Toward a Compatible Filesystem Interface

Michael J. Karels
Marshall Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

As network or remote filesystems have been implemented for UNIX,† several stylized interfaces between the filesystem implementation and the rest of the kernel have been developed. Notable among these are Sun Microsystems' Virtual Filesystem interface (VFS) using vnodes, Digital Equipment's Generic File System (GFS) architecture, and AT&T's File System Switch (FSS). Each design attempts to isolate filesystem-dependent details below a generic interface and to provide a framework within which new filesystems may be incorporated. However, each of these interfaces is different from and incompatible with the others. Each of them addresses somewhat different design goals. Each was based on a different starting version of UNIX, targetted a different set of filesystems with varying characteristics, and uses a different set of primitive operations provided by the filesystem. The current study compares the various filesystem interfaces. Criteria for comparison include generality, completeness, robustness, efficiency and esthetics. Several of the underlying design issues are examined in detail. As a result of this comparison, a proposal for a new filesystem interface is advanced that includes the best features of the existing implementations. The proposal adopts the calling convention for name lookup introduced in 4.3BSD, but is otherwise closely related to Sun's VFS. A prototype implementation is now being developed at Berkeley. This proposal and the rationale underlying its development have been presented to major software vendors as an early step toward convergence on a compatible filesystem interface.

1. Introduction

As network communications and workstation environments became common elements in UNIX systems, several vendors of UNIX systems have designed and built network file systems that allow client process on one UNIX machine to access files on a server machine. Examples include Sun's Network File System, NFS [Sandberg85], AT&T's recently-announced Remote File Sharing, RFS [Rifkin86], the LOCUS distributed filesystem [Walker85], and Masscomp's extended filesystem [Cole85]. Other remote filesystems have been implemented in research or university groups for internal use, notably the network filesystem in the Eighth Edition UNIX system [Weinberger84] and two different filesystems used at Carnegie-Mellon University [Satyanarayanan85]. Numerous other remote file access methods have been devised for use within individual UNIX processes, many of them by modifications to the C I/O library similar to those in the Newcastle Connection [Brownbridge82].

---

† UNIX is a registered trademark of AT&T.

This is an update of a paper originally presented at the September 1986 conference of the European UNIX Users' Group. Last modified April 16, 1991.

Multiple network filesystems may frequently be found in use within a single organization. These circumstances make it highly desirable to be able to transport filesystem implementations from one system to another. Such portability is considerably enhanced by the use of a stylized interface with carefully-defined entry points to separate the filesystem from the rest of the operating system. This interface should be similar to the interface between device drivers and the kernel. Although varying somewhat among the common versions of UNIX, the device driver interfaces are sufficiently similar that device drivers may be moved from one system to another without major problems. A clean, well-defined interface to the filesystem also allows a single system to support multiple local filesystem types.

For reasons such as these, several filesystem interfaces have been used when integrating new filesystems into the system. The best-known of these are Sun Microsystems' Virtual File System interface, VFS [Kleiman86], and AT&T's File System Switch, FSS. Another interface, known as the Generic File System, GFS, has been implemented for the ULTRIX‡ system by Digital [Rodriguez86]. There are numerous differences among these designs. The differences may be understood from the varying philosophies and design goals of the groups involved, from the systems under which the implementations were done, and from the filesystems originally targetted by the designs. These differences are summarized in the following sections within the limitations of the published specifications.

2. Design goals

There are several design goals which, in varying degrees, have driven the various designs. Each attempts to divide the filesystem into a filesystem-type-independent layer and individual filesystem implementations. The division between these layers occurs at somewhat different places in these systems, reflecting different views of the diversity and types of the filesystems that may be accommodated. Compatibility with existing local filesystems has varying importance; at the user-process level, each attempts to be completely transparent except for a few filesystem-related system management programs. The AT&T interface also makes a major effort to retain familiar internal system interfaces, and even to retain object-file-level binary compatibility with operating system modules such as device drivers. Both Sun and DEC were willing to change internal data structures and interfaces so that other operating system modules might require recompilation or source-code modification.

AT&T's interface both allows and requires filesystems to support the full and exact semantics of their previous filesystem, including interruptions of system calls on slow operations. System calls that deal with remote files are encapsulated with their environment and sent to a server where execution continues. The system call may be aborted by either client or server, returning control to the client. Most system calls that descend into the file-system dependent layer of a filesystem other than the standard local filesystem do not return to the higher-level kernel calling routines. Instead, the filesystem-dependent code completes the requested operation and then executes a non-local goto (*longjmp*) to exit the system call. These efforts to avoid modification of main-line kernel code indicate a far greater emphasis on internal compatibility than on modularity, clean design, or efficiency.

In contrast, the Sun VFS interface makes major modifications to the internal interfaces in the kernel, with a very clear separation of filesystem-independent and -dependent data structures and operations. The semantics of the filesystem are largely retained for local operations, although this is achieved at some expense where it does not fit the internal structuring well. The filesystem implementations are not required to support the same semantics as local UNIX filesystems. Several historical features of UNIX filesystem behavior are difficult to achieve using the VFS interface, including the atomicity of file and link creation and the use of open files whose names have been removed.

A major design objective of Sun's network filesystem, statelessness, permeates the VFS interface. No locking may be done in the filesystem-independent layer, and locking in the filesystem-dependent layer may occur only during a single call into that layer.

A final design goal of most implementors is performance. For remote filesystems, this goal tends to be in conflict with the goals of complete semantic consistency, compatibility and modularity. Sun has chosen performance over modularity in some areas, but has emphasized clean separation of the layers within the filesystem at the expense of performance. Although the performance of RFS is yet to be seen, AT&T

‡ ULTRIX is a trademark of Digital Equipment Corp.

seems to have considered compatibility far more important than modularity or performance.

3. Differences among filesystem interfaces

The existing filesystem interfaces may be characterized in several ways. Each system is centered around a few data structures or objects, along with a set of primitives for performing operations upon these objects. In the original UNIX filesystem [Ritchie74], the basic object used by the filesystem is the inode, or index node. The inode contains all of the information about a file except its name: its type, identification, ownership, permissions, timestamps and location. Inodes are identified by the filesystem device number and the index within the filesystem. The major entry points to the filesystem are *namei*, which translates a filesystem pathname into the underlying inode, and *iget*, which locates an inode by number and installs it in the in-core inode table. *Namei* performs name translation by iterative lookup of each component name in its directory to find its inumber, then using *iget* to return the actual inode. If the last component has been reached, this inode is returned; otherwise, the inode describes the next directory to be searched. The inode returned may be used in various ways by the caller; it may be examined, the file may be read or written, types and access may be checked, and fields may be modified. Modified inodes are automatically written back the filesystem on disk when the last reference is released with *iput*. Although the details are considerably different, the same general scheme is used in the faster filesystem in 4.2BSD UNIX [Mckusick85].

Both the AT&T interface and, to a lesser extent, the DEC interface attempt to preserve the inode-oriented interface. Each modify the inode to allow different varieties of the structure for different filesystem types by separating the filesystem-dependent parts of the inode into a separate structure or one arm of a union. Both interfaces allow operations equivalent to the *namei* and *iget* operations of the old filesystem to be performed in the filesystem-independent layer, with entry points to the individual filesystem implementations to support the type-specific parts of these operations. Implicit in this interface is that files may be conveniently be named by and located using a single index within a filesystem. The GFS provides specific entry points to the filesystems to change most file properties rather than allowing arbitrary changes to be made to the generic part of the inode.

In contrast, the Sun VFS interface replaces the inode as the primary object with the vnode. The vnode contains no filesystem-dependent fields except the pointer to the set of operations implemented by the filesystem. Properties of a vnode that might be transient, such as the ownership, permissions, size and timestamps, are maintained by the lower layer. These properties may be presented in a generic format upon request; callers are expected not to hold this information for any length of time, as they may not be up-to-date later on. The vnode operations do not include a corollary for *iget*; the only external interface for obtaining vnodes for specific files is the name lookup operation. (Separate procedures are provided outside of this interface that obtain a ‘‘file handle’’ for a vnode which may be given to a client by a server, such that the vnode may be retrieved upon later presentation of the file handle.)

4. Name translation issues

Each of the systems described include a mechanism for performing pathname-to-internal-representation translation. The style of the name translation function is very different in all three systems. As described above, the AT&T and DEC systems retain the *namei* function. The two are quite different, however, as the ULTRIX interface uses the *namei* calling convention introduced in 4.3BSD. The parameters and context for the name lookup operation are collected in a *nameidata* structure which is passed to *namei* for operation. Intent to create or delete the named file is declared in advance, so that the final directory scan in *namei* may retain information such as the offset in the directory at which the modification will be made. Filesystems that use such mechanisms to avoid redundant work must therefore lock the directory to be modified so that it may not be modified by another process before completion. In the System V filesystem, as in previous versions of UNIX, this information is stored in the per-process *user* structure by *namei* for use by a low-level routine called after performing the actual creation or deletion of the file itself. In 4.3BSD and in the GFS interface, these side effects of *namei* are stored in the *nameidata* structure given as argument to *namei*, which is also presented to the routine implementing file creation or deletion.

The ULTRIX *namei* routine is responsible for the generic parts of the name translation process, such as copying the name into an internal buffer, validating it, interpolating the contents of symbolic links, and indirecting at mount points. As in 4.3BSD, the name is copied into the buffer in a single call, according to the location of the name. After determining the type of the filesystem at the start of translation (the current directory or root directory), it calls the filesystem's *namei* entry with the same structure it received from its caller. The filesystem-specific routine translates the name, component by component, as long as no mount points are reached. It may return after any number of components have been processed. *Namei* performs any processing at mount points, then calls the correct translation routine for the next filesystem. Network filesystems may pass the remaining pathname to a server for translation, or they may look up the pathname components one at a time. The former strategy would be more efficient, but the latter scheme allows mount points within a remote filesystem without server knowledge of all client mounts.

The AT&T *namei* interface is presumably the same as that in previous UNIX systems, accepting the name of a routine to fetch pathname characters and an operation (one of: lookup, lookup for creation, or lookup for deletion). It translates, component by component, as before. If it detects that a mount point crosses to a remote filesystem, it passes the remainder of the pathname to the remote server. A pathname-oriented request other than open may be completed within the *namei* call, avoiding return to the (unmodified) system call handler that called *namei*.

In contrast to the first two systems, Sun's VFS interface has replaced *namei* with *lookupname*. This routine simply calls a new pathname-handling module to allocate a pathname buffer and copy in the pathname (copying a character per call), then calls *lookuppn*. *Lookuppn* performs the iteration over the directories leading to the destination file; it copies each pathname component to a local buffer, then calls the filesystem *lookup* entry to locate the vnode for that file in the current directory. Per-filesystem *lookup* routines may translate only one component per call. For creation and deletion of new files, the lookup operation is unmodified; the lookup of the final component only serves to check for the existence of the file. The subsequent creation or deletion call, if any, must repeat the final name translation and associated directory scan. For new file creation in particular, this is rather inefficient, as file creation requires two complete scans of the directory.

Several of the important performance improvements in 4.3BSD were related to the name translation process [McKusick85][Leffler84]. The following changes were made:

1. A system-wide cache of recent translations is maintained. The cache is separate from the inode cache, so that multiple names for a file may be present in the cache. The cache does not hold "hard" references to the inodes, so that the normal reference pattern is not disturbed.
2. A per-process cache is kept of the directory and offset at which the last successful name lookup was done. This allows sequential lookups of all the entries in a directory to be done in linear time.
3. The entire pathname is copied into a kernel buffer in a single operation, rather than using two subroutine calls per character.
4. A pool of pathname buffers are held by *namei*, avoiding allocation overhead.

All of these performance improvements from 4.3BSD are well worth using within a more generalized filesystem framework. The generalization of the structure may otherwise make an already-expensive function even more costly. Most of these improvements are present in the GFS system, as it derives from the beta-test version of 4.3BSD. The Sun system uses a name-translation cache generally like that in 4.3BSD. The name cache is a filesystem-independent facility provided for the use of the filesystem-specific lookup routines. The Sun cache, like that first used at Berkeley but unlike that in 4.3, holds a "hard" reference to the vnode (increments the reference count). The "soft" reference scheme in 4.3BSD cannot be used with the current NFS implementation, as NFS allocates vnodes dynamically and frees them when the reference count returns to zero rather than caching them. As a result, fewer names may be held in the cache than (local filesystem) vnodes, and the cache distorts the normal reference patterns otherwise seen by the LRU cache. As the name cache references overflow the local filesystem inode table, the name cache must be purged to make room in the inode table. Also, to determine whether a vnode is in use (for example, before mounting upon it), the cache must be flushed to free any cache reference. These problems should be corrected by the use of the soft cache reference scheme.

A final observation on the efficiency of name translation in the current Sun VFS architecture is that the number of subroutine calls used by a multi-component name lookup is dramatically larger than in the other systems. The name lookup scheme in GFS suffers from this problem much less, at no expense in violation of layering.

A final problem to be considered is synchronization and consistency. As the filesystem operations are more stylized and broken into separate entry points for parts of operations, it is more difficult to guarantee consistency throughout an operation and/or to synchronize with other processes using the same filesystem objects. The Sun interface suffers most severely from this, as it forbids the filesystems from locking objects across calls to the filesystem. It is possible that a file may be created between the time that a lookup is performed and a subsequent creation is requested. Perhaps more strangely, after a lookup fails to find the target of a creation attempt, the actual creation might find that the target now exists and is a symbolic link. The call will either fail unexpectedly, as the target is of the wrong type, or the generic creation routine will have to note the error and restart the operation from the lookup. This problem will always exist in a stateless filesystem, but the VFS interface forces all filesystems to share the problem. This restriction against locking between calls also forces duplication of work during file creation and deletion. This is considered unacceptable.

5. Support facilities and other interactions

Several support facilities are used by the current UNIX filesystem and require generalization for use by other filesystem types. For filesystem implementations to be portable, it is desirable that these modified support facilities should also have a uniform interface and behave in a consistent manner in target systems. A prominent example is the filesystem buffer cache. The buffer cache in a standard (System V or 4.3BSD) UNIX system contains physical disk blocks with no reference to the files containing them. This works well for the local filesystem, but has obvious problems for remote filesystems. Sun has modified the buffer cache routines to describe buffers by vnode rather than by device. For remote files, the vnode used is that of the file, and the block numbers are virtual data blocks. For local filesystems, a vnode for the block device is used for cache reference, and the block numbers are filesystem physical blocks. Use of per-file cache description does not easily accommodate caching of indirect blocks, inode blocks, superblocks or cylinder group blocks. However, the vnode describing the block device for the cache is one created internally, rather than the vnode for the device looked up when mounting, and it is located by searching a private list of vnodes rather than by holding it in the mount structure. Although the Sun modification makes it possible to use the buffer cache for data blocks of remote files, a better generalization of the buffer cache is needed.

The RFS filesystem used by AT&T does not currently cache data blocks on client systems, thus the buffer cache is probably unmodified. The form of the buffer cache in ULTRIX is unknown to us.

Another subsystem that has a large interaction with the filesystem is the virtual memory system. The virtual memory system must read data from the filesystem to satisfy fill-on-demand page faults. For efficiency, this read call is arranged to place the data directly into the physical pages assigned to the process (a "raw" read) to avoid copying the data. Although the read operation normally bypasses the filesystem buffer cache, consistency must be maintained by checking the buffer cache and copying or flushing modified data not yet stored on disk. The 4.2BSD virtual memory system, like that of Sun and ULTRIX, maintains its own cache of reusable text pages. This creates additional complications. As the virtual memory systems are redesigned, these problems should be resolved by reading through the buffer cache, then mapping the cached data into the user address space. If the buffer cache or the process pages are changed while the other reference remains, the data would have to be copied ("copy-on-write").

In the meantime, the current virtual memory systems must be used with the new filesystem framework. Both the Sun and AT&T filesystem interfaces provide entry points to the filesystem for optimization of the virtual memory system by performing logical-to-physical block number translation when setting up a fill-on-demand image for a process. The VFS provides a vnode operation analogous to the *bmap* function of the UNIX filesystem. Given a vnode and logical block number, it returns a vnode and block number which may be read to obtain the data. If the filesystem is local, it returns the private vnode for the block device and the physical block number. As the *bmap* operations are all performed at one time, during process startup, any indirect blocks for the file will remain in the cache after they are once read. In addition, the interface provides a *strategy* entry that may be used for "raw" reads from a filesystem device, used to read data blocks into an address space without copying. This entry uses a buffer header (*buf* structure) to describe the I/O operation instead of a *uio* structure. The buffer-style interface is the same as that used by disk drivers internally. This difference allows the current *uio* primitives to be avoided, as they copy all data to/from the current user process address space. Instead, for local filesystems these operations could be done internally with the standard raw disk read routines, which use a *uio* interface. When loading from a remote filesystems, the data will be received in a network buffer. If network buffers are suitably aligned, the data may be mapped into the process address space by a page swap without copying. In either case, it should be possible to use the standard filesystem read entry from the virtual memory system.

Other issues that must be considered in devising a portable filesystem implementation include kernel memory allocation, the implicit use of user-structure global context, which may create problems with reentrancy, the style of the system call interface, and the conventions for synchronization (sleep/wakeup, handling of interrupted system calls, semaphores).

6. The Berkeley Proposal

The Sun VFS interface has been most widely used of the three described here. It is also the most general of the three, in that filesystem-specific data and operations are best separated from the generic layer. Although it has several disadvantages which were described above, most of them may be corrected with minor changes to the interface (and, in a few areas, philosophical changes). The DEC GFS has other advantages, in particular the use of the 4.3BSD *namei* interface and optimizations. It allows single or multiple components of a pathname to be translated in a single call to the specific filesystem and thus accommodates filesystems with either preference. The FSS is least well understood, as there is little public information about the interface. However, the design goals are the least consistent with those of the Berkeley research groups. Accordingly, a new filesystem interface has been devised to avoid some of the problems in the other systems. The proposed interface derives directly from Sun's VFS, but, like GFS, uses a 4.3BSD-style name lookup interface. Additional context information has been moved from the *user* structure to the *nameidata* structure so that name translation may be independent of the global context of a user process. This is especially desired in any system where kernel-mode servers operate as light-weight or interrupt-level processes, or where a server may store or cache context for several clients. This calling interface has the additional advantage that the call parameters need not all be pushed onto the stack for each call through the filesystem interface, and they may be accessed using short offsets from a base pointer (unlike global variables in the *user* structure).

The proposed filesystem interface is described very tersely here. For the most part, data structures and procedures are analogous to those used by VFS, and only the changes will be treated here. See [Kleiman86] for complete descriptions of the vfs and vnode operations in Sun's interface.

The central data structure for name translation is the *nameidata* structure. The same structure is used to pass parameters to *namei*, to pass these same parameters to filesystem-specific lookup routines, to communicate completion status from the lookup routines back to *namei*, and to return completion status to the calling routine. For creation or deletion requests, the parameters to the filesystem operation to complete the request are also passed in this same structure. The form of the *nameidata* structure is:

```
/*
 * Encapsulation of namei parameters.
 * One of these is located in the u. area to
 * minimize space allocated on the kernel stack
 * and to retain per-process context.
 */
struct nameidata {
                /* arguments to namei and related context: */
        caddr_t ni_dirp;                /* pathname pointer */
        enum    uio_seg ni_seg;         /* location of pathname */
        short   ni_nameiop;             /* see below */
        struct  vnode *ni_cdir;         /* current directory */
        struct  vnode *ni_rdir;         /* root directory, if not normal root */
        struct  ucred *ni_cred;         /* credentials */

                /* shared between namei, lookup routines and commit routines: */
        caddr_t ni_pnbuf;               /* pathname buffer */
```

```
        char      *ni_ptr;                   /* current location in pathname */
        int       ni_pathlen;                /* remaining chars in path */
        short     ni_more;                   /* more left to translate in pathname */
        short     ni_loopcnt;                /* count of symlinks encountered */

                  /* results: */
        struct    vnode *ni_vp;              /* vnode of result */
        struct    vnode *ni_dvp;             /* vnode of intermediate directory */

/* BEGIN UFS SPECIFIC */
        struct diroffcache {                 /* last successful directory search */
                  struct  vnode *nc_prevdir; /* terminal directory */
                  long    nc_id;             /* directory's unique id */
                  off_t   nc_prevoffset;     /* where last entry found */
        } ni_nc;
/* END UFS SPECIFIC */
};

/*
 * namei operations and modifiers
 */
#define   LOOKUP        0         /* perform name lookup only */
#define   CREATE        1         /* setup for file creation */
#define   DELETE        2         /* setup for file deletion */
#define   WANTPARENT    0x10      /* return parent directory vnode also */
#define   NOCACHE       0x20      /* name must not be left in cache */
#define   FOLLOW        0x40      /* follow symbolic links */
#define   NOFOLLOW      0x0       /* don't follow symbolic links (pseudo) */
```

As in current systems other than Sun's VFS, *namei* is called with an operation request, one of LOOKUP, CREATE or DELETE. For a LOOKUP, the operation is exactly like the lookup in VFS. CREATE and DELETE allow the filesystem to ensure consistency by locking the parent inode (private to the filesystem), and (for the local filesystem) to avoid duplicate directory scans by storing the new directory entry and its offset in the directory in the *ndirinfo* structure. This is intended to be opaque to the filesystem-independent levels. Not all lookups for creation or deletion are actually followed by the intended operation; permission may be denied, the filesystem may be read-only, etc. Therefore, an entry point to the filesystem is provided to abort a creation or deletion operation and allow release of any locked internal data. After a *namei* with a CREATE or DELETE flag, the pathname pointer is set to point to the last filename component. Filesystems that choose to implement creation or deletion entirely within the subsequent call to a create or delete entry are thus free to do so.

The *nameidata* is used to store context used during name translation. The current and root directories for the translation are stored here. For the local filesystem, the per-process directory offset cache is also kept here. A file server could leave the directory offset cache empty, could use a single cache for all clients, or could hold caches for several recent clients.

Several other data structures are used in the filesystem operations. One is the *ucred* structure which describes a client's credentials to the filesystem. This is modified slightly from the Sun structure; the "accounting" group ID has been merged into the groups array. The actual number of groups in the array is given explicitly to avoid use of a reserved group ID as a terminator. Also, typedefs introduced in 4.3BSD for user and group ID's have been used. The *ucred* structure is thus:

```
/*
 * Credentials.
 */
struct ucred {
        u_short     cr_ref;                     /* reference count */
        uid_t       cr_uid;                     /* effective user id */
        short       cr_ngroups;                 /* number of groups */
        gid_t       cr_groups[NGROUPS];         /* groups */
        /*
         * The following either should not be here,
         * or should be treated as opaque.
         */
        uid_t   cr_ruid;                        /* real user id */
        gid_t   cr_svgid;                       /* saved set-group id */
};
```

A final structure used by the filesystem interface is the *uio* structure mentioned earlier. This structure describes the source or destination of an I/O operation, with provision for scatter/gather I/O. It is used in the read and write entries to the filesystem. The *uio* structure presented here is modified from the one used in 4.2BSD to specify the location of each vector of the operation (user or kernel space) and to allow an alternate function to be used to implement the data movement. The alternate function might perform page remapping rather than a copy, for example.

```
/*
 * Description of an I/O operation which potentially
 * involves scatter-gather, with individual sections
 * described by iovec, below.  uio_resid is initially
 * set to the total size of the operation, and is
 * decremented as the operation proceeds.  uio_offset
 * is incremented by the amount of each operation.
 * uio_iov is incremented and uio_iovcnt is decremented
 * after each vector is processed.
 */
struct uio {
        struct      iovec *uio_iov;
        int         uio_iovcnt;
        off_t       uio_offset;
        int         uio_resid;
        enum        uio_rw uio_rw;
};

enum    uio_rw { UIO_READ, UIO_WRITE };
```

```
/*
 * Description of a contiguous section of an I/O operation.
 * If iov_op is non-null, it is called to implement the copy
 * operation, possibly by remapping, with the call
 *       (*iov_op)(from, to, count);
 * where from and to are caddr_t and count is int.
 * Otherwise, the copy is done in the normal way,
 * treating base as a user or kernel virtual address
 * according to iov_segflg.
 */
struct iovec {
        caddr_t   iov_base;
        int       iov_len;
        enum      uio_seg iov_segflg;
        int       (*iov_op)();
};

/*
 * Segment flag values.
 */
enum    uio_seg {
        UIO_USERSPACE,        /* from user data space */
        UIO_SYSSPACE,         /* from system space */
        UIO_USERISPACE        /* from user I space */
};
```

7. File and filesystem operations

With the introduction of the data structures used by the filesystem operations, the complete list of filesystem entry points may be listed. As noted, they derive mostly from the Sun VFS interface. Lines marked with **+** are additions to the Sun definitions; lines marked with **!** are modified from VFS.

The structure describing the externally-visible features of a mounted filesystem, *vfs*, is:

```
/*
 * Structure per mounted file system.
 * Each mounted file system has an array of
 * operations and an instance record.
 * The file systems are put on a doubly linked list.
 */
struct vfs {
        struct vfs      *vfs_next;              /* next vfs in vfs list */
+       struct vfs      *vfs_prev;              /* prev vfs in vfs list */
        struct vfsops   *vfs_op;                /* operations on vfs */
        struct vnode    *vfs_vnodecovered;      /* vnode we mounted on */
        int             vfs_flag;               /* flags */
!       int             vfs_fsize;              /* fundamental block size */
+       int             vfs_bsize;              /* optimal transfer size */
!       uid_t           vfs_exroot;             /* exported fs uid 0 mapping */
        short           vfs_exflags;            /* exported fs flags */
        caddr_t         vfs_data;               /* private data */
};
```

```
  /*
   * vfs flags.
   * VFS_MLOCK lock the vfs so that name lookup cannot proceed past the vfs.
   * This keeps the subtree stable during mounts and unmounts.
   */
  #define   VFS_RDONLY     0x01    /* read only vfs */
+ #define   VFS_NOEXEC     0x02    /* can't exec from filesystem */
  #define   VFS_MLOCK      0x04    /* lock vfs so that subtree is stable */
  #define   VFS_MWAIT      0x08    /* someone is waiting for lock */
  #define   VFS_NOSUID     0x10    /* don't honor setuid bits on vfs */
  #define   VFS_EXPORTED   0x20    /* file system is exported (NFS) */

  /*
   * exported vfs flags.
   */
  #define   EX_RDONLY      0x01    /* exported read only */
```

The operations supported by the filesystem-specific layer on an individual filesystem are:

```
/*
 * Operations supported on virtual file system.
 */
struct vfsops {
!       int     (*vfs_mount)(       /* vfs, path, data, datalen */ );
!       int     (*vfs_unmount)(     /* vfs, forcibly */ );
+       int     (*vfs_mountroot)();
        int     (*vfs_root)(        /* vfs, vpp */ );
!       int     (*vfs_statfs)(      /* vfs, vp, sbp */ );
!       int     (*vfs_sync)(        /* vfs, waitfor */ );
+       int     (*vfs_fhtovp)(      /* vfs, fhp, vpp */ );
+       int     (*vfs_vptofh)(      /* vp, fhp */ );
};
```

The *vfs_statfs* entry returns a structure of the form:

```
/*
 * file system statistics
 */
struct statfs {
!       short   f_type;           /* type of filesystem */
+       short   f_flags;          /* copy of vfs (mount) flags */
!       long    f_fsize;          /* fundamental file system block size */
+       long    f_bsize;          /* optimal transfer block size */
        long    f_blocks;         /* total data blocks in file system */
        long    f_bfree;          /* free blocks in fs */
        long    f_bavail;         /* free blocks avail to non-superuser */
        long    f_files;          /* total file nodes in file system */
        long    f_ffree;          /* free file nodes in fs */
        fsid_t  f_fsid;           /* file system id */
+       char    *f_mntonname;     /* directory on which mounted */
+       char    *f_mntfromname;   /* mounted filesystem */
        long    f_spare[7];       /* spare for later */
};

typedef long fsid_t[2];           /* file system id type */
```

The modifications to Sun's interface at this level are minor. Additional arguments are present for the *vfs_mount* and *vfs_umount* entries. *vfs_statfs* accepts a vnode as well as filesystem identifier, as the information may not be uniform throughout a filesystem. For example, if a client may mount a file tree that spans multiple physical filesystems on a server, different sections may have different amounts of free space. (NFS does not allow remotely-mounted file trees to span physical filesystems on the server.) The final additions are the entries that support file handles. *vfs_vptofh* is provided for the use of file servers, which need to obtain an opaque file handle to represent the current vnode for transmission to clients. This file handle may later be used to relocate the vnode using *vfs_fhtovp* without requiring the vnode to remain in memory.

Finally, the external form of a filesystem object, the *vnode*, is:

```
/*
 * vnode types. VNON means no type.
 */
enum vtype              { VNON, VREG, VDIR, VBLK, VCHR, VLNK, VSOCK };

struct vnode {
        u_short         v_flag;                 /* vnode flags (see below) */
        u_short         v_count;                /* reference count */
        u_short         v_shlockc;              /* count of shared locks */
        u_short         v_exlockc;              /* count of exclusive locks */
        struct vfs      *v_vfsmountedhere;      /* ptr to vfs mounted here */
        struct vfs      *v_vfsp;                /* ptr to vfs we are in */
        struct vnodeops *v_op;                  /* vnode operations */
+       struct text     *v_text;                /* text/mapped region */
        enum vtype      v_type;                 /* vnode type */
        caddr_t         v_data;                 /* private data for fs */
};

/*
 * vnode flags.
 */
#define VROOT      0x01    /* root of its file system */
#define VTEXT      0x02    /* vnode is a pure text prototype */
#define VEXLOCK    0x10    /* exclusive lock */
#define VSHLOCK    0x20    /* shared lock */
#define VLWAIT     0x40    /* proc is waiting on shared or excl. lock */
```

The operations supported by the filesystems on individual *vnode*s are:

```
/*
 * Operations on vnodes.
 */
struct vnodeops {
!       int     (*vn_lookup)(     /* ndp */ );
!       int     (*vn_create)(     /* ndp, vap, fflags */ );
+       int     (*vn_mknod)(      /* ndp, vap, fflags */ );
!       int     (*vn_open)(       /* vp, fflags, cred */ );
        int     (*vn_close)(      /* vp, fflags, cred */ );
        int     (*vn_access)(     /* vp, fflags, cred */ );
        int     (*vn_getattr)(    /* vp, vap, cred */ );
        int     (*vn_setattr)(    /* vp, vap, cred */ );

+       int     (*vn_read)(       /* vp, uiop, offp, ioflag, cred */ );
+       int     (*vn_write)(      /* vp, uiop, offp, ioflag, cred */ );
!       int     (*vn_ioctl)(      /* vp, com, data, fflag, cred */ );
        int     (*vn_select)(     /* vp, which, cred */ );
+       int     (*vn_mmap)(       /* vp, ..., cred */ );
        int     (*vn_fsync)(      /* vp, cred */ );
+       int     (*vn_seek)(       /* vp, offp, off, whence */ );

!       int     (*vn_remove)(     /* ndp */ );
!       int     (*vn_link)(       /* vp, ndp */ );
!       int     (*vn_rename)(     /* src ndp, target ndp */ );
!       int     (*vn_mkdir)(      /* ndp, vap */ );
!       int     (*vn_rmdir)(      /* ndp */ );
!       int     (*vn_symlink)(    /* ndp, vap, nm */ );
        int     (*vn_readdir)(    /* vp, uiop, offp, ioflag, cred */ );
        int     (*vn_readlink)(   /* vp, uiop, ioflag, cred */ );

+       int     (*vn_abortop)(    /* ndp */ );
+       int     (*vn_lock)(       /* vp */ );
+       int     (*vn_unlock)(     /* vp */ );
!       int     (*vn_inactive)(   /* vp */ );
};

/*
 * flags for ioflag
 */
#define IO_UNIT     0x01    /* do io as atomic unit for VOP_RDWR */
#define IO_APPEND   0x02    /* append write for VOP_RDWR */
#define IO_SYNC     0x04    /* sync io for VOP_RDWR */
```

The argument types listed in the comments following each operation are:

ndp A pointer to a *nameidata* structure.

vap A pointer to a *vattr* structure (vnode attributes; see below).

fflags File open flags, possibly including O_APPEND, O_CREAT, O_TRUNC and O_EXCL.

vp A pointer to a *vnode* previously obtained with *vn_lookup*.

cred A pointer to a *ucred* credentials structure.

uiop A pointer to a *uio* structure.

ioflag Any of the IO flags defined above.

com	An *ioctl* command, with type *unsigned long*.
data	A pointer to a character buffer used to pass data to or from an *ioctl*.
which	One of FREAD, FWRITE or 0 (select for exceptional conditions).
off	A file offset of type *off_t*.
offp	A pointer to file offset of type *off_t*.
whence	One of L_SET, L_INCR, or L_XTND.
fhp	A pointer to a file handle buffer.

Several changes have been made to Sun's set of vnode operations. Most obviously, the *vn_lookup* receives a *nameidata* structure containing its arguments and context as described. The same structure is also passed to one of the creation or deletion entries if the lookup operation is for CREATE or DELETE to complete an operation, or to the *vn_abortop* entry if no operation is undertaken. For filesystems that perform no locking between lookup for creation or deletion and the call to implement that action, the final pathname component may be left untranslated by the lookup routine. In any case, the pathname pointer points at the final name component, and the *nameidata* contains a reference to the vnode of the parent directory. The interface is thus flexible enough to accommodate filesystems that are fully stateful or fully stateless, while avoiding redundant operations whenever possible. One operation remains problematical, the *vn_rename* call. It is tempting to look up the source of the rename for deletion and the target for creation. However, filesystems that lock directories during such lookups must avoid deadlock if the two paths cross. For that reason, the source is translated for LOOKUP only, with the WANTPARENT flag set; the target is then translated with an operation of CREATE.

In addition to the changes concerned with the *nameidata* interface, several other changes were made in the vnode operations. The *vn_rdrw* entry was split into *vn_read* and *vn_write*; frequently, the read/write entry amounts to a routine that checks the direction flag, then calls either a read routine or a write routine. The two entries may be identical for any given filesystem; the direction flag is contained in the *uio* given as an argument.

All of the read and write operations use a *uio* to describe the file offset and buffer locations. All of these fields must be updated before return. In particular, the *vn_readdir* entry uses this to return a new file offset token for its current location.

Several new operations have been added. The first, *vn_seek*, is a concession to record-oriented files such as directories. It allows the filesystem to verify that a seek leaves a file at a sensible offset, or to return a new offset token relative to an earlier one. For most filesystems and files, this operation amounts to performing simple arithmetic. Another new entry point is *vn_mmap*, for use in mapping device memory into a user process address space. Its semantics are not yet decided. The final additions are the *vn_lock* and *vn_unlock* entries. These are used to request that the underlying file be locked against changes for short periods of time if the filesystem implementation allows it. They are used to maintain consistency during internal operations such as *exec*, and may not be used to construct atomic operations from other filesystem operations.

The attributes of a vnode are not stored in the vnode, as they might change with time and may need to be read from a remote source. Attributes have the form:

```
/*
 * Vnode attributes.  A field value of -1
 * represents a field whose value is unavailable
 * (getattr) or which is not to be changed (setattr).
 */
struct vattr {
        enum vtype      va_type;        /* vnode type (for create) */
        u_short         va_mode;        /* files access mode and type */
!       uid_t           va_uid;         /* owner user id */
!       gid_t           va_gid;         /* owner group id */
        long            va_fsid;        /* file system id (dev for now) */
!       long            va_fileid;      /* file id */
        short           va_nlink;       /* number of references to file */
        u_long          va_size;        /* file size in bytes (quad?) */
+       u_long          va_size1;       /* reserved if not quad */
        long            va_blocksize;   /* blocksize preferred for i/o */
        struct timeval  va_atime;       /* time of last access */
        struct timeval  va_mtime;       /* time of last modification */
        struct timeval  va_ctime;       /* time file changed */
        dev_t           va_rdev;        /* device the file represents */
        u_long          va_bytes;       /* bytes of disk space held by file */
+       u_long          va_bytes1;      /* reserved if va_bytes not a quad */
};
```

8. Conclusions

The Sun VFS filesystem interface is the most widely used generic filesystem interface. Of the interfaces examined, it creates the cleanest separation between the filesystem-independent and -dependent layers and data structures. It has several flaws, but it is felt that certain changes in the interface can ameliorate most of them. The interface proposed here includes those changes. The proposed interface is now being implemented by the Computer Systems Research Group at Berkeley. If the design succeeds in improving the flexibility and performance of the filesystem layering, it will be advanced as a model interface.

9. Acknowledgements

The filesystem interface described here is derived from Sun's VFS interface. It also includes features similar to those of DEC's GFS interface. We are indebted to members of the Sun and DEC system groups for long discussions of the issues involved.

10. References

Brownbridge82 — Brownbridge, D.R., L.F. Marshall, B. Randell, ‘‘The Newcastle Connection, or UNIXes of the World Unite!,’’ *Software− Practice and Experience*, Vol. 12, pp. 1147-1162, 1982.

Cole85 — Cole, C.T., P.B. Flinn, A.B. Atlas, ‘‘An Implementation of an Extended File System for UNIX,’’ *Usenix Conference Proceedings*, pp. 131-150, June, 1985.

Kleiman86 — ‘‘Vnodes: An Architecture for Multiple File System Types in Sun UNIX,’’ *Usenix Conference Proceedings*, pp. 238-247, June, 1986.

Leffler84 — Leffler, S., M.K. McKusick, M. Karels, ‘‘Measuring and Improving the Performance of 4.2BSD,’’ *Usenix Conference Proceedings*, pp. 237-252, June, 1984.

McKusick84 McKusick, M.K., W.N. Joy, S.J. Leffler, R.S. Fabry, ‘‘A Fast File System for UNIX,’’ *Transactions on Computer Systems*, Vol. 2, pp. 181-197, ACM, August, 1984.

McKusick85 McKusick, M.K., M. Karels, S. Leffler, ‘‘Performance Improvements and Functional Enhancements in 4.3BSD,’’ *Usenix Conference Proceedings*, pp. 519-531, June, 1985.

Rifkin86 Rifkin, A.P., M.P. Forbes, R.L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, ‘‘RFS Architectural Overview,’’ *Usenix Conference Proceedings*, pp. 248-259, June, 1986.

Ritchie74 Ritchie, D.M. and K. Thompson, ‘‘The Unix Time-Sharing System,’’ *Communications of the ACM*, Vol. 17, pp. 365-375, July, 1974.

Rodriguez86 Rodriguez, R., M. Koehler, R. Hyde, ‘‘The Generic File System,’’ *Usenix Conference Proceedings*, pp. 260-269, June, 1986.

Sandberg85 Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, ‘‘Design and Implementation of the Sun Network Filesystem,’’ *Usenix Conference Proceedings*, pp. 119-130, June, 1985.

Satyanarayanan85 Satyanarayanan, M., *et al.*, ‘‘The ITC Distributed File System: Principles and Design,’’ *Proc. 10th Symposium on Operating Systems Principles*, pp. 35-50, ACM, December, 1985.

Walker85 Walker, B.J. and S.H. Kiser, ‘‘The LOCUS Distributed Filesystem,’’ *The LOCUS Distributed System Architecture*, G.J. Popek and B.J. Walker, ed., The MIT Press, Cambridge, MA, 1985.

Weinberger84 Weinberger, P.J., ‘‘The Version 8 Network File System,’’ *Usenix Conference presentation*, June, 1984.