Interposition Mechanism for the OSF/1 AD Version 2 File System

David Noveck

OSF Research Institute
dnoveck@osf.org

March 6, 1994

Abstract

An interposition mechanism is used in OSF/1 AD Version 2 to allow ADFS, a multi-node single-image file system, to be built on top of existing local file systems. The existing mount structure and vnode have been split to allow multiple layers of behavior to be implemented by separate behavior objects distinct from the objects defining file and file system identity, respectively. Changes to the VOP and VFS op interfaces have been made to allow interposition to be implemented without requiring extensive changes to existing file system code. This document explains the details of the interposition mechanism and its use by ADFS.

1 Introduction

An interposition mechanism[1] is used in OSF/1 AD Version 2 to allow the behavior of file system objects (files and file systems) to be modified in order to support coherent cached distributed access without extensive changes to the underlying local file systems.

An overall description of the architecture of ADFS, the multi-node single-image file system for OSF/1 AD Version 2, is given in [1]. This document explains the details of the interposition mechanism and its use by ADFS.

The base OSF/1 vnode implementation combines the identity and behavior for a single object into one data structure (vnode or mount structure). The interposition mechanism splits identity and behavior so that code outside the file system can continue doing operations on a single object identity (the vnode, for example) while allowing the implementation to be composed of multiple layers of behavior implemented by separate bodies of code. This has been done in a manner so as to reduce the changes required to existing code both within existing file systems such as UFS and NFS and with those using the existing VOP mechanism from outside.

[1]This work was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Materiel Command (AFMC).

This implementation has many elements in common with work to allow ‘stackable’ file systems but is not be as general. (See [2] and [3]). Because the possible uses of interposition to implement vnode stacking are so various, there are many, often competing, requirements for such a mechanism. The interposition mechanism does not try to meet all these requirements but focuses on the subset needed to cleanly implement ADFS. Because of the limited set of requirements, we have been able to avoid major reworking of large sections of file system code. Such major rework would almost certainly have been required to accommodate a general implementation of stackable file systems.

2 Data Structures

In order to allow multiple levels of behavior to exist for a single object, two new data structures, the *pvnode* and the *pvfs* are defined, each representing one layer of behavior associated with a file or file system respectively. In order to accommodate existing code in which behavior and identity are combined in a single data structure, one of these objects is embedded within the object corresponding to the associated identity. A pvnode is embedded within each vnode and a pvfs is embedded within each mount structure. Existing code which depends on a single data structure thus continues to work, but additional behavior structures may be added to allow composition of file behavior.

In the rest of the Data Structures section, pvnodes and vnodes will be used as illustrations. In each case, the reader may infer a parallel example with pvfs’s and mount structures.

2.1 Behavior Objects

The data structures representing behavior, the pvnode and pvfs, allow multiple levels of behavior to be composed by linking the objects together and allow the code implementing some piece of file system behavior to maintain its state in an associated interposer-specific data area. Changes to the data items within the behavior object proper should be few and localized to routines designed to establish or remove interposition layers or to allocate or deallocate vnodes. An exception is the use of the m_op and v_op compatibility defines. (See next section for details). Once the behavior objects have been set up, the fields within them are available for general read-only use in implementing interposition layers.

The following are the data items contained in a pvnode.

- pv_op is a pointer to a struct vnodeops specifying the ops vector to be applied for this layer in executing one of the VOP file system operations.

- pv_next is a pointer to the pvnode representing the next lower level of file system behavior for the current vnode. Ops defined in the vector pointed to by pv_op may invoke the ops for this lower layer after doing any desired setup or operand modification. Interposer ops may dispense with calling the lower level entirely whenever they can maintain proper operation semantics on their own (through caching, for instance). It is the responsibility of each layer to provide valid file system semantics to the layer below. The pvnodes linked together by the pv_next field are referred to as the *behavior chain.*

- pv_vnode is a pointer to the vnode for the object being represented. All of the pvnodes on the behavior chain must have the same pv_vnode pointer.

- pv_data is a void pointer to a file system-specific data structure which contains whatever state the ops at this layer need to maintain about the current file.

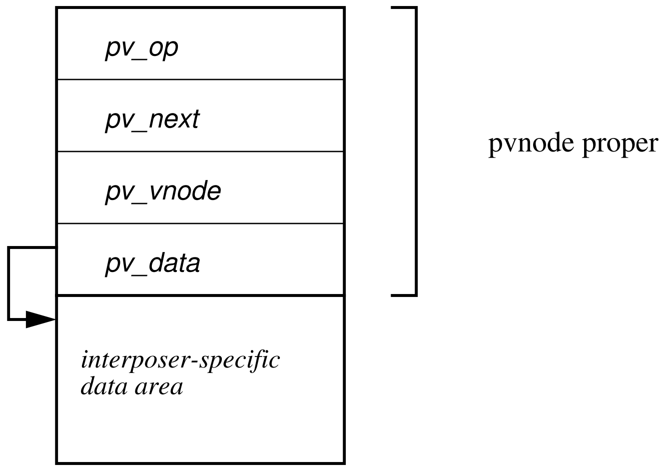

Figure 1. Typical Independent pvnode with associated data

Pvnodes are embedded inside vnodes to provide base file system functionality or they may be allocated independently. Allocation of such independent pvnodes is the responsibility of the module providing that behavioral layer. The normal practice is to allocate the pvnode together with the file-system-specific data.

An independent pvnode can be linked above the pvnode for the base file system with its ops getting control before the ops of the base file system. On the other hand, it may be linked below the pvnode for the base file system in which case its ops only get control after and with the consent of the ops for the base file system. The former case is used to impose a new level of behavior on a file system without the file system's cooperation or knowledge. We refer to this as interposition-ahead. The latter case, which can be used by file system code to divide the required function so that separate features are handled by separate objects, we refer to as delegation.

The behavior objects are small and are intended to remain so. In order to preserve maximum flexibility, additional data should be added only to the associated interposer-specific data structures. When such data is common to many interposers, a new common behavior layer is to be preferred to adding data to the common behavior objects themselves.

2.2 Identity Objects

The vnode and mount structure for OSF/1.3 contain a mixture of identity-related and behavior-related items. They also include a substantial amount of data which is used by a number of different file systems but which is not logically part of the file or file system identity but is instead file system-specific data used in implementing behavior. Reforming this situation by putting each item in the appropriate data structure would be a major undertaking requiring substantial changes to every piece of file system code.

To avoid the costs of a complete cleanup while still allowing interposition to be implemented, we made the following concessions to existing code. The vnode is used as an example. The mount structure is handled identically.

- Instead of splitting a vnode into a pure behavior object and a pure identity object, we only do half the split. We create the pure behavior object as discussed above but retain the vnode as combining identity and behavior albeit in a somewhat cleaner way than in the current system. We embed the behavior object as the first item within the identity object. This behavior object represents the behavior of a base file system.

- Existing behavioral data is retained in the vnode for the use of base file systems.

This approach has the following advantages:

- Existing code outside the a specific filesystem which expects to have access to the current data structure and pass it to VOP's continues to work.

- Existing filesystem VOP's which expect to be called with a vnode pointer can now be called with a pvnode pointer since this is the same address.

- New file system and interposer code can be written aware of the identity/behavior split and expecting a pvnode pointer as the first parameter to the routine implementing the VOP.

- This change makes it easier to implement a full stacking mechanism in the future.

Figure 2 shows the new construction of the vnode. The pvnode is the first item in the structure

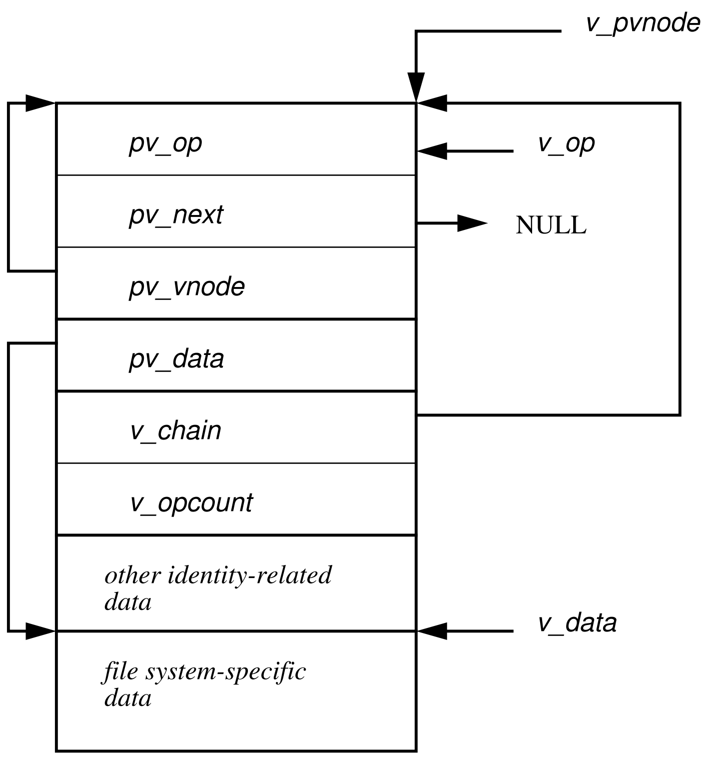

Figure 2. Vnode with only base behavior

and is named v_pvnode. The field v_op will be defined as v_pvnode.pv_op to allow existing filesystem code that refers to v_op to continue to work. The behavioral layers implementing the file's behavior are found starting at v_chain and proceeding through the pv_next field.

An additional change to the vnode is to add a new v_opcount which is maintained to have a count of VOP's in progress. This will be used to allow synchronization between ongoing operations and changes in the pvnode chain.

2.3 Interposition-ahead

Interposition-ahead is implemented as shown in Figure 3. An existing vnode is combined with an independent pvnode linked ahead of it in the behavior chain to allow interception of VOP's by interposer code with the possibility of passing requests along to the base functionality. The figure only shows one level of interposition although any number is possible.

Interposition-ahead can be used to add functionality to a base file system transparently. Interposers will be used to implement the token logic for shared distributed caching of file data and metadata by obtaining the necessary tokens in an interposing layer. See section 4.5 for more information.

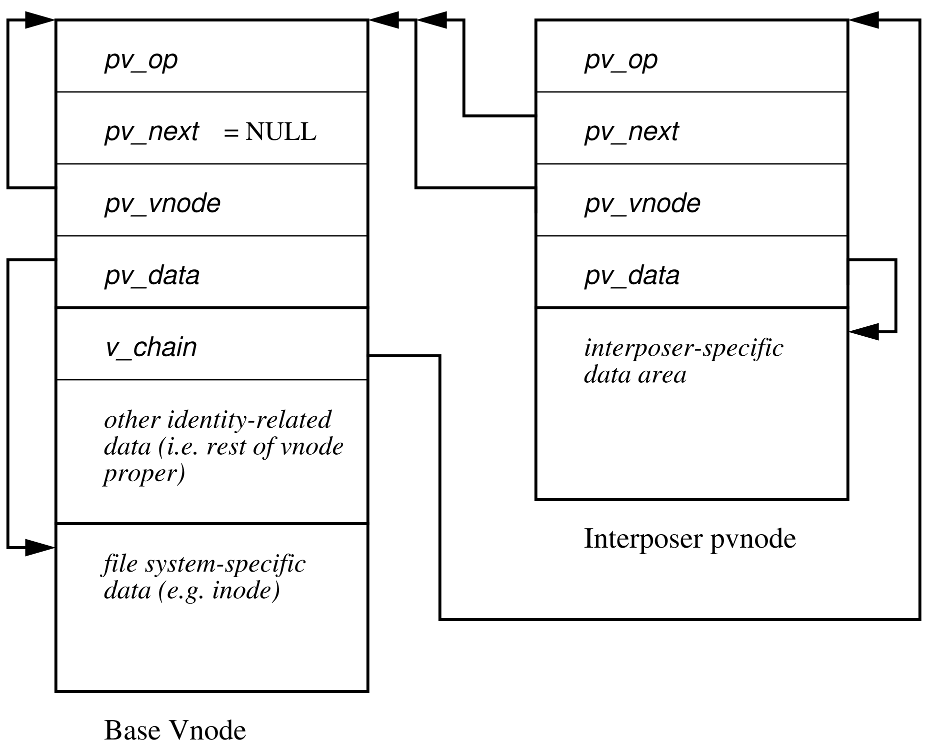

Figure 3. Vnode with one level of interposition-ahead installed

2.4 Delegation

The same data structures which are used for interposition-ahead can be relinked to provide a delegation mechanism. Figure 4 shows a vnode and a pvnode connected to implement delegation.

An example of a situation in which delegation is useful is the implementation of special files. In handling special files, we need a combination of two types of behavior.

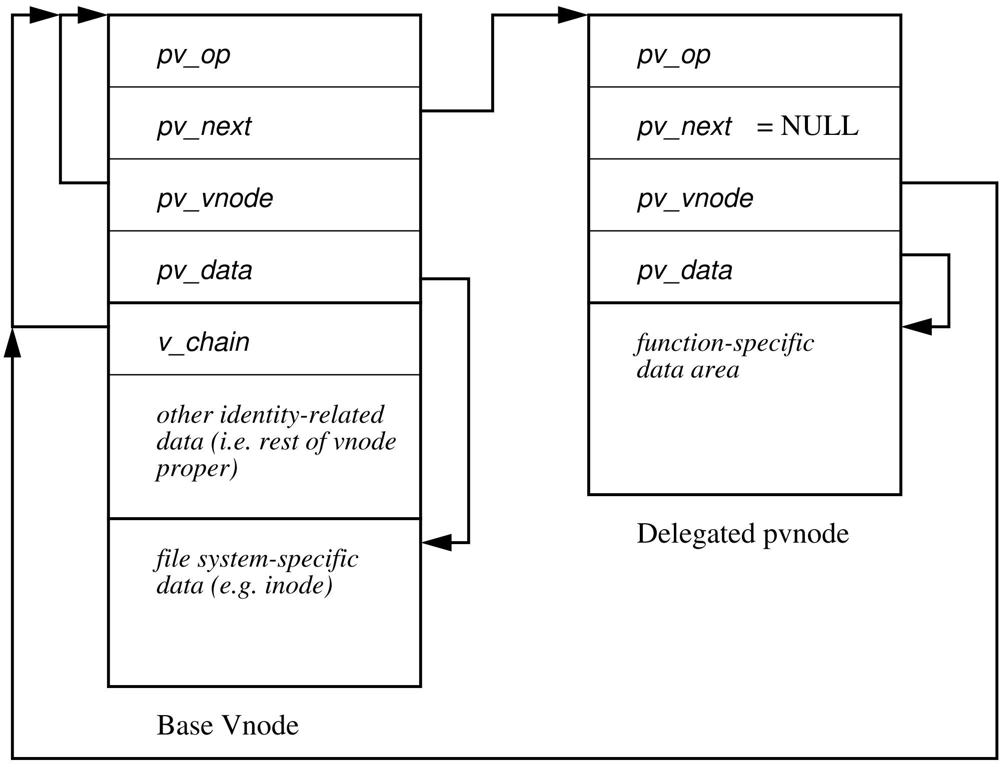

Figure 4. Vnode with one level of delegation installed

- Actions relating to the names and attributes of special files need to be handled by a file system implementing a name tree. This could be UFS locally or ADFS remotely.

- Actions affecting the device associated with the special file are handled differently from normal UFS or ADFS files. They require special handling which should be implemented only once for each type of file in a fifo or device special file system for local objects or a remote special file system for remote objects.

Note that the special file and the object referenced by it may each be local or remote independently. Both the name layer and the object layer should be written independently of the remoteness of the other layer.

Delegation of read, write and ioctl VOP's allows just such isolation. UFS and ADFS will each have one special ops vector providing for delegation of those VOP's after handling access and modified times within the name and attribute layer. This special ops vector would be used for all types of special files. Depending on the type of special file, the fifo, device or remote special file systems would be delegated to handle the read, write and ioctl VOP's. The special file systems would have no knowledge of which file system, UFS or ADFS, would be delegating these operations to it.

2.5 Locking

Locking in general remains the same as in the OSF/1 system. All added fields (i.e. v_chain, v_opcount) are also controlled by v_lock. Instead of having a lock for the pvnode, the entire chain of pvnodes found from v_chain is governed by the v_lock of the base vnode. This locking applies to pv_next. The other fields in the pvnode should never be changed while they are accessible as part of a pvnode. Normally, the application will have its own lock for the interposer-specific data area.

Handling of pvfs's and mount structures is exactly parallel. The added fields m_chain and m_opcount are controlled by m_lock. Associated pvfs's are also controlled by this lock in the same way.

See section 4 for a discussion of how these primitive locking facilities are used in inserting and removing interposers.

3 Ops Interface

The OSF/1.3 VOP and VFS op interfaces have been retained with only the minimal changes needed to support interposition. The code that calls VOP's and VFS op's continues to deal with vnodes and mount structures to represent the identity of the selected file or file system, just as they do today. The OP defines actually call the target routine in the file system code based on the pv_op or pvfs_op field in the first pvnode or pvfs on the chain. This behavior object is also be the first parameter passed to the target routine. This behavior object may be the one at the front of the corresponding identity object or it may correspond to an interposer. Note that in the case of a call to existing file system code, the first parameter will be cast implicitly from behavior object to identity object with the result that existing file system code will continue to work even though the op routines expect a vnode or mount structure address as the first parameter.

The VOP and VFS op macros have been enhanced to maintain the v_opcount field which allows us to make sure that another thread is not traversing the behavior chain when it is modified.

Corresponding to VOP and VFS op macros, there are now PVOP and PVFS macros in which the caller explicitly provides a pvnode or pvfs as the first parameter. These macros are used most often in interposers calling the next layer down in the list or in base file system in which there is partial delegation. Because any mount funneling and operation counting is assumed to have happened at the identity op layer (VOP or VFS op), these macros have no provision for either of these and provide their return values directly, rather than in a special return parameter. To simplify the generation of the appropriate macro's, they are generated, together with the definition of the vnodeops and vfsops by an m4 script using a single definition of ops and the associated parameters.

VOP's or VFS op's which return the address of vnodes (sometimes in nameidata's) continue to do so. This is true whether they are invoked as VOP's or PVOP's.

Within OSF/1 there exists, under the option FS_SEC_CHANGE, a definition for a vnsecops structure parallel to the vnodeops structure. There is, also under this option, an associated v_secop field in the vnode and associated VOP's. These VOP's have not been integrated into this new structure. Should their functions be needed in the future, this special structure will be deleted and these peculiar VOP's implemented as extensions to the standard vnodeops structure.

We illustrate below the OSF/1 definition for VOP_READ and the new ones for VOP_READ and PVOP_READ. The code shown is intended to be illustrative of the actions performed by the macros and is not the actual definition. Instead, defines are selectively expanded to clarify the functions performed. Additionally, backslashes for continuations, some parameter levels, etc. are removed in the interests of readability.

The OSF/1 definition of VOP_READ is represented as below.

```
#define VOP_READ(vn_ptr, ioflag, uio, cred, result)
{
    FUNNEL_DECL(struct mount *mount = (vn_ptr)->v_mount;
    MOUNT_FUNNEL(mount);
    result = (*(vn_ptr->v_op->vn_read))
                    (vn_ptr, ioflag, uio, cred);
    MOUNT_UNFUNNEL(mount);
}
```

The new definition of VOP_READ is as follows:

```
#define VOP_READ(vn_ptr, ioflag, uio, cred, result)
{
    FUNNEL_DECL(struct mount *mount = (vn_ptr)->v_mount;
    MOUNT_FUNNEL(mount);
    VN_LOCK(vn_ptr);
    (vn_ptr)->v_opcount++;
    VN_UNLOCK(vn_ptr);
    result = (*(vn_ptr->v_chain->pv_op->vn_read))
                    (vn_ptr->v_chain, ioflag, uio, cred);
    VN_LOCK(vn_ptr);
    (vn_ptr)->v_opcount--;
    VN_UNLOCK(vn_ptr);
    MOUNT_UNFUNNEL(mount);
}
```

The definition of PVOP_READ is as follows:

```
#define PVOP_READ(pvn_ptr, ioflag, uio, cred)
    (*(pvn_ptr->pv_op->vn_read))(pvn_ptr, ioflag, uio, cred)
```

Note that The MOUNT_FUNNEL code will not exist in most cases. Also, in some configurations, operation counting may not be performed.

3.1 Identity Ops

A small set of VOP's are implemented without allowing behavioral interposition. Unlike the ops discussed above, the routine selected depends on the ops found using pv_op for the pvnode contained in the target vnode, not the first one in the interposition chain. Such ops are referred to *identity ops* since they find their ops vector directly from the corresponding identity object without allowing behavioral interposition. This approach is used in such cases as VOP_STRATEGY to avoid making changes in VOP interfaces where there is no requirement for interposition. (See section 3.2).

Identity ops are implemented in the same way that current VOP's are. They do not do operation counting and will make no reference to the behavior chain.

For identity ops there are no corresponding PVOP's or PVFS ops. This quickly catches any invalid attempt to interpose on these ops.

3.2 Changes in Ops

There are a number of ops which need to be handled specially to support interposition. The following are the issues that we had to deal with:

- Some VOP's do not have the vnode address as the first parameter. These include VOP_STRATEGY which take a **struct buf ***, VOP_OPEN which takes a **struct vnode ****, and a number of directory related VOP's which take a **struct nameidata ***. Also, VFS_VPTOFH takes a **struct vnode *** instead of a **struct mount ***.

- Some VOP's deal with more than one existing vnode. This creates problems since the interposition is only effective on the first vnode, the object to which the op is directed. In [2], this problem is addressed by radical surgery on the VOP interface, breaking many VOP's into smaller operations, each of which deals with only one vnode. This causes a very extensive change to the VOP interface. Such a big change can be avoided in the case of ADFS.

- A number of the OSF/1 conventions regarding reference counting for vnodes across VOP's cause problems for interposition.

In dealing with these issues, we have done the minimum set of changes necessary to allow us to implement a clean interposition mechanism that meets the needs of ADFS. Changes that might be required for other uses of interposition have not been done.

The following lists the changes to the VOP and VFS op interface which have been made:

- The VOP_OPEN has a new **struct vnode *** first parameter which is converted into a **struct pvnode *** parameter for passing to the file system and/or interposer's open routine. The **struct vnode **** parameter previously passed as the first parameter has become the second parameter to allow the identity of the opened file to be changed by the open routine.

- VOP_STRATEGY has been retained in its current form without provision for interposition. It is implemented as an identity op. This approach was taken because such interposition on VOP_STRATEGY is not needed for ADFS and because modifying the system to allow meaningful interposition it would require significant changes to the buffer cache code to implement. This problem is left for the future.

- VOP_ABORTOP has gotten a new **struct vnode *** first parameter which is equal to the ni_dvp field of the nameidata previously passed as the first parameter. This allows interposition on the directory. If interposition on the file is ever needed in the future, this interface will have to be changed. It is unclear whether any interposition will ever be required for this VOP. Currently, all implementations of it within the OSF/1 source tree are no-ops.

- VFS_VPTOFH has been converted to a VOP, which it should have been in the first place anyway. Leaving it a VFS op hopelessly complicates the construction of relevant defines in the context of interposition.

- The VOP's, VOP_CREATE, VOP_MKDIR, VOP_MKNOD and VOP_SYMLINK have been changed to have a **struct vnode *** first parameter. Even though these VOP's deal with two vnodes, they pose no problems for interposition. The second vnode is a new one returned by the VOP. The interposer can add its own interposition to the new vnode after calling down to the lower layer.

- VOP_REMOVE and VOP_RMDIR have also been changed to have a **struct vnode *** first parameter. These VOP's deal with two vnodes, the directory from which an entry is to be removed and the vnode for the object to be removed. Solving this problem required addition of a new VOP, VOP_REMLINK. This is discussed in section 3.2 below.

- VOP_LINK previously had a **struct vnode *** first parameter which designates the file to which a link is to be made. In OSF/1, the op vector is taken from the directory vnode of the **struct nameidata *** second parameter. This was easily changed to reference the first parameter in conformity with other VOP's since both must be on the same file system. Its reference to two vnodes is more of a problem, however. ADFS needs interposition on the directory vnode only. Swapping the parameters would therefore satisfy ADFS's needs but would make this VOP different from its relatives in other UNIX's. Instead a new VFS_LINK has been created to allow interposition with calls to VOP_LINK routed through a vn_link routine. The details will be discussed in section 3.3 below.

- VOP_RENAME has two **struct nameidata *** parameters, one each for source and destination names. Addition of an initial **struct vnode *** parameter could have been done but it would not solve its reference to multiple vnodes. Instead, the VOP has been left as it is as an identity VOP. A new VFS_RENAME was added to allow interposition. Details are discussed in section 3.3 below.

- The OSF/1 convention in which VOP's that change directories are defined to do a vrele on the directory and the file (ni_dvp and ni_vp) passed to it has been changed. This convention, if not changed, would prevent an interposer from assuming that the objects passed to were still valid after calling down to the next lower behavior layer. Since this would make interposers clumsy to write it was decided to change it. Vnodes VREF'ed on entry to a VOP will remain VREF'ed on return.

- Previously, VOP_CREATE and VOP_MKDIR returned the address of the new vnode created in the nameidata structure passed to it. It is VREF'ed so as to be valid for as long as the caller needs it. Unfortunately, VOP_MKNOD and VOP_SYMLINK simply vrele the new vnode and do not return its address. This has been changed so that all of these VOP's work like VOP_CREATE. This allows ADFS server code to create interposers when necessary.

These changes to the VOP interface require only small local edits. Most of the VOP's changed only appear in two places. The changes to the target op routines must be made to each file system's ops but these are also simple and local in nature

3.3 Multiple Vnode Reference by VOP_REMOVE and VOP_RMDIR

The two remove operations inherently refer to multiple vnodes for which interposer action may be required. This leads to the problem that interposition may only be in effect on the vnode which is not the first parameter of the VOP thereby depriving us of a hook for the interposition. We would have no opportunity to get control in our interposition layer since VOP's only follow the behavior chain for the first vnode.

Although we have a nameidata whose ni_vp is supposed to be the object to be deleted, there is no locking to assure this. In OSF/1, UFS assumes that it can determine whether the directory has changed with regard to the target entry since the lookup on ni_vp was done and return an appropriate error.

This approach is not valid since it makes the assumption that the existence of a different vnode with the same name allows the file system to validly return ENOENT. The logic of this is there must have been an instant when neither the original file nor the new file with the target name existed and that we may therefore act as though the VOP was executed during the time when neither file existed. The possibility of concurrent renames invalidates this logic since POSIX requires that a rename over an existing file always show either the old or the new file as existing. The gap required by the existing ENOENT return is defined not to be there.

Therefore, we do not rely on this to key the token revocation. Instead we want to get control when we are sure of the vnode to be referenced. Since one of the reasons that we must get control is to revoke the attribute token since the link count is changing, the best place to do that is precisely when the link count is to be decremented. The method used is a new VOP, VOP_REMLINK.

VOP_REMLINK(struct vnode *vp, int count, ...) is defined as decrementing the link count for the subject vnode by the specified amount. (See section 3.5 for a full definition.) It may be called with a count of zero in those cases in which a name is being invalidated but the overall link count is not changing.

A VOP_REMLINK call is used inside the UFS implementations of VOP_REMOVE, VOP_RMDIR and VOP_RENAME (see section 3.3 for details) to give us a hook for interposition.

The following illustrates what might happen on a typical case of file removal:

- A system call or NFS server code calls VOP_REMOVE specifying the directory from which the removal is to occur as the first parameter.

- Interposers for pvnodes linked ahead of the base file system on the behavior chain for the directory get their chance to execute their PVOP_REMOVE routines with each layer invoking the next one down.

- The base file system's PVOP_REMOVE (in this example ufs_remove) is executed.

- After removing the file entry from the directory, ufs_remove decrements the link count for the file by doing a VOP_REMLINK on the file being unlinked.

- Interposers for pvnodes linked ahead of the base file system on the behavior chain for the file being removed get their chance to execute their PVOP_REMLINK routines with each layer invoking the next one down.

- The base file system's PVOP_REMLINK (in this example, ufs_remlink) is executed. This routine does the actual decrement of the link count.

File systems for which such interposition is not needed for the ADFS project may implement VOP_REMLINK as returning ENOSYS, since it will never be called from outside the file system. This is the case for NFS in AD Version 2 since all NFS operations will be implemented remotely without distributed caching. Such an implementation will not need interposition.

3.4 Interposition Changes for VOP_LINK and VOP_RENAME

In the case of VOP_LINK and VOP_RENAME, we have multiple vnodes on which we might want to do interposition actions, only one of which is the object to which the request is directed and for which interpostion is implemented. In the case of VOP_RENAME, there are also one or two vnodes not available at the VOP interface, the file or directory to be renamed and possibly a vnode for a file to removed because it was renamed over.

The solution is to make these operations which deal with multiple vnodes into VFS ops. Since all the vnodes dealt with must be within a single file system, there is a well defined object to which the request can be directed which subsumes all of the objects currently dealt with by the current VOP's. Successful interposition merely requires the easily-met restriction that there always be a file system interposer whenever interposition is desired on any of its constituent files.

To minimize code changes, the current VOP interfaces have been retained. Existing file systems have a VFS_LINK and VFS_RENAME routine which simply returns ENOSYS. Direct access to the current VOP's is disallowed. Instead, those who previously called VOP_LINK or VOP_RENAME now call the new routines vn_link or vn_rename. Here is an example of an execution of vn_rename.

- A system call or NFS server code call vn_rename passing source and target nameidata's.
- The vn_rename routine would check that all vnodes were for the same file system and return the proper errors if so.
- If all vnodes were from the same file system, VFS_RENAME would be called on that file system.
- If there were no interposers, for this file system, we would fall into a file system routine which would return ENOSYS. If vn_rename received this, it would instead call VOP_RENAME to perform the operation.
- If there were an interposer, its PVFS_RENAME routine could examine each of the vnodes involved and obtain the appropriate tokens for each, in whatever order it chose.
- After obtaining tokens, the PVFS_RENAME routine could then pass on the operation to the next level by doing a PVFS_RENAME on the pvfs in m_chain.
- If that call return ENOSYS (the most likely case), it could then perform the operation by doing a VOP_RENAME. To prevent problems with the insertion of new interposers while this is going on, the operation count on all of the vnodes involved is incremented for the duration of this operation.

The VOP_REMLINK mechanism deals with the vnodes other than the source and target directories. For the renamed object, a VOP_REMLINK(vp, 0, ...) tells us that some link is going away, requiring revocation for the name token in ADFS, while a possible VOP_REMLINK(vp, 1, ...) for a renamed-over file tells us that an unlink is being done, requiring revocation of the name token and the attribute token.

3.5 PVOP_REMLINK

PVOP_REMLINK is defined as returning a boolean_t which is TRUE iff the file's link count is being decremented to zero. Its parameters are as follows:

struct pvnode *pvp

The pvnode for the file whose link count if being decremented. The base PVOP will actually decrement the link count while interposers will use this to key token revocation.

int decrement

The amount to decrement the link count. Zero indicates that the link count is not changing but that the file is having a link with one name replaced by a link with another. This allows revocation of a token associated with name caching, for instance. A decrement of two implies that one of the links being removed is a self-reference (“.” in a directory). A decrement above two is not valid.

char *name
int namelen

Together indicates the name of the link being deleted. This is provided only for the use of interposers which may want to treat the deletion of some names (e.g. “..”) differently.

struct vnode *dirvp

Indicates the directory from which the link is being removed. This is provided also for possible use by interposers. The existence of a second vnode parameter is not a problem for interposers since they would be able to interpose on VOP_REMOVE, VOP_RMDIR or VOP_RENAME to catch the directory.

3.6 Passthrough

In order to simplify the writing of interposition layers, facilities are provided to allow the implementor to avoid writing code for VOP’s and VFS ops where no interposition functionality is needed; i.e. where the only function is to pass the operation down to the next level. Only ops that do something have to be written.

A set of interposer ops that only pass operations through is generated by an m4 script from the definition of VOP’s and VFS ops. The generated .c file also includes routines to set a vnodeops or vfsops structure so that all of the entries refer to these passthrough ops. The creator can then set the ops for which there is actually an implementation leaving the passthrough implementations for everything else.

The default passthrough routines do a panic if an identity op should ever be encountered. Because of the way identity ops are defined they should never be validly seen by interposers.

Similar technology is used for a delegation layer, generating default ops that all generate panic’s to catch anything delegated where a handler does not exist.

4 Inserting and Removing Interposers

4.1 Allocating and Deallocating Behavior Objects

Allocation and deallocation of interposer data structures is the sole responsibility of the layer implementing an interposition layer. No general allocation facilities are provided. This has been done instead of using a single pool as is done for vnodes. We avoided using the single pool for the following reasons:

- The interposer-specific data is most conveniently allocated with the pvnode or pvfs and this structure may vary a great deal in size from interposer to interposer.

- There is no reason to suppose that storage usage by different interposer types will have any tendency to trade off against one another which would make a single pool an advantage. This behavior might reasonably be expected in the case of vnodes where user programs might access various files periodically changing their focus of interest among various file systems.

4.2 Basic Interposer Insertion and Deletion

Holding only v_lock, an interposer can be inserted in the behavior chain. All VOP's which occur after the insertion will see the new chain. VOP's in progress when the chain is changed may or may not see the new interposer depending on where it is inserted relative to the pvnodes for which execution is proceeding when the insertion occurs. In any case, each VOP will see a coherent chain.

Whether the above procedure is valid from the application point of view depends on the nature of the interposer. In the case of delegation for special file handling, the file system can, by only creating the interposer at VOP_OPEN time, be certain that no VOP's for which the new interposer will be entered (VOP_READ, VOP_WRITE, VOP_IOCTL) can have VOP's validly in execution when the interposer is inserted.

Removing an interposer from the chain with v_lock held assures the integrity of the chain but does not prevent VOP's executing from seeing a chain without integrity since they are not prevented from accessing a pvnode that has been removed from the chain and deallocated.

Since the vfs layer assures that a vnode may not be valid accessed by any other thread at VOP_RECLAIM time, we may remove interposers at this time simply by holding the v_lock.

4.3 Synchronization Interposer

An interposer layer has been implemented to allow more controlled insertion and deletion of other interposers into the behavior chains for vnodes and mount structures. This interposer uses the operation count, v_opcount or m_opcount, incremented and decremented by the VOP and VFS op macros.

This interposer is designed to be inserted at the top of the behavior chain. Its function is to stop all new ops from proceeding past the interposer into the rest of the behavior chain, allowing earlier ops to drain away leaving the object in an idle state. Once the object has been put in an idle state, those that need to modify the behavior chain may do so guaranteed that no request is traversing the chain. After all those needing to make such changes have done so, waiting requests are released to execute and the behavior object for the synchronization interposer freed.

The synchronization interposer maintains the followings data:

- A state variable si_state with two possible values, SI_OPS_HOLD and SI_OPS_RELEASE.

- A count, si_ops_waiting, of ops waiting for the changes to the behavior chain to be completed. This count is used to determine when all active ops are in this waiting state.

- A count, si_changes_waiting, of actors waiting for ops to be quiesced before making their changes to the behavior chain. This count is used to determine when all pending changes have been made and the waiting ops may be released.

- A count, si_users_left, of ops which still need to reference the interposer data structure. This count is used to determine when the structure may be freed.

Those needing to change the behavior chain in a controlled way execute the following algorithm:

- Allocate an interposer from the pool maintained for the appropriate type of synchronization interposer and initialize si_state to SI_OPS_HOLD, si_ops_waiting to zero and si_changes_waiting to one.

- With v_lock or m_lock held, insert the interposer at the top of the behavior chain for the object. If there is already such a synchronization interposer already there, increment si_changes_waiting on that object and free the newly allocated object instead of inserting it in the chain.

- Periodically check (every 50 ms or so, its very unlikely to require multiple iterations) to see if si_ops_waiting is equal to the count of ops in effect (v_opcount or m_opcount). This check is done with v_lock or m_lock held. If the counts are equal, with the lock held, decrement si_changes_waiting and make the desired change. It is certain that there are no active ops. Return if si_changes_waiting is non-zero.

- If si_changes_waiting was zero, we continue. Still with the lock held, set si_users_left to the operation count. Remove the interposer from the behavior chain. If the operation count was zero, deallocate the interposer. Otherwise, set the state to SI_OPS_RELEASE, do a wakeup, and leave it to the last op to deallocate the interposer.

All ops for the interposer go to a simple stub which calls a common routine and then passes the op down to the next behavioral layer, the address of the behavior object being returned by this common routine. These stub routines are generated by an m4 script in the same way as the passthrough interposer us generated. The algorithm executed by the common routine is as follows:

- If si_state is SI_OPS_HOLD, increment si_ops_waiting and sleep on the interposer data structure. This is all done under v_lock or m_lock.

- Keep sleeping until the state changes to SI_OPS_RELEASE.

- Under v_lock or m_lock, decrement si_users_left. Fetch pv_next or pvfs_next to use to determine the next behavioral layer.

- If the si_users_left count was just decremented to zero, it is freed before returning. It was already removed from the behavior chain by the thread initiating the behavior change before setting the state to SI_OPS_RELEASE.

4.4 Inserting and Removing ADFS Interposers

ADFS server-side code uses interposition to prevent local operations outdating data cached by remote nodes. The interposing routine can obtain the tokens required for the local operation allowing the consequent token revocation on the remote nodes to invalidate the cached data.

Interposers are only created when the object is known on remote nodes. The ADFS server routines which send handles for objects to remote nodes create the interposers before sending these handles. In fact, the handles contain the addresses of the interposing behavior objects. The continuing execution of VOP's started before the interposer was created could be a problem since the operations being performed might have required that tokens be held if they were started after the interposer was created. Such an operation might outdate data held on remote nodes if they were allowed to continue execution after the interposer was created. To prevent this, the synchronization interposer is used to make sure that all such operations have terminated before the interposer is created.

Deletion of ADFS interposers happens at VOP_RECLAIM time when the synchronization interposer is not required. It would be possible to be more aggressive and remove the interposer when all remote references to a file go away but the added complexity of such an approach is not justified.

Because interposers are only deleted at reclaim time and because they are inserted in a controlled way for delegation, the synchronization interposer is not needed when dynamic interposition-ahead is not done. This is the case on compute nodes where the full ADFS caching file system will not be present. Build options allow the synchronization interposer not to be included in this case.

5 Interfaces to Interposition Support Routines

5.1 Routines to Set Up Ops Vectors

```
void init_passthru_vnops(
      struct vnodeops * ops_to_initialize)
```

```
void init_passthru_vfsops(
      struct vfsops * ops_to_initialize)
```

These routines set up an operations vector to a set of default ops for an interposer. Each ops routine merely passes the operation down to the next layer in the behavior chain. Ops routines corresponding to identity ops just call panic. By using these routines, an interposing layer which needs to affect only a subset of valid VOP's or VFS op's can initialize its ops vector and then modify that ops vector to reflect the interposing routine which it needs to implement. Null interposing routines will remain those set up by the initializing routine.

```
void init_panic_vnops(
      struct vnodeops * ops_to_initialize)
```

```
void init_panic_vfsops(
      struct vfsops * ops_to_initialize)
```

These routines initialize an ops vector with routines that just call panic. This allows a delegated behavior layer to trap ops calls it is supposed to support but for which it has no corresponding implementation. It simply initializes its ops vector using one of these routines and then updates the ops vector to reflect the routines actually implemented.

5.2 Basic Routines to Modify Behavior Chain

These routines are used to modify the behavior chain immediately. The caller must hold the lock (v_lock or m_lock) on the object whose behavior chain is to modified in order to ensure the integrity of the chain.

```
boolean_t pvnode_insert_immed(
      vnode_t * object_to_change,
      pvnode_t * behavior_to_insert,
      boolean_t insert_before,
      pvnode_t * before_or_after_what)
```

```
boolean_t pvfs_insert_immed(
      mount_t * object_to_change,
      pvfs_t * behavior_to_insert,
```

```
    boolean_t insert_before,
    pvfs_t * before_or_after_what)
```

These routines are used to insert a behavior object into the behavior chain for a vnode or mount structure. The identity object whose behavior chain is to be modified is specified by object_to_change. The behavior object to insert is specified by behavior_to_insert. The location in the behavior chain at which the insertion is to be done is specified by insert_before and before_or_after_what. If before_or_after_what is non-NULL, then behavior_to_change is inserted before or after before_or_after_what as indicated by the boolean insert_before. If before_or_after_what is NULL, then the insertion is done at the start of the behavior chain if insert_before is TRUE and at the tail of the chain otherwise.

If there is already an interposer on the behavior chain with the same ops pointer field (pv_op or pvfs_op), the routine returns FALSE and does nothing. This applies whether the interposer is found in the position specified for insertion or in some other position in the chain. When such an interposer is not found, the specified interposer is added to the list and TRUE is returned.

The identity object pointer (pv_vnode or pvfs_mount) is set before doing the insertion. Use of a behavior object embedded inside an identity object (as evidenced by a self-pointing identity object pointer) is invalid and results in a panic.

The insertion happens immediately. It is up to the caller to ensure that the possible existence of ongoing ops poses no problems.

```
void pvnode_delete_immed(
    vnode_t * object_to_change,
    pvnode_t * behavior_to_delete)

void pvfs_delete_immed(
    mount_t * object_to_change,
    pvfs_t * behavior_to_delete)
```

These routines are used to remove an object from the behavior chain. The identity object whose behavior chain is to be modified is specified by object_to_change. The object to delete from the chain is specified by behavior_to_delete.

If the specified object is not on the behavior chain or is embedded in the identity object, a panic results.

The deletion happens immediately. It is the responsibility of the caller to ensure that there can be no ongoing ops while the change is made.

5.3 Synchronized Routines to Modify Behavior Chain

These routines use the synchronization interposer to modify the behavior chain at a time at which there are no ongoing ops. Use of these routines from the implementation of an op will result in a deadlock unless special provisions are made to withdraw the issuing op from the operation count while the routine is called.

Since ongoing ops need to complete, the caller may not hold locks such as v_lock or m_lock which would prevent these ops from completing normally.

It is the caller's responsibility to make sure that the object to be changed does not go away during the pendency or the requested change. This can be done by VREF'ing a vnode or VREF'ing the root vnode for a mount structure.

```
boolean_t pvnode_insert_synch(
      vnode_t * object_to_change,
      pvnode_t * behavior_to_insert,
      boolean_t insert_before,
      pvnode_t * before_or_after_what)

boolean_t pvfs_insert_synch(
      mount_t * object_to_change,
      pvfs_t * behavior_to_insert,
      boolean_t insert_before,
      pvfs_t * before_or_after_what)
```

These routines are used to insert a behavior object into the behavior chain for a vnode or mount structure. Their interfaces are exactly the same as those of pvnode_insert_immed and pvfs_insert_immed as described in section 5.2.

```
void pvnode_delete_synch(
      vnode_t * object_to_change,
      pvnode_t * behavior_to_delete)

void pvfs_delete_synch(
      mount_t * object_to_change,
      pvfs_t * behavior_to_delete)
```

These routines are used to remove an object from the behavior chain. Their interfaces are exactly the same as those for pvnode_delete_immed and pvfs_delete_immed as described in section 5.2

References

[1] Roy, P., Noveck, D., Netterwala, D. The File System Architecture of OSF/1 AD Version 2. *OSF Research Institute Operating Systems Collected Papers*, Volume 2

[2] Skinner, G., Wong, T., "Stacking" Vnodes: A Progress Report., 1993 Summer Usenix

[3] Unix International Stackable Files Work Group, Requirements for Stackable Files

[4] Noveck, D., The Token Module for OSF/1 AD Version 2, *OSF Research Institute Operating Systems Collected Papers*, Volume 3