xFS Space Manager Design

Doug Doucette

1.0 Introduction

This is the design document for the Space Manager module of xFS. The following areas of the design are described: Functionality and External Interfaces; On-Disk Structures; Internal Components and Interfaces; Internal Data Structures and Algorithms; User Library and Utility Support; Performance Characterization; Implementation Plan and Schedule.

2.0 Functionality and External Interfaces

2.1 Overview

The Space Manager manages the allocation of disk space within a file system. It is responsible for mapping a file (a sequence of bytes) into a sequence of disk blocks. The internal structure of the file system: allocation (cylinder) groups, inodes, and freespace management are controlled by the Space Manager, as well as the above mapping function.

The space layout choices in the design are influenced by the requirements to support very large files and file systems efficiently. The Space Manager is responsible for optimizing the layout of blocks in a file, determining rate guarantee information for each file, and keeping related files close to each other on the disks.

The internal details of space management are hidden from the users and from the Name Manager layer, except that users are allowed to determine whether a file is sufficiently contiguous and if not, to have the file's space be reallocated so that it is more contiguous.

2.2 External Interfaces Provided

All exported interfaces are call-based, not message-based. Control and administration messages may come from another node but will be handled by the system call and administration layer and turned into a local call.

2.2.1 Interfaces to xFS Components

- inode manipulation routines (analogous to iget, iput, efs_iupdat)
- inode allocation and freeing routines (analogous to efs_ialloc, efs_ifree)

2.2.2 Vnode (per-file) Interfaces

- a block mapping and allocation routine (VOP_BMAP). Will accept a starting offset and length, vnode pointer, read/write flag, and return an array of data structures describing the location of the blocks. For writes, the disk space is allocated as necessary.
- VOP_SETATTR for the AT_SIZE attribute, which sets the file size. HSM support would have us add another operation to allow "hole-punching" in a file's address space.
- VOP_INACTIVE can result in the file being removed and its blocks truncated. This invokes the same code that a VOP_SETATTR of AT_SIZE to 0 does.

2.2.3 VFS (per file system) Interfaces

- file system extension and contraction need to be new vfsops routines.
- statistics are handled by VFS_STATVFS.

2.3 External Interfaces Used

2.3.1 Interfaces to xFS Components

- Buffer cache interfaces
- Log Manager interfaces
- Volume Manager interfaces (for rate guarantees?)

2.3.2 Interfaces to IRIX Kernel

- Memory allocation
- Memory mapping interfaces (for VOP_MAP, etc.)
- Locks and semaphores

3.0 On-Disk Structures

This section describes the logical structure and physical layout of all disk space for which the Space Manager is responsible.

3.1 Superblock

The superblock is the root of all file system information. The superblock is located at the beginning of the file system (offset 0); this is a departure from historical behavior of UNIX file systems, which start at offset 512 (typically). To avoid confusion between xFS and EFS file systems, offset 512 of an xFS file system must contain a value other than the magic number value of an EFS file system's superblock.

The superblock contains enough information to find all the other pieces of the file system. There is an incore copy of it which is part of the information belonging to a mounted file system.

The following fields, at least, are present in the superblock:

- xFS magic number
- xFS version
- File system unique id
- Last name file system mounted as
- Logical block size (lbsz, in bytes, 2^9 .. 2^{16})
- Unreliable extent size (in lbsz)
- Physical sector size (bytes)
- Inode size (bytes, 2^7 .. 2^{11}) and some information about how the space in the inode is divided up, such as a minimum size for each of the data and attribute areas of the inode
- Data block allocation mechanism, choice of bitmap or B-tree, maybe others
- Small files allocated in inodes, or not
- Allocation group size (in lbsz)
- Total file system data sub-volume size (in lbsz)
- Total file system unreliable sub-volume size (in extents)
- Logical block number of bitmap for unreliable sub-volume extents
- Logical block number of summary information for unreliable sub-volume bitmap
- Total number of allocated and free inodes
- Total number of free data sub-volume blocks
- Total number of blocks allocated as data and as metadata in the data sub-volume
- Total number of unreliable sub-volume extents free

The only fields which change during normal operation are the statistical fields, containing information used by **df**. Changes to this information must be logged, to keep the file system consistent, if the information is trusted across reboots. Alternatively, the information could be computed at mount time, and never written to disk at all (except possibly at unmount time). This would avoid the overhead of logging changes to the superblock at the cost of scanning all the allocation structures of the file system at mount time. The current plan is to avoid the mount-time scan, and log the superblock changes.

The file system size and total fields also change when the file system is re-sized dynamically; these changes must be logged.

To promote resilience to faults, it is necessary to have multiple copies of the super block information that allows us to find the rest of the file system. We will put a copy of the superblock as it is initially formed with each allocation group, and at the end of the file system.

3.2 Allocation group header

Each file system data sub-volume is divided into allocation groups, each (except possibly the last) having the same size. The size is chosen at the time the file system is created. The size chosen will be in the range of 16MB to 1GB, with the default size being the total file system size divided by eight; subject to the overriding minimum allocation group size, there shall be a minimum of eight allocation groups. There may also be some rounding done to the allocation group size, details to be figured out.

The primary reason to divide a file system into allocation groups is to promote parallelism in space allocation in the file system: locking on allocation information can be done separately per allocation group. This allows improved performance, especially in a multiprocessor. The allocation groups are made a uniform size to make it easier to find them in the event of an unreadable block; if the allocation groups were of a variable size, then either each one would have to be readable to figure out where the next one was, or there would need to be an index containing all the allocation group addresses.

Each allocation group is composed of: a superblock (only the first one gets values updated after file system creation time), at location 0 bytes; an allocation group header (fit into the first logical block along with the superblock data); and a bunch of data pointed at by the allocation group header. All the “pointers” in the file system metadata are 64-bit logical block numbers, with exceptions noted below being 32-bit block numbers relative to the start of the allocation group.

The following fields are present in the allocation group header:

- Allocation group header magic number (for checking)
- Allocation group header version number
- Allocation group sequence number (for checking), starting from 0
- If the bitmap allocation scheme is used, the location (relative block number) and size (lbsz) of the free block bitmap [see section 3.3, “Data block freelist”] and summary information.
- If the two B-tree allocation scheme is used, the location (relative block number) of each of the B-tree roots. We might choose to fit one or both roots into the allocation group header’s logical block.
- Location (relative block number) of the “inode” which contains the inode table; this might instead be stored next to the allocation group header [see section 3.4, “Inode table”]

The number of free and allocated blocks and inodes could be maintained per allocation group. For accuracy the changes would need to be logged, implying extra log activity not otherwise needed to log the allocation group header. Instead, this information will be present only in the superblock, which is sufficient for **df**’s purposes. Alternatively, the information could be ignored on disk, and only used in memory. Further discussion of in-core structures is left for later in the document. For now we assume that per-allocation group information for **df** is not stored on disk.

3.3 Data block freelist

There are two basic schemes under consideration for data block allocation. In both cases, the designs do not keep any information in kernel memory, just in buffers. That is, all information is read from and written to disk, and all changes are logged. This makes the designs scale better with respect to memory usage than designs which gobble up some memory per allocation group.

In the first scheme, a bitmap covers all the logical blocks in the allocation group, including the header information and the bitmap itself. The bitmap is a single extent of logical blocks taken from inside the allocation group's blocks. The bitmap moves only if the file system is extended (for the old last allocation group of the file system) or shrunk (for the new last allocation group of the file system). Bits in the bitmap are set for free blocks, clear for allocated blocks. To avoid having to scan the entire bitmap to find free extents of a given size, additional information is stored. For each block of the bitmap, for each possible extent size in the allocation group that is a power of two (2^k), we keep a count of the number of free extents of size 2^k to 2^{k+1}-1 starting in the block. The "blocks" are file system logical blocks. This information occupies a variable amount of space depending on the allocation group and logical block size of the file system; worst case for 512 byte blocks and 1GB allocation groups is 21KB. For 4KB blocks and 1GB allocation groups the size of the information is 288 bytes (assuming that 4KB is taken as the bitmap block size). Each count entry is 16 bits. The information is ordered so that all entries related to a given size are together. These entries are searched to find a bitmap block that must describe a free extent large enough to satisfy our request. Then the bitmap block is searched to find the starting location; the search must be successful. A slight alteration of this scheme restricts each count to cover only free sections inside a single bitmap block; one scheme will be chosen depending on performance.

In the second scheme, the allocation information is kept in a pair of B-trees. Both B-trees contain as data the pairs (starting free block, free block count) for all the free extents in the allocation group. One B-tree is indexed by the starting free block, the other by the free block count (and secondarily by the starting free block, to make the keys unique). Block allocations first search one B-tree and then update both B-trees in the buffer cache (and log the changes). Once the log entry is made, the B-tree buffers can be released to be written to disk when practical. Assuming that the buffer cache implements basically an LRU scheme for metadata, this means that only blocks which are actually being referenced will be in memory.

The current belief is that scheme one will be implemented first, but both schemes will be implemented eventually. Performance analysis will show us whether both schemes must be retained, and under what circumstances one outperforms the other.

3.4 Inode table

We all dislike the old (UFS, EFS) scheme of having all the inodes in the allocation group be contiguously (and statically) allocated. Therefore, xFS uses a scheme where the inodes are allocated on-demand, in small groups. This implies either that the inodes are stored in a single variable-sized extent, or that there is a high-level index pointing to chunks of inodes. The single-extent scheme is simple but prone to failure of allocation if the file system is fragmented, so we will ignore it. An index scheme could work in one of two general ways: either with fixed-size chunks of inodes and a single-extent index, or with a B-tree (or a sequence of extent pointers) for the inode “file” as is done for regular files (see section 3.7, “Data and Attribute block representation”). We have chosen the latter method.

Each allocation group header contains the root of a B-tree representing the inode space and an inode number: the B-tree represents a “file” which contains all the inode space for the allocation group, and the inode number refers to the first inode in the inode free list for the allocation group. Inodes on the inode free list are linked together (by inode number) through a field in the inode. The “file” containing the inodes is extended when necessary, with preallocation when possible: as is done for any other file, an attempt will be made to extend the inode table contiguously. Note, however, that sequential access to the inode table is rare, so its performance is not that important; the real reason for keeping the number of extents of the inode table small is to keep the B-tree representing it flatter.

The B-tree entries contain the low 32 bits of the inode number for the first inode in the inode extent (see section 3.5, “Inode numbers”), the number of inodes in the extent (always a multiple of the number of inodes in a block), and the relative (to the allocation group header) disk block number of the start of the extent. The B-tree is indexed by the inode number field. This is different from the B-tree used for the bmap function only in the units and the sizes of the fields; the algorithms will be the same.

The issue of having an inode bitmap vs. having a freelist are really an independent dimension in the comparison of inode management schemes. In the bitmap case, allocation and freeing both require changing the inode and the bitmap word. In the freelist case, allocation and freeing both require changing the allocation group header and the inode. The freelist case may require less overall storage, since the freelist pointers are stored in the inodes. Theoretically, the freelist case allows less parallelism than the bitmap case, since the bitmap is broken into pieces which could be locked independently. In practice, this does not appear to be a significant restriction, since inode allocations can proceed in parallel in each allocation group. The bitmap case allows for allocating inodes near each other more easily; it’s not clear how important this is. Therefore we have decided to go with the inode freelist scheme for xFS.

If we decided to use a bitmap scheme instead of a freelist, the most likely design is to scatter inode-sized chunks of the free inode bitmap at the appropriate interval through the inode “file”. For instance, if the inode size is 256 bytes, then at 256 * 8 = 2048 inode intervals, the data at that offset in the inode “file” would be a piece of bitmap instead of an inode. Thus to find a free inode “near” an allocated one, calculate and read the appropriate bitmap block, and look nearby the allocated inode’s bit position.

Another possible design is to have an independent set of extents for the bitmap. Then the whole set of issues about dealing with arbitrary sized bitmaps arises; it might be practical to restrict the bitmap to be a small number of extents, either fixed or variable sized. In any case, more disk I/O is required to manipulate the bitmap unless it’s trivial to find the right bitmap block. The other disadvantage of an unadorned bitmap vs. a freelist, is that it’s harder to find a free inode in the normal case, when most inodes are allocated.

3.5 Inode numbers

Inodes contain the information defining each file, directory, etc. in the file system. Each inode is named by its inode number, or inumber. Inodes are numbered sequentially through the entire file system in traditional UNIX file systems. In such a file system, there are the same number of inodes in each allocation group (cylinder group), and so there is no difficulty in locating a particular inode. In xFS, there is a variable number of inodes in each allocation group, and so having the traditional numbering scheme would mean that there would be great difficulty in translating an inumber to an inode disk address.

The inode number in xFS is divided into two bitfields. The more significant bitfield is the allocation group number, the less significant is the inode number within the allocation group. For the moment, the two bitfields are each 32 bits, and the inumber is thus a 64 bit integer. The difficulty with dividing up a 32 bit integer into allocation group number and inode number is the possibility of making a file system large enough that one or the other would run out of space in the bitfield. Since we want this file system design to be good for ten years or so, we will choose to spend this space in the disk format now.

3.6 Inode disk format

The bulk of the design for the inode disk format is in the Directory Manager and Attribute Manager design documents. The issues to be addressed by the Space Manager design are:

- How fields in the inode describe the space allocated for the (parts of the) file
- The size of the inode, in particular the size of the area dedicated to holding small files and small attribute areas

Each file has two address spaces, one for data and one for attributes. The sizes of the two byte streams are arbitrary and independent from each other. The inode has to contain enough information to find both byte streams. We want “short” files and attribute sections to fit completely inside the inode, in the space where the block pointers would be for a longer file. This gives us a lower bound on the space taken up by those pointers. For symbolic links, the maximum length is 1024 bytes; a typical length is more like 100 to 200 bytes for a long symlink. The other major file type that might easily fit is a directory. It would be nice if a directory with a few short entries would fit in the inode. One reason to keep in mind, that symbolic links and directories are more important here than regular files, is that they are accessed through special calls, not by read/write/mmap.

A total inode size of 512 bytes gives us about 450 bytes of storage, enough to satisfy the above pretty well, assuming that the attributes use negligible space. Going to 1024 bytes starts to waste a lot of storage, unless the attributes are several hundred bytes long on average. Most files in a normal file system will be in one extent, and need very little space to represent their storage. Setting the size to 256 bytes will give us around 200 bytes of storage, which is enough for many symlinks, and even some directories. For now we will assert that the size is selected at file system creation time, and is allowed to vary between file systems.

For the moment we will assume simply that a moderate-sized undifferentiated block of bytes is available to the Space Manager in each inode. It was not an absolute requirement that the inode size be a power of 2, but it does make everything go faster. Since we have a use for the space, we can pad the inode size to be a power of 2 easily.

We will assume that the inode contains flags which tell the Space Manager the current usage of the block of bytes. We need the following information: size of the data section (implies the size and starting offset of the attribute section); flags for each section giving its representation. The representation values are: is contained in the inode; is pointed to by direct extent pointers in the inode; the root of a B-tree which points to the files extents is contained in the inode. Possible additional representation values, should we wish to make one section small, are: inode contains a block number, where the block contains either the direct pointers or the B-tree root. We will not necessarily implement all the possible representations, at first, or ever. See section 3.7, "Data and Attribute block representation".

The split between data and attribute space, in the initial implementation, will want to be done simply, but could become more complex and dynamic in later releases. One possibility for the first implementation is to split the space 50/50 between the two, then apply the "does it fit in the space" algorithm to each independently to determine which representation to use. Another possibility is to place a block pointer for direct pointers to the attribute space (or the root of its B-tree) in the inode (i.e., make it small and fixed size, and use the rest of the space for the data block or its B-tree root). The scheme can be more dynamic, at the cost of a more complex implementation, allowing an arbitrary split between the spaces which changes over time.

The dynamic scheme in the design at the moment is as follows. Each of the data and attribute areas have a minimum number of bytes reserved for them in every inode, namely, the amount needed to represent the data or attributes indirectly (a small B-tree root). The free space, if any, is left between the data and attribute information. The Space Manager block mapping routine will never be called on data or attributes whose representation is "contained in the inode"; the upper layer code is responsible for I/O to those bytes. If the data or attributes are being extended and will not fit with the current representation, one of two things happen:

1. If data is growing, and the data would fit if the attributes occupied their minimum space, then the upper layer calls the attribute layer to remove its direct data from the inode and make it direct. The upper data layer then moves its new data into the vacated space.
2. Otherwise, the growing section won't fit. It must kick itself out of the inode. For data, this means calling the Space Manager to bmap the block of data (which will make a B-tree or indirect representation) then putting the data there (in one log operation). For some kinds of data (e.g. directories, attributes) this may mean a format change, as well: the data representation is block-structured.

3.7 Data and Attribute block representation

The mapping from the address spaces of a file to disk blocks in the volume is implemented either by a B-tree or by a set of extent descriptors. For the B-tree case, the information in each node of the tree is (starting file offset [in logical blocks], starting volume block number, length of extent [in logical blocks]). The B-tree is keyed on the starting file offset field. The root block of the B-tree is stored in the inode; this means that the root block is a different size than the other blocks of the B-tree, which are of the logical block size of the file system.

The B-tree is composed of (potentially) multiple levels. At each level except the root level there are multiple blocks. Each nonleaf block contains keys (the starting file offset values) and pointers to other blocks; each leaf node contains only nodes (as above). The keys and pointers alternate (logically) in each block; when a pointer lies between two keys, the data (starting file offsets) in the block pointed to by the pointer lies between the two keys' values. Each block has K keys and $K+1$ pointers. When a block becomes full and an insertion is necessary, one of two operations is performed. First, a rotation is attempted. Rotation attempts to move overflow nodes to both neighboring blocks. If both neighboring blocks are full, then the block is split: half of the information is moved to a new block, and the parent block points to two blocks instead of one. [Alternatively, two blocks can be split producing three new blocks. This keeps the tree bushier.] The procedure is followed recursively until the root is reached or there is room for the new information in the parent block. Deletion operations do essentially the opposite work if the remaining blocks are not full enough. Search operations find the nearest lower or equal block number in the tree, then check to see if the block requested exists.

For the extent descriptor case, we will have a pair of arrays: one of extent sizes (lbsz, 32 bits) and one of pointers to extents (64 bit volume block numbers). If there is a hole in the file it is represented by a 0 extent pointer. This scheme can be used for all files with a small number of extents. The scheme requires a linear search through the array to find a particular block, since the file offset information is implied by the cumulative sizes. As an alternative, the file offset (another 64 bits) could be stored as well, and we could leave out the zero-pointers for holes in files. This would mean that binary search was possible; on the other hand, fewer descriptors would fit, for the normal case where the files do not have holes. This information is stored in the inode directly. To compress this information down to 128 bits the information can be stored as 21 bits for the size, 52 bits for the volume block number, and 55 bits for the file block number (for instance).

The extent descriptor method is used unless that representation does not fit in the inode.

3.8 Unreliable Sub-Volume Space Management

The unreliable sub-volume is divided into a number of fixed-size pieces. The size is a multiple of the file system block size, and is set at mkfs time, and stored in the superblock. The size is expected to be relatively large, say 1Mbyte or so. Free space in this sub-volume can therefore be represented by a simple bitmap. The bitmap must be reliable and so is stored in the data sub-volume; the superblock points to it. In order to speed allocation further, a count is kept per bitmap block, of the number of bits set. This set of counts is saved as an extent in the data sub-volume, and also pointed to by the superblock.

4.0 Internal Components and Interfaces

- Allocation Group Manager (includes allocation group selection for file and directory creation)
- Block and Extent Allocator (for freespace within an allocation group, or the unreliable sub-volume); space may be allocated anywhere in the object, or near a particular block, or at an exact location
- Inode Allocator (allocation group specified or arbitrary)

- File Space Manager (includes bmap routine)
- Statistics Management (includes VFS_STATVFS and performance monitoring)
- Reorganizer Interfaces
- Interfaces out to Log Manager

5.0 Internal Data Structures and Algorithms

5.1 File System Structure

For each mounted file system there is an incore (allocated) structure containing or pointing to information pertaining to the file system, the *vfs* structure. This structure includes a field *vfs_data* which is private data for the file system implementation. As is done for EFS, this field will point to a structure (*xfs_mount* for xFS, *mount* for EFS) which contains some per-file system information. Included in the *xfs_mount* structure will be the following information:

- pointer back to the vfs structure
- vnode pointer for the block device for the data region of the volume
- vnode pointer for the block device for the log region of the volume
- inode pointer to the incore root inode of the file system
- pointer to the list of incore inodes for the file system
- some fields for quotas; in EFS there are flags, an inode pointer, and a size
- some statistics for our own use in tuning the implementation
- a copy of the superblock structure
- an array of short structures, one per allocation group (struct *cg* in EFS)

5.2 Buffering vs. Allocation

The critical data structures where there is some question about incore (allocated) vs. buffered are inodes and data and inode allocation bitmaps (or alternate data structures). For inodes, there is certainly a precedent for caching them incore, into a fixed-size pool of incore inode structures. The incore inodes contain an ondisk inode as well as pointers and other information. The allocation strategy for this inode pool must be examined. The next question for inodes is whether the B-tree representing the file's on-disk structure is pulled into memory or pulled on-demand into buffers. To allow support of very large files, this will need to use buffers.

The free data block bitmaps (or B-trees) are potentially very large, indicating the use of buffers to reference them. We do not believe we can afford to copy these into allocated memory for large file systems.

The inode allocation structures, while not as large, can probably be buffered as well without a substantial loss in performance. This assumption must be checked against the real implementation, though.

5.3 File Contiguity

One very important algorithm remains to be designed: how file contiguity is achieved. As this is purely a performance matter, the implementation (whatever it is) can be left until the rest of the file system implementation is functional. However, the performance of the file system will be truly awful without some attention paid to this problem.

A first-stage solution is equivalent to that used by EFS, approximately. When space is allocated for a file, extra space is allocated at the end of the extent. The space can be returned if unused when the file is closed, or this can be done by an asynchronous reorganizer process (***fsr***).

A second-stage solution is to delay block allocations until larger groups of blocks are accumulated, then dump the blocks into a single extent if possible. This requires changes to VOP_BMAP and related stuff. The new blocks must be reserved instead of bmap'ed. All the blocks must be pinned in the buffer cache (they cannot be written since no block is assigned), with a vnode/offset assigned to them but no physical volume address. This needs more examination to see what the actual interfaces would be.

6.0 User Library and Utility Support

- mkfs, extendfs, shrinkfs
- file system tuning (tunefs)
- file system reorganizer (fsr)
- file system internals dumping (fsdb or equivalent)
- file system repair program (fsck equivalent) for disaster recovery
- library interface to determine extent structure of a file
- interfaces provided for reorganizer to restructure the file "atomically"; for EFS this is done through the fsctl driver

7.0 Performance Characterization

I'm not comfortable supplying anything here until the other sections are more filled in. The strategy is to make performance reasonable on small filesystems by using simpler algorithms and representations until more complex algorithms are actually needed.

8.0 Implementation Plan and Schedule

- Implement simpler block allocation scheme and all other basic algorithms (inode allocation, bmap) for the simulation version of the file system.
- Implement B-tree version of block allocation, also in the simulation version. The B-tree implementation can improve over time if this helps get the file system released.
- Implement direct (in-inode) file support.
- At some point, switch over to live kernel use. This should happen in November, according to the last schedule we put together.