xFS Project Description

Doug Doucette

1.0 Scope of project

This project is concerned with producing a “next generation” file system for IRIX. Included in this is the underlying volume manager, disk driver, buffer cache and virtual memory support. Also included are additional semantics added for reasons such as standardization, new products, etc.

2.0 High-level goals

- Large systems must be saleable as scientific file and compute servers, as commercial data processing servers, and as digital media servers.
- The same software must be able to run on all supported SGI machines, in particular small machines must be supported well.
- The filesystem should replace EFS completely, i.e. it should do everything that EFS does.
- The volume manager should replace the current two volume managers completely.
- The filesystem must out-perform EFS on benchmarks that represent useful activity.
- The filesystem must support high availability by recovering quickly from failures and by keeping its disk-based data in a consistent state at all times.
- The filesystem and volume manager must support future extensions in certain specific areas, i.e. high availability, distributed file systems, and user transactions.

3.0 Detailed requirements

The goals from section 2 should guide us in determining a detailed set of requirements for the project. These are broken up into groups below. Items in the “implementation” sections are not really requirements in the same sense that the “functionality” items are; they represent our current ideas about how to fulfill the functional requirements. In each section, the items are not in any particular order.

3.1 File system functionality

- Implement asynchronous I/O, direct I/O, and synchronous I/O as is done in EFS, in addition to “normal” (buffered) I/O.
- Efficient support for very large files, where very large means a 64 bit size. There should be little or no performance penalty to access blocks in different areas of the file. Some disk space penalty (for indices, for example) is allowed to increase performance. Linear searches through the filesystem data structures to get to blocks at the end of a large file are unacceptable.

- Efficient support for sparse files. Arbitrary “holes” must be supported, areas of the file which have never been written and which read back as zeroes. The representation must be disk-space efficient as well as cpu-time efficient in retrieval of old data and insertion of new data. There is no requirement to detect blocks of zeroes being written in order to replace them with holes (nor is it forbidden). This capability is important for some scientific and compute-intensive applications, as well as for Hierarchical Storage Management (HSM).
- Efficient support for very small files, under 1kb or so. A normal root or usr filesystem has many such files, as does a filesystem which contains program sources. Most symbolic links fit into this category, as well.
- Efficient support for large directories, both for searches and for insertions and deletions. This implies some index scheme, to avoid linear searches through a long directory.
- The time to recover from failure does not increase with the size of the filesystem. The time is allowed to increase with the level of activity in the filesystem at the time of the failure. The recovery scheme must not scan all inodes, or all directories, to ensure consistency. This implies that consistency is guaranteed by use of a log, since the alternative (synchronous behavior as in MS-DOS) is unacceptably slow.
- Recovery never backs out changes that were “committed” after returning successfully to the user. Some operations must be synchronous, at least as far as the log writes are concerned. Certainly this includes file creation and deletion, and does not include ordinary (buffered) writes. [There is already some disagreement about this section.]
- Supports ACLs and other POSIX.6 functionality. This includes some form of support for Mandatory Access Controls, Information Labeling (?), and auditing.
- Supports extents, logically contiguous regions in a single file. It is not a requirement that the extents be exposed in the programming interface to the user. This is primarily a performance issue but we may choose to make the extent sizes visible or settable per-file. We must also be able to ensure the user that their files are contiguous, implying that there is some way to display layout information, and some way to make a file contiguous (fsr, for example).
- Supports multiple logical block sizes, ranging from the disk sector size up to something large like 64k or 256k. The block size is set at filesystem creation time. It is the minimum unit of allocation in the filesystem.
- Supports multiple physical sector sizes. This allows us to support different disk hardware without a built-in reliance on a particular formatted sector size. Smaller sector sizes yield less total useable disk space, so more efficient use of current disks can be made by increasing the sector size.
- Allow the filesystem to change size on-line, possible automatically as well as by administrative command. The filesystem’s underlying space (volume) can grow, so the filesystem must be able to use the new space. It is also possible to allow communication between the volume manager and the filesystem so that the filesystem will ask the volume manager to grow the underlying volume when the filesystem is getting full. It is not a requirement to allow on-line shrinking of a filesystem; it is a requirement to allow off-line shrinking.

- Allow the separation of filesystem space between inodes and data to change on-line. This implies dynamic allocation of the space for inodes as the only reasonable implementation. Off-line change doesn't imply anything about the allocation mechanisms; they could still be static in that case. Note that any mechanism which yields different numbers of inodes in each allocation group implies some sort of indexing scheme to find the inodes.
- High throughput for file server and compute server applications. In particular, the NFS performance must make our system price/performance competitive. Compute server applications need high single-file throughput.
- Extremely high throughput for video server applications. This means that sequential access to large files must be very fast. This might be done via hints from the application about necessary read performance, rather than by the default mechanisms.
- Fast, guaranteed response time for digital media and other real-time applications. Preallocation of blocks for POSIX 1003.4 functionality. We don't really know how to do this yet. It's possible this should be pushed to user mode, and probable that it won't make our first release.
- High throughput for random access to very large databases, via direct I/O and asynchronous I/O. These applications will want to bypass the buffer cache and implement their own cacheing.
- Backup and HSM interfaces for Epoch and similar systems are supported. File migration and backup tools are supported or supplied by us. Backup tools allow full and incremental backups in a reasonable length of time.
- It must be possible to restore filesystems from backup media after a disaster, in a reasonably short amount of time. File restore must also allow selection of individual files to restore, and must allow the backup media to be remote from the filesystem.

3.2 File system implementation

- The file system is implemented under vnodes, possibly extended from the current ones. Other file systems (excluding EFS) in IRIX continue to run with little or no implementation effort. EFS must continue to run, but may have impaired performance.
- File system is implemented as a "journalled" filesystem. This is implied by the requirement for a small recovery time for large filesystems.
- Implement using message passing and kernel threads. The former allows later distribution in a network. The latter allows greater performance and ease of implementation.
- Implement so that a user-mode "simulation" of the filesystem is functional and usable for debugging and performance modelling.
- Support large, sparse files with a B-tree representation of the data blocks in the files. This makes the performance of these files acceptable. Any equivalent index scheme will do as well.
- Delay allocation of user data blocks when possible to make blocks more contiguous. This allows us to make extents large without requiring the user to specify extent size, and without requiring a filesystem reorganizer to fix the extent sizes up after the fact. Users may still require that certain files be contiguous, and so we will need a file system reorganizer for that purpose.

- Store symlinks and other small files in the inode when possible. By doing so, we save a disk block and the time to read it.
- Support directories with some form of indexed structure, so that searches are faster for large directories. Some form of B-tree will work.
- Support low-power machines with the ability to turn off the disk drives when they are not in use, and turn them on again when needed.

3.3 Volume manager functionality

- Mirroring (plexing) of storage. Flexibility is required, we do not want to require duplexing entire disks, or limiting to two plexes.
- Disk striping. Required for performance on large systems.
- Concatenation of storage sections. Should be able to build arbitrarily large volumes, up to the 64-bit limit.
- On-line re-sizing of volumes. Concatenation can be used for this.
- Separate logging and data sub-volumes. Logging sub-volumes are needed by the filesystem. Each should be sized independently in each portion of the volume.
- High throughput for file server applications. The performance penalty for using the volume manager should be vanishingly small in normal operation.
- Support of RAID devices at their full performance. Implies large transfers generated from the filesystem code, through the volume manager, down to the RAID driver.
- Fast, guaranteed response time for digital media and other real-time applications. We might want to restrict this to certain disk types, if there are disk types we think cannot meet such requirements.
- Support of multiple-access (dual-ported) disk controllers. Necessary for real high-availability applications.
- Dynamic relocation of control for a volume. Necessary for high availability.
- Support multiple logical sector sizes (one per volume). This supports the equivalent filesystem requirement.
- The new and old volume managers (lv, not Veritas) must be able to run in the same system. It is not a requirement to run a Veritas volume in the same system.
- EFS and non filesystem applications must be able to run on top of the new volume manager. Ordinary driver interfaces must be presented to these clients, even if xFS doesn’t use them.

3.4 Volume manager implementation

- Logging of written blocks when volume is incomplete. Necessary for high availability (fast recovery from a disk failure).
- Implement using message passing and kernel threads, to allow later distribution in a network.

4.0 Buffer cache and virtual memory support

- Need to be able to keep blocks from going to disk until prerequisite blocks have gone out. This allows us to keep consistency with the logging filesystem.
- May need to finish the memory mapped files implementation, it is not complete in IRIX 5.0 (memcntl interface is stuffed out). This implies some additional virtual memory work.

5.0 Possible requirements

- Support DFS vnode interfaces. This has to wait until we figure out what is happening with DFS.