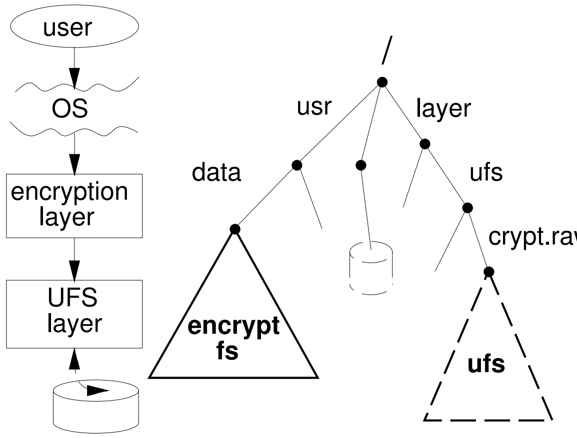

Figure 8: An encryption layer is stacked over a UFS. The encryption layer `/usr/data` is instantiated form an existing ufs layer `/layer/ufs/crypt.raw`.

continues the previous example by stacking an encryption layer over the UFS. In this figure, an encryption layer is created with a new name (`/usr/data`) after specifying the lower layer (`/layer/ufs/crypt.raw`). Alternatively, if no new name is necessary or desired, the new layer can be mounted to the same place in the namespace[5]. Stacks with fan-out typically require each lower layer be named when constructed.

Stack construction does not necessarily proceed from the bottom up. Sophisticated file systems may create lower layers on demand. The Ficus distributed file system takes this approach in its use of volumes. Each volume is a subtree storing related files. To insure that all sites maintain a consistent view about the location of the thousands of volumes in a large scale distributed system, 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 (and lower stack layers) is automatically located and mounted.

4.3.2 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 stack layer and are then forwarded down the stack as required. If an action requires creation of a new vnode (such as referencing 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 the private data of the vnode it constructs. Should a layer employ fan-out, each of its vnodes will reference several lower-level vnodes similarly.

Since vnode references are used both to bind layers and 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 general kernel can be used between layers; layers treat all incoming operations identically.

Although the current implementation does not explicitly support stack configuration at a per-file granularity, there is nothing in the model which prohibits finer configuration control. A typed-file layer, for example, could maintain layer configuration details for each file and automatically create the proper layers when the file is opened.

4.3.3 Stack Data Caching

Individual stack layers often wish to cache data, both for performance and to support memory mapped files. If each layer may independently cache data pages, writes to different cached copies can cause cache aliasing problems and possible data loss[6].

[5] Mounts to the same name are currently possible only in BSD 4.4-derived systems. If each layer is separately named, standard access control mechanisms can be used to mediate access to lower layers.

[6] Imagine modifying the first byte of a page in one stack layer and the last byte of the same page in another layer. Without some cache consistency policy, one of these modifications will be lost.

A cache manager coordinates page caching in the UCLA interface. Before any stack layer may cache a page, it must acquire "ownership" of that page from the cache manager. The cache manager immediately grants ownership if the page was not previously cached. If the page is cached by another layer, the other layer is called upon to flush its cache before ownership is transferred. This approach is analogous to callbacks or token passing in a distributed file system.

A consistent page naming policy is required to identify cached data pages between different layers. The cache manager identifies pages by a pair: [stack identifier, file offset]. Stack layers that provide semantics that violate this naming convention are required to conceal this difference from their clients. For example, a replication service must coordinate stack identifiers between its clients, itself, and its several lower layers providing replica storage. To conceal this fact the replication layer will claim to be the bottom of stack to its clients. A compression layer presents another example, since file offsets have different meanings above and below a compression layer. The compression layer will explicitly coordinate cache behavior above and below itself.

Attribute and name caching by layers present similar problems; we are currently investigating solutions in these areas.

4.4 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 modify only a few operations, passing most to the next lower layer unchanged. For example, although an encryption layer would encrypt and decrypt all data accessed by read and write requests, it may not need to modify operations for directory manipulation. Since the inter-layer interface is extensible and therefore new operations may always be added, an intermediate layer must be prepared to forward arbitrary, new operations.

One way to pass operations to a lower layer is to implement, for each operation, a routine that explicitly invokes the same operation in the next lower layer. This approach would fail to adapt automatically to the addition of new operations, requiring modification of all existing layers when any layer adds a new operation. The creation of new layers and new operations would be discouraged, and the use of and unmodified third-party layers in the middle of new stacks would be impossible.

What is needed is a single *bypass* routine which forwards new operations to a lower level. Default routines (discussed in Section 4.2) provide the capability to have a generic routine intercept unknown operations, but the standard vnode interface provides no way to process this operation in a general manner. To handle multiple operations, a single routine must be able to handle the variety of arguments used by different operations. It must also be possible to identify the operation taking place, and to map any vnode arguments to their lower level counterparts[7].

Neither of these characteristics are possible with existing interfaces where operations are implemented as standard function calls. And, of course, support for these characteristics must have absolutely minimal performance impact.

The UCLA interface accommodates these characteristics by explicitly managing operations' arguments as collections. In addition, meta-data is associated with each collection, providing the operation identity, argument types, and other pertinent information. Together, this explicit management of operation invocations allows arguments to be manipulated in a generic fashion and efficiently forwarded between layers, usually with pointer manipulation.

These characteristics make it possible for a simple bypass routine to forward all operations to a lower layer in the UCLA interface. By convention, we expect most file system layers to support such a bypass routine. More importantly, these changes to the interface have minimal impact on performance. For example, passing meta-data requires only one additional argument to each operation. See Section 5.1 for a detailed analysis of performance.

4.5 Inter-machine Operation

A *transport layer* is a stackable layer that transfers operations from one address space to another. Because vnodes for both local and remote file systems accept the same operations, they may be used interchangeably, providing network transparency. Sections 3.5 and 3.6 describe some of the ways to configure layers this transparency allows.

Providing a bridge between address spaces presents several potential problems. Different machines might have differently configured sets of operations. Heterogeneity can make basic data types incompatible. Finally, methods to support variable length and dynamically allocated data structures for traditional kernel interfaces do not always generalize when crossing address space boundaries.

[7] Vnode arguments change as a call proceeds down the stack, much as protocol headers are stripped off as network messages are processed. No other argument processing is required in order to bypass an operation between two layers in the same address space.

For two hosts to inter-operate, it must be possible to identify each desired operation unambiguously. Well-defined RPC protocols such as NFS insure compatibility by providing only a fixed set of operations. Since restricting the set of operations frequently restricts and impedes innovation, each operation in the UCLA interface is assigned a universally unique identifier when it is defined[8]. Inter-machine communication of arbitrary operations uses these labels to reject locally unknown operations.

Transparent forwarding of operations across address space boundaries requires not only that operations be identified consistently, but also that arguments be communicated correctly in spite of machine heterogeneity. Part of the meta-data associated with each operation includes a complete type description of all arguments. With this information, an RPC protocol can marshal operation arguments and results between heterogeneous machines. 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 stacks on top of existing local file systems, using the vnode interface above and below. But NFS was not designed as a transport layer; its supported operations are not extensible and its implementations define particular caching semantics. We extend NFS to automatically bypass new operations. We have also prototyped a cache consistency layer providing a separate consistency policy.

In addition to the use of an NFS-like inter-address space transport layer we employ a more efficient transport layer operating between the user and the kernel level. Such a transport layer provides "system call" level access to the UCLA interface, allowing user-level development of file system layers and providing user-level access to new file system functionality. The desire to support a system-call-like transport layer placed one additional constraint on the interface. Traditional system calls expect the user to provide space for all returned data. We have chosen to extend this restriction to the UCLA interface to make the user-to-kernel transport layer universal. In practice this restriction has not been serious since the client can often make a good estimate of storage requirements. If the client's first guess is wrong, information is returned allowing the client to correctly repeat the operation.

[8] Generation schemes based on host identifier and time-stamp support fully distributed identifier creation and assignment. We therefore employ the NCS UUID mechanism.

4.6 Centralized Interface Definition

Several aspects of the UCLA interface require precise information about the characteristics of the operation taking place. Network transparency requires a complete definition of all operation types, and a bypass routine must be able to map vnodes from one layer to the next. The designer of a file system employing new operations must provide this information.

Detailed interface information is needed at several different places throughout 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*. Similar to the data description language used by RPC packages, this description lists each operation, its arguments, and the direction of data movement. An interface "compiler" translates this into forms convenient for automatic manipulation.

4.7 Framework Portability

The UCLA interface has proven to be quite portable. Initially implemented under SunOS 4.0.3, it has since been ported to SunOS 4.1.1. In addition, the in-kernel stacking and extensibility portions of the interface have been ported to BSD 4.4. Although BSD's *namei* approach to pathname translation required some change, we are largely pleased with our framework's portability to a system with an independently derived vnode interface. Section 5.3 discusses portability of individual layers.

While the UCLA interface itself has proven to be portable, portability of individual layers is somewhat more difficult. None of the implementations described have identical sets of vnode operations, and pathname translation approaches differ considerably between SunOS and BSD.

Fortunately, several aspects of the UCLA interface provide approaches to address layer portability. Extensibility allows layers with different sets of operations to co-exist. In fact, interface additions from SunOS 4.0.3 to 4.1.1 required no changes to existing layers. When interface differences are significantly greater, a compatibility layer (see Section 3.5) provides an opportunity to run layers without change. Ultimately, adoption of a standard set of core operations (as well as other system services) is required for effortless layer portability.

5 Performance and Experience

While a stackable file system design offers numerous advantages, file system layering will not be widely accepted if layer overhead is such that a monolithic file system performs significantly better than one formed from multiple layers. To verify layering performance, overhead was evaluated from several points of view.

If stackable layering is to encourage rapid advance in filing, it must have not only good performance, but it also must facilitate file system development. Here we also examine this aspect of "performance", first by comparing the development of similar file systems with and without the UCLA interface, and then by examining the development of layers in the new system.

Finally, compatibility problems are one of the primary barriers to the use of current filing abstractions. We conclude by describing our experiences in applying stacking to resolve filing incompatibilities.

5.1 Layer Performance

To examine the performance of the UCLA interface, we consider several classes of benchmarks. First, we examine the costs of particular parts of this interface with "micro-benchmarks". We then consider how the interface affects overall system performance by comparing a stackable layers kernel to an unmodified kernel. Finally we evaluate the performance of multi-layer file systems by determining the overhead as the number of layers changes.

The UCLA interface measured here 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 5.1.2 used the new interface throughout the new kernel, while those in Section 5.1.3 used it only within file systems.

5.1.1 Micro-benchmarks

The new interface changes the way every file system operation is invoked. To minimize overhead, operation calls must be very inexpensive. Here we discuss two portions of the interface: the method for calling an operation, and the bypass routine. Cost of operation invocation is key to performance, since it is an unavoidable cost of stacking no matter how layers themselves are constructed.

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 order-of-magnitude comparison[9].

We began by considering the cost of invoking an operation in the vnode and the UCLA interfaces. On a Sun-3 platform, the original vnode calling sequence translates into four assembly language instructions, while the new sequence requires six instructions[10]. We view this overhead as not significant with respect to most file system operations.

We are also interested in the cost of the bypass routine. We envision a number of "filter" file system layers, each adding new abilities to the file system stack. File compression or local disk caching are examples of services such layers might offer. These layers pass many 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[11]. About one-third of these instructions are not in the main flow, being 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 5.1.3 examines the overall performance impact of a multi-layered file system.

5.1.2 Interface performance

While instruction counts are useful, actual implementation performance measurements are essential for evaluation. The first step compares a kernel supporting only the UCLA interface with a standard kernel.

To do so, we consider two benchmarks: the modified Andrew benchmark [8, 4] and the recursive copy and removal of large subdirectory trees. In addition, we examine 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, the brevity of the first four phases relative to granularity makes accuracy difficult. In addition, the long compile phase dominates overall benchmark results. Nevertheless, taken as a whole, this benchmark probably characterizes "normal use" better than a file-system

[9] 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.

[10] We found a similar ratio on SPARC-based architectures, where the old sequence required five instructions, the new eight. In both cases these calling sequences do not include code to pass arguments of the operation.

[11] 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.

intensive benchmark such as a recursive copy/remove.

The results from the benchmark can be seen in Table 1. Overhead for the first four phases averages about two percent. Coarse timing granularity and the very short run times for these benchmarks limit their accuracy. The compile phase shows only a slight overhead. We attribute this lower overhead to 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 recursive copy and remove times. 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) instead of total elapsed time. This greatly exaggerates the impact of layering, since all overhead is in the kernel and system time is usually small compared to the elapsed, "wall clock" time a user actually experiences. As can be seen in Table 2, system time overhead averages about 1.5%.

5.1.3 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 UCLA interface within all file systems and the vnode interface throughout the rest of the kernel[12]. At the base of the stack we placed a Berkeley fast file system, modified to use the UCLA interface. Above this layer we mounted from zero to six null layers, each of which merely forwards all operations to the next layer of the stack. We ran the benchmarks described in the previous section upon those file system stacks. This test is by far the worst possible case for layering since each added layer incurs full overhead without providing any additional functionality.

Figure 9 shows the results of this study. 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 10 compares recursive copy and remove system times (the modified Andrew benchmark does not report system time statistics). 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. Slightly better performance for the case of one layer in Figure 10 results from a slight caching effect of the null layer compared to the standard UFS. Differences in benchmark overheads are the result of differences in the ratio 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.

5.2 Layer Implementation Effort

An important goal of stackable file systems and this interface is to ease the job of new file system development. Importing functionality with existing layers saves a significant amount of time in new development, but this savings must be compared to the effort required to employ stackable layers. The next three sections compare development with and without the UCLA interface, and examine how layering can be used for both large and small filing services. We conclude that layering simplifies both small and large projects.

5.2.1 Simple layer development

A first concern when developing new file system layers was that the process would prove to be more complicated than development of existing file systems. Most other kernel interfaces do not support extensibility; would this facility complicate implementation?

To evaluate complexity, we choose to examine the size of similar layers implemented both with and without the UCLA interface. A simple "pass-through" layer was chose for comparison: the loopback file system under the traditional vnode interface, and the null layer under the UCLA interface[13] We performed this comparison for

[12] To improve portability, we desired to modify as little of the kernel as possible. Mapping between interfaces occurs automatically upon first entry of a file system layer.

[13] In SunOS the null layer was augmented to exactly reproduce the semantics of the loopback layer. This was not necessary in

Phase	Vnode interface time	%RSD	UCLA interface time	%RSD	Percent Overhead
MakeDir	3.3	16.1	3.2	14.8	−3.03
Copy	18.8	4.7	19.1	5.0	1.60
ScanDir	17.3	5.1	17.8	7.9	2.89
ReadAll	28.2	1.8	28.8	2.0	2.13
Make	327.1	0.4	328.1	0.7	0.31
Overall	394.7	0.4	396.9	0.9	0.56

Table 1: Modified Andrew benchmark results running on kernels using the vnode and the UCLA interfaces. Time values (in seconds, timer granularity one second) are the means of elapsed time from 29 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.

Phase	Vnode interface time	%RSD	UCLA interface time	%RSD	Percent Overhead
Recursive Copy	51.57	1.28	52.55	1.11	1.90
Recursive Remove	25.26	2.50	25.41	2.80	0.59
Overall	76.83	0.87	77.96	1.11	1.47

Table 2: Recursive copy and remove benchmark results running on kernels using the vnode and UCLA interfaces. Time values (in seconds, timer granularity 0.1 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.

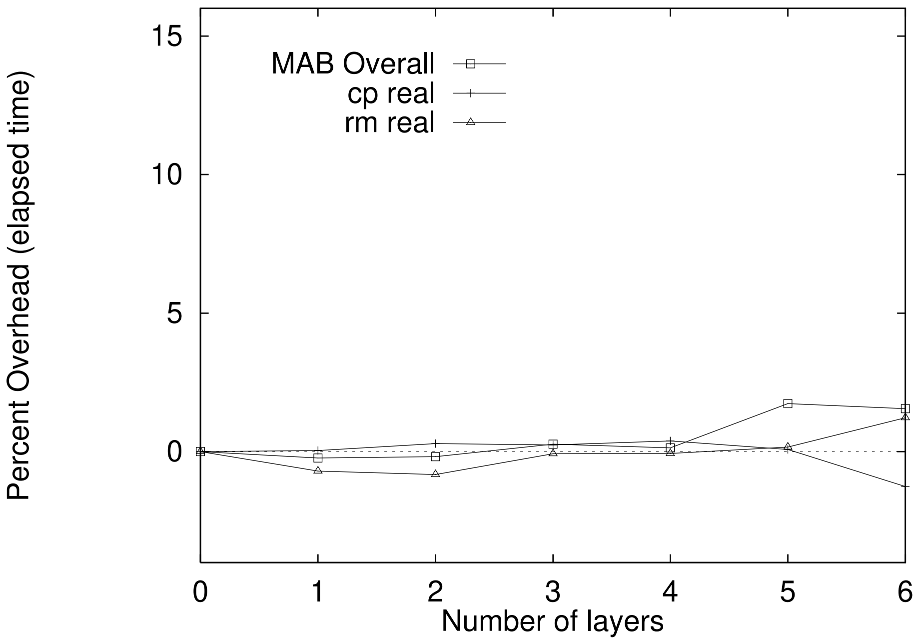

Figure 9: Elapsed time of recursive copy/remove and modified Andrew benchmarks as layers are added to a file system stack. Each data point is the mean of four runs.

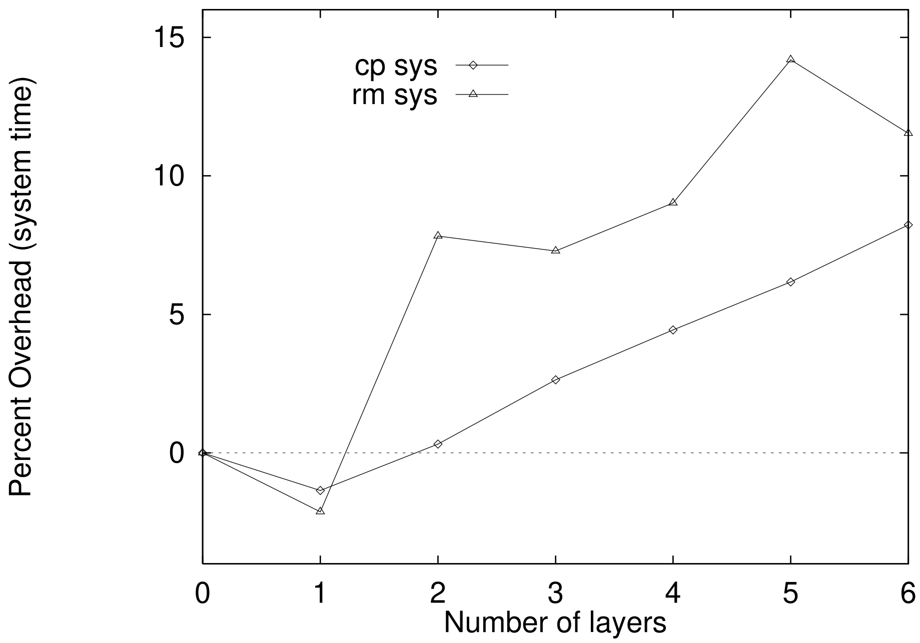

Figure 10: System time of recursive copy/remove benchmarks as layers are added to a file system stack (the modified Andrew benchmark does not provide system time). Each data point is the mean of four runs. Measuring system time alone of a do-nothing layer represents the worst possible layering overhead.

	SunOS	BSD
loopback-fs	743 lines	1046 lines
null layer	632 lines	578 lines
difference	111 lines (15%)	468 lines (45%)

Table 3: Number of lines of comment-free code needed to implement a pass-through layer or file system in SunOS 4.0.3 and BSD 4.4.

both the SunOS 4.0.3 and the BSD 4.4 implementations, measuring complexity as numbers of lines of comment-free C code[14].

Table 3 compares the code length of each service in the two operating systems. Closer examination revealed that the majority of code savings occurs in the implementation of individual vnode operations. The null layer implements most operations with a bypass routine, while the loopback file system must explicitly forward each operation. In spite of a smaller implementation, the services provided by the null layer are also more general; the same implementation will support the addition of future operations.

For the example of a pass-through layer, use of the UCLA interface enabled improved functionality with a smaller implementation. Although the relative difference in size would be less for single layers providing multiple services, a goal of stackable layers is to provide sophisticated services through multiple, reusable layers. This goal requires that minimal layers be as simple as possible.

We are currently pursuing strategies to further reduce the absolute size of null layer code. We expect to unify vnode management routines for null-derived layers, centralizing this common service.

5.2.2 Layer development experience

The best way to demonstrate the generality of a new design technique is through its use by different parties and in application to different problems.

To gain more perspective on this issue students were invited to design and develop new layers as part of a graduate class at UCLA. While all were proficient programmers, their kernel programming experience ranged from none to considerable. Five groups of one or two students were each provided with a null layer and a user-level development environment.

BSD Unix.

[14] While well commented code might be a better comparison, the null layer was quite heavily commented for pedagogical reasons, while the loopback layer had only sparse comments. We chose to eliminate this variable.

All projects succeeded in provided functioning prototype layers. Prototypes include a file-versioning layer, an encryption layer, a compression layer, second class replication as a layer, and an NFS consistency layer. 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 included time to become familiar with the development environment, as well as layer design and implementation.

Review of the development of these layers suggested three primary contributions of stacking to this experiment. First, by relying on a lower layer to provide basic filing services, detailed understanding of these services was unnecessary. Second, by beginning with a null layer, new implementation required was largely focused on the problem being solved rather than peripheral framework issues. Finally, the out-of-kernel layer development platform provided a convenient, familiar environment compared to traditional kernel development.

We consider this experience a promising indication of the ease of development offered by stackable layers. Previously, new file system functionality required in-kernel modification of current file systems, requiring knowledge of multi-thousand line file systems and low-level kernel debugging tools. With stackable layers, students in the class were able to investigate significant new filing capabilities with knowledge only of the stackable interface and programming methodology.

5.2.3 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 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 three-year existence. Its developers' computing environment (including Ficus development) is completely supported in Ficus, and it is now in use at various sites in the United States.

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 does not.

Ficus makes good use of stackable concepts such as extensibility, cooperating layers, an extensible transport layer, and out-of-kernel development. Extensibility is widely used in Ficus to provide replication-specific operations. The concept of cooperating layers is fundamental to the Ficus architecture, where some services must be provided "close" to the user while others must be close to data storage. Between the Ficus layers, the optional transport layer has provided easy access to any replica, leveraging location transparency as well. Finally, the out-of-kernel debugging environment has proved particularly important in early development, saving significant development time.

As a full-scale example of the use of stackable layering and the UCLA interface, Ficus illustrates 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.

5.3 Compatibility Experiences

Extensibility and layering are powerful tools to address compatibility problems. Section 3.5 discusses several different approaches to employ these tools; here we consider how effective these tools have proven to be in practice. Our experiences here primarily concern the use and evolution of the Ficus layers, the user-id mapping and null layers, and stack-enabled versions of NFS and UFS.

Extensibility has proven quite effective in supporting "third party"-style change. The file system layers developed at UCLA evolve independently of each other and of standard filing services. Operations are frequently added to the Ficus layers with minimal consequences on the other layers. We have encountered some cache consistency problems resulting from extensibility and our transport layer. We are currently implementing cache coherence protocols as discussed in Section 4.3.3 to address this issue. Without extensibility, each interface change would require changes to all other layers, greatly slowing progress.

We have had mixed experiences with portability between different operating systems. On the positive side, Ficus is currently accessible from PCs running MS-DOS (see Figure 11). The PC runs an NFS implementation to communicate with a UNIX host running Ficus. Ficus requires more information to identify files than will fit in an NFS file identifier, so we employ an additional "shrinkfid" layer to map over this difference.

Actual portability of layers between the SunOS and BSD stacking implementations is more difficult. Each operating system has a radically different set of core vnode operations and related services. For this reason, and because of licensing restrictions we chose to reimplement the null and user-id mapping layers for the BSD port. Although we expect that a compatibility layer could mask interface differences, long term interoperability requires not only a consistent stacking framework

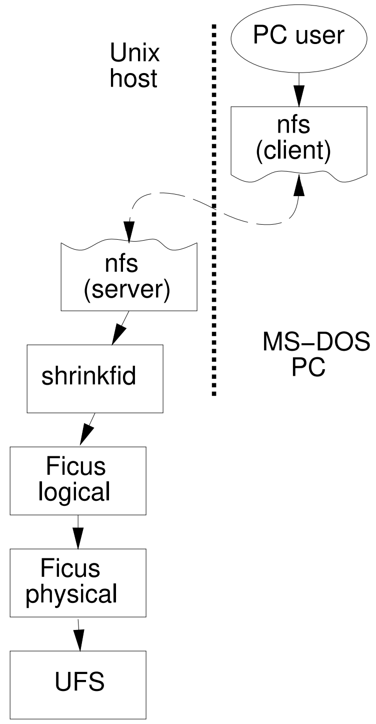

Figure 11: Access to UNIX-based Ficus from a PC running MS-DOS. NFS bridges operating system differences; the shrinkfid layer addresses minor internal interface differences.

but also a common set of core operations and related operating system services.

Finally, we have had quite good success employing simple compatibility layers to map over minor interface differences. The shrinkfid and umap layers each correct deficiencies in interface or administrative configuration. We have also constructed a simple layer which passes additional state information closes through extensible NFS as new operations.

6 Related Work

Stackable filing environments build upon several bodies of existing work. UNIX shell programming, Streams, and the x-kernel present examples of stackable development, primarily applied to network protocols and terminal processing. There is also a significant relationship between stacking and object-oriented design. Sun's vnode interface provides a basis for modular file systems. Finally, Rosenthal has presented a prototype stackable filing interface independently descended from these examples. We will consider each of these in turn.

6.1 Other Stackable Systems

The key characteristics of a stackable file system are its symmetric interface and a flexible method of joining these layers. UNIX shell programming provides an early example of combining independently developed modules with a syntactically identical interface [9].

Ritchie applied these principles to one kernel subsystem with the Streams device I/O system [10]. Ritchie's system constructs terminal and network protocols by composing stackable modules which may be added and removed during operation. Ritchie's conclusion is that Streams significantly reduce complexity and improve maintainability of this portion of the kernel. Since their development Streams have been widely adopted.

The x-kernel is an operating system nucleus designed to simplify network protocol implementation by implementing all protocols as stackable layers [5]. Key features are a uniform protocol interface, allowing arbitrary protocol composition; run-time choice of protocol stacks, allowing selection based on efficiency; and very inexpensive layer transition. The x-kernel demonstrates the effectiveness of layering in new protocol development in the network environment, and that performance need not suffer.

Shell programming, Streams, and the x-kernel are all important examples of stackable environments. They differ from our work in stackable file systems primarily in the richness of their services and the level of performance demands. The pipe mechanism provides only a simple byte-stream of data, leaving it to the application to impose structure. Both Streams and the x-kernel also place very few constraints or requirements on their interface, effectively annotating message streams with control information. A stackable file system, on the other hand, must provide the complete suite of expected filing operations under reasonably extreme performance requirements

Caching of persistent data is another major difference between Streams-like approaches and stackable file systems. File systems store persistent data which may be repeatedly accessed, making caching of frequently accessed data both possible and necessary. Because of the performance differences between cached and non-cached data, file caching is mandatory in production systems. Network protocols operate strictly with transient data, and so caching issues need not be addressed.

6.2 Object-orientation and Stacking

Strong parallels exist between "object-oriented" design techniques and stacking. Object-oriented design is frequently characterized by strong data encapsulation, late binding, and inheritance. Each of these has a counterpart in stacking. Strong data encapsulation is required; without encapsulation one cannot manipulate layers as black boxes. Late binding is analogous to run-time stack configuration. Inheritance parallels a layer providing a bypass routine; operations inherited in an object-oriented system would be bypassed through a stack to the implementing layer.

Stacking differs from object-oriented design in two broad areas. Object-orientation is often associated with a particular programming language. Such languages are typically general purpose, while stackable filing can be instead tuned for much more specific requirements. For example, languages usually employ similar mechanisms (compilers and linkers) to define a new class of objects and to instantiate individual objects. In a stackable filing environment, however, far more people will configure (instantiate) new stacks than will design new layers. As a result, special tools exist to simplify this process.

A second difference concerns inheritance. Simple stackable layers can easily be described in object-oriented terms. For example, the compression stack of Figure 3 can be thought of as a compression sub-class of normal files; similarly a remote-access layer could be described as a sub-class of "files". But with stacking it is not uncommon to employ multiple remote-access layers. It is less clear how to express this characteristic in traditional object-oriented terms.

6.3 Modular File Systems

Sun's vnode interface [7] served as a foundation for our stackable file systems work. Section 2 compares stackable filing and the standard vnode interface. We build upon its abstractions and approach to modularity to provide stackable filing.

The standard vnode interface has been used to provide basic file system stacking. Sun's loopback and translucent file systems [3], and early versions of the Ficus file system were all built with a standard vnode interface. These implementations highlight the primary differences between the standard vnode interface and our stackable environment; with support for extensibility and explicit support for stacking, the UCLA interface is significantly easier to employ (see Section 5.2.1).

6.4 Rosenthal's Stackable Interface

Rosenthal [12] has also recently explored stackable filing. Although conceptually similar to our work, the approaches differ with regard to stack configuration, stack view consistency, and extensibility.

Stack configuration in Rosenthal's model is accomplished by two new operations, *push* and *pop*. Stacks are configured on a file-by-file basis with these operations, unlike our subtree granularity configuration. Per-file configuration allows additional configuration flexibility, since arbitrary files can be independently configured. However, this flexibility complicates the task of maintaining this information; it is not clear how current tools can be applied to this problem. A second concern is that these new operations are specialized for the construction of linear stacks. Push and pop do not support more general stack fan in and fan out.

Rosenthal's stack model requires that all users see an identical view of stack layers; dynamic changes of the stack by one client will be perceived by all other clients. As a result, it is possible to push a new layer on an existing stack and have all clients immediately begin using the new layer. In principle, one might dynamically add and remove a measurements layer during file use. This approach also can be used to implement mounts as a new vnode pushed over the mount point.

However, it is not clear that this facility is widely needed. Because stack layers typically have semantic content, a client will expect stack contents to remain unchanged during use. Consider a compression layer. Clearly if it were used to write the file, the corresponding decompression service needs to be employed to read the data. This argues that a more global dynamic change may not be necessary, and, to the extent that it adds complexity and overhead, undesirable.

In addition, insuring that all stack clients agree on stack construction has a number of drawbacks. As discussed in Section 3.3, access to different stack layers is often useful for special tasks such as backup, debugging, and remote access. Such diverse access is explicitly prohibited if only one stack view is allowed. Insuring a common stack top also requires very careful locking in a multiprocessor implementation, at some performance cost. Since the UCLA interface does not enforce atomic stack configuration, it does not share this overhead.

The most significant problem with Rosenthal's 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" (see Section 3.3). Rosenthal's guarantee of a single stack view for all stack users does not make sense with multiple stack tops. Furthermore, with transport layers, the correct stack top could be in another address space, making it impossible to keep a top-of-stack pointer. For all these reasons, our stack model explicitly permits different clients to access the stack at different layers[15].

A final difference between Rosenthal's vnode interface and the UCLA interface concerns extensibility. Rosenthal discusses the use of versioning layers to map between different interfaces. While versioning layers work well to map between pairs of layers with conflicting semantics, the number of mappings required grows exponentially with the number of changes, making this approach unsuitable for wide-scale, third party change. A more general solution to extensibility is preferable, in our view.

7 Conclusion

The focus of this work is to improve the file system development process. This has been approached in several ways. Stacking provides a framework allowing re-use of existing filing services. Higher level services can be built quickly by leveraging the body of existing file systems; improved low-level facilities can immediately have a wide-reaching impact by replacing existing services. Formal mechanisms for extensibility provide a consistent approach to export new services, even from lower layers of a sophisticated stack. When these facilities are provided in an address space independent manner, this framework enables a number of new development approaches.

[15] While Rosenthal's model can be extended to support non-linear stacking [1], the result is, in effect two different "stacking" methods.

Widespread adoption of a framework such as that described in this paper will permit independent development of filing services by many parties, while individual developers can benefit from the ability to leverage others' work while moving forward independently. By opening this field previously largely restricted to major operating systems vendors, it is hoped that the industry as a whole can progress forward more rapidly.

Acknowledgments

The authors thank Tom Page and Richard Guy for their many discussions regarding stackable file systems. We would also particularly like to thank Kirk McKusick for working with us to bring stacking to BSD 4.4. We would like to acknowledge the contributions of Dieter Rothmeier, Wai Mak, David Ratner, Jeff Weidner, Peter Reiher, Steven Stovall, and Greg Skinner to the Ficus file system, and of Yu Guang Wu and Jan-Simon Pendry for contributions to the null layer. Finally, we wish to thank the reviewers for their many helpful suggestions.

References

[1] Unix International Stackable Files Working Group. Requirements for stackable files. Internal memorandum., 1992.

[2] 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*, pages 63–71. USENIX, June 1990.

[3] David Hendricks. A filesystem for software development. In *USENIX Conference Proceedings*, pages 333–340. USENIX, June 1990.

[4] 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.

[5] 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*, pages 91–101. ACM, December 1989.

[6] Michael J. Karels and Marshall Kirk McKusick. Toward a compatible filesystem interface. In *Proceedings of the European Unix User's Group*, page 15. EUUG, September 1986.

[7] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun Unix. In *USENIX Conference Proceedings*, pages 238–247. USENIX, June 1986.

[8] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Conference Proceedings*, pages 247–256. USENIX, June 1990.

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

[10] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[11] R. Rodriguez, M. Koehler, and R. Hyde. The generic file system. In *USENIX Conference Proceedings*, pages 260–269. USENIX, June 1986.

[12] David S. H. Rosenthal. Evolving the vnode interface. In *USENIX Conference Proceedings*, pages 107–118. USENIX, June 1990.

[13] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *USENIX Conference Proceedings*, pages 119–130. USENIX, June 1985.

[14] David C. Steere, James J. Kistler, and M. Satyanarayanan. Efficient user-level file cache management on the Sun vnode interface. In *USENIX Conference Proceedings*, pages 325–332. USENIX, June 1990.

[15] Neil Webber. Operating system support for portable filesystem extensions. In *USENIX Conference Proceedings*, pages 219–228. USENIX, January 1993.