The Virtual Filesystem Interface in 4.4BSD†

Marshall Kirk McKusick Consultant and Author
Berkeley, California

ABSTRACT: This paper describes the virtual filesystem interface found in 4.4BSD. This interface is designed around an object oriented virtual file node or "vnode" data structure. The vnode structure is described along with its method for dynamically expanding its set of operations. These operations have been divided into two groups: those to manage the hierarchical filesystem name space and those to manage the flat filestore. The translation of pathnames is described, as it requires a tight coupling between the virtual filesystem layer and the underlying filesystems through which the path traverses. This paper describes the filesystem services that are exported from the vnode interface to its clients, both local and remote. It also describes the set of services provided by the vnode layer to its client filesystems. The vnode interface has been generalized to allow multiple filesystems to be stacked together. After describing the stacking functionality, several examples of stacking filesystems are shown.

† To appear in *The Design and Implementation of the 4.4BSD Operating System*, by Marshall Kirk McKusick, et al., ©1995 by Addison-Wesley Publishing Company, Inc. Reprinted with the permission of the publisher.

1. The Virtual Filesystem Interface

In early UNIX systems, the file entries directly referenced the local filesystem inode, see Figure 1 [Leffler et al. 1989]. This approach worked fine when there was a single filesystem implementation. However, with the advent of multiple filesystem types, the architecture had to be generalized. The new architecture had to support import of filesystems from other machines including other machines that were running different operating systems.

One alternative would have been to connect the multiple filesystems into the system as different file types. However, this approach would have required massive restructuring of the internals of the system, since current directories, references to executables, and several other interfaces used inodes instead of file entries as their point of reference. Thus, it was easier and more logical to add a new object oriented layer to the system below the file entry and above the inode as shown in Figure 2. This new layer was first implemented by Sun Microsystems who called it the virtual node or *vnode* layer [Kleiman 1986]. Interfaces in the system that had previously referred to local inodes were changed to reference generic vnodes. A vnode used by a local filesystem would refer to an inode. A vnode used by a remote filesystem would refer to a protocol control block that

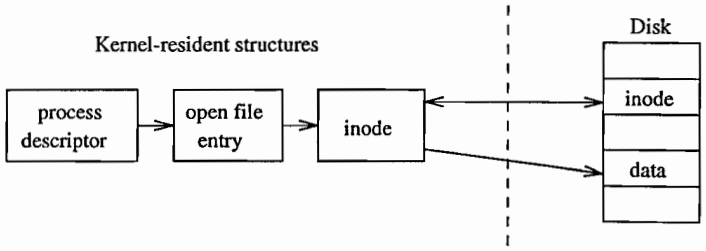

Figure 1. Old layout of kernel tables.

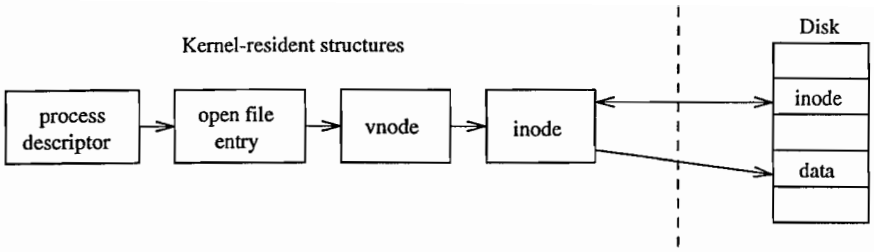

Figure 2. New layout of kernel tables showing a local filesystem.

described the location and naming information necessary to access the remote file.

2. *Contents of a Vnode*

The vnode is an extensible object oriented interface. It contains information that is generically useful independent of the underlying filesystem object that it represents. The information stored in a vnode includes:

- flags for locking the vnode and identifying generic attributes. An example generic attribute is a flag to show that a vnode represents an object that is the root of a filesystem.

- various reference counts. These counts include the number of file entries that reference the vnode for reading and/or writing, the number of file entries that are open for writing that reference the vnode, and the number of pages and buffers that are associated with the vnode.

- a pointer to the mount structure describing the filesystem that contains the object represented by the vnode.

- various information used to do file readahead.

- a reference to state about special devices, sockets, fifos, and mount points.

- a pointer to the set of vnode operations defined for the object. These operations are described in Section 3.

- a pointer to private information needed for the underlying object. For the local filesystem, this pointer will reference inodes; for NFS it will reference an nfsnode.
- the type of the underlying object. The type information is not strictly necessary, since a vnode client could always ask about the type of the underlying object. However, since the type is often needed, the type of underlying objects does not change, and it does take time to call through the vnode interface, the object type is cached in the vnode.
- the list of clean and dirty buffers associated with the vnode. All valid buffers in the system are identified by the vnode and logical block within the object that the vnode represents. All the buffers that have been modified, but not yet written back are stored on the dirty buffer list. All buffers that have not been modified, or have been written back since they were last modified are stored on the clean list. By having all the dirty buffers grouped onto a single list, the cost of doing an *fsync* system call to flush all the dirty blocks associated with a file is proportional to the amount of dirty data. In 4.3BSD the cost was proportional to the smaller of the size of the file or the size of the buffer pool. The list of clean buffers is used when a file is deleted. Since the file will never be read again, the kernel can immediately cancel any pending I/O on its dirty buffers, and reclaim all its clean and dirty buffers and place them at the head of the buffer free list, ready for immediate reuse.
- a count of the number of buffer write operations in progress. To speed the flushing of dirty data, this operation is done by doing asynchronous writes on all the dirty buffers at once. For local filesystems, this simultaneous push causes all the buffers to be put into the disk queue so that they can be sorted into an optimal order to minimize seeking. For remote filesystems, this simultaneous push causes all the data to be presented to the network at once so that it can maximize its throughput. System calls that cannot return until the data is on stable store (such as *fsync*), can sleep on the count of pending output operations waiting for it to reach zero.

The vnode itself is connected into several other structures within the kernel (see Figure 3). Each mounted filesystem within the kernel is represented by a generic mount structure that also includes a specific pointer to a filesystem specific control block. All the vnodes associated with a mount point are linked together on a list headed by the generic mount structure. Thus, when doing a *sync* system call for a filesystem, the kernel can traverse this list to visit all the files active within that filesystem. Also shown in the figure are the lists of clean and dirty

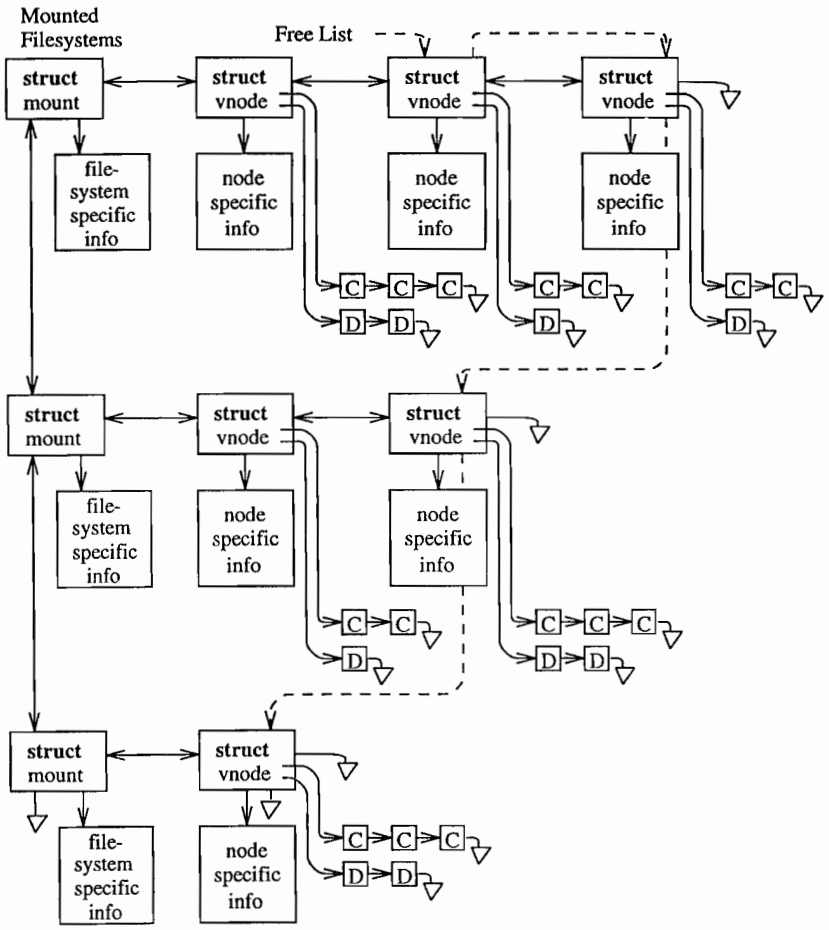

Figure 3. Vnode linkages. D—dirty buffer; C—clean buffer.

buffers associated with each vnode. Finally, there is a free list that links together all the vnodes in the system that are not actively being used. The free list is used when a filesystem needs to allocate a new vnode so that it can open a new file (see Section 6).

3. *Vnode Operations*

Vnodes are designed as an object-oriented interface. Thus, they are manipulated by passing requests to the underlying object through a set of defined operations. Because of the many varied filesystems that are supported in 4.4BSD, the set

of operations defined for vnodes is a large and extensible set. Unlike the original Sun Microsystems vnode implementation, 4.4BSD allows dynamic addition of vnode operations at system boot time. As part of the booting process, each filesystem registers the set of vnode operations that it is able to support. The kernel then builds a table that lists the union of all operations supported by any of the filesystems. From that table it builds an operations vector for each filesystem. Supported operations are filled in with the entry point registered by the filesystem. Filesystems may opt to have unsupported operations filled in with either a default routine (typically a routine to bypass the operation to the next lower layer [see Section 7]) or to return the characteristic error "operation not supported."

In 4.3BSD, the local filesystem code provided both the semantics of the hierarchical filesystem naming and the details of the on-disk storage management. These functions are only loosely related. To enable experimentation with other disk storage techniques without having to reproduce the entire naming semantics, 4.4BSD split the naming and storage code into separate modules. This split is evident at the vnode layer where there are a set of operations defined for hierarchical filesystem operations and a separate set of operations defined for storage of variable sized objects using a flat namespace. About sixty percent of the traditional Berkeley fast filesystem (FFS) became the name space management and the remaining forty percent became the code implementing the 4.4BSD on-disk file storage.

The 4.4BSD kernel provides two different filestore managers: the traditional FFS and a more recent addition, the log-structured filesystem (LFS). The FFS filestore was designed on the assumption that buffer caches would be small and thus that files would need to be read often. It spends great effort placing files likely to be accessed together in the same general location on the disk. The LFS filestore was designed for fast machines with large buffer caches. It assumes that writing data to disk is the bottleneck, and tries to avoid seeking by writing all data together in the order that it was created. It assumes that active files will remain in the buffer cache, so is little concerned with the time that it takes to retrieve files from the filestore [Seltzer et al. 1993].

3.1. Hierarchical Filesystem Management

The vnode operations defined for doing hierarchical filesystem operations are shown in Figure 4. They are derived from the set of operators first defined in the Sun Microsystems vnode implementation and have been augmented by additional operators added by Berkeley.

pathname searching	lookup
name creation	create, mknod, link, symlink, mkdir
name change/deletion	rename, remove, rmdir
attribute manipulation	access, getattr, setattr
object interpretation	open, readdir, readlink, mmap, close
process control	advlock, ioctl, select
object management	lock, unlock, inactive, reclaim, abortop

Figure 4. Hierarchical filesystem operations.

The most complex operator is the one for doing a name lookup; it is described in Section 4.

There are five operators for creating names. The operator to be used depends on the type of object to be created. The *create* operator creates regular files and also is used by the networking code to create AF_LOCAL domain sockets. The *link* operator creates additional names for existing objects. The *symlink* operator creates a symbolic link. The *mknod* operator creates block and character special devices and fifos. The *mkdir* operator creates directories.

There are three operators defined to change or delete existing names. The *rename* operator deletes a name for an object in one location and creates a new name for the object in another location. The implementation of this operator is complex when dealing with the movement of a directory from one part of the filesystem tree to another. The *remove* operator removes a name. If the removed name is the last reference to the object, the space associated with the underlying object is reclaimed. The *remove* operator operates on all object types except directories; they are removed using the *rmdir* operator.

Three operators are supplied for object attributes. The attributes are retrieved from an object using the *getattr* operator; they are stored using the *setattr* operator. Access checks for a given user are provided by the *access* operator.

Five operators are provided for interpreting objects. The *open* and *close* operators have only peripheral use for regular files, but when used on special devices are used to notify the appropriate device driver of device activation or shutdown. The *readdir* operator converts the filesystem specific format of a directory to the standard list of directory entries expected by an application. Note that the interpretation of the contents of a directory are provided by the hierarchical code; the flat filestore code considers a directory as just another object holding data. The *readlink* operator returns the contents of a symbolic link. As with directories, the flat filestore code considers a symbolic link as just another object holding data.

The *mmap* operator prepares an object to be mapped into the address space of a process.

Three operators are provided to allow process control over objects. The *select* operator allows a process to find out if an object is ready to be read or written. The *ioctl* operator passes control requests to a special device. The *advlock* operator allows a process to acquire or release an advisory lock on an object. None of these operators operates on the representation of the object in the filestore. They are simply using the object for naming or directing the desired operation.

There are five operations for management of the objects. The *inactive* and *reclaim* operators are discussed in Section 6. The *lock* and *unlock* operators allow the vnode clients to provide hints to the code implementing operations on the underlying objects. Stateless filesystem such as NFS ignore these hints. However, stateful filesystems can use them to avoid doing extra work. For example, an *open* system call requesting that a new file be created requires two steps. First a *lookup* call is done to see if the file already exists. Before starting the lookup, a *lock* request is done on the directory being searched. While scanning through the directory checking for the name, the lookup code also identifies a location within the directory that contains enough space to hold the new name. If the lookup returns successfully (meaning that the name does not already exist), the *open* code verifies that the user has permission to create the file. If the user is not eligible to create the new file, then the *abortop* operator is called to release any resources held in reserve. Otherwise the *create* operation is called. If the filesystem is stateful and has been able to lock the directory, then it can simply create the name in the previously identified space since it knows that no other processes will have had access to the directory. Once the name is created, an *unlock* request is made on the directory. If the filesystem is stateless, then it cannot lock the directory, so the *create* operator must rescan the directory to find space and to verify that the name has not shown since the lookup.

3.2. Filestore Management

The vnode operations defined for doing the datastore filesystem operations are shown in Figure 5. There are fewer of these operators and they are semantically simpler than those used for managing the namespace.

There are two operators for allocating and freeing objects. The *valloc* operator creates a new object. The identity of the object is a number returned by the operator. The mapping of this number to a name is the responsibility of the namespace code. An object is freed by the *vfree* operator. The object to be freed is identified only by its number.

object creation/deletion	valloc, vfree
attribute update	update
object read/write	vget, blkatoff, read, write, fsync
change in space allocation	truncate

Figure 5. Datastore filesystem operations.

The attributes of an object are changed by the *update* operator. This layer does no interpretation of these attributes, they are simply fixed-size auxiliary data stored outside the main data area of the object. They are typically used to store file attributes such as the owner, group, permissions, etc.

There are five operators for manipulating existing objects. The *vget* operator retrieves an existing object from the filestore. The object is identified by its number, and must have been previously created by *valloc*. The *read* operator copies data from an object to a user specified location. The *blkatoff* operator is similar to the *read* operator except that it simply returns a pointer to a kernel memory buffer with the requested data instead of copying it. This operator is designed to increase the efficiency of operations where the namespace code is going to interpret the contents of an object (i.e., directories) instead of just returning it to a user process. The *write* operator copies data from a user specified location to an object. The *fsync* operator requests that all data associated with the object be moved to stable storage (usually by writing it all to disk).

The final datastore operation is *truncate*. This operation changes the amount of space associated with an object. Despite its name, it may be used to both increase and decrease the size of an object.

4. Pathname Translation

The translation of a pathname requires a series of interactions between the vnode interface and the underlying filesystems. The pathname translation process proceeds as follows:

1. The pathname to be translated is copied in from the user process or, for a remote filesystem request, is extracted from the network buffer.
2. The starting point of the pathname is determined as either the root directory or the current directory. The vnode for the appropriate directory becomes the *lookup directory* used in the next step.

3. The vnode layer calls the filesystem specific *lookup* operation passing it the remaining components of the pathname and the current *lookup directory*. Typically, the underlying filesystem will search the *lookup directory* for the next component of the pathname and return the resulting vnode (or an error if the name does not exist).
4. If an error is returned, the top level returns the error. If the pathname has been exhausted, the pathname lookup is done and the returned vnode is the result of the lookup. If the pathname has not been exhausted, and the returned vnode is not a directory, then the vnode layer returns the "not a directory" error. If there are no errors, the top layer checks to see if the returned directory has another filesystem mounted on top of it. If it does, then the *lookup directory* becomes the mounted filesystem, otherwise the *lookup directory* becomes the vnode returned by the lower layer. The lookup then iterates with step 3.

While it may seem a bit inefficient to call through the vnode interface for each pathname component, it is usually necessary to do so. The reason is that the underlying filesystem does not know which directories are being used as mount points. Since a mount point will redirect the lookup to a new filesystem, it is important that the current filesystem not proceed past a mounted directory. While it might be possible for a local filesystem to be knowledgeable about which directories are mount points, it is nearly impossible for the server of an exported filesystem to know the mount points being used within that filesystem by all its clients. Consequently, the conservative approach of traversing only a single pathname component per *lookup* call is used. There are a few instances where a filesystem will know that there are no further mount points in the remaining path, and will traverse the rest of the pathname. An example is crossing into a *portal*, described in Section 7.3.

5. *Exported Filesystem Services*

The vnode interface provides a set of services that are made available from all the filesystems supported under the interface. The first of these is the ability to support the update of generic mount options. These options include:

noexec do not execute any files on the filesystem. This option is often used when a server exports binaries for a different architecture that cannot be executed on the server itself. The kernel will even refuse to

execute shell scripts; to run a shell script, its interpreter must be invoked explicitly.

nosuid do not honor the set-user-id or set-group-id flags for any executables on the filesystem. This option is useful when mounting a filesystem of unknown origin.

nodev do not allow any special devices on the filesystem to be opened. This option is often used when a server exports device directories for a different architecture. The values of the major/minor numbers are nonsensical on the server.

Together, these options allow reasonably secure mounting of untrusted or foreign filesystems. It is not necessary to unmount and remount the filesystem to change these flags; they may be changed while a filesystem is mounted. Additionally a filesystem that is mounted read-only can be upgraded to allow writing. Conversely, a filesystem that allows writing may be downgraded to read-only provided no files are open for modification. The filesystem may be forcibly downgraded to read-only by requesting that any files open for writing have their access revoked.

Another service exported from the vnode interface is the ability to get information about a mounted filesystem. The *statfs* system call returns a buffer that gives the number of used and free disk blocks and inodes along with the filesystem mount point, and the device, location, or program from which the filesystem is mounted. The *getfsstat* system call returns information about all the mounted filesystems. This interface avoids the need to track the set of mounted filesystems outside the kernel as is done in many other UNIX variants.

6. *Filesystem Independent Services*

The vnode interface not only supplies an object oriented interface to the underlying filesystems, but also provides a set of management routines that can be used by the client filesystems. These facilities are described in this section.

When the last file entry reference to a file is closed, the usage count on the vnode drops to zero and the vnode interface calls the *inactive* vnode operation. The *inactive* call notifies the underlying filesystem that the file is no longer being used. The filesystem is permitted to keep the file on its hash chains so that it can be reactivated quickly (i.e., without doing disk or network I/O) if the file is reopened.

In addition to calling the *inactive* vnode operation when the reference count drops to zero, the vnode is placed on a system wide free list. Unlike most vendors' implementation of vnodes that have a fixed number of vnodes allocated

to each filesystem type, the 4.4BSD kernel keeps a single system-wide collection of vnodes. When an application opens a file that does not currently have an in-memory vnode, the client filesystem calls the *getnewvnode()* routine to allocate a new one. The *getnewvnode()* routine removes the least recently used vnode from the front of the free list, and calls the *reclaim* operation to notify the filesystem currently using the vnode that it is about to be reused. The *reclaim* operation writes back any dirty data associated with the underlying object, removes the underlying object from any lists that it is on (such as hash lists used to look it up), and frees up any auxiliary storage that was being used by the object. The vnode is then returned for use by the new client filesystem.

The benefit of having a single global vnode table is that the kernel memory dedicated to vnodes is used more efficiently. Consider a system that is willing to dedicate memory for 1000 vnodes. If the system supports ten filesystem types, then each filesystem type will get 100 vnodes. If most of the activity moves to a single filesystem (e.g., during the compilation of a kernel located in a local filesystem), all the active files will have to be kept in the 100 vnodes dedicated to that filesystem while the other 900 vnodes sit idle. In a 4.4BSD system, all 1000 vnodes could be used for the active filesystem, allowing a much bigger set of files to be cached in memory. If the center of activity moved to another filesystem (e.g., compiling a program on an NFS mounted filesystem), the vnodes would migrate from the previously active local filesystem over to the NFS filesystem. Here too, there would be a much larger set of cached files than if only 100 vnodes were available using a partitioned set of vnodes.

The *reclaim* operation is really a disassociation of the underlying filesystem object from the vnode itself. This ability combined with the ability to associate new objects with the vnode provides functionality with usefulness that goes far beyond simply allowing vnodes to be moved from one filesystem to another. By replacing an existing object with an object from the dead filesystem—a filesystem in which all operations except *close* fail—the kernel can provide a revocation of the object. Internally this revocation of an object is provided by the *vgone()* routine.

The revocation service is used for session management to revoke all references to the controlling terminal when the session leader exits. This revocation works as follows. All open descriptors within the session reference the vnode for the special device representing the session terminal. When *vgone()* is called on this vnode, the underlying special device is detached from the vnode and replaced with the dead filesystem. Any further operations on the vnode will result in errors since the open descriptors no longer reference the terminal. Eventually, all the processes will exit and close their descriptors causing the reference count to drop to zero. The *inactive* routine for the dead filesystem, recognizing that it will never be

possible to get a reference to the vnode again, returns the vnode to the front of the free list for immediate reuse.

The revocation service is also used to allow forcible unmounting of filesystems. If an active vnode is found when attempting to unmount a filesystem, the kernel simply calls the *vgone()* routine to disassociate it from the filesystem object. Processes with open files or current directories within the filesystem find that they have simply vanished, as if someone had done a remove operation on them. It is also possible to downgrade a mounted filesystem from read-write to read-only. Instead of revoking access on every active file within the filesystem, only those files with a non-zero number of references for writing are revoked.

Finally, the ability to revoke objects is exported to processes through the *revoke* system call. This system call can be used to ensure controlled access to a device such as a pseudo-terminal port. First the ownership of the device is changed to the desired user, then the device name is revoked to eliminate any interlopers that already had it open. Once revoked, only the new owner of the device is able to open it.

6.1. The Name Cache

Name cache management is another service that is provided by the vnode management routines. The interface provides a facility to add a name and its corresponding vnode, lookup a name to get the corresponding vnode, and to delete a specific name from the cache. In addition to providing a facility for deleting specific names, the interface also provides an efficient way to invalidate all names that reference a specific vnode. Directory vnodes can have many names that reference them, notably the .. entries in all their immediate descendents. The revocation of all names for a vnode could be done by scanning the entire name table looking for references to the vnode in question. This approach would be slow given that the name table may store thousands of names. Instead, each vnode is given a *capability*—a 32-bit number guaranteed to be unique. When all the numbers have been exhausted, all outstanding capabilities are purged, and numbering restarts from scratch. Purging is possible, as all capabilities are easily found in kernel memory and only needs to be done if the machine remains running for nearly a year. When an entry is made in the name table, the current value of the vnode's capability is copied to the associated name entry. A vnode's capability is invalidated each time it is reused by *getnewvnode()* or when specifically requested by a client (e.g., when a file is being renamed) by assigning a new capability to the vnode. When a name is found during a cached lookup, the capability assigned to the name is compared with that of the vnode. If they match, the lookup is successful; if they do not match, the cache entry is freed and failure is returned.

The cache management routines also allow for negative cacheing. If a name is looked up in a directory and is not found, that name can be entered in the cache along with a null pointer for its corresponding vnode. If the name is later looked up, it will be found in the name table, and thus the kernel can avoid scanning the entire directory to determine that the name is not there. If a directory is modified, then potentially one or more of the negative entries may be wrong. So when the directory is modified, all the negative names for that directory must be invalidated by assigning the directory vnode a new capability. Negative cacheing provides a significant improvement because of path searching in command shells. When executing a command, many shells will look at each path in turn looking for the executable. Commonly run executables will be searched for repeatedly in directories in which they do not exist. Negative cacheing speeds these searches.

An obscure but tricky issue has to do with detecting and properly handling special device aliases. Special devices and fifos are hybrid objects. Their naming and attributes (such as owner, time stamps, and permissions) are maintained by the filesystem in which they reside. However, their operations (such as read and write) are maintained by the kernel on which they are being used. Since a special device is identified solely by its major and minor number, it is possible for two or more instances of the same device to appear within the filesystem namespace, possibly in different filesystems. Each of these different names has its own vnode and underlying object, yet all these vnodes must be treated as one from the perspective of identifying blocks in the buffer cache and in other places where the vnode and logical block number are used as a key. To ensure that the set of vnodes are treated as a single vnode, the vnode layer provides a routine *checkalias()* that is called each time a new special device vnode comes into existence. This routine looks for other instances of the device, and if found links them together so that they can be treated as one.

7. Stackable Filesystems

The early vnode interface was simply an object-oriented interface to an underlying filesystem. As the demand grew for new filesystem features, it became desirable to find ways of providing them without having to modify the existing and stable filesystem code. One approach is to provide a mechanism for stacking several filesystems on top of each other [Rosenthal 1990]. The stacking ideas were refined and implemented in the 4.4BSD system [Heidemann and Popek 1994]. The bottom of a vnode stack tends to be a disk-based filesystem, while the layers used above it are typically things that transform their arguments and pass them on to a lower layer.

In all UNIX systems, the *mount* command causes a disk-based filesystem to take a special device as a source and map it into a mount point in the existing filesystem. When a filesystem is mounted on a directory, the previous contents of the directory are hidden; only the contents of the root of the newly mounted filesystem are visible. To most users, the effect of the series of mount commands done at system startup is the creation of a single seamless filesystem tree.

Stacking also uses the *mount* command to create new layers. The *mount* command pushes a new layer onto a vnode stack; an *unmount* command removes a layer. Like the mounting of a filesystem, a vnode stack is visible to all processes running on the system. The *mount* command identifies the underlying layer in the stack, creates the new layer, and attaches it into the filesystem namespace. The new layer can be attached to the same place as the old layer (covering it up), or to a different place in the tree (allowing both layers to be visible). An example is shown in the next section.

If layers are attached to different places in the namespace, then the same file will be visible in multiple places. Access to the file under the name of the new layer's namespace will go to the new layer, while access under the old layer's namespace will go only to the old layer.

When a file access (like open, read, stat, or close) occurs to a vnode in the stack, that vnode has several options:

- Do the requested operations and return a result.
- Pass the operation without change to the next lower vnode on the stack. When the operation returns from the lower vnode, it may further interpret the result, or simply return the result it received.
- Modify the operands provided with the request, then pass it to the next lower vnode. When the operation returns from the lower vnode, it may further interpret the result, or simply return the result it received.

If an operation is passed to the bottom of the stack without any layer taking action on it, then the interface will return the error "operation not supported."

Vnode interfaces released before 4.4BSD implemented vnode operations as indirect function calls. The requirements that intermediate stack layers bypass operations to lower layers and that new operations can be added into the system at boot time means that this approach no longer works. Filesystems must be able to bypass operations that may not have been defined at the time that the filesystem was implemented. In addition to passing through the function, it must also pass through the parameters of the function which are of unknown type and number.

To resolve these two problems in a clean and portable way, the arguments to a vnode operation and the operation name are placed into an argument structure. This argument structure is then passed as a single parameter to the vnode operation. Thus, all calls on a vnode operation will always have exactly one parameter which is the pointer to its argument structure. If the vnode operation is one that is supported by the filesystem, then it will know what the arguments are and how to interpret them. If it is an unknown vnode operation, then the generic bypass routine can call the same operation in the next lower layer passing it the same argument structure that it received. In addition, the first argument of every operation is a pointer to the vnode operation description. This description provides a bypass routine information about the operation, including its name and the location of its parameters. An example access check call and its implementation for the UFS filesystem are shown in Figure 6. Note that the vop_*access*_args structure is normally declared in a header file, but is declared at the function site to simplify the example.

7.1. Simple Filesystem Layers

The simplest filesystem layer is *nullfs*. It does no transformations on its arguments, simply passing through all requests that it receives and returning all results that it gets back. While it provides no useful functionality if it is simply stacked on top of an existing vnode, it can be used to provide a loopback filesystem by mounting its source vnode at some other location in the filesystem tree than its source. The code for *nullfs* is also an excellent starting point for designers that want to build their own filesystem layers. Examples that could be built include a compression layer or an encryption layer.

A sample vnode stack is shown in Figure 7. It shows a local filesystem on the bottom of the stack that is being exported from /**local** via an NFS layer. Clients within the administrative domain of the server can directly import the /**local** filesystem since they are all presumed to use a common mapping of user-ids to user names.

The *umapfs* filesystem works much like the *nullfs* filesystem in that it provides a view of the file tree rooted at the /**local** filesystem on the /**export** mount point. In addition to providing a copy of the /**local** filesystem at the /**export** mount point, it transforms the credentials of each system call made to files within the /**export** filesystem. The transformation is done using a mapping that was provided as part of the *mount* system call that created the *umapfs* layer.

The /**export** filesystem can be exported to clients from an outside administrative domain that uses different user-ids and group-ids. When an NFS request comes in for the /**export** filesystem, the *umapfs* layer modifies the credential from

```
{
    ...
    /*
     * Check for read permission on file ``vp''.
     */
    if (error = VOP_ACCESS(vp, VREAD, cred, p))
        return (error);
    ...
}

/*
 * Check access permission for a file.
 */
int
ufs_access(ap)
    struct vop_access_args {
       struct vnodeop_desc *a_desc;    /* operation description */
       struct vnode *a_vp;             /* file to be checked */
       int a_mode;                     /* access mode sought */
       struct ucred *a_cred;           /* user asking for permission */
       struct proc *a_p;               /* associated process, if any */
    } *ap;
{

    if (permission granted)
        return (1);
    return (0);

}
```

Figure 6. Call to and function header for "access" vnode operation.

the foreign client by mapping the ids used on the foreign client to the corresponding ids used on the local system. The requested operation with the modified credential is passed down to the lower layer corresponding to the /**local** filesystem, where it is processed identically to a local request. When the result is returned to the mapping layer, any returned credentials are inversely mapped to convert them from the local ids to the outside ids, and this result is sent back as the NFS response.

There are three benefits to this approach:

1. There is no computational cost of mapping imposed on the local clients.
2. There are no changes required to the local filesystem code or the NFS code to support mapping.

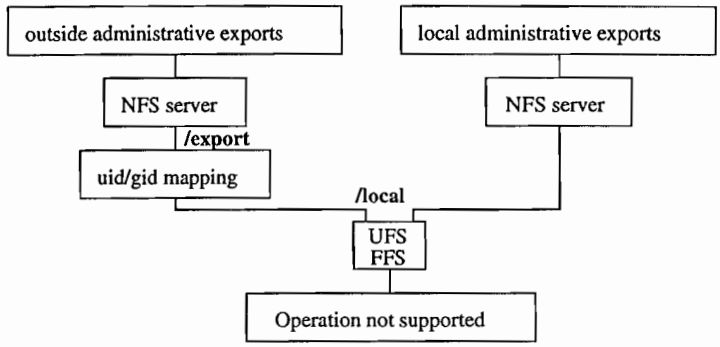

Figure 7. Stackable Vnodes.

3. Each outside domain can have its own mapping. Domains with simple mappings consume small amounts of memory and run quickly; domains with large and complex mappings can be supported without impacting the performance of the simpler environments.

Vnode stacking is an effective approach to add filing extensions such as the *umapfs* service.

7.2. The Union Mount Filesystem

The *union* filesystem is another example of a middle filesystem layer. Like the *nullfs* it does not store data, it just provides a name space transformation. It is loosely modelled on the work on the 3-D filesystem [Korn and Krell 1989], the work on the Translucent filesystem [Hendricks 1990], and the work on the Automounter [Pendry and Williams 1994]. The *union* filesystem takes an existing filesystem and transparently overlays it on another filesystem. Unlike most other filesystems, a union mount does not cover up the directory on which it is mounted. Instead, it shows the logical merger of both directories and allows both directory trees to be simultaneously accessible.

A small example of a union mount stack is shown in Figure 8. Here, the bottom layer of the stack is the **src** filesystem that includes the source for the **shell** program. Being a simple program, it contains only one source and one header file. The upper layer that has been union mounted on top of **src** initially contains just the **src** directory. When the user changes directory into **shell**, a directory of the same name is created in the top layer. Directories in the top layer that correspond to directories in the lower layer are only created as they are encountered while traversing around the top layer. If the user were to run a recursive traversal of the tree rooted at the top of the union mount location, the result would be a complete

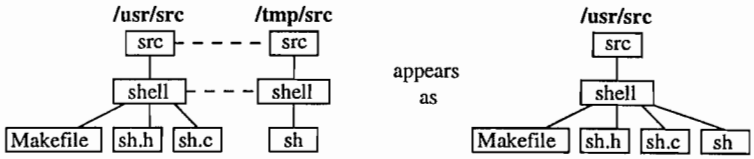

Figure 8. A union mounted filesystem.

tree of directories matching the underlying filesystem. In our example, the user now types *make* in the **shell** directory. The **sh** executable is created in the upper layer of the union stack. To the user, a directory listing shows the sources and executable all apparently together as shown on the right of Figure 8.

All filesystem layers except the top one are treated as if they were read-only. If a file residing in a lower layer is opened for reading, a descriptor is returned for that file. If a file residing in a lower layer is opened for writing, the kernel first copies the file to the top layer, then returns a descriptor referencing the copy of the file. The result is that there are two copies of the file: the original unmodified copy in the lower layer, and the modified copy of the file in the upper layer. When the user does a directory listing, any duplicate names in the lower layer are suppressed. When a file is opened, a descriptor for the file in the uppermost layer in which the name appears is returned. Thus, once a file has been copied to the top layer, instances of the file in lower layers become inaccessible.

The tricky part of the *union* filesystem is handling the removal of files that reside in a lower layer. Since the lower layers may not be modified, the only way to remove a file is to hide it by creating a *whiteout* directory entry in the top layer. A whiteout is an entry in a directory that has no corresponding file; it is distinguished by having an inode number of one. If a whiteout is found while searching for a name, the lookup is stopped and the "no such file or directory" error is returned. Thus, the file with the same name in a lower layer appears to have been removed. If a file is removed from the top layer, it is only necessary to create a whiteout entry for it if there is a file with the same name in the lower level that would reappear.

When a process creates a file with the same name as a whiteout entry, the whiteout entry is replaced with a regular name that references the new file. Since the new file is being created in the top layer, it will mask out any files with the same name in a lower layer. When doing a directory listing, whiteout entries and the files that they mask are usually not shown. However, there is an option that causes them to appear.

One feature that has long been missing in UNIX systems is the ability to recover files after they have been deleted. For the union filesystem, file recovery can be trivially implemented simply by removing the whiteout entry to expose

the underlying file. The LFS filesystem also has the ability to recover deleted files since it never overwrites previously written data. Deleted versions of files are not reclaimed until the filesystem becomes nearly full and the LFS garbage collector runs. Files can be recovered (for filesystems that provide file recovery) using a special option to the remove command or by using the *undelete* system call.

When removing a directory whose name appears in a lower layer, a whiteout entry is created just as it would be for a file. However, if the user later attempts to create a directory with the same name as the previously deleted directory, the union filesystem must treat it specially to avoid having the previous contents from the lower layer directory reappear. When creating a directory that replaces a whiteout entry, the union filesystem sets a flag in the directory metadata to show that this directory should be treated specially. When a directory scan is done, the kernel returns information about only the top level directory; it suppresses the list of files from the directories of the same name in the lower layers.

The *union* filesystem can be used for many purposes:

- Allowing several different architectures to build from a common source base. The source pool is NFS mounted onto each of several machines. On each host machine a local filesystem is union mounted on top of the imported source tree. As the build proceeds, the binaries appear in the local filesystem that is layered above the source tree. This not only avoids contaminating the source pool with different binaries, but also speeds the compilation since most of the filesystem traffic is being done on the local filesystem.

- Allows compilation of sources on read-only media such as CD-ROMs. A local filesystem is union mounted above the CD-ROM sources. It is then possible to change into directories on the CD-ROM and have the appearance of being able to edit and compile in that directory.

- Creation of a private source directory. The user creates a source directory in their own work area, then union mounts the system sources underneath it. This feature is possible because the restrictions on the *mount* command have been relaxed. Any user can do a mount if they own the directory on which the mount is being done and they have appropriate access permissions on the device or directory being mounted (read permission is required for a read-only mount, read-write permission is required for a read-write mount). Only the user that did the mount or the superuser can unmount a filesystem.

Additional information about the union filesystem is available in the Winter 1995 Usenix Proceedings [Pendry and McKusick 1995].

7.3. Other Filesystems

There are several other filesystems included as part of 4.4BSD. The *portal* filesystem mounts a process onto a directory in the file tree. When using a pathname that traverses the location of the portal, the remainder of the path is passed to the process mounted at that point. The process interprets the path in whatever way it sees fit, then returns a descriptor to the calling process. This descriptor may be for a socket connected to the portal process. If so, further operations on the descriptor will be passed to the portal process for it to interpret. Alternatively, the descriptor may be for a file elsewhere in the filesystem.

Consider a portal process mounted on **/dialout** used to manage a bank of dialout modems. When a process wanted to connect to an outside number, it would open **/dialout/15105551212/9600** to specify that it wanted to dial 1-510-555-1212 at 9600 baud. The portal process would get the final two pathname components. Using the last component it would determine that it should find an unused 9600 baud modem. It would use the other component as the number to which to place the call. It would then write an accounting record for future billing, and return the descriptor for the modem to the process.

There are several filesystems that are designed to provide a convenient interface to kernel information. The *procfs* filesystem is normally mounted at **/proc** and provides a view of the running processes in the system. Its primary use is for debugging, but it also provides a convenient interface for collecting information about the processes in the system. A directory listing of **/proc** produces a numeric list of all the processes in the system. Each process entry is itself a directory that contains:

ctl	a file to control the process allowing it to be stopped, continued, and signalled.
file	the executable for the process.
mem	the virtual memory of the process.
regs	the registers for the process.
status	a text file containing information about the process.

The *fdesc* filesystem is normally mounted on **/dev/fd** and provides a list of all the active file descriptors for the currently running process. An example where this is useful is specifying to an application that it should read input from its standard input. Here you can use the pathname **/dev/fd/0** instead of having to come up with a special convention like using the name – to tell the application to read from its standard input.

The *kernfs* filesystem is normally mounted on /**kern** and contains files that have various information about the system. It includes things like the hostname, time of day, version of the system, etc.

Finally there is the *cd9660* filesystem. It allows ISO-9660 compliant filesystems, with or without Rock Ridge extensions, to be mounted. The ISO-9660 filesystem format is most commonly used on CD-ROMs.

References

1. J. S. Heidemann, G. J. Popek, File-System Development with Stackable Layers, *ACM Transactions on Computer Systems* 12(1)(February 1994), 58–89.
2. D. Hendricks, A Filesystem for Software Development, *USENIX Association Conference Proceedings*, June 1990, 333–340.
3. S. R. Kleiman, Vnodes: An Architecture for Multiple File System Types in Sun UNIX, *USENIX Association Conference Proceedings*, June 1986, 238–247.
4. D. Korn, E. Krell, The 3-D File System, *USENIX Association Conference Proceedings*, June 1989, 147–156.
5. S. Leffler, M. McKusick, M. Karels, J. Quarterman, in *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, Reading, MA, January 1989, 205, ISBN 0-201-06196-1.
6. J. Pendry, M. McKusick, Union mounts in 4.4BSD-Lite, *USENIX Association Conference Proceedings*, January 1995.
7. J. Pendry, N. Williams, AMD - The 4.4BSD Automounter Reference Manual, in *4.4BSD System Manager's Manual*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994, 13:1–57.
8. D. Rosenthal, Evolving the Vnode Interface, *USENIX Association Conference Proceedings*, June 1990, 107–118.
9. M. Seltzer, K. Bostic, M. K. McKusick, C. Staelin, An Implementation of a Log-Structured File System for UNIX, *USENIX Association Conference Proceedings*, January 1993, 307–326.