A Layered Approach to File System Development[*]

John S. Heidemann   Gerald J. Popek

Department of Computer Science
University of California, Los Angeles

Technical Report CSD-910007

March 1991

Abstract

This paper discusses the stackable layers *approach to file system design. With this approach, a file system is constructed from several layers, each implementing one portion of the file system well. Each layer is bounded above and below by an identical interface framework. The symmetry of the interface, coupled with run-time stack definition, make layer configuration flexible and facilitate experimentation and new file system development.*

Addition of new file system functionality to existing environments often requires changes to current interfaces. To address this issue, stackable layers are joined by an extensible interface. Any layer can add to such an interface; existing layers continue to function without modification.

Stackable architectures benefit from new development techniques. This paper examines development methods unique to stackable systems, and concludes with an analysis of the performance of layered file systems.

1 Introduction

The utility of modular structures in systems software is widely recognized. There are the numerous software development advantages, including ease of development, testing, and maintenance. Here we wish to concentrate on one important aspect of modular development, namely the ability of independent parties to contribute function and innovation when interfaces are wisely chosen, generally available and compatibly maintained. Commonly used operating systems especially benefit from this practice, since improvements may be so widely leveraged. If independent parties can work together effectively, then surely sub-groups within a development group are also well served.

Micro-kernels and protocol stacks are two important illustrative examples of efforts to develop suitable interfaces in UNIX. A micro-kernel such as Mach [1] or Chorus [11] divides the operating system into two parts: a core, typically responsible for providing a virtual memory; and processing service, and the remainder of the operating system services that run within the framework provided by the core. In the case of Mach and UNIX, as a figure of merit, the core is of the order of 15% of the total of the operating system kernel. This intra-kernel boundary is an important structuring tool, and may become widely available enough that third parties can build on it. A Mach-style interface however does not set the structure for the remaining 85%, although better practices appear generally encouraged.

The formal framework for protocol stacks provided in UNIX System V Streams [9] goes a step further by providing a structure *within* the network communications software and portions of the device support of the operating system. The interface among protocol layers is fixed; the syntax is the same at all layer boundaries. As a result, third parties have successfully built commercial quality layers that integrate well with other protocol modules. It is not uncom-

mon for the communications software in the operating system to represent another 15% of the kernel, but in this case with a structure that allows multiple, independent groups to contribute to the communications functions. We believe that this ability is one of the reasons why UNIX is a preferred base for networking and distributed systems software in engineering and commercial use[1].

The value of micro-kernels and protocol stacks is encouraging enough to motivate one to consider analogous efforts for other parts of the system software. As one examines the services typically provided in an operating system, the file system is an obvious candidate in which to introduce a firm structure. Today, the *entire* file system can be substituted via the Virtual File System (VFS) interface, but it is not easy to replace or enhance separate portions of the file system, keeping the physical disk management and installing a new directory layer, for example. Since the file system can easily compose 25% of the operating system code, and arguably critical and highly visible code at that, an internal structure that enables third parties to add their value and services could make a major difference in the success of the host operating system, and allow rapid evolution in the services available to users.

However, an internal file system interface is subject to a number of serious requirements if they are to be successful. Most of all, the interface must be very well defined so that independent groups can build their services without cross group coordination. Certainly such an interface must be extensible, in the sense of allowing new functions to be added even though old layers do not recognize the functions and cannot explicitly handle them. New layers must be easy to add. Rich flow of control is needed among modules to support sophisticated caching strategies. Last, but perhaps most important, the interface must be highly efficient, as the file system is often in the "tight loop" of users' computations. A ten percent slow-down in the file system due to its composition as a half dozen layers of software can easily translate to a ten percent slowdown in the system overall, making the approach unacceptable.

The research reported in this paper set out to provide a interface within file system software suitable for wide and heavy use in demanding situations, but still maintaining the characteristics mentioned above. We envisioned a situation where a filing service could

be composed from a number of independent layers, each provided by developers working separately. Examples include physical disk directory management, directory service, selectively replicated files, file versions, encryption for secure storage on servers, single system image synchronization among a group of workstations, a compression layer, and a long term caching service. Each of these examples have been or are being built as stackable layers using the framework described below. One of those cases, replication, is discussed further in a companion paper [7].

The body of this paper describes the stackable filing layers framework; the properties we tried to achieve and why, performance results, remaining work, and our conclusions from the experience gained thus far.

2 Requirements of a Layered Interface

The success of a file system interface depends on its ability to promote the growth of future filing environments. Based on the experiences of existing file system interfaces and our work with stackable layering, we consider several features critical to a layered interface.

Extensibility. An interface must be extensible, allowing addition of new operations as needed. Nearly all interfaces must adapt to meet future requirements. As an example, the continuous evolution of Sun's vnode interface is described by Rosenthal [10].

Current kernel interfaces discourage evolution. Addition of a new operation often requires entire kernel source code availability and modifications to the implementation of existing file systems. These characteristics make it difficult for third parties to offer new file system services, or for several new file systems to function concurrently. Ideally, new file system functions could be added as easily as new device drivers are today.

Stackablity. File systems frequently implement very similar abstractions. Most file systems must coordinate disk access or file and directory allocation, for example.

File system module stacking is an effective method of code reuse, employing existing implementations for well known abstractions. Rather than each file system providing all user services with a monolithic implementation, separable services are placed in individual layers. These layers then serve as powerful

---

[1] In fact, commercial systems such as Netware-386 have adopted the Streams framework, presumably for similar reasons.
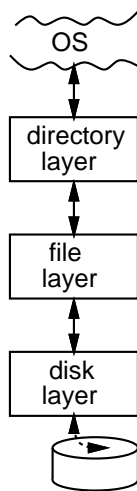
Figure 1: A simple file system stack.

building blocks for future work. Because layers are bounded by symmetric interfaces, combining layers is easy. Often new functionality can be achieved simply by slipping a new layer into currently available stacks.

For 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 inode-level file support, and the top layer provides hierarchical directory services. Figure 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 provide twice the apparent storage.

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 1 again. If the interface supported by the file layer were extended to provide atomic commit, for example, this interface should be available above the directory layer, even though that layer was completed before the commit service was even designed.

Well defined. Most kernel interfaces are described only by paper documentation, relying on programmer vigilance about the types and contents of arguments. For effective use of stackable layers, this approach is insufficient. Meta-data about each operation is very important to allow stackable layers to deal with operations in a generic way. Enough information should be present to allow a layer to package operations to execute in another address space. All relevant information about an interface should be available for layer use at run time.

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.

Rich flow of control. File systems are for the most part passive, responding to user actions. In a stackable file system, user actions can be imagined as passing down through each layer of the stack for processing. Occasionally, however, a user's action needs to pass up the stack, or even "sideways" to another stack. For example, cache invalidation in a multi-layer system can be viewed as a lower layer calling an upper layer to purge its cached data, and failure recovery in a distributed system typically requires consulting peers of the same logical layer. These actions do not proceed naturally down a stack of layers, but instead naturally progress up and sideways between layers. Providing this kind of interaction within the framework of a file system interface minimizes the number of separate constructs required in development.

Opaqueness. 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 optimizations make assumptions about the status of the file system. These optimizations break when confronted with radically new file systems.

3   A New Interface

Our original stackable file system efforts were built using a standard file system interface. As our work progressed, we were frustrated by its lack of extensibility and its limited support for file system stacking. To address these issues, we have adopted a new interface for our future work.

We begin by discussing the vnode interface, a common UNIX file system interface. After reviewing its

structure, we discuss our modifications of this interface to provide stacking and extensibility. Finally, we examine the new interface in light of the goals of Section 2.

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 [5] is a good example of this approach, separating file systems from the remainder of the kernel with an object-oriented interface. 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 consists of two primary data structures. A vfs structure identifies a "file system", or subtree of files, to the operating system. Vnode structures represent individual files within each file system. To provide data abstraction, access to these data types is restricted to a set of operations[2]. By convention, all file systems provide the same set of operations.

The vnode interface supports multiple file system implementations. Although all file systems provide the same set of operations, each may implement them in different ways. To insure that the correct implementation is invoked for a given vnode, each vnode type has associated with it an *operations vector*. This vector lists each vnode operation, associating an operation with the code which implements it. Operations on a vnode are then invoked by an indirect function call through this vector. Arguments to the operation are simply parameters to this function call.

File systems with this interface are configured with the UNIX mount mechanism. Mounting is the process of connecting several independent file system subtrees into the global file system name space. Each subtree represents a group of files with similar characteristics, such as files from a given disk partition or remote host.

Subtrees are made available with the mount system call. To allow configuration of different kinds of file system subtrees, a collection of "private data" specific to the involved file system is included with each mount system call. For example, a local file system would list the disk partition in this private data,

and a remote file system would list the host and path name of the remote subtree.

This mount mechanism has been used to provide some file system stacking with the standard vnode interface. Sun Microsystems' NFS [12], loopback, and translucent [2] 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 The new interface

Our concern with the existing vnode interface is that it is not extensible and it does little to facilitate stacking. To meet these needs, our new interface incorporates several improvements over the standard interface.

To provide extensibility, we construct operations vectors dynamically. In the standard vnode interface, the operations vector is defined by convention. All file systems assume, for example, that the open operation is the first, the close operation the second, and so on. The new interface instead constructs operations vectors dynamically when the operating system is configured. To do this the union of all supported operations is taken, and each operation is assigned a position in the vector. Then a custom operations vector is built for each vnode type.

With an extensible interface, the complete set of operations is not necessarily known when a file system is implemented. A file system must therefore be prepared to handle general operations it does not explicitly implement. In the new interface, each layer provides a default routine to handle this case. Layers at the base of the stack may log the unknown operation and return an error as a default. We expect intermediate file system layers to provide a *bypass routine* which will pass unknown operations to a lower layer by default.

The default routine must be able to handle many different operations. The new interface supports this in two ways. First, rather than passing operation arguments as parameters of the subroutine implementing the operation, they are grouped into a structure, and a pointer to this structure is passed. This method 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 parameter is added to each operation. This argument contains meta-information about the operation: what operation it is, the number

---

[2]Actually, some public data, such as a type and a reference count, is directly available for efficiency. Significant actions are provided by vnode operations.
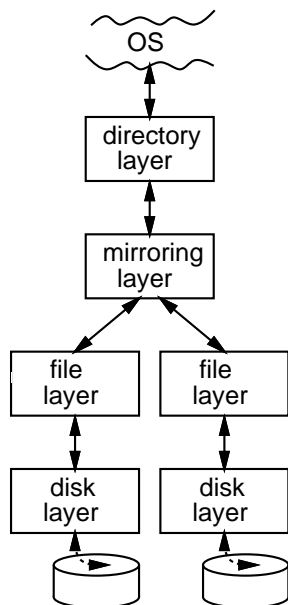
Figure 2: A tree of file system layers.

and types of its arguments, and so on. This description information and the arguments structure extend the object-oriented style provided by the vnode interface to the implementation of the interface itself. The original interface gave the user the ability to perform operations on a vnode without regard to its type; this modification allows a bypass routine to forward an operation to a lower level without regard to the operation involved.

Like the standard vnode interface, the new vnode interface constructs file system stacks at the file system granularity. A complete file system stack is built by creating each layer with a mount command. Each new layer is given a name in the file system name space, allowing potential user access at that level of the stack. This name also serves to identify the layer when another layer is mounted above it.

File system stacks are not necessarily linear. Trees are also possible, with one file system presiding over several lower layers. Figure 2 shows how a tree of file systems might be used to provide disk mirroring. Stack creation proceeds as before, except that to create the mirroring layer, the names of both of its lower level layers must be provided to the mount call.

As an example, Figure 3 shows the use of the get-attributes vnode operation in the fstat system call. The operation is invoked with the **VOP_GETATTR** macro. This macro expands to encode the opera-

tion's arguments in a structure and invoke the operation indirectly through the operations vector. Assuming the operation is performed on a local disk, the **ufs_getattr** routine will be called which performs the operation and returns the results to the user.

3.3 Flow of Control

As observed earlier, file systems are usually passive, responding only to operations from the user. Control typically flows from the user down through layers of the file system.

There are several important applications of file systems which require a richer flow of control. Cache consistency algorithms in distributed systems make use of callbacks, operations invoked on the client by the server, to inform the client of modified data. Similarly, consider a consistency layer connecting several machines providing single system image semantics for the shared file system. Such a layer needs to communicate with the consistency layers of other machines.

An RPC protocol is one approach to providing this kind of non-linear flow of control. However, rather than add another interface to a file system, it would be attractive to generalize the vnode interface to support non-linear flow of control. We have considered different approaches to providing this functionality. Our current method focuses on providing richer flow of control with an NFS-like protocol; we are not entirely satisfied with the degree of transparency this approach offers.

4 Comparison to Other Layering Methods

Ritchie introduced the concept of stackable protocols with the Streams I/O subsystem [9]. His work demonstrated the flexibility a symmetric interface provides in protocol configuration, and has been widely adopted in commercial UNIX systems.

Since that work, the concept of stackable protocols has been applied more widely to kernel interfaces. Independently, Rosenthal has developed a prototype of the vnode interface with Streams-like stacking. The x-kernel has applied the concepts of layered design and late binding to a variety of kernel interfaces. Here we compare these layered design approaches and the standard vnode interface to the work described in this paper.

```
struct vnode {
    int (**v_ops)();
    ...
};

int vop_getattr_offset;/*set at configuration*/
struct vnodeop_desc vop_getattr_desc;
struct vop_getattr_args {
    struct vnodeop_desc *a_desc;
    struct vnode *a_vp;
    struct vattr *a_vap;
    struct ucred *a_cred;
};
#define USES_VOP_GETATTR \
        struct vop_getattr_args getattr_a
#define VOP_GETATTR(VP,VA,C) \
    (getattr_a.a_desc=&vop_getattr_desc, \
    getattr_a.a_vp=(VP), \
    getattr_a.a_vap=(VA), \
    getattr_a.a_cred=(C), \
    (*(VP)->v_ops[vop_getattr_offset])(VP,VA,C))

fstat(uap)
    struct a { int fd; struct stat *buf; } *uap;
{
    USES_VOP_GETATTR;
    struct vattr va;
    vp = FDTOVP(uap->fd);
    VOP_GETATTR(vp,&va,u.u_cred);
    vattr_to_stat (va, uap->buf);
}

ufs_getattr(ap)
    struct vop_getattr_args *ap;
{
    struct inode *ip = VTOI(ap->vp);
    inode_to_vattr(ip, ap->va); /* get stats */
}
```

Figure 3: Declarations for the get-attributes vnode operation, and its use in the fstat system call (without error handling code).

4.1   The standard vnode interface

The new interface differs from the standard vnode interface both in support for extensibility, and facilities for stacking.

Most existing vnode interfaces provide no support for extensibility. Workstation vendors change the interface between releases of the operating system to provide new filing facilities; this ability is not available to third parties. The System V Release 4 vnode interface acknowledges the need for extension by reserving extra space in the operations vector. This space is not of general use, however, because no mechanism is provided to coordinate its use among multiple independent software vendors.

Support for creating new stacks in the old and new interfaces is very similar. Both operate on a file system granularity with the mount mechanism. The support the new interface offers for handling operations in a general way allows the creation of bypass routines, making long-term stackable development much easier.

4.2   Rosenthal's stackable vnode interface

Rosenthal [10] has also developed a prototype file system supporting vnode stacking. While we share many of the same goals, our approaches differ significantly on several points. Stacking occurs at different granularities, and support for dynamic change of stacks and methods of extensibility differ.

Rosenthal proposes to stack at the vnode granularity, rather than the file system granularity. His design allows vnodes within the same file system to have completely different stacks. Our design instead restricts stack composition to a file system basis where vnodes within the same file system have similar stacks.

An ability to configure the filing environment at a fine granularity is, in principle, desirable. One can imagine several files within the same directory, one compressed, the other encrypted and compressed, and so on. While this flexibility is desirable, it requires an underlying filing system with typed files to allow each file to identify its stack separately. Currently, no such filing system is widely available. Whether the additional complexity which results can be managed in practice is not yet clear. This area is an important one for future research.

Another important difference between our work and Rosenthal's concerns when stacks can be manipulated. Rosenthal allows new layers to be pushed

on an active vnode. User operations are always forwarded to the current top of stack, seeing this new layer. With our approach, the new layer would be given a new name in the file system name space; to see the new layer, user requests must be directed at this new name.

Rosenthal's approach has the advantage that the user will see new layers as they are created and added to the stack, since operations are automatically redirected to the stack top. But operations should not always be redirected. A file system layer may implement some operations by performing them directly on the lower layer. These operations cannot be redirected. Therefore, two methods to invoke vnode operations must be provided: one always going to the stack top, and another without redirection.

Like stacks of communications protocols, it is necessary to have the "right" collection of file stack layers in order to successfully manipulate the filing environment. Generally, the stack used to read a file must have the same semantic interpreters in it as those which were used to write the file in the first place. If encryption, compression, and an extended directory service with encoded attributes were used to write the bits on the disk, then they should typically be used to read those bits, by default. It is for this reason that we chose to use a relatively static configuration method for building stacks.

There are some other advantages that result from not pushing and popping stack layers at run time. Since the top of stack does not change, clients which have their own pointers to data linked to a vnode at the top are not disturbed. A good example is the virtual memory manager of some operating systems, which accesses data through the file system. Like any other file system client, it accesses files via vnodes at the top of the stack; but then retains its own references to pages that belong to the file represented by that vnode and linked in memory to it. If the stack top were changed, all those pointers would have to change; alternately, the page cache would have to be flushed[3].

Introducing such dynamic change into an environment which is typically static should be very carefully weighed before it is done; surprises often await. Furthermore, we question the persuasiveness of the

motivations for dynamic layer manipulation.

Finally, the approaches to extensibility differ significantly between Rosenthal's work and ours. Rosenthal employs a versioning layer to map between one version of the vnode interface and another. Such a layer provides a "compatibility mode" for layers using old interface versions until they can be updated. While this method is also possible with our interface, the more gradual interface evolution permitted by our extensible design provides a much more flexible alternative.

4.3   The x-kernel

The x-kernel [4] is designed around the concept of layered protocols. Although originally focused on network protocols, recent work has addressed file systems as well [8].

The scope of the x-kernel work is quite different from that of this paper. The x-kernel seeks to provide a complete new kernel environment, while our work is targeted specifically at the file system portion of existing UNIX systems. Because the x-kernel provides the entire computing environment, it is able to provide all kernel services with a homogeneous, layered interface.

Although the x-kernel provides a number of different protocols, it does not address the issue of evolution of individual protocols.

5   Experiences in Layered Design

Effective use of stackable layering benefits from techniques somewhat different from those used in traditional file system development. This section outlines some of the lessons learned in our layer development.

5.1   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.

To illustrate this point, we present two examples of file system layering. First, consider the standard UNIX file system. It implements three basic abstractions: a file system (a disk partition), file level access

---

[3]Consider the sequence where one reader maps the file into his address space, another module is pushed onto the stack, and a subsequent open followed by reads are done. Since pages in the SunOS virtual memory system are indexed by vnode and file offset, the first pages will be indexed by one vnode and the later reads by another. This problem was first encountered by Rosenthal.
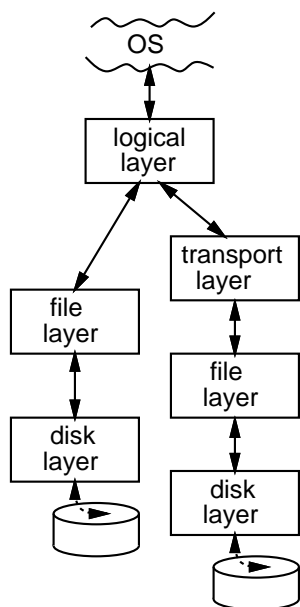
Figure 4: The Ficus stack of layers. The left stack provides access to a local replica. The right stack shows the addition of a transport layer to allow remote replica access.

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, or Ficus for example) which would enjoy efficient inode-level file access without the overhead and complication of directories.

The Ficus replicated file system is a second example of layered file system design. Figure 4 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 replication 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. We are interested in divid-

ing this layer into a file-mapping layer, an extended attributes layer, and a Ficus-specific layer. Two of these layers would then be useful in other contexts.

5.2 Cooperating layers

The previous section encouraged 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. Further details about the Ficus implementation and use of cooperating layers can be found in [7].

5.3 Use of meta-data

Meta-data, information about the operation that is taking place, is an important part of any interface. One important reason why current file system interfaces are unsuitable for use in a stackable environment is that they lack necessary meta-data.

For a kernel interface, the most important meta-data is the identity of the operation and the types of its arguments. With this information, implementing a bypass routine or a transport layer becomes possible.

As an optimization, it often helps to have meta-data present in several forms. For example, an RPC protocol may prefer a list of argument types, while for speed, a bypass routine must quickly access particular arguments. Duplicating this information in different forms improves performance.

5.4 Network transparency

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, and high performance in the local case, as described in Section 5.5.

For this 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. 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. Rather than providing a monolithic networked file system, it layers on top of existing local file systems. Internally, NFS uses a vnode-like RPC interface. But NFS was not designed to serve as a transport layer. Instead, it was specialized for remote file access. Its stateless service complicates its use as a semantics-free transport layer. To address the protocol's lack of extensibility, we have modified it and included support for a bypass routine.

5.5 Uses of transport layers

Transport layers are a powerful component in the construction of distributed file systems. With the configuration flexibility of a stackable environment, transport layers have many additional applications. In the role of remote access, they provide a bridge between hardware and software incompatibilities. In addition, transport layers can be used to gain many of the advantages of a micro-kernel approach to operating system design.

Transport layers can provide access to resources which would not otherwise be available. For example, a new file system might exist only for a particular operating system. By mounting a transport layer above that file system, most features of the new ser-
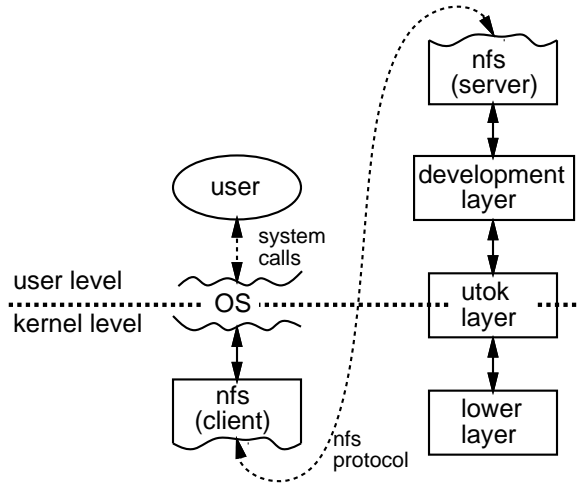

Figure 5: User-level layer development via transport layers.

vice become available to all machines supporting the transport layer. Similarly, a machine lacking hardware resources such as disk space could make use of another machine for data storage by placing a transport layer just above the disk-level file system layer.

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 5 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 good performance for an out-of-kernel file system [13].

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.

6   Performance

The interface described in this paper has been implemented as a modification of SunOS 4.0.3. Two implementations have been made, one converting the entire kernel to use the new interface, another using the new interface only for new file systems and supporting the old interface throughout the rest of the kernel.

To examine the performance of the new interface, we consider several classes of benchmarks. First, we carefully examine the costs of particular parts of the new interface with "micro-benchmarks". We then consider how the interface modifications affect overall system performance by comparing a modified kernel 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. Finally, we compare the implementation effort of similar file systems under both the new and the old interfaces.

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 6.2 used the new interface throughout the new kernel, while those in Section 6.3 used it only within file systems.

6.1   Micro-benchmarks

Parts of the new vnode 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 order-of-magnitude comparison[4].

We began by considering the cost of invoking an operation in the old and the new interfaces. Figure 3 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[5]. 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 imagine 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[6]. 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 6.3 examines the overall performance impact of a multi-layered file system.

6.2   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 new interface with a standard kernel.

To examine overall performance, we consider two benchmarks: the modified Andrew benchmark [6, 3] 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.

The results from the benchmark can be seen in Table 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

---

[4] 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.

[5] We found a similar ratio on SPARC-based architectures,

where the old sequence required five instructions, the new eight.

[6] 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.

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 get a more accurate assessment of performance of the new interface, we examined an additional benchmark. These 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 alone to exaggerate the impact of layering. Our first additional phase recursively copies this data, the second recursively removes it. As can be seen in Table 2, overhead averages about 1.5%.

6.3   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 new interface within all file systems and the old interface throughout the rest of the kernel[7]. At the base of the stack we placed a conventional UNIX file system, modified to use the new 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 the worst case, since each layer provides full layer overhead without any additional functionality.

Figure 6 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. Because all overhead is in the kernel, and the total time spent

in the kernel is only one-tenth of total 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. Differences in benchmark overheads are the result of differences in the ratio between the number of vnode operations and benchmark length. Elapsed time results indicate that under normal load usage, a layered file system architecture will be virtually undetectable. System time costs imply that during heavy file system use a small overhead will be incurred when numerous layers are involved.

6.4   Layer implementation effort

The goal of stackable file systems and this interface is to ease the job of developing new file systems. Clearly, importing functionality from existing layers saves a significant amount of time. Ficus, for example, borrows network transport and low-level disk storage facilities from pre-existing file systems, for great savings in implementation effort. In addition to code reuse, we would hope that implementing in a stackable file system framework is as easy as building conventional file systems. To address these questions, we compare two very similar file systems as developed under each interface.

The loopback file system in SunOS duplicates a portion of the file system name space. Modifications to either copy of the name space appear in the other. This file system is provided in SunOS 4.0 under the vnode interface.

Our null layer, implemented under the new interface, provides very similar characteristics. The null layer forwards all operations to the next layer down the stack. Since each layer has a name visible in the file system name space, both the null layer and the underlying file system are accessible to the user.

Table 3 shows the number of lines of C code needed to implement the loopback file system and the null layer. The amount of support code needed for each implementation is very similar, as are implementations of the mount protocol. The null layer implementation for vnode operations is much shorter, however, since the loopback file system requires special case code to pass each operation down. The services the null layer provides are also more general, since the same implementation will handle all future added operations.

For the example of a pass-through layer, the null file system layer provides better functionality with

---

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

	Old interface		New 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 1: Modified Andrew benchmark results running on kernels using the old and new vnode interfaces. Time values (in seconds, accurate to one second) are the means of elpased 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.

	Old interface		New 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 2: Recursive copy and remove benchmark results running on kernels using the old and new vnode 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.
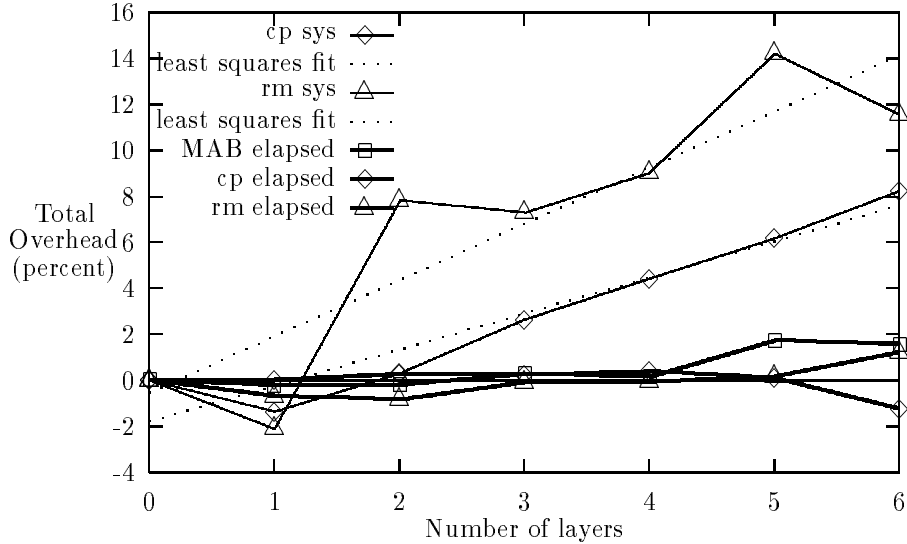

Figure 6: Performance of file system stacks with varying numbers of layers under the new interface. Recursive copy and recursive remove system times and overall modified Andrew benchmark (MAB) times are shown. Dotted lines indicate linear least squares approximations of the data. Each data point is the mean of four runs.

module	loopback file system	null layer
node.h	10	12
info.h	25	37
subr.c	200	199
vfsops.c	135	173
vnodeops.c	373	211
total	743	632

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 3: Number of lines of code needed to implement a pass-through layer or file system.

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 effort.

7 Future Work

Current file systems suffer from their monolithic origins. Using stackable layers, a more modular approach is appropriate. Existing file systems should be broken into several layers, each of which implements only one abstraction. The UFS itself could be divided into several layers, one implementing the concept of a disk partition, one files, and another directories.

New file systems built on top of others will often need to extend the data structures of lower levels. NFS, for example, needed to add a generation number to the inode, and replication in Ficus requires additions to the superblock, the inode, and the directory entry. When a new file system abstraction is implemented, its corresponding data structure must be extensible to allow future layers to build on it. We are currently investigating methods to make file system data structures more extensible.

The vnode interface is a kernel interface for files. Its counterpart for whole file systems is the VFS interface. Modifications to make the VFS interface extensible need to be examined. One approach under consideration is to make the file system vfs data structure a special type of vnode, thereby taking advantage of the mechanisms for vnode extensibility.

Finally, it is important to note that there are currently many slightly different versions of the vnode interface. Standardization on some core set of vnode operations is important to widespread acceptance of the interface. Extensibility mechanisms described in this paper can be used to provide features not widely agreed upon.

8 Conclusions

We have been surprised at how successful stackable layers seem to be in achieving the goals we set out for them. Initial experience suggests that they do represent an interface well enough and extensibly enough defined that third parties can indeed build value added layers for new filing services, or replace services built in this manner by others. Gone for example are the problems of coordinating addition of new operations, or worrying about unimplemented services.

A wide variety of filing services have been provided under this interface. That these services have been built by very small groups or even individuals, sometimes in a very short period of time, demonstrates the power of this approach to enable the user's filing environment to evolve rapidly and see rich improvements in functionality. At the same time, much heavier enhancements employing extensive cross system filing protocols have been equally well provided, as discussed in a companion paper [7].

The retrofitting of stackable layers to Unix systems already equipped with the VFS interface has been reasonably straightforward. No part of the kernel needed modification other than that directly related to the file system interface. While this fact may reveal as much about the quality of the rest of the system's modular construction as the definition and implementation of stackable layers, it is reassuring

nevertheless.

The decision to limit ourselves to stacks that are not dynamically built, but instead are constructed at system startup, is somewhat more controversial. Certainly that is a limitation, as discussed earlier, but it led to such simplification that on balance we believe that it is the right choice at this point.

We chose to overload the *mount* function to construct a file system stack. This choice was made because it allowed incremental development of concepts and made use of already existing naming facilities. However, this use of mount for two concepts complicates the user view. Furthermore, the ability to customize the filing environment on a file-by-file basis may be desirable. We intend to re-examine this decision.

Most of all, the fact that for most benchmarks of interest, a first implementation can perform as well as this one does gives promise that wide use of modular filing structures is indeed feasible, and in light of the earlier observations, especially desirable.

Acknowledgments

References

[1] 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*, pages 93–113. USENIX, June 1986.

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

[3] 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.

[4] 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.

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

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

[7] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report CSD-910005, University of California, Los Angeles, March 1991.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.