# Vnodes: An Architecture for Multiple File System Types in Sun UNIX

*S.R. Kleiman*

Sun Microsystems
sun!srk

## 1. Introduction

This paper describes an architecture for accommodating multiple file system implementations within the Sun UNIX† kernel. The file system implementations can encompass local, remote, or even non-UNIX file systems. These file systems can be "plugged" into the kernel through a well defined interface, much the same way as UNIX device drivers are currently added to the kernel.

## 2. Design Goals

- Split the file system implementation independent and the file system implementation dependent functionality of the kernel and provide a well defined interface between the two parts.
- The interface must support (but not require) UNIX file system access semantics. In particular it must support local disk file systems such as the 4.2BSD file system[1], stateless remote file systems such as Sun's NFS[2], statefull remote file systems such as AT&T's RFS, or non-UNIX file systems such as the MSDOS file system[3].
- The interface must be usable by the server side of a remote file system to satisfy client requests.
- All file system operations should be atomic. In other words, the set of interface operations should be at a high enough level so that there is no need for locking (hard locking, not user advisory locking) across several operations. Locking, if required, should be left up to the file system implementation dependent layer. For example, if a relatively slow computer running a remote file system requires a supercomputer server to lock a file while it does several operations, the users of the supercomputer would be noticeably affected. It is much better to give the file system dependent code full information about what operation is being done and let it decide what locking is necessary and practical.

## 3. Implementation goals and techniques

These implementation goals were necessary in order to make future implementation easier as the kernel evolved.

- There should be little or no performance degradation.
- The file system independent layer should not force static table sizes. Most of the new file system types use a dynamic storage allocator to create and destroy objects.
- Different file system implementations should not be forced to use centralized resources (e.g inode table, mount table or buffer cache). However, sharing should be allowed.
- The interface should be reentrant. In other words, there should be no implicit references to global data (e.g. `u.u_base`) or any global side effect information passed between operations (e.g. `u.u_dent`). This has the added benefit of cutting down the size of the per user global data area (u area). In addition, all the interface operations return error codes as the return value. Overloaded return codes and u.u_error should not be used.
- The changes to the kernel should be implemented by an "object oriented" programming approach. Data structures representing objects contain a pointer to a vector of generic operations on the object. Implementations of the object fill in the vector as appropriate. The complete interface to the object is

† UNIX is a trademark of AT&T

specified by its data structure and its generic operations. The object data structures also contain a pointer to implementation specific data. This allows implementation specific information to be hidden from the interface.

- Each interface operation is done on behalf of the current process. It is permissible for any interface operation to put the current process to sleep in the course of performing its function.

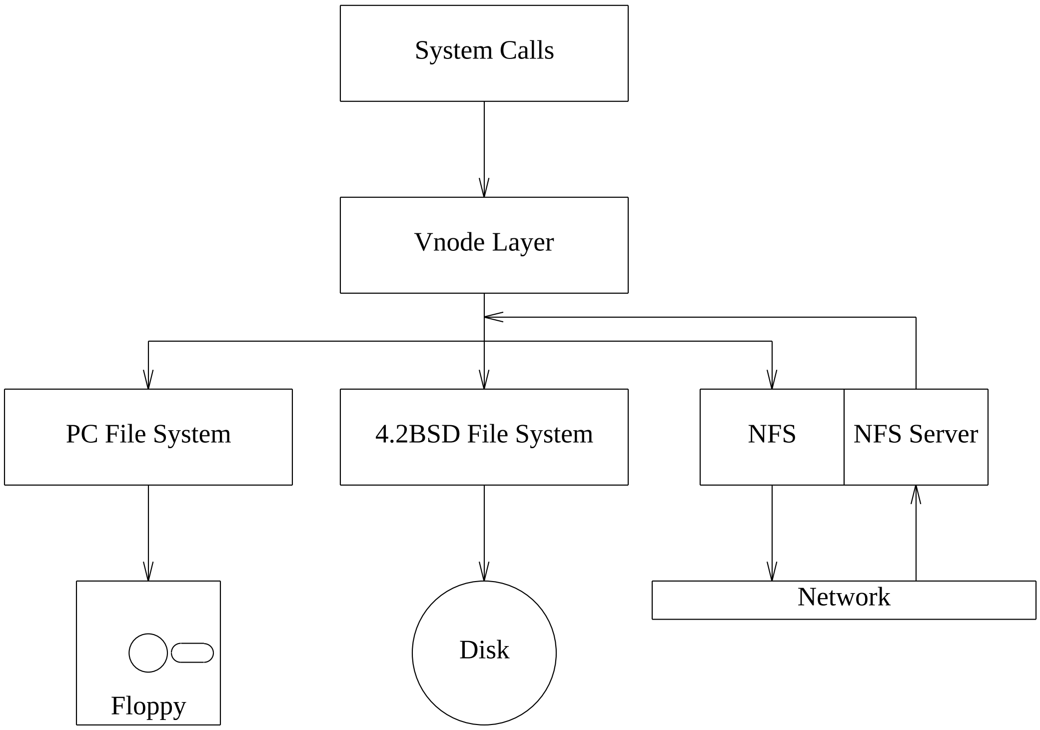


**Figure 1.** Vnode architecture block diagram

**4. Operation**

The file system dependent/independent split was done just above the UNIX kernel inode layer. This was an obvious choice, as the inode was the main object for file manipulation in the kernel. A block diagram of the architecture is shown in Figure 1. The file system independent inode was renamed *vnode* (virtual node). All file manipulation is done with a vnode object. Similarly, file systems are manipulated through an object called a *vfs* (virtual file system). The *vfs* is the analog to the old mount table entry. The file system independent layer is generally referred to a the *vnode layer*. The file system implementation dependent layer is called by the file system type it implements (e.g. 4.2BSD file system, NFS file system). Figure 2 shows the definition of the vnode and vfs objects.

**4.1 Vfs's**

Each mounted vfs is linked into a list of mounted file systems. The first file system on the list is always the root. The private data pointer (`vfs_data`) in the vfs points to file system dependent data. In the 4.2BSD file system, `vfs_data` points to a mount table entry. The public data in the vfs structure contains data used by the vnode layer or data about the mounted file system that does not change.

Since different file system implementations require different mount data, the *mount*(2) system call was changed. The arguments to *mount*(2) now specify the file system type, the directory which is the mount point, generic flags (e.g. read only), and a pointer to file system type specific data. When a mount system

```
struct vfs {
        struct vfs      *vfs_next;              /* next vfs in list */
        struct vfsops   *vfs_op;                /* operations on vfs */
        struct vnode    *vfs_vnodecovered;      /* vnode we cover */
        int             vfs_flag;               /* flags */
        int             vfs_bsize;              /* native block size */
        caddr_t         vfs_data;               /* private data */
};
struct vfsops {
        int     (*vfs_mount)();
        int     (*vfs_unmount)();
        int     (*vfs_root)();
        int     (*vfs_statfs)();
        int     (*vfs_sync)();
        int     (*vfs_fid)();
        int     (*vfs_vget)();
};

enum vtype      { VNON, VREG, VDIR, VBLK, VCHR, VLNK, VSOCK, VBAD };
struct vnode {
        u_short         v_flag;                 /* vnode flags */
        u_short         v_count;                /* reference count */
        u_short         v_shlockc;              /* # of shared locks */
        u_short         v_exlockc;              /* # of exclusive locks */
        struct vfs      *v_vfsmountedhere;      /* covering vfs */
        struct vnodeops *v_op;                  /* vnode operations */
        union {
                struct socket   *v_Socket;      /* unix ipc */
                struct stdata   *v_Stream;      /* stream */
        };
        struct vfs      *v_vfsp;                /* vfs we are in */
        enum vtype      v_type;                 /* vnode type */
        caddr_t         v_data;                 /* private data */
};
struct vnodeops {
        int     (*vn_open)();
        int     (*vn_close)();
        int     (*vn_rdwr)();
        int     (*vn_ioctl)();
        int     (*vn_select)();
        int     (*vn_getattr)();
        int     (*vn_setattr)();
        int     (*vn_access)();
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_bmap)();
        int     (*vn_strategy)();
        int     (*vn_bread)();
        int     (*vn_brelse)();
};
```

**Figure 2.** *Vfs* and *vnode* objects

call is performed, the vnode for the mount point is looked up (see below) and the `vfs_mount` operation for the file system type is called. If this succeeds, the file system is linked into the list of mounted file systems, and the `vfs_vnodecovered` field is set to point to the vnode for the mount point. This field is null in the root vfs. The root vfs is always first in the list of mounted file systems.

Once mounted, file systems are named by the path name of their mount points. Special device name are no longer used because remote file systems do not necessarily have a unique local device associated with them. *Umount*(2) was changed to *unmount*(2) which takes a path name for a file system mount point instead of a device.

The root vnode for a mounted file system is obtained by the `vfs_root` operation, as opposed to always referencing the root vnode in the vfs structure. This allows the root vnode to be deallocated if the file system is not being referenced. For example, remote mount points can exist in "embryonic" form, which contains just enough information to actually contact the server and complete the remote mount when the file system is referenced. These mount points can exist with minimal allocated resources when they are not being used.

**4.2 Vnodes**

The public data fields in each vnode either contain data that is manipulated only by the vfs layer or data about the file that does not change over the life of the file, such as the file type (`v_type`). Each vnode contains a reference count (`v_count`) which is maintained by the generic vnode macros `VN_HOLD` and `VN_RELE`. The vnode layer and file systems call these macros when vnode pointers are copied or destroyed. When the last reference to a vnode is destroyed, the `vn_inactive` operation is called to tell the vnode's file system that there are no more references. The file system may then destroy the vnode or cache it for later use. The `v_vfsp` field in the vnode points to the vfs for the file system to which the vnode belongs. If a vnode is a mount point, the `v_vfsmountedhere` field points to the vfs for another file system. The private data pointer (`v_data`) in the vnode points to data that is dependent on the file system. In the 4.2BSD file system `v_data` points to an in core inode table entry.

Vnodes are not locked by the vnode layer. All hard locking (i.e. not user advisory locks) is done within the file system dependent layer. Locking could have been done in the vnode layer for synchronization purposes without violating the design goal; however, it was found to be not necessary.

**4.3 An example**

Figure 3 shows an example vnode and vfs object interconnection. In figure 3, `vnode1` is a file or directory in a 4.2BSD type file system. As such, it's private data pointer points to an inode in the 4.2BSD file system's inode table. `Vnode1` belongs to `vfs1`, which is the root vfs, since it is the first on the vfs list (`rootvfs`). `Vfs1`'s private data pointer points to a mount table entry in the 4.2BSD file system's mount table. `Vnode2` is a directory in `vfs1`, which is the mount point for `vfs2`. `Vfs2` is an NFS file system, which contains `vnode3`.

**4.4 Path name traversal**

Path name traversal is done by the *lookuppn* routine (lookup path name), which takes a path name in a path name buffer and returns a pointer to the vnode which the path represents. This takes the place of the old *namei* routine.

If the path name begins with a "/", Path name traversal starts at the vnode pointed to by either `u.u_rdir` or the root. Otherwise it starts at the vnode pointed to by `u.u_cdir` (the current directory). *Lookuppn* traverses the path one component at a time using the `vn_lookup` vnode operation. `Vn_lookup` takes a directory vnode and a component as arguments and returns a vnode representing that component. If a directory vnode has `v_vfsmountedhere` set, then it is a mount point. When a mount point is encountered going down the file system tree, *lookuppn* follows the vnode's `v_vfsmountedhere` pointer to the mounted file system and calls the `vfs_root` operation to obtain the root vnode for the file system. Path name traversal then continues from this point. If a root vnode is encountered (VROOT flag in `v_flag` set) when following "..", *lookuppn* follows the `vfs_vnodecovered` pointer in the vnode's associated vfs to obtain the covered vnode. If a symbolic link is encountered *lookuppn* calls the

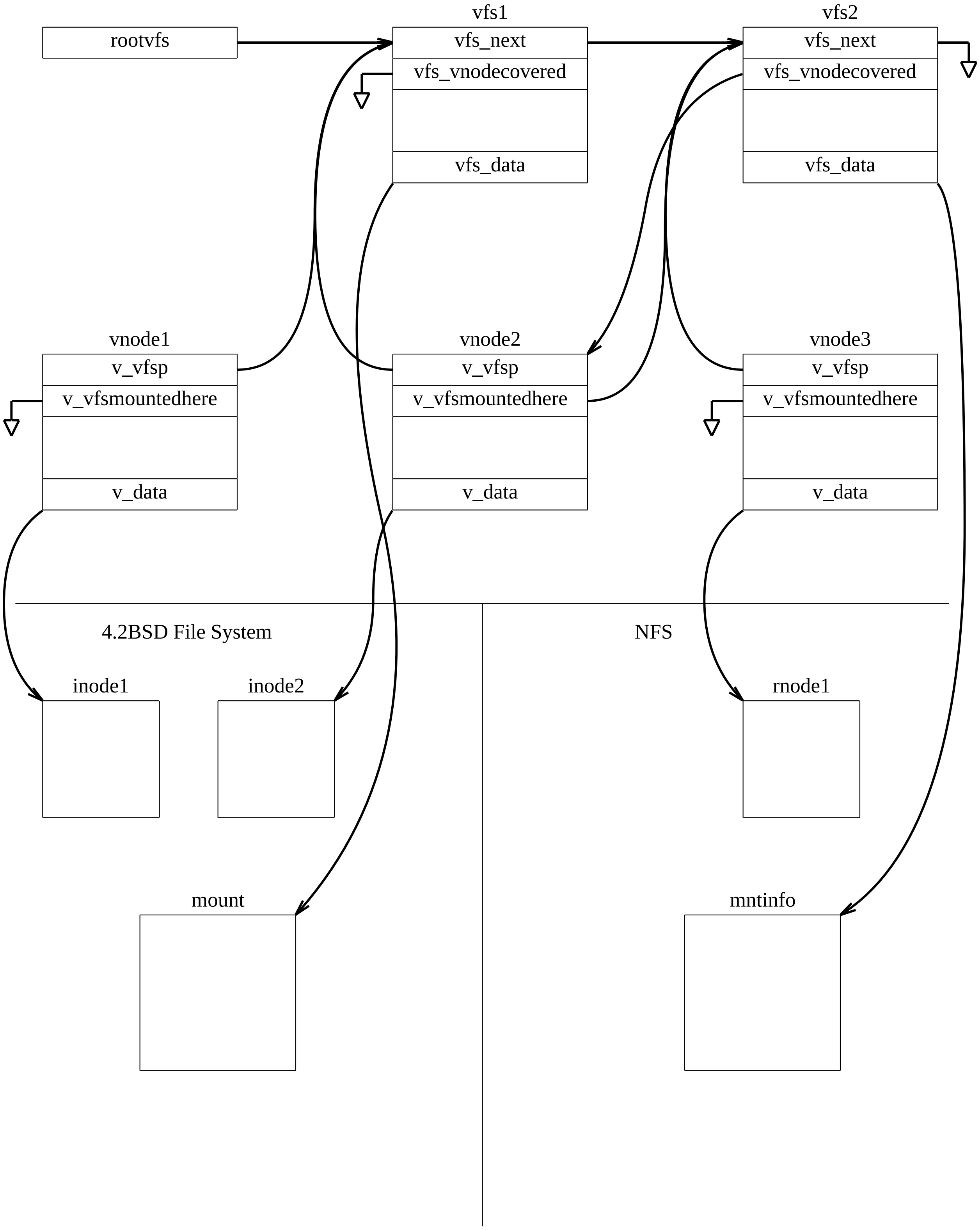


**Figure 3.** Example vnode layer object interconnection

`vn_readlink` vnode operation to obtain the symbolic link. If the symbolic link begins with a "/", the path name traversal is restarted from the root (or `u.u_rdir`); otherwise the traversal continues from the last directory. The caller of lookuppn specifies whether the last component of the path name is to be followed if it is a symbolic link. This process continues until the path name is exhausted or an error occurs. When *lookuppn* completes, a vnode representing the desired file is returned.

**4.5 Remote file systems**

The path name traversal scheme implies that files on remote file systems appear as files within the normal UNIX file name space. Remote files are not named by any special constructs that current programs don't understand[4]. The path name traversal process handles all indirection through mount points. This means that in a remote file system implementation, the client maintains its own mount points. If the client mounts another file system on a remote directory, the remote file system will never see references below the new mount point. Also, the remote file system will not see any ".." references at the root of the remote file system. For example, if the client has mounted a server's "/usr" on his "/usr" and a local file system on "/usr/local" then the path "/usr/local/bin" will access the local root, the remote "/usr" and the local "/usr/local" without the remote file system having any knowledge of the "/usr/local" mount point. Similarly, the path "/usr/.." will access the local root, the remote "/usr" and the local root, without the remote file system (or the server) seeing the ".." out of "/usr".

**4.6 New system calls**

Three new system calls were added in order to make the normal application interface file system implementation independent. The *getdirentries*(2) system call was added to read directories in a manner which is independent of the on disk directory format. *Getdirentries* reads directory entries from an open directory file descriptor into a user buffer, in file system independent format. As many directory entries as can fit in the buffer are read. The file pointer is is changed so that it points at directory entry boundaries after each call to *getdirentries*. The *statfs*(2) and *fstatfs*(2) system calls were added to get general file system statistics (e.g. space left). *Statfs* and *fstatfs* take a path name or a file descriptor, respectively, for a file within a particular file system, and return a `statfs` structure (see below).

**4.7 Devices**

The device interfaces, *bdevsw* and *cdevsw*, are hidden from the vnode layer, so that devices are only manipulated through the vnode interface. A special device file system implementation, which is never mounted, is provided to facilitate this. Thus, file systems which have a notion of associating a name within the file system with a local device may redirect vnodes to the special device file system.

**4.8 The buffer cache**

The buffer cache routines have been modified to act either as a physical buffer cache or a logical buffer cache. A local file system typically uses the buffer cache as a cache of physical disk blocks. Other file system types may use the buffer cache as a cache of logical file blocks. Unique blocks are identified by the pair (vnode-pointer, block-number). The vnode pointer points to a device vnode when a cached block is a copy of a physical device block, or it points to a file vnode when the block is a copy of a logical file block.

**5. VFS operations**

In the following descriptions of the vfs operations the *vfsp* argument is a pointer to the vfs that the operation is being applied to.

| | |
|---|---|
| vfs_mount(*vfsp*,*pathp*,*datap*) | Mount *vfsp* (i.e. read the superblock etc.). *Pathp* points to the path name to be mounted (for recording purposes), and *datap* points to file system dependent data. |
| vfs_unmount(*vfsp*) | Unmount *vfsp* (a.e. sync the superblock). |
| vfs_root(*vfsp*,*vpp*) | Return the root vnode for this file system. *Vpp* points to a pointer to a vnode for the results. |

vfs_statfs(*vfsp*,*sbp*) Return file system information. *Sbp* points to a statfs structure for the results.

```
struct statfs {
        long f_type;            /* type of info */
        long f_bsize;           /* block size */
        long f_blocks;          /* total blocks */
        long f_bfree;           /* free blocks */
        long f_bavail;          /* non-su blocks */
        long f_files;           /* total # of nodes */
        long f_ffree;           /* free nodes in fs */
        fsid_t f_fsid;          /* file system id */
        long f_spare[7];        /* spare for later */
};
```

vfs_sync(*vfsp*) Write out all cached information for *vfsp*. Note that this is not necessarily done synchronously. When the operation returns all data has not necessarily been written out, however it has been scheduled.

vfs_fid(*vfsp*,*vp*,*fidpp*) Get a unique file identifier for *vp* which represents a file within this file system. *Fidpp* points to a pointer to a fid structure for the results.

```
struct fid {
        u_short fid_len;        /* length of data */
        char fid_data[1];       /* variable size */
};
```

vfs_vget(*vfsp*,*vpp*,*fidp*) Turn unique file identifier *fidp* into a vnode representing the file associated with the file identifier. *vpp* points to a pointer to a vnode for the result.

**6. Vnode operations**

In the following descriptions of the vnode operations, the *vp* argument is a pointer to the vnode to which the operation is being applied; the *c* argument is a pointer to a credentials structure which contains the user credentials (e.g. uid) to use for the operation; and the *nm* argument is a pointer to a character string containing a name.

vn_open(*vpp*,*f*,*c*) Perform any open protocol on a vnode pointed to by *vpp* (e.g. devices). If the open is a "clone" open the operation may return a new vnode. *F* is the open flags.

vn_close(*vp*,*f*,*c*) Perform any close protocol on a vnode (e.g. devices). Called on the closing of the last reference to the vnode from the file table, if vnode is a device. Called on the last user close of a file descriptor, otherwise. *F* is the open flags.

vn_rdwr(*vp*,*uiop*,*rw*,*f*,*c*) Read or write vnode. Reads or writes a number of bytes at a specified offset in the file. *Uiop* points to a *uio* structure which supplies the I/O arguments. *Rw* specifies the I/O direction. *F* is the I/O flags, which may specify that the I/O is to be done synchronously (i.e. don't return until all the volatile data is on disk) and/or in a unit (i.e. lock the file to write a large unit).

vn_ioctl(*vp*,*com*,*d*,*f*,*c*) Perform an *ioctl* on vnode *vp*. *Com* is the command, *d* is the pointer to the data, and *f* is the open flags.

vn_select(*vp*,*w*,*c*) Perform a *select* on *vp*. *W* specifies the I/O direction.

vn_getattr(*vp*,*va*,*c*) Get attributes for *vp*. *Va* points to a *vattr* structure.

```
struct vattr {
        enum vtype      va_type;      /* vnode type */
        u_short         va_mode;      /* acc mode */
        short           va_uid;       /* owner uid */
        short           va_gid;       /* owner gid */
        long            va_fsid;      /* fs id */
        long            va_nodeid;    /* node # */
        short           va_nlink;     /* # links */
        u_long          va_size;      /* file size */
        long            va_blocksize; /* block size */
        struct timeval  va_atime;     /* last acc */
        struct timeval  va_mtime;     /* last mod */
        struct timeval  va_ctime;     /* last chg */
        dev_t           va_rdev;      /* dev */
        long            va_blocks;    /* space used */
};
```

This must map file system dependent attributes to UNIX file attributes.

vn_setattr(*vp*,*va*,*c*) — Set attributes for *vp*. *Va* points to a *vattr* structure, but only mode, uid, gid, file size, and times may be set. This must map UNIX file attributes to file system dependent attributes.

vn_access(*vp*,*m*,*c*) — Check access permissions for *vp*. Returns error if access is denied. *M* is the mode to check for access (e.g. read, write, execute). This must map UNIX file protection information to file system dependent protection information.

vn_lookup(*vp*,*nm*,*vpp*,*c*) — Lookup a component name *nm* in directory *vp*. *Vpp* points to a pointer to a vnode for the results.

vn_create(*vp*,*nm*,*va*,*e*,*m*,*vpp*,*c*) — Create a new file *nm* in directory *vp*. *Va* points to an *vattr* structure containing the attributes of the new file. *E* is the exclusive/non-exclusive create flag. *M* is the open mode. *vpp* points to a pointer to a vnode for the results.

vn_remove(*vp*,*nm*,*c*) — Remove a file *nm* in directory *vp*.

vn_link(*vp*,*tdvp*,*tnm*,*c*) — Link the vnode *vp* to the target name *tnm* in the target directory *tdvp*.

vn_rename(*vp*,*nm*,*tdvp*,*tnm*,*c*) — Rename the file *nm* in directory *vp* to *tnm* in target directory *tdvp*. The node can't be lost if the system crashes in the middle of the operation.

vn_mkdir(*vp*,*nm*,*va*,*vpp*,*c*) — Create directory *nm* in directory *vp*. *Va* points to an *vattr* structure containing the attributes of the new directory and *vpp* points to a pointer to a vnode for the results.

vn_rmdir(*vp*,*nm*,*c*) — Remove the directory *nm* from directory *vp*.

vn_readdir(*vp*,*uiop*,*c*) — Read entries from directory *vp*. *Uiop* points to a *uio* structure which supplies the I/O arguments. The uio offset is set to a file system dependent number which represents the logical offset in the directory when the reading is done. This is necessary because the number of bytes returned by *vn_readdir* is not necessarily the number of bytes in the equivalent part of the on disk directory.

vn_symlink(*vp*,*lnm*,*va*,*tnm*,*c*) — Symbolically link the path pointed to by *tnm* to the name *lnm* in directory *vp*.

vn_readlink(*vp*,*uiop*,*c*) — Read symbolic link *vp*. *Uiop* points to a *uio* structure which supplies the I/O arguments.

vn_fsync(*vp*,*c*) — Write out all cached information for file *vp*. The operation is synchronous and does not return until the I/O is complete.

vn_inactive(*vp*,*c*) The *vp* is no longer referenced by the vnode layer. It may now be deallocated.

vn_bmap(*vp*,*bn*,*vpp*,*bnp*) Map logical block number *bn* in file *vp* to physical block number and physical device. *Bnp* is a pointer to a block number for the physical block and *vpp* is a pointer to a vnode pointer for the physical device. Note that the returned vnode is not necessarily a physical device. This is used by the paging system to premap files before they are paged. In the NFS this is a null mapping.

vn_strategy(*bp*) Block oriented interface to read or write a logical block from a file into or out of a buffer. *Bp* is a pointer to a buffer header which contains a pointer to the vnode to be operated on. Does not copy through the buffer cache if the file system uses it. This is used by the buffer cache routines and the paging system to read blocks into memory.

vn_bread(*vp*,*bn*,*bpp*) Read a logical block *bn* from a file *vp* and return a pointer to a buffer header in *bpp* which contains a pointer to the data. This does not necessarily imply the use of the buffer cache. This operation is useful avoid extra data copying on the server side of a remote file system.

vn_brelse(*vp*,*bp*) The buffer returned by *vn_bread* can be released.

### 6.1 Kernel interfaces

A veneer layer is provided over the generic vnode interface to make it easier for kernel subsystems to manipulate files:

vn_open Perform permission checks and then open a vnode given by a path name.

vn_close Close a vnode.

vn_rdwr Build a *uio* structure and read or write a vnode.

vn_create Perform permission checks and then create a vnode given by a path name.

vn_remove Remove a node given by a path name.

vn_link Link a node given by a source path name to a target given by a target path name.

vn_rename Rename a node given by a source path name to a target given by a target path name.

VN_HOLD Increment the vnode reference count.

VN_RELE Decrement the vnode reference count and call *vn_inactive* if this is the last reference.

Many system calls which take names do a *lookuppn* to resolve the name to a vnode then call the appropriate veneer routine to do the operation. System calls which work off file descriptors pull the vnode pointer out of the file table and call the appropriate veneer routine.

## 7. Current status

The current interface has been in operation since the summer of 1984, and is a released Sun product. In general, the system performance degradation was nil to 2% depending on the benchmark. To date the 4.2BSD file system, the Sun Network File System, and an MSDOS floppy disk file system have been implemented under the interface, with other file system types to follow. In addition, a prototype "/proc" file system[5] has been implemented. It is also possible to configure out all the disk based file systems and run with just the NFS. Throughout this time the interface definition has been stable, with minor additions, even though several radically different file system types were implemented. Vnodes has been proven to provide a clean, well defined interface to different file system implementations.

Sun is currently discussing with AT&T and Berkeley the merging of this interface with AT&T's File System Switch technology. The goal is to produce a standard UNIX file system interface. Some of the current issues are:

- Allow multiple component lookup in `vn_lookup`. This would require file systems that implemented this to know about mount points.
- Cleaner replacements for `vn_bmap`, `vn_strategy`, `vn_bread`, and `vn_brelse`.
- Symlink handling in the file system independent layer.
- Eliminate redundant lookups.

### 8. Acknowledgements

Bill Joy is the designer of the architecture, and provided much help in its implementation. Dan Walsh modified *bio* and implemented parts of the device interface as well as parts of the 4.2BSD file system port. Russel Sandberg was the primary NFS implementor. He also built the prototype "/proc" file system and improved the device interface. Bill Shannon, Tom Lyon and Bob Lyon were invaluable in reviewing and evolving the interface.

*REFERENCES*

1. M.K. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX, ACM TOCS, 2, 3, August 1984, pp 181-197.
2. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem", USENIX Summer 1985, pp 119-130.
3. IBM, "DOS Operating System Version 2.0", January 1983.
4. R. Pike, P. Weinberger, "The Hideous Name" USENIX Summer 1985, pp 563-568.
5. T.J. Killian, "Processes as Files", USENIX Summer 1985, pp 203-207.