xFS Attribute Manager Design

Curtis Anderson

1.0 Introduction

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

The Attribute Manager is accessible via extended VNODE calls and provides the ability to attach a (name, value) pair to any filesystem object. This is accomplished by using the split-inode support in the Space Manager and storing specialized structures in the second “fork”. The names use the same character set as pathnames do, while the values are short arrays of arbitrary bytes.

The reader will notice the extreme similarity between the Attribute Manager and the NameSpace 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 attribute 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 attribute-related entry points and arguments have to be similar to, but an extension of, the existing VFS/VNODE interfaces.

2.1.2 Fast for Small Attribute Lists

The attribute operations must be relatively fast for small numbers of attributes, but operations with large numbers of attributes are not required to be fast. In fact, the proposed implementation will have nearly the same performance characteristics as that of directories, but there is still no commitment to fast access when the number of attributes grows large.

2.1.3 Associated with all NameSpace Objects

It should be possible to associate arbitrary attributes with any type of namespace object, including directories and symlinks.

2.1.4 Internationalization

It should be possible to store attributes whose names are in an international character set.

2.1.5 Location Independence (Distributed Naming)

Hooks must be available to support the planned distributed filesystem. This implies that attributes must be accessible remotely just as they are accessed locally.

2.1.6 SGI-Enhanced NFS Support

It is highly desirable that SGI’s superset of the NFS protocol (used between consenting SGI systems) support attribute operations.

NOTE: The enhanced protocol has not been defined yet and there is some concern about the relative timing of the enhanced NFS versus the distributed filesystem support that will appear as part of xFS in the future.

2.1.7 Size of Attribute Values

An attribute should be able to hold a value of up to 2KB.

NOTE: There may be a call for larger values, possibly up to 16MB, but this is being resisted.

2.1.8 ACLs

The requirement for Access Control Lists (ACLs) is not limited to US Military Orange Book security, but should also accommodate the requirements of similar “commercial security” implementations.

ACLs and other security-related tags should be treated as special forms of arbitrary attributes.

2.2 Functionality (ie: Attribute Semantics)

This section describes what attributes are and what they can do. The implementation of attributes is described in a following section.

Arbitrary attributes are (name, value) pairs that applications and/or the system can associate will filesystem objects. One example is tape header information for files that have migrated to off-line storage. Another example is the character set used to encode text within a file.

Equivalents of the timestamps, etc, will not be maintained for the individual attributes (eg: there will be no indication of when a particular attribute was last changed).

2.2.1 Attribute Naming

Attribute names will be composed of exactly the same character sets as file names, and will have the same length restrictions. Attribute names will support international character sets, subject to the above length restrictions. The intention is that people in Japan can use Kanji names for their attributes.

All attribute names should be stored as a byte string and a length; in the long term, we cannot depend on a NULL character terminating a string. Current interface styles require NULL terminated strings, but the on-disk structures should support (text, length) pairs.

Name conflicts between different users of attribute names (applications) will be handled the same way that filename conflicts between different applications is handled now: the last one overwrites the previous ones. It is up to the application to provide a sufficiently differentiated attribute name such that other well-behaved applications will not stomp on it. For example: “SGI_toto_...” could be used as a prefix for all attribute names used by the “Toto” package produced by SGI.

2.2.2 Attribute Access Model

The model used to access attributes can be called the “small fixed data” model. In this case, attributes are second class objects in the filesystem, and do not appear in the namespace. The value of an attribute can only be a small object of a few kilobytes or less.

In this model, the way that attributes are accessed and managed is very similar to that way that the stat() system call works. A get/set/delete operation is performed on the pathname or open file descriptor of interest. One argument to the operation is the attribute name, while the other is a pointer to a block of memory that contains/will-contain the complete image of the value of the attribute being operated on.

Existing file attributes for the “data fork” such as owner, group ID, timestamps, etc. might be made available through this type of interface.

The list of attributes that are attached to an object can be obtained with functions that parallel the semantics of readdir() and friends for directories.

2.2.3 ACLs

ACLs and other security-related items that are usually associated with namespace objects will be treated as special forms of arbitrary attributes. They will be accessed in the same way as the existing file attributes.

If the user has permission to read the existing UNIX attributes (eg: permissions bits), then the user will have permission to read the names and values of all the arbitrary attributes associated with an object. If the user has permission to write the existing UNIX attributes (eg: permissions bits), then the user will have permission to create, modify, or delete arbitrary attributes associated with an object.

NOTE: is this too limiting? What about groups working together on files that have 0777 permissions, shouldn’t anyone be able to change an attribute value?

2.2.4 Filesystem Level Attributes

There will be some system-predefined attributes that will be associated with the filesystem itself instead of any particular object in that filesystem. They will be available as attributes of any given object within the filesystem, but will require different syscall arguments to obtain than the attributes of that object. Note that these are not user-defined, the names are compiled into the kernel.

This parallels the use of statfs() where superblock information is available via any object in the filesystem.

2.2.4.1 Root vs. Non-Root NameSpaces

To facilitate the implementation of trusted system services that depend on the values of arbitrary attributes, there must be attributes that only a superuser application can manipulate (create, delete, or modify), independently of the permissions of the object they are attached to. However, if the object is unlinked, then all attributes would be unlinked with it.

The chosen technique is to completely segregate normal arbitrary attributes and only-root-modifiable attributes into separate namespaces. Given that normal user attributes and only-root-modifiable attributes are in separate namespaces, there will be no possibility of name collisions or user misuse of trusted attributes.

The interface to attribute operations will include an indication of what namespace is being accessed. At this time, we only know of root versus non-root but there may be call for more capabilities and therefore more namespaces. Note that this is orthogonal to filesystem level attributes.

2.2.5 Searching for Inodes with Specific Attributes

It would be a nice feature for desktop users (for example) to find the pathnames of all files in a filesystem that have the given attribute, or the given value for the given attribute. Fortunately, this is being left up to user mode applications to implement, ie: it is not in the kernel and not in this project.

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 newly defined VFS/VNODE operations will point into the Attribute manager code. Since they are new, the semantics of each call are not yet defined in detail.

- vop_attribute_list - return a list of all attribute names in the way that getdents() works.
- vop_attribute_get - read an attribute and value.
- vop_attribute_set - write (possibly creating) an attribute and value.
- vop_attribute_create - create an attribute and value, fail if attribute already exists.
- vop_attribute_remove - remove an attribute.
- vop_attribute_multi - take a list of attribute operations and loop across them all.

3.1.1 IRIX Components Used

Lots of them.

3.2 Dependencies on Other xFS Components

This section lists functional and algorithmic dependencies of the Attribute manager on other modules in xFS. The Attribute 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.

3.2.1 Space Manager

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

3.2.1.1 Manage the inode pool

3.2.1.2 Manage the resource/data fork split in the inode

3.2.1.3 Provide the bmap() routine

3.2.2 Log/Recovery Manager

3.2.2.1 Provide log write interfaces

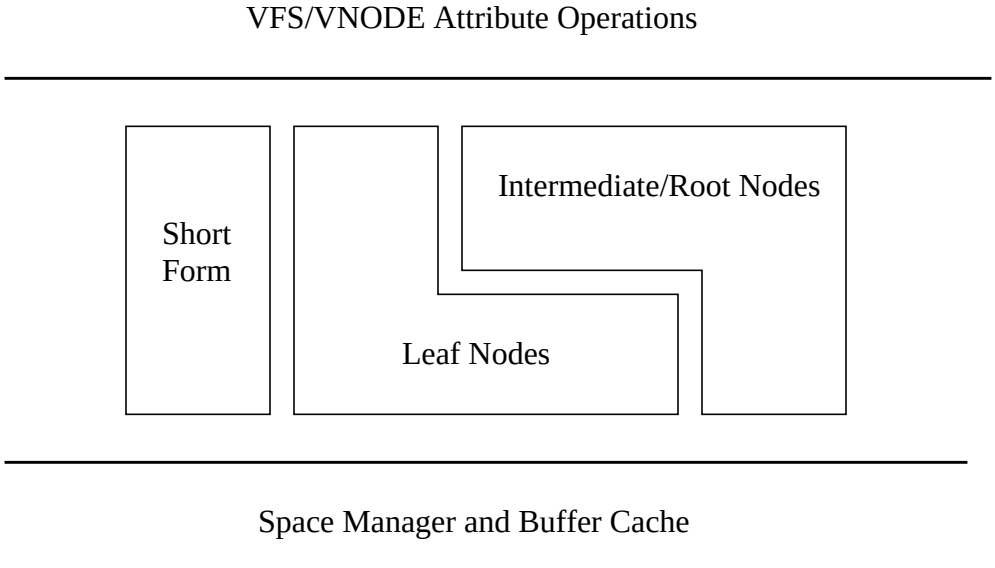
3.2.2.2 Call back on log recovery

text

4.0 Major Components

4.1 List of Components

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



- Short Form Attribute Routines - manage an extremely compact representation of attribute entries intended to maximize the number of entries that can fit into the literal space inside an inode.
- Leaf Node Attribute Routines - manage the contents of the leaf nodes of an attribute list. Leaf nodes are used in large attribute list B-trees and as a special case when the only node is a single leaf node. They are indexed on the attribute ID number.
- Intermediate/Root Node Attribute Routines - implement a B*-tree using the attribute ID number as the key, using the leaf node routines defined above to actually store the attributes.

4.2 Internal Interfaces

For each of the three components of the Attribute Manager

- There is a *create* routine to build a new block of this type.
- There is an *addname* routine to add a name to a block with this structure.
- There is a *removename* routine to remove a name from a block with this structure.
- There is a *lookup* routine to search for a name in a block with this structure.
- There is a migration routine to the adjacent block structure. For example: migrating from short form to leaf node, from leaf node to full B-tree, and equivalent routines for migrating back again.
- There is a *getdents* routine to return entries from the attribute list in sequence. This is equivalent to what readdir() and friends do for directories.

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 an attribute 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 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 below.

When an attribute list 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 quite a few attribute operations. Caches will alleviate some of the cost of such IOs, at the cost of cache management.

When an attribute list 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.

4.3.1 Filesystem Level Attributes

One inode in the filesystem as a whole must be partially dedicated to attribute storage. Those filesystem level attributes that are writable and have persistent storage will be stored as attributes on inode number 0. They will use the same structures and algorithms as other arbitrary attributes.

The data fork of this inode is available for some other use.

4.3.2 Root vs. Non-Root NameSpaces

To implement separate namespaces, we will have to differentiate the attribute names in each namespace. We will add a non-legal character to the end of all root-only attribute names, thus ensuring that they do not collide with a user attribute of the same name and have a unique hash value to ease searching.

4.3.2.1 Small Attribute List Support

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

The structure for small attribute list entries is:

count of entries			
3	5	app	Excel
4	5	lang	ASCII
4	12	path	/icons/excel
4	5	size	32bit
4	9	priv	eyes-only

```
struct xfs_attr_shortform {
   struct xfs_attr_sf_hdr {
      unchar count;
   } hdr;
   struct xfs_attr_sf_entry {
      unchar namelen;
      unchar valuelen;
      unchar name[1];
      unchar value[1];
   } list[1];
};
```

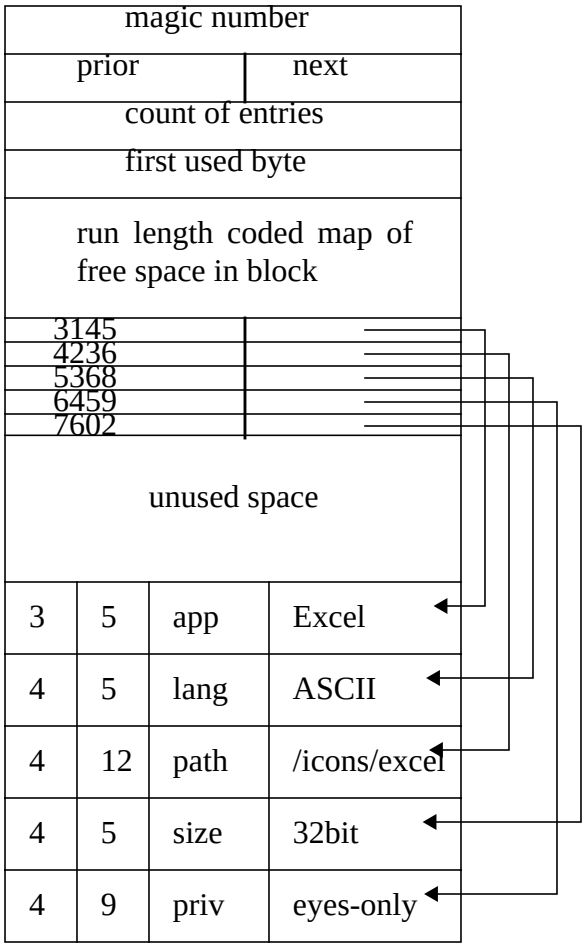
4.3.2.2 Large Attribute List Support

An attribute list 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 attribute name. 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 attribute list, there will be no intermediate nodes, just the single leaf node.

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

The B-tree will contain its “data” (ie: the value of the named attribute) 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. Unfortunately, the attributes are sorted on attribute name hash value, not alphabetically, so an application would probably want to sort them before displaying them.

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




```
struct xfs_attr_leafblock {
   struct xfs_attr_leaf_hdr
      ushort magic;
      xblkno forw;
      xblkno back;
      ushort count;
      ushort firstused;
      ushort pad1;
      struct xfs_attr_leaf_map
{
         ushort base;
         ushort size;
      } freemap[LEAF_MAPSIZE];
   } hdr;
   struct xfs_attr_leaf_entry{
      uint hashval;
      ushort nameidx;
      ushort pad2;
   } leaves[1];
   struct xfs_attr_leaf_name {
      unchar namelen;
      unchar valuelen;
      unchar name[1];
      unchar value[1];
   } namelist[1];
```

Packed at the front are the attribute hash values and offsets of each of the strings. The offsets are sorted on attribute hash value. Packed at the back are the strings themselves.

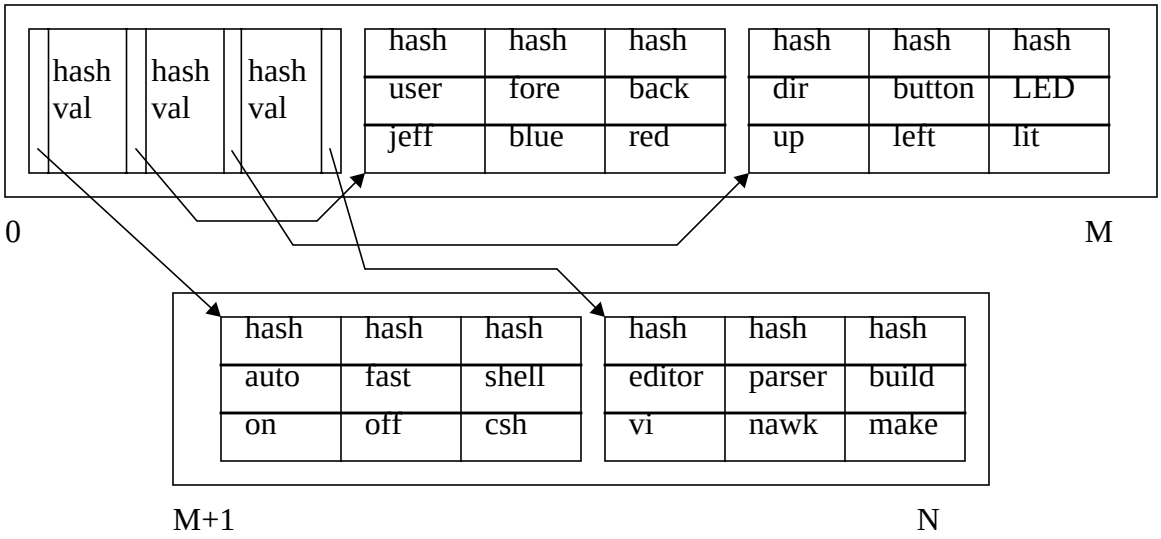
The structure for the B-tree root or the intermediate nodes in a large attribute list is:

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_attr_intnode {
   struct xfs_attr_int_hdr {
      ushort magic;
      unchar leavesnext;
      unchar freeblks;
      xblkno freechain;
      xblkno after;
      ushort count;
      ushort pad1[3];
   } hdr;
   struct xfs_attr_int_entry {
      uint hashval;
      xblkno before;
      ushort pad2;
   } btree[1];
};
```

Packed at the front are the B-tree elements. Each element contains the associated attribute name 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 attribute name hash value are pointed to by the *after* field in the header structure.

The overall structure of the B*-tree used in a large attribute list 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.

4.3.3 ACL Storage

Access Control Lists and other security-related attributes will be stored just like all other attributes on filesystem objects. The fact that the kernel will interpret their values doesn't affect their storage representation.

4.4 Working Data Structures (ie: In-Memory)

4.4.1 Inode Table

We will have a traditional incore inode table. The attribute manager will understand that when an inode is marked as containing literal data, it must manage the attribute 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 and the regular-file write code have 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) or a small file 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.

4.4.2 Filesystem Level Attributes

The filesystem attribute inode will be in the system inode table and the attribute fork will be accessed via the existing attribute access functions.

4.4.3 Attribute Name Spaces for Root vs. Non-Root

There are no special structures for the root-only attribute namespace.

4.4.4 Attribute Structure

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

4.4.5 ACL Storage

Access Control Lists and other security-related attributes will be stored just like all other attributes on filesystem objects. The fact that the kernel will interpret their values doesn't affect their storage representation.

4.5 Algorithms

In this section pseudocode is provided for each possible operation.

• VOP_ATTRIBUTE_LIST

`pseudocode`

• VOP_ATTRIBUTE_GET

`pseudocode`

• VOP_ATTRIBUTE_SET

`pseudocode`

• VOP_ATTRIBUTE_CREATE

`pseudocode`

• VOP_ATTRIBUTE_REMOVE

`pseudocode`

• VOP_ATTRIBUTE_MULTI

`pseudocode`

4.6 Performance Characterization

4.6.1 Inode Size and Structure

It is expected that almost all namespace objects will have either no attributes or many attributes, not just one or two. The desktop process (“WorkSpace”) will be attaching several attributes to most files, character set encoding information may be attached to files for the international markets, and backup/restore/hierarchical-storage management processes will be storing information in attributes.

The size of the inodes in a filesystem will be at mkfs time, per filesystem. This impacts the percentage of inodes where the attribute list can be stored inside the inode. We should figure out the expected size of the attribute stream for a “normal” file before we decide on a default inode size for the personal workstation class of machines.

4.6.2 Space Required for Attribute Names and Values

Quite a bit of space will be required on disk to store however many attribute names plus attribute values there are on every object. There will probably be a relatively few distinct attribute names, but with a very high replication factor.

In the future, this can be changed by the addition of a per-allocation-unit repository of the attribute names that are used in that AU. This change would be transparent to users and could be migrated to via an off-line filesystem modification utility. It would save most of the space occupied by the multiple copies of the attribute names, but at the cost of adding a potential performance bottleneck.

4.6.3 Scaling of Attribute Operations

Since there is no central resource being contended for here, there should be no scaling problems.

4.6.4 Access Time for Security Attributes

Security related attributes will be accessed on most (if not all) file operations in a secure environment. Their performance will be an optimized example of accessing a set of arbitrary attributes at every file operation.

4.6.5 Filesystem Level Attributes

These are all compiled into the kernel, but use the existing functions and interfaces and so should have the same performance characteristics as normal user attributes.

4.6.6 Attribute Name Spaces for Root vs. Non-Root

There will be no difference in performance for accessing the attribute list for normal attributes versus accessing the attribute list for root-only attributes.

4.6.7 Small Attribute Structure

The efficiency of not having to seek the disk heads out to read another block before we can access the attribute list will be a big win. For small attribute lists, simply reading the inode will gets us the contents of the attribute list and allow us to continue the operation. That percentage depends completely on the size of the inode, the size of the data fork, and how many attributes there are.

4.6.8 Large Attribute List Structure

In a B*-tree, all the keys are in the leaves so those leaves can be linked together to give quick sequential access to all the keys in the tree. Unfortunately, the attributes will be sorted on the hash value of their names rather than alphabetically so applications will probably want to sort the list before displaying it.

The access time to find a give key in a B*-tree is a logarithmic function of the attribute list size, not a linear relationship. For our application, it is expected that only in rare cases will attribute lists use more than a leaf node. This is subject to verification, however.

4.6.9 Available Parallelism

Reading/writing attribute is impacted by the level of parallelism provided for reading/writing a file because the underlying structure used by the Attribute Manager for an attribute list is that of a file.

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

4.6.10 Logging Bandwidth Required

Since normal attributes will use the same block structuring as directories, the amount of log bandwidth required per attribute operation should be the same as the equivalent operation for a directory, either:

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

4.6.11 Effect on Disk Seek Patterns

Obviously, when the attributes are 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 attributes are 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 attribute list,
• if the desired attribute is not in the first extent (unlikely), more seeks and reads will be required.

4.7 Initialization Procedure

Mkfs will create the initial contents of the attribute fork (nothing) on each inode and will create the filesystem level attribute fork on the reserved inode.

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 Call Interface

• *attr_list*() - a relative of *getdents*() that returns attribute names and sizes for an object.
• *attr_get*() - return the value associated with the given attribute name for the given object.
• *attr_set*() - set/create the given (name, value) pair on the given object.
• *attr_create*() - create a (name, value) pair on an object, fail if name already exists.
• *attr_remove*() - remove the given attribute name from the given object.
• *attr_multi()* - take a list of attribute operations and loop across them all.

5.1.1 Attribute Operation Arguments

The *flags* arguments for the following calls include:

```
#define ATTR_DONTFOLLOW  0x01        /* do not follow symlinks */
#define ATTR_NOCREATE    0x02        /* don't create on set op */
#define ATTR_FILESYSTEM  0x10        /* incl filesystem attrs */
#define ATTR_USER        0x20        /* incl user attrs */
#define ATTR_ROOT        0x40        /* incl root-only attrs */
```

They mean:

• ATTR_DONTFOLLOW - don't follow symlinks, use on any operation that takes a pathname.
• ATTR_NOCREATE - don't auto-create, fail if name doesn't exist on a set operation.
• ATTR_FILESYSTEM - include filesystem attributes in namespace to operate on.
• ATTR_USER - include local-to-object non-root attributes in namespace to operate on.
• ATTR_ROOT - include local-to-object root-only attributes in namespace to operate on.

5.1.2 Single Attribute Operations

The single-attribute operations include:

5.1.2.1 attr_list()

```
int attr_list(char *path, struct attr_list_struct *list, int len,
              int flags);
int attr_listf(int fd, struct attr_list_struct *list, int len,
              int flags);
```

where:

```
struct attr_list_struct {
     int valuelen;       /* length of associated value */
     int namelen;        /* length of name (including NULL) */
     char name[1];       /* text of name (NULL terminated) */
};
```

Like *getdents()*, successive calls will return more of the attribute list. There are also related routines: *attr_open()*, *attr_rewind()*, *attr_close()* that work just like their *getdents()* equivalents.

The *attr_list()* calls take any combination of the *ATTR_FILESYSTEM*, *ATTR_USER*, and *ATTR_ROOT* flags. The namespaces will be listed in the order: user, root, and then filesystem, but note that there is no indication of where a given attribute name came from when more than one namespace is specified.

5.1.2.2 attr_get()

```
int attr_get(char *path, char *attrname, char *value, int *len,
                int flags);
int attr_getf(int fd, char *attrname, char *value, int *len,
                int flags);
```

For the *attr_get()* calls, the *length* argument initially contains the size of the allocated *value* buffer. After the syscall, the *length* argument contains the actual length of the associated value. If the initial *length* is not sufficient to hold the attribute's value, the *length* argument is set to the required size and an error is returned.

The *attr_get()* calls take any combination of the *ATTR_FILESYSTEM*, *ATTR_USER*, and *ATTR_ROOT* flags. The namespaces will be searched in the order: user, root, and then filesystem.

5.1.2.3 attr_set()

```
int attr_set(char *path, char *attrname, char *value, int len,
                int flags);
int attr_setf(int fd, char *attrname, char *value, int len,
                int flags);
```

The *attr_set()* calls take only one of the *ATTR_FILESYSTEM*, *ATTR_USER*, and *ATTR_ROOT* flags. If one is not specified, *ATTR_USER* is assumed. Note that to set or create an *ATTR_ROOT* attribute requires superuser permissions.

The *ATTR_NOCREATE* flag may also be set. If so, the call will fail if an attribute with this name does not exist in the given namespace.

5.1.2.4 attr_create()

```
int attr_create(char *path, char *attrname, char *value, int len,
                int flags);
int attr_createf(int fd, char *attrname, char *value, int len,
                int flags);
```

The *attr_create()* calls take only one of the *ATTR_FILESYSTEM*, *ATTR_USER*, and *ATTR_ROOT* flags. If one is not specified, *ATTR_USER* is assumed. Note that to create an *ATTR_ROOT* attribute requires superuser permissions.

This call will fail if an attribute with this name already exists in the given namespace.

5.1.2.5 attr_remove()

```
int attr_remove(char *path, char *attrname, int flags);
int attr_removef(int fd, char *attrname, int flags);
```

The *attr_remove()* calls take any combination of the *ATTR_FILESYSTEM*, *ATTR_USER*, and *ATTR_ROOT* flags. The namespaces will be searched in the order: user, root, and then filesystem. Note that to remove an *ATTR_ROOT* attribute requires superuser permissions.

This call will fail if an attribute with this name does not exist.

5.1.3 Multi-Attribute Operations

There is also a multi-attribute operation where an array of sub-operations is passed in. Each operation has its own opcode, arguments, and return value. The argument/result structure follows:

```
struct attr_multi_op {
     int operation;      /* set/create/remove operation code */
     char *attrname;     /* the attribute name to operate on */
     char *value;        /* the attribute value to use/set */
     int *len;           /* the max/used length of the value */
     int flags;          /* flags for this sub-operation */
     int error;          /* error for this sub-operation */
};
```

The *operation* field can contain the following values:

```
#define ATTR_OP_GET      0x1         /* do an attr_get() */
#define ATTR_OP_SET      0x2         /* do an attr_set() */
#define ATTR_OP_CREATE   0x3         /* do an attr_create() */
#define ATTR_OP_REMOVE   0x4         /* do an attr_remove() */
```

In all of the multi-attribute operations, the fields in each sub-operation may contain the same values (with the same semantics) as the arguments to the corresponding single-attribute operations above.

Where the single-attribute operation would have returned an error code from the function call, the *error* field in the sub-operation structure will contain that error code. The *attr_multi()* function call itself will return an error to the caller only when atomic operations are requested (see the next section).

```
int attr_multi(char *path, struct attr_multi *args, int count,
               int flags);
int attr_multif(int fd, struct attr_multi *args, int count,
               int flags);
```

Note that the *flags* argument to the call itself (not the sub-operation) does not have the same semantics as the *flags* argument to the single-attribute operations (see the next section). The *flags* argument can contain the following values:

```
#define ATTR_ATOMIC      0x1         /* atomic multi-attr op */
```

5.1.4 Atomic Multi-Attribute Operations

If the *ATTR_ATOMIC* flag is set in the *flags* argument to the *attr_multi()* call, then all the sub-operations must be successful or none of them will succeed. In this way, multiple attributes can be set/created/removed atomically. Note that this has no meaning for the *ATTR_OP_GET* opcode.

In the following discussion, please assume that *ATTR_ATOMIC* has been set in the flags argument to an *attr_multi()* call.

If any of the sub-operations is an *ATTR_OP_SET* that has the *ATTR_NOCREATE* flag set and that operation would fail because an attribute with that name does not already exist, then the *error* field is set and the whole *attr_multi()* call will fail before doing any of the sub-operations.

If any of the sub-operations is an *ATTR_OP_CREATE* and that creation would fail because an attribute with that name already exists, then the *error* field is set and the whole *attr_multi()* call will fail before doing any of the sub-operations

If any of the sub-operations is an *ATTR_OP_REMOVE* and that removal would fail because an attribute with that name does not exist, then the *error* field is set and the whole *attr_multi()* call will fail before doing any of the sub-operations.

Note that only *ATTR_OP_SET* sub-operations with the *ATTR_NOCREATE* flag set, *ATTR_CREATE* sub-operations, and *ATTR_REMOVE* sub-operations can cause a failure such that the whole *attr_multi()* operation will fail.

Each of the sub-operations is checked for failure, even if one has already been found that will cause the whole *attr_multi()* call to fail. The error field in each sub-option will be set according to whether that sub-operation would have succeeded or not.

The semantics described for atomic multiple attribute access can be used to:

- atomically set a group of attributes only if they all exist
- atomically set a group of attributes only if another attribute can be removed
- atomically set a group of attributes only if another attribute can be created
- atomically create a set of attributes at the same time
- atomically remove a set of attributes at the same time
- atomically create a group of attributes only if another attribute already exists and can be set
- atomically remove a group of attributes only if another attribute already exists and can be set
- etc...

5.2 Utilities

There will need to be:

- a utility to manipulate arbitrary user attributes in a generic way.
- modifications to ls(1) to show security related attributes and values (at a minimum).
- modifications to backup utilities.
- modifications to file transfer and interchange utilities (tar, cpio, rcp, cp, ...).
- modifications to the SGI-NFS protocol to pass attribute operations between consenting systems.
- etc, etc, etc...

6.0 Implementation Plan and Schedule

6.1 Features in Not Version 1

Arbitrary attributes may not be in the March release.

7.0 Initial Test Plan