xFS NameSpace Manager Design

Curtis Anderson

1.0 Introduction

This document describes the requirements and proposed implementation for the namespace management functions of the xFS filesystem. Note that only the namespace features are defined here, attribute management, space management and other features are described elsewhere.

The NameSpace Manager supports all the traditional name-related VNODE operations by storing specialized structures (a directory) inside the logical address space provided by a “file”. The Space Manager provides the “file” abstraction and allows the caller to set the inode type as it desires.

The reader will notice the extreme similarity between the NameSpace Manager and the Attribute Manager designs. This is intentional, they are parallel in the system architecture and will in fact share portions of code.

2.0 Requirements and Functionality

2.1 Requirements

Here are the external requirements for the namespace management code in xFS.

2.1.1 Standard VNODE Operations and Semantics

This filesystem has to operate in a fairly standard VFS/VNODE environment, so all the existing namespace-related entry points and arguments have to be available and working.

2.1.2 Fast for Large and Small Directories

The namespace operations must be fast for any size of directory, not just “normal” sizes. One of the goals of the namespace design is to efficiently support very large single directories.

2.1.3 Internationalization

It should be possible to store file names in an international character set, and file contents should be tagged so that processes can determine the character set used inside them.

2.1.4 Location Independence (Distributed Naming)

Hooks must be available to support the planned distributed filesystem. These will take the form of “mount point” type inodes and possibly remote device access, but may not be available in the first release.

2.1.5 IPC Rendezvous Nodes

An inode type will be provided that is similar to named pipes or sockets. It will provide an interprocess communication conduit that will have more capabilities that a named pipe or socket.

Specifically, it will be able to take part in naming operations. There will be a way for a user process to ask that for any naming operations that attempt to pass through the inode, the remaining piece of the pathname will be passed out to the process. The intent is to allow active user processes to be full participants in the namespace. It should be very easy to build a new filesystem type in user mode with these IPC nodes.

IPC nodes may not be available in the first release.

2.2 Functionality (ie: NameSpace Semantics)

This section describes all the objects that are visible in the namespace, and their associated semantics. The implementation of these objects is described in a following section.

2.2.1 Object Types

This section lists the different types of objects that exist in the namespace, and what their semantics are.

2.2.1.1 Files, Directories, Symlinks

These types will have the same semantics that they do in IRIX today. Note that directories will have a different internal structure, but that readdir() and friends will be supported.

2.2.1.2 Pipes, Sockets, Device Nodes

Until the distributed filesystem support appears, these types will have the same semantics that they do in IRIX today. When the distributed system support appears, these questions will need to be answered:

- How will a pipe act if the two opening processes are on different machines?
- What should the “originating address” of a socket be if the system managing the filesystem is not the system managing the process?
- Should we provide transparent access to devices that are attached to remote machines?

2.2.1.3 Mount Points

These are a special inode type that is just a holder for the UUID of the filesystem we want access to. It is somewhat analogous to a symlink in that the UUID specifies where to continue pathname resolution.

When a pathname is being resolved and one of these inodes is found in the middle of the pathname, the filesystem will return a message to the upper layers of the kernel saying that pathname resolution should continue from the indicated point on the indicated system (ie: whatever is left of the pathname, on the system that is managing the indicated filesystem).

One method of multi-hop pathname resolution would have the kernel on the system that ran into the mountpoint inode pass a message on to the kernel managing the “next” filesystem to continue pathname resolution, without involving the “originator” system. This has a severe drawback when one system in that chain fails.

Returning a message to the originating kernel rather than just passing it along allows the originating kernel to know who is doing processing on its behalf. The originating kernel can then tell if the remote system has gone down or is just slow.

During system boot or when a fragmented network is reconnected, mountpoint inodes come in very handy. As soon as the volume manager has a complete image of a volume ready, it can invoke the filesystem recovery manager. As soon as that is done, the filesystem UUID can be announced to the world and operations can start. Note that an administrative “mount” operation is

not required. In the case of reassembling a fragmented network, this technique works especially well, since only the fact that the filesystem UUIDs are now available needs to be transferred and the whole “fragmented” section of the network becomes visible again.

2.2.1.4 IPC Nodes

IPC nodes are complicated beasties. Processes attached to them can:

- Participate in pathname operations via passing pathname fragments in and out.
- Provide “file descriptor” support in that they can respond to filesystem operations (eg: read/write/stat/etc.).
- Lots more neato things....

2.2.2 Pathnames

The pathnames that xFS will support will be 8-bit clean, and will be stored as a (bytes, length) pair, but the interface will still use NULL-terminated strings (ie: the unicode standard will not be supported in this release). We should plan ahead for the possibility of supporting the unicode standard where a NULL byte doesn’t terminate a string.

A pathname will be limited to MAXPATHLEN bytes. Note that in a multi-byte character set, the number of “characters” will be less than the number of bytes. This also applies to the max length of a file name (a pathname component).

The upper layer VFS/VNODE code will need to be able to find “/”, and the xFS filesystem code will need to be able to find “.” and “..”, but that seems to be the only characters that the kernel needs to parse for in the pathname.

2.2.3 Mounting of Filesystems

Until the distributed filesystem support comes along, we may not implement mount-point type inodes, so we will use a traditional *mount* command to initiate kernel activity on a filesystem.

2.2.3.1 MountPoint Type Inodes

This is an inode type that contains the UUID of the new filesystem, and is used something like a traditional symlink. It provides the UUID of the filesystem to continue pathname resolution with.

We must support the library interfaces that let programs read the /etc/mtab file. The filesystems that are currently mounted will depend on other nodes in the local cluster being up, and network connections to more remote systems being up. The list of currently mounted filesystems could be a fairly dynamic thing.

The /etc/mtab file will be managed just the way it is today by non-xFS filesystems, but xFS filesystems will not be recorded in the /etc/mtab file. The library routines will present all of the non-xFS entries in the file, and will then use system calls to query the kernel as to what xFS filesystems are currently mounted.

3.0 External Interfaces

Listed here are the interfaces provided to external callers, the interfaces into other kernel code used by this module, and the dependencies that this module has on the kernel and other modules.

3.1 Supported VNODE Operations

The following VFS/VNODE operations will point into the NameSpace manager code. The semantics of each call are already defined.

- vop_open
- vop_getattr
- vop_setattr
- vop_access
- vop_lookup
- vop_create
- vop_remove
- vop_link
- vop_rename
- vop_mkdir
- vop_rmdir
- vop_readdir
- vop_symlink
- vop_readlink
- vop_pathconf

3.2 IRIX Components Used

Lots of them.

3.3 Dependencies on Other xFS Components

This section lists functional and algorithmic dependencies of the NameSpace manager on other modules in xFS. The NameSpace manager should be able to get all of its work done via calls to Space manager routines, buffer cache routines, and to log/recovery manager routines. It will also need to interact with the Attribute Manager to manage the space inside an inode.

3.3.1 Space Manager

The NameSpace manager is intended to be layered on top of the Space manager.

3.3.1.1 Manage the inode pool

3.3.1.2 Provide the bmap() routine

3.3.2 Log/Recovery Manager

3.3.2.1 Provide Log Write Interfaces

3.3.3 Attribute Manager

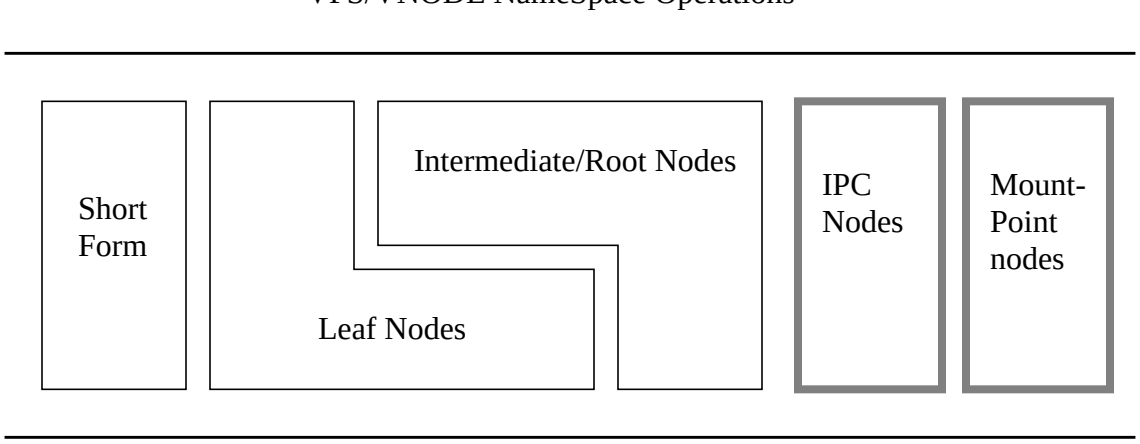
3.3.3.1 Provide a PushOut procedure

text goes here

4.0 Major Components

4.1 List of Components

Here is a partial diagram of the NameSpace Manager components and their interactions:



- Short Form Directory Routines - manage an extremely compact representation of directory entries intended to maximize the number of entries that can fit into the literal space inside an inode.
- Leaf Node Directory Routines - manage the contents of the leaf nodes of a directory. Leaf nodes are used in large directory B-trees and as a special case when the only node is a single leaf node. They are indexed on a hash value computed from the file name.
- Intermediate/Root Node Directory Routines - implement a B*-tree using a hash value computed from the file name as a key, using the leaf node routines defined above to actually store the directory entries.
- IPC Nodes - implement the ability for a user process to be a full participant in namespace operations and in file descriptor based I/O. Used to implement special filesystems in user-mode.
- Mount Point Nodes - used as a symbolic continuation/redirection point in the namespace of a distributed system.

4.2 Internal Interfaces

For each of the three components of the NameSpace Manager that relate to directories:

- There is a *create* routine to build a new block of this type.
- There is an *addname* routine to add a name to a directory with this structure.
- There is a *removename* routine to remove a name from a directory with this structure.
- There is a *lookup* routine to search for a name in a directory with this structure.
- There is a migration routine to the adjacent directory structure. For example: migrating from shortform to leaf node, from leaf node to full B-tree, and equivalent routines for migrating back again.
- There is a *getdents* routine to support readdir() and friends.

The routines listed above are usually just wrappers that call work routines that are used in common by the various components. For example, the internal routine to add a name to a leaf node is called by the leaf node code and by the B*-tree code when it has reached the bottom of the tree.

There may be additional special purpose routines as well as those listed above.

4.3 Algorithms

In this section pseudocode is provided for each possible operation.

4.3.0.1 vop_open, vop_lookup, vop_access, and vop_create

pseudocode

4.3.0.2 vop_getattr and vop_setattr

pseudocode

4.3.0.3 vop_remove

pseudocode

4.3.0.4 vop_link

pseudocode

4.3.0.5 vop_rename

pseudocode

4.3.0.6 vop_mkdir, vop_rmdir, and vop_readdir

pseudocode

4.3.0.7 vop_symlink and vop_readlink

pseudocode

4.3.0.8 vop_pathconf

pseudocode

4.4 Permanent Data Structures (ie: On-Disk)

The Space Manager will split the on-disk inode into three pieces: the UNIX guk, the data fork, and the attribute fork. The UNIX guk is pretty traditional, while the data and attribute fork sections both have the same structure: they will either contain extent pointers, or literal data.

The Space Manager will make the size of the literal area of each fork known to the namespace and attribute routines so that they can use space-efficient optimized structures when their data will fit into the inode itself, and can use time-efficient structures when their data will not fit into the inode. If the namespace code needs more space in the inode, it can call into the attribute manager with a request that the attributes be moved out of the inode and into extents. This is covered in more detail in the External Interfaces sections of this document and the Attribute Manager Design document.

When a directory is small, it is possible to store it inside the inode in place of extent pointers. This has a tremendous appeal in that it will save an IO at something like half of all naming operations. Name caches will alleviate some of the cost of such IOs, at the cost of cache management.

When a directory has too many entries to fit into an inode, there is little choice but to fall back on the familiar scheme of creating blocks of entries stored in a structure that looks like a regular file.

4.4.1 Small Directory Support

We will use a space-efficient structure when we try to fit a directory into the literal area of an inode. The entries will be packed into a flat structure and sorted.

Unlike traditional UNIX directories, when an xFS directory is in this compacted form “.” and “..” will not have explicit representations. The parent directory will simply have a dedicated field, and the self-reference “.” will be calculated on the fly.

The structure for small directory entries is:

parent dir inumber		
count of entries		
I-6523	4	able
I-6546	7	charlie
I-1263	5	baker
I-9833	4	zulu
I-4253	5	delta

```
struct xfs_dir_shortform {
     struct xfs_dir_sf_hdr {
               unchar par-
ent[8];
               unchar count;
     } hdr;
     struct xfs_dir_sf_en-
try {
               unchar inum-
ber[8];
               unchar
namelen;
               unchar
name[1];
```

4.4.2 Large Directory Support

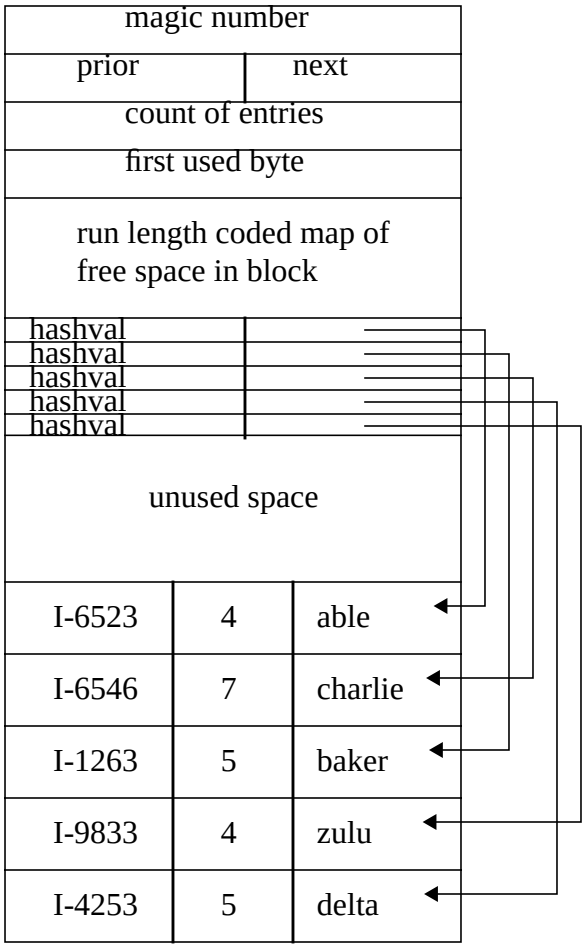
A directory that does not fit into the literal area of an inode will be structured as a B-tree keyed on the hash value of the filenames stored in that directory. The root, intermediate nodes, and the leaves of the B-tree will each be sized to exactly fit into a filesystem logical block. When a single leaf node is sufficient to store the whole directory, there will be no intermediate nodes, just the single leaf node. For our application, it is expected that only in rare cases will directories use more than that a leaf node.

Note that we are using a hash value calculated from the filename as the key into the B-tree and leaf blocks, not the filename itself.

In the B-tree and leaf node directory format, rather than complicate the B-tree structure by having dedicated fields for “.” and “..” as in the compact directory format, we will represent the parent directory, “..”, and the self-reference, “.”, with actual entries.

The B-tree will contain its “data” (ie: the inode number of the referenced object) only at the leaf level of the tree. This is known more formally as a B*-tree (or sometimes as a B+-tree). This differs from a plain B-tree in that all the keys are in the leaves, rather than spread through the tree. Since all keys are in the leaves, the leaf nodes can be linked together to give quick sequential access to all the keys in the tree.

The structure for each B-tree leaf block in a large directory is:




```
struct xfs_dir_leafblock {
     struct xfs_dir_leaf_hdr
               ushort magic;
               xblkno forw;
               xblkno back;
               ushort count;
               ushort
firstused;
               ushort pad1;
               struct xfs_-
dir_leaf_map {
                         ush-
ort base;
                         ush-
ort size;
               } freema-
p[LEAF_MAPSIZE];
     } hdr;
     struct xfs_dir_leaf_entry
{
               uint hashval;
               ushort nameidx;
               ushort pad2;
     } leaves[1];
     struct xfs_dir_leaf_name
```

Packed at the front are the hash value and offsets of each of the strings. The entries are sorted on hash value. Packed at the back are the inode numbers, string lengths, and the strings themselves.

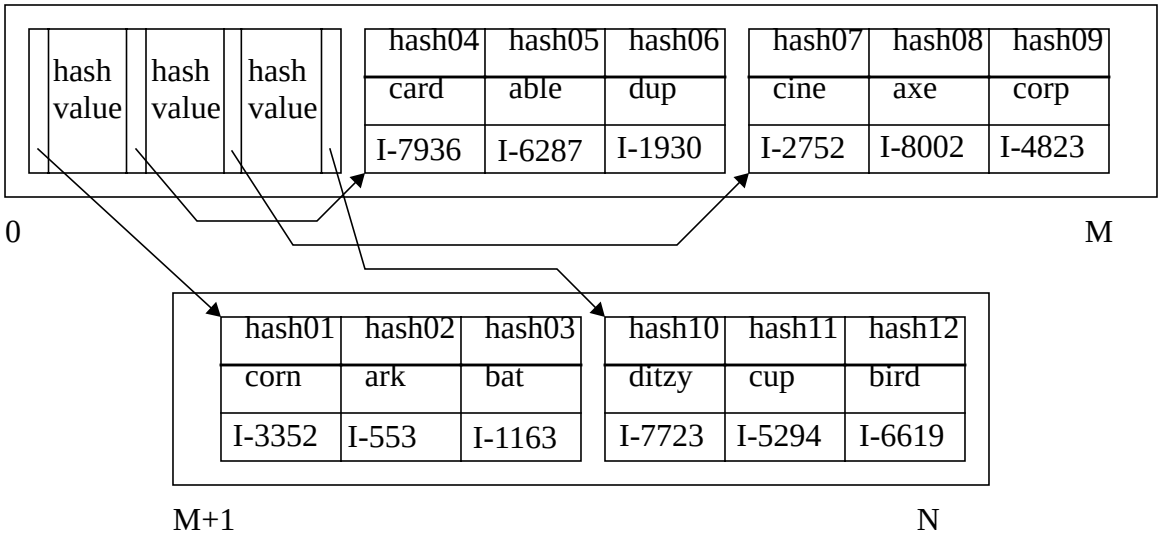
The structure for the B-tree root or the intermediate nodes in a large directory is:

magic number	
leaves next	
count of free blocks	
free block chain	
node after all entries	
count of entries	
hashval	block number
hashval	block number
hashval	block number
hashval	block number
unused space	

```
struct xfs_dir_intnode {
     struct xfs_dir_int_hdr {
               ushort magic;
               unchar leavesn-
ext;
               unchar free-
blks;
               xblkno
freechain;
               xblkno after;
               ushort count;
               ushort pad1[3];
     } hdr;
     struct xfs_dir_int_entry
{
               uint hashval;
               xblkno before;
               ushort pad2;
```

Packed at the front are the B-tree elements. Each element contains the associated hash value and the block number of the B-tree block that contains all the nodes between this key and the key from the prior entry in this B-tree block. Entries after the last hash value are pointed to by the *after* field in the header structure.

The overall structure of the B*-tree used in a large directory is:



The B-tree is embedded inside a linear array of blocks, ie: a file structure. Pointers to B-tree blocks are relative to the file, not to the filesystem.

4.5 Working Data Structures (ie: In-Memory)

4.5.1 Inode Table

We will have a traditional incore inode table. The namespace manager will understand that when an inode is marked as containing literal data, it must manage the directory structures (in the compressed format) inside the inode and not in a buffer.

Since we have a split inode and space is being shared between the primary fork and the attribute fork, the namespace manager has the right to call a function in the attribute manager asking that any attributes stored in the literal area of the inode be moved out into newly allocated blocks. This routine will be called if a small directory (ie: in the inode) grows large enough to need the space occupied by the attributes in the inode, but not too large that it won't fit into the whole literal area of the inode.

4.5.2 Directory Structure

The directory structures for both small and large directories have already been described. The namespace manager code will use those structures either in the incore inode itself, or in buffers that have been read from disk.

4.6 Performance Characterization

4.6.1 Inode Size and Structure

This table compares the approximate storage efficiency of small structured directories (ie: inside the inode) for various sizes of inodes for a typical kernel source tree on campus. This table assumes that an inode will contain 64 bytes of fixed fields and have 9 bytes of overhead per entry.

TABLE 1. Small Format Directory Efficiency

	Dirs that fit
128B inode	27%
256B inode	62%
512B inode	85%
1024B inode	95%

The size of the inodes in a filesystem will be stored in the superblock for that filesystem. In the first release we may only support a single inode size, but we want the flexibility in the data structures to allow us to have different inode sizes per filesystem.

4.6.2 Small Directory Structure

The efficiency of not having to seek the disk heads out to read another block before we can access the directory will be a big win. For a large percentage of directories, simply reading the inode will gets us the contents of the directory and allow us to continue the naming operation. This table assumes that an inode contains 64 bytes of fixed fields.

TABLE 2. Number of User-Settable Keys in a Small Directory versus Inode Size

	128B inode	**256B inode**	**512B inode**	**1KB inode**
8 Byte Filenames	3	10	25	55
32 Byte Filenames	1	4	10	23

4.6.3 Large Directory Structure

A large directory will be structured as a B*-tree keyed on the hash value of the filenames. A B*-tree improves on a plain B-tree in that since all the keys are in the leaves, those leaves can be linked together to give quick sequential access to all the keys in the tree. The access time to find a give key in a B*-tree is a logarithmic function of the directory size, not a linear relationship as in most other UNIX namespace schemes.

For our application, it is expected that only in rare cases will directories use more than a single leaf node.

TABLE 3. Number of User-Settable Keys in a Large Directory Leaf versus Logical Block Size

	512B LBS	1KB LBS	2KB LBS	4KB LBS	8KB LBS
8 Byte Filenames	19	39	80	162	326
32 Byte Filenames	9	20	41	82	166

4.6.4 Directory Inode Allocation Policy

One possibility for improving naming efficiency is to have the inode allocation policy try to cluster directory inodes within the inode list of an AllocGroup. The intent is to increase our odds of reading in about-to-be-accessed inodes along with the inode of current interest. It would increase our cache hit ratio.

4.6.5 Available Parallelism

Reading/writing a directory entry is impacted by the level of parallelism provided for reading/writing a file because the underlying structure used by the NameSpace Manager for a directory is that of a file.

Directory blocks will live in buffers, buffers are exclusively locked when accessed, and most directory operations will take place under the auspices of the transaction manager (which will hold buffer block locks until the transaction completes), so operations on a single directory inode will essentially be single threaded.

4.6.6 Logging Bandwidth Required

Directory operations will usually require one of:

• log the directory inode with the literal directory inside, or
• log the changed block(s) in the directory B-tree.

In the first case, we will log the whole inode, while in the second case we will log as many full blocks as have changed in the operation.

Note that directory operations are usually a part of a larger operation such as removing a file. In such a case, an inode would be modified as part of the same transaction, thereby requiring more log bandwidth than just the operation on the directory per se.

4.6.7 Effect on Disk Seek Patterns

Obviously, when the directory is literally inside the inode, there is no impact on disk seek patterns (other than the lack of a required seek to access a data block).

When the directory is not literally inside the inode, a disk seek and block read will be required. To be more specific:

• a seek and read of the whole first extent of the directory

If the desired filename is not in the first extent (unlikely), more seeks and reads will be required.

4.7 Initialization Procedure

Mkfs will create the *root* and *lost+found* directories when it lays out the filesystem.

4.8 Logging Actions

4.8.1 Normal Operation

4.8.1.1 Types of Log Records

4.8.2 Recovery

4.8.2.1 Actions For Each Type of Log Record

4.9 Disk Transfer Policies

4.9.1 To Disk

4.9.1.1 Xfer size

4.9.1.2 Logging

4.9.2 From Disk

4.9.2.1 Xfer sizes

5.0 User Interface

5.1 System Calls

5.2 Utilities

6.0 Implementation Plan and Schedule

6.1 Features Not in Version 1

6.1.1 IPC Nodes

IPC nodes may not appear in the March release.

6.1.2 Mount Points

Mount point nodes may not appear in the March release.

7.0 Initial Test Plan