Computer Science Department Technical Report

University of California

Los Angeles, CA 90024-1596

AN EXTENSIBLE, STACKABLE INTERFACE FOR FILE SYSTEM DEVELOPMENT

John S. Heidemann

Gerald J. Popek

December 1990

CSD-900044

An Extensible, Stackable Interface for File System Development*

John S. Heidemann Gerald J. Popek†

Department of Computer Science
University of California, Los Angeles

December 19, 1990

Abstract

Desire to support several file system implementations in the same kernel sparked Sun Microsystems' creation of the vnode interface. Although this interface provides good support several simultaneous file systems. Furthermore, the interface often requires change to support new file systems; change to the interface requires updating all existing file systems. To support further file system development, we propose the *stackable layers* design methodology, supported by an *extensible* vnode interface.

The stackable layers approach to file system design separates a file system implementation into reusable layers in a way analogous to the System V STREAMS I/O subsystem. Our approach permits any file system layer to add new operations; other layers adjust accordingly. This report discusses an extensible file system interface, and the special demands required to support stacking. It also describes an implementation of these ideas, including several aspects of performance.

1 Introduction

One thing common to all versions of the UNIX[1] operating system is change. Each version is a little different; each product adds a few new features to make it stand out. Perhaps no area of UNIX has seen more change than the file system, which has evolved from the original version, to the Berkeley Fast File System, and to several remote file systems. Each version has brought new features to the user and new changes to the kernel.

Sun's vnode interface [4] was a response to this relentless change. This interface specifies how a file system interacts with the rest of the kernel, allowing several different file systems to coexist on the same machine. This interface has encouraged the development of several new file systems.

The vnode interface allows the substitution

*This work is sponsored by the Defense Advanced Research Projects Agency under contract number F29601-87-C-0072. John Heidemann is also sponsored by a USENIX scholarship for the 1990-91 academic year. The authors can be reached at 3804 Boelter Hall, UCLA, Los Angeles, CA 90024, or by e-mail to johnh@cs.ucla.edu.

†This author is also affiliated with Locus Computing Corporation.

[1] UNIX is a trademark of AT&T Bell Laboratories.

of one file system for another, but only when they provide very similar services. Adding truly new services almost always requires changing the interface to support new operations that could not have been anticipated by the original designers. Rosenthal [8] describes changes of the SunOS vnode interface with each new operating system release.

Change to the vnode interface requires changes to each existing file system. This makes the cost of interface changes very high, and it makes it impossible to change the interface if source code for all file systems is not available.

If third parties are to play a major role in file system development, a less costly method of extending the vnode interface is essential. An interface which is easily *extensible* is needed. This method must be equally accessible to third party developers as to workstation manufacturers, and it must function without full source code availability.

The Ficus file system [6] is a distributed file system supporting replication with optimistic concurrency control. In its development, we have found the ability to add new operations critical to the success of our efforts.

Another approach we have found vital is the *stackable layers* method of file system design. Instead of each file system standing alone, file system layers are composed into stacks. Each layer implements one file system abstraction well. Layers may then be combined to form more sophisticated systems. Because each layer has the same syntactic interface above and blow, layers which do not alter semantics can be "slipped in" anywhere in a stack. This feature is reminiscent of System V STREAMS modules [7], or commands in a UNIX shell pipeline.

By speeding the development of new functionality, stackable file systems also accelerate the rate of interface change. In addition, stackable techniques such as cooperating layers [1] require the addition of new operations to an interface to accommodate close communication between pairs of layers.

An interface which is both stackable and extensible presents special problems to the developer. Consider Figure 1, showing middle-fs stacked on top of lower-fs. If middle-fs was designed after lower-fs, it could be built to pass all operations supported by lower-fs through. For example, middle-fs might be a layer that selectively compresses files to save disk space, and lower-fs might be a the System V or Berkeley file system. But consider replacing lower-fs with a new file system, say an extents-based file system. This new file system adds new operations to control its unique features. If middle-fs must be explicitly modified to pass these new operations, this will present a powerful inertia against development of new file systems and new stackable layers.

These are some of the problems that must be faced by an interface that claims to be both stackable and extensible. This paper discusses these problems in more detail, and then presents modifications to the vnode interface to make it both stackable and extensible. Finally, implementation of these ideas is discussed, and initial performance data are presented.

2 Requirements

The design of our interface has two primary goals:

- An interface must be able to adapt to change; it must be *extensible*.
- *Stacking* vnodes is a valuable technique in file system development which should be supported by the interface.

These goals are very similar to those of Rosenthal [8]. Stacking is a goal we have in

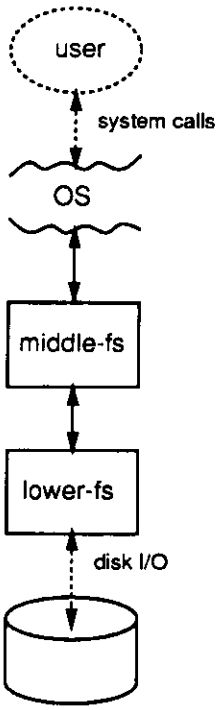


Figure 1: A two-level stack of file systems.

common, but Rosenthal's concern for versioning is a special case of extensibility.

When examined more closely, these goals suggest several more specific requirements:

- A layer must be able to specify a default action for operations it doesn't implement. This default action might be to return an error, or to pass the operation down to the next layer.
- *Transport layers* can be expected to extend the interface between different address spaces. A transport layer must be able to forward operations to other processes, even operations added after the implementation of the transport layer.
- Sophisticated transport layers are likely to provide crash recovery, stateless behavior, or cache coherency. These layers must be extensible to provide these characteristics for new operations.
- It should be possible to add new file systems to an binary-only UNIX distribution. Source code for the new file system and UNIX should not be required.
- The object-oriented character of the vnode interface must be preserved.
- Performance of the new interface must be comparable to the existing vnode interface. The cost of stacking must not be too great, or it will not be practical to use.

3 A New Vnode Interface

The above goals are not met by the standard vnode interface, as it does not allow operations to be added easily, and it does little to facilitate stacking. A revised interface can meet these challenges while preserving the object-oriented flavor of the vnode interface.

3.1 The existing interface

Operations in the standard vnode interface are identified by a constant offset into the vnode operations (vnodeops) vector. For example, **open** is the first operation, **rdwr** the third, and so on. These offsets are defined by convention for each version of an operating system.

Because these offsets are defined only by convention, no method exists to add new operations to this interface. When a new operation is added, a new version of the operating system is created and all existing file systems must be modified to accommodate the new interface. Modifications, or even availability of all file systems is an expensive requirement unavailable to many developers and users.

One approach to future expansion is that taken by System V Release 4, which adds "filler" space to the end of the vnodeops vector. This reserves space for future operations, legitimatizing the need for expansion. But as a long-term solution, this approach is little better than the current policy, since there is no mechanism to assign these unused entries to developers.

Like the operations themselves, the arguments of each operation are specified only by convention. Each subroutine implementing an operation is expected to conform to this convention. Because this is only specified by convention, inter-operation with other computers must be accomplished by better specified protocols. NFS [9], for example, converts vnode operations to a standard network protocol when communicating with other machines.

3.2 An extensible interface

One fundamental problem with the existing vnode interface is that too much of the interface is left to convention. When the interface changes as operations are added, these conventions break down and all existing code must be updated. To solve this problem, we propose more formal methods for interface definition.

The existing vnode interface implements all operations as indirect procedure calls through the vnodeops vector. Automatic configuration of this data structure makes extensible operations possible.

To configure the new interface, each new file system provides a list of operations it supports. This list is all that the designer of a new file system need supply. An example of this list appears in Figure 2, identifying each vnode type, the names of the operations it supports, and pointers to the procedures which implement these operations.

This list is examined at boot time, and the union of all operations is taken. This identifies all operations that the vnode interface must support. The list in Figure 2 includes six unique operations: **open**, **close**, **rdwr**, **new_a_op**, **new_b_op_1**, **new_b_op_2**.

Next, one entry in the vnodeops vector is assigned to each operation. This dynamic allocation of slots is unique to the new interface; in the standard interface these slots are assigned statically by the operating system designer. As each offset is assigned, it is recorded in a global variable for reference when the operation is later invoked. Figure 3 shows one possible assignment of slots for our example.

Finally, a vnodeops vector is built for each type of vnode. Each slot of each vnodeops vector defines a particular operation for that vnode type. A pointer to the subroutine implementing this operation is placed in the slot for supported operations. Each vnode type defines a default routine which is assigned to all operations not otherwise implemented by that vnode type. Simple file systems may simply return the error "operation not supported"; more sophisticated file systems may do more.

This configuration can be thought of as compiling the list of operations from Figure 2 into a table that can be accessed efficiently.

List of provided operations

vnode type	operation name	implementation
anode	open	a_open
anode	close	a_close
anode	rdwr	a_rdwr
anode	new_a_op	a_a_op
anode	default	a_default
bnode	open	b_open
bnode	close	b_close
bnode	rdwr	b_rdwr
bnode	new_b_op_1	b_b_op_1
bnode	new_b_op_2	b_b_op_2
bnode	default	b_default
nfs_node	open	nfs_open
nfs_node	close	nfs_close
nfs_node	rdwr	nfs_rdwr
nfs_node	new_b_op_1	nfs_b_op_1
nfs_node	new_b_op_2	nfs_b_op_2
nfs_node	default	nfs_default
null_node	default	null_default

Figure 2: A list of operations provided by several different file systems. The first column is the type of vnode, the second is the name of the operation. The final column specifies the subroutine that implements this operation for this kind of vnode.

operation name	vnodeops offset
open	0
close	1
rdwr	2
new_a_op	3
new_b_op_1	4
new_b_op_2	5

Figure 3: Offsets assigned to the operations listed in Figure 2.

Resulting vnodeops vectors

vnode type	vnodeops vector					
anode	a_open	a_close	a_rdwr	a_a_op_1	a_default	a_default
bnode	b_open	b_close	b_rdwr	b_default	b_b_op_1	b_b_op_2
nfs_node	nfs_open	nfs_close	nfs_rdwr	nfs_default	nfs_b_op_1	nfs_b_op_2
null_node	null_default	null_default	null_default	null_default	null_default	null_default

Figure 4: Computed vnodeops vectors. Unimplemented operations are filled in with the default routine, provided operations with a pointer to the subroutine.

The vnodeops vector is a particularly fast method for finding the correct routine for invocation. Alternatives, such as each layer using a switch statement to select the correct routine leave the problem of efficiency to the compiler, which often fails to do the best job possible.

Figure 5 shows how operations are invoked in the traditional and the extensible vnode interfaces.

To invoke an operation in the old interface, first the vnode's vnodeops vector was found. Each operation was taken from a well known, constant offset into this vector. The operation was then invoked by an indirect procedure call.

Operations in an extensible interface are invoked the same way, but instead of a well known constant offset into the vnodeops vector, the offset dynamically assigned at boot time is used. In this way operations can be placed anywhere in the vector, allowing any number of new operations are supported.

3.3 File system stacking

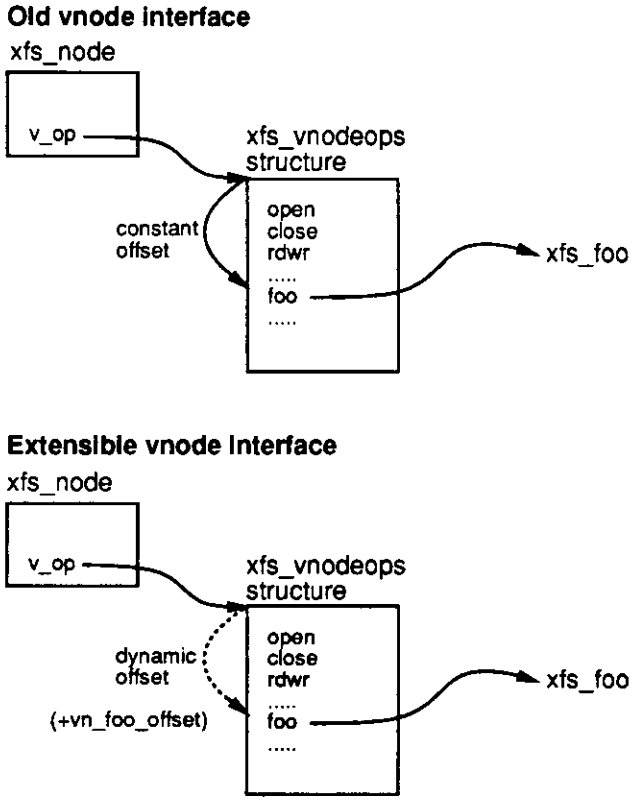
In addition to extensibility, file system stacking is an important tool in file system development. This section examines how file system stacks are created with the new interface.

A file system stack is a composition of stackable layers identified by name in the file system name space. Stacks are often linear, consisting of one layer placed directly on top of another. Stacks can also form a tree, where one layer has several others above or below it. Each stack is associated with a name in the file system name space; users access data stored by the stack through using files prefixed by this name.

User actions to a file system stack are converted to vnode operations sent to the top layer of the stack. This layer interprets these operations as required, possibly calling upon lower layers for service. In particular, actions which return a new vnode to the user typically call upon the next layer down the stack to create an appropriate lower level vnode. Each layer's vnode then holds a reference count and the status of this lower level vnode as part of its private data.

There are two aspects to the UNIX file naming service. The first occurs within a particular file system where the name hierarchy is connected by entries in directories. The second connect the entire tree of an individual file system into the global hierarchy by overlaying it on top of a leaf in the name space. We generalize this second mechanism to build file system stacks, mounting each layer of the stack to a unique name in the name space.

Mounting is done by the kernel with the mount system call. It requires the file system type (NFS or UFS, or the name of some stackable layer) and the location of the new




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

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

Figure 5: Invoking operations with traditional and extensible interfaces.

file system in the name space. What makes this mechanism suitable for stacking is that mount also takes a pointer to a set of parameters specific to the file system type. Stackable layers use this private information to identify the next lower layer of the stack.

For example, consider creating a two level file system. The upper layer could be a compression layer, and the lower may be the standard UNIX file system. One would first mount the UNIX file system, giving it a name in the name space (such as `/layers/raw-data`) and telling it the name of the disk device (`/dev/sd1d`). To place the compression layer over the file system, mount is then called for the second layer. One would mount the compression layer on to some directory (`/usr/data`), providing it the name of the standard file system (`/layers/raw-data`) as its lower layer. Users would then access the top layer of the stack (`/usr/data`).

This approach has many of the same drawbacks and advantages of the standard UNI mount mechanism. All file system levels appear in the name space, so it is possible for users to interfere with them, just as it is possible for a user to write to a raw disk partition. T' blem is typically solved by denying u: ss to lower-level file systems. On the ot: and, the name space an nethods for dealing with it are well known.

R senthal [8] uses a different method for bu g vnode stacks. He describes a method all g layers to be inserted between the r and vnode stacks currently in use. Our p- proach to stacking instead requires that the user explicitly use the path name of the new top-of-stack to begin using the new layer. Because most file systems alter the semantics of the file system stack, it does not usually make sense to insert layers between the user and file currently being used. We have not found this to be a serious limitation in our development of file systems using stacking.

3.4 Stacking and Extensibility

One of the most powerful features of a stackable interface is that file systems can be layered together, each adding functionality to the whole. Often layers in the middle of a stack will pass most operations to a lower layer unchanged. For example, the only purpose of an early version of the Ficus logical layer was to select the replica for use; all other operations were then forwarded directly to this replica for handling. This section discusses methods to forward operations in an extensible environment.

One way to pass operations down to a lower level is to implement, for each operation, a routine which explicitly invokes the same operation in the next lower layer. This approach does not work when the interface is extensible, since new operations can be added at any time. With this approach, addition of a new operation would require addition of this forwarding routine to all existing stackable layers.

To avoid constant changes to existing layers, an interface that is both stackable and extensible must allow a default routine to handle "all other cases". This *bypass* routine can forward operations not understood to a lower layer, or it may simply return an error. Figure 6 shows how a bypass routine might forward operations to a lower level.

A bypass routine must handle forward g an unknown set of operations to a lower level, where each operation has some unknown number of arguments. Information to make this possible does not exist in the standard vnode interface. The new vnode interface makes a bypass routine possible by formalizing information about operation arguments, just as formal management of the vnodeops vector allowed extensibility.

The revised vnode interface describes arguments in two ways. First, rat r tha passing operation arguments as parameters directly to

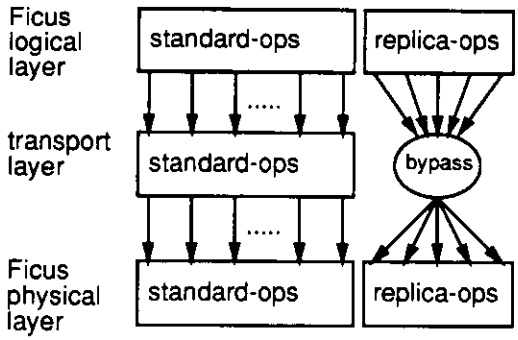


Figure 6: A bypass routine forwards operations to a lower level.

the subroutine implementing the operation, they are grouped into a structure and a pointer to this structure is passed. This allows arguments to be collectively identified by a generic pointer, and it avoids repeatedly copying arguments when passing through several layers of a file system stack.

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

These modifications allow a single bypass routine to pass all operations to a lower layer. Figure 7 shows a simplified implementation of a bypass routine. It extracts the lower-level vnode pointer from its level's private data, and invokes the same operation on the lower level.

A null layer which would simply pass operations down to the next level would consist of a bypass routine alone. This simple layer is very easy to implement with the new interface, and can serve as a basis for more complicated layers.

3.5 Network transparency

A *transport layer* is a stackable layer which transfers operations from one address space to another. NFS is an example of such a layer, providing access to remote file systems by mapping vnode operations into a network protocol and back[2].

The object-oriented flavor of the vnode interface allows remote access to be *network transparent* to the programmer. A vnode which refers to a file on another machine accepts the same operations with the same interface as a local vnode, the programmer never needs to know the difference.

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.

[2]NFS modifies the semantics of the interface to make it stateless. An ideal transport layer would be semantics-free.

```
struct vnode_arg_description {
    int operation_offset;
    ...
    };

struct generic_vnode_args {
    struct vnode_arg_description *desc;
    struct vnode *vp;
    };

struct null_node {
    struct vnode *lowervp;
    struct vnode vnode;
    };

#define VTONULL(VP) \
    ((struct null_node*)((VP)->v_data))

#define VCALL(VP,OFFSET,AP) \
    ((*(VP)->v_op[OFFSET])(AP))

int
null_bypass (ap)
    struct generic_vnode_args *ap;
{
    ap->vp = VTONULL(ap->vp)->lowervp;
    return VCALL(ap->vp,
        ap->desc->operation_offset, ap);
}
```

Figure 7: An implementation of a bypass routine.

To forward an operation to another address space, the types of each operation argument must be known, allowing a network RPC protocol to marshal that operation and its arguments. This information is part of the metadata carried along with each operation. Thus a transport layer may be thought of as a semantics-free RPC protocol with a stylized way of marshaling and delivering arguments.

4 Implementation and 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 for only 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 effect 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 different 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 a two 70 Mb Maxtor XT-1085 hard disks, except for Section 4.3 which used a single 300 Mb Maxtor XT-8380S hard disk.

4.1 Micro-benchmarks

Parts of the new vnode interface we knew would be called at least once per 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, we consider it appropriate for order-of-magnitude comparisons[3].

We began by considering the cost of invoking an operation in the old and the new interfaces. Figure 7 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[4]. We view this overhead as not significant with respect to most file system operations.

We were also interested in the cost of the bypass routine. We imagine many "filter" file system layers, each adding an important feature to the file system stack. File compression or local disk caching are examples of services such layers might offer. These layers would pass most operations directly to the next layer down, modifying the user's actions only rarely (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 very inexpensive. A complete bypass routine amounts to about 54 assembly language instructions[5]. About one-third of this is used only for certain argument combinations, reducing the cost of forwarding simple vnode operations to only 34 instructions. Although this cost is significantly more than a simple subroutine call, we feel 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 4.3 examines the overall performance impact of a multi-layered file system.

4.2 Interface performance

Encouraged by results of the previous section, we anticipated very 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 [5, 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, since timing is done only to a one second resolution. The compile phase shows only a slight overhead. We attribute

[3] Obviously, 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.

[4] We found a similar ratio on RISC-based architectures.

[5] 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.

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 benchmark, we augmented the Andrew benchmark with two additional phases. Both phases operate on large amounts of data (a 4.8 Mb `/usr/include` directory tree) to extend the duration of the benchmark. Our first additional phase recursively copies this data, the second recursively removes it. As can be seen in Table 2, overhead averages a little more than 1%.

4.3 Multiple layer performance

Since the stackable layers design philosophy advocates using several layers to implement what has traditionally been provided for by a single layer, the cost of layer transitions must be minimal. To examine the overall impact of a multi-layer file system, we analyzed 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 supporting the new interface within all file systems and the old interface throughout the rest of the kernel. At the base of the stack we placed a UFS modified to use the new interface. Above this layer we mounted from zero to ten null layers, layers which merely forward all operations to the next layer of the stack. Upon this file system stack we ran the benchmarks used in the last section.

Figure 8 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% overhead; the recursive copy benchmark, there is slightly more than a 1.8% overhead per layer. Recursive remove indicates 3% overhead per layer. These overheads were computed by least squares fits to the sample data, yielding 0.98 correlations for the two later tests, 0.77 for the Andrew benchmark. Differences in overhead per benchmark are the result of differences in the ratio of vnode operations to benchmark length. These results indicate that under normal load usages, a layered file system architecture will be virtually undetectable, but under heavy file system use a small overhead will be felt.

We are investigating ways to reduce layer overhead. Currently it appears that most overhead occurs creating new null layer vnodes. A more sophisticated vnode pool in that layer would likely improve performance.

4.4 Layer implementation effort

The goal of stackable file systems and this interface is to ease the job of developing new file systems. Clearly, importing a 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. In addition to this, we would hope that stackable file system layers are as easy to implement as currently existing file systems. To address this question, we compare two very similar file systems as developed under each interface.

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

The 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 visible to the user.

Table 3 shows the number of lines of C code needed to implement the loopback file system

Phase	Old interface time	%RSD	New interface 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) are the means of 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 (times have only one second accuracy).

Test	Old interface time	%RSD	New interface time	%RSD	overhead
Copy	51.57	1.28	52.54	1.38	1.88
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) are the means of twenty sample runs; %RSD indicates the percent relative standard deviation overhead is the percent overhead of the new interface.

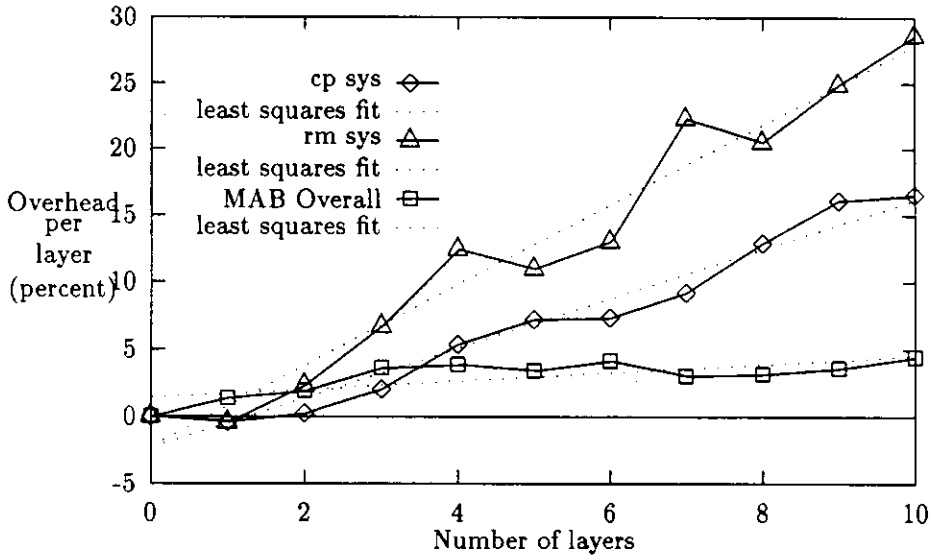


Figure 8: Performance of file system stacks with varying numbers of layers under the new interface. Recursive copy and recursive remove system times and overall modified Andrew benchmark times are shown. Dotted lines indicate linear least squares approximations of the data. Each data point is the mean of four runs.

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.

The loopback file system and the null layer are not necessarily the best example of the cost of file system implementation, since neither implements significant functionality. The Ficus file system is a distributed file system supporting replication. For an analysis of its implementation with stackable layers, see [2].

5 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 in to 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're currently investigating methods to make file system data structures more extensible.

The NFS Version 3 [10] and NeFS [11] proposals expand upon the NFS protocol to accommodate. They present one approach to extensible data structures and operations. The relationship between this work and stackable file systems needs careful consideration.

As another file system layer example, a measurements layer has been designed and is being built at UCLA. We hope to use it to collect trace data and analyze performance. Such a layer is particularly exciting since it can be slid in anywhere in a stack to collect performance information with no modification to the target file system code.

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 we're considering is to make the file system vfs data structure a special type of vnode, thereby taking advantage of the mechanisms for vnode extensibility.

6 Conclusions

This report discusses how change and demand for new file system features requires the ability to add new operations to a file system interface. The current vnode interface makes change a very expensive proposition; modifications presented in this paper allow easy addition of new operations with little overhead.

The combination of an extensible interface and stackable layers creates difficulties when layers are called upon to perform new operations. To avoid requiring changes to old levels, we present the supporting concept of a generic bypass routine.

We have implemented a prototype of these concepts at UCLA. We find performance good, with only a slight overhead required for layer traversal.

We have found this combination of a stackable and extensible file system interface to offers a powerful base for the design of new file

file	lines of code
lnode.h	10
loinfo.h	25
lo_subr.c	200
lo_vfsops.c	135
lo_vnodeops.c	373
total loopback	743
nullnode.h	12
nullinfo.h	37
null_subr.c	199
null_vfsops.c	173
null_vnodeops.c	211
total null	632

node.h This file defines the vnode structure for that file system.

info.h This file provides declarations for mounting.

subr.c This implements utility routines for the file system, such as node management.

vfsops.c This implements the file system mount protocol.

vnodeops.c This provides all vnode operations.

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

systems.

Acknowledgments

The authors would like to thank Yuguang Wu for implementation of the null layer, and Richard Guy and Tom Page for their many helpful comments on this paper. The would also like to acknowledge the contributions of Dieter Rothmeier and Wai Mak to the implementation of the Ficus file system.

References

[1] 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.

[2] John S. Heidemann. Stackable layers: an architecture for file system development. Master's thesis, University of California, Los Angeles, 1991. In progress.

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

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

[6] Thomas W. Page, Jr., Gerald J. Popek, Richard G. Guy, and John S. Heidemann. The Ficus distributed file system: Replication via stackable layers. Technical Report CSD-900009, University of California, Los Angeles, April 1990.

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

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

[9] 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.

[10] Sun Microsystems. Sun network filesystem protocol specification, version 3, *draft*. Available for anonymous ftp on uunet.uu.net as networking/NFS.spec.Z, November 1988.

[11] Sun Microsystems. Network extensible file system protocol specification, *draft*. Available for anonymous ftp on titan.rice.edu as public/nefs.doc.ps, February 1990.