Porting the SGI XFS File System to Linux

Jim Mostek, William Earl, and Dan Koren
SGI

Russell Cattelan, Kenneth Preslan, and Matthew O'Keefe
Sistina Software, Inc.

Abstract

In late 1994, SGI released an advanced, journaled file system called XFS on IRIX, their System-V-derived version of UNIX. Since that time, XFS has proven itself in production as a fast, highly scalable file system suitable for computer systems ranging from the desktop to supercomputers. In early 1999, SGI announced that XFS would be released under an open source license and integrated into the Linux kernel. In this paper, we outline the history of XFS, its current architecture and implementation, our porting strategy for migrating XFS to Linux, and future plans, including coordinating our work with the Linux hacker community.

1 Introduction to XFS

In the early 1990's, SGI realized its existing EFS (Extent File System) would be inadequate to support the new application demands arising from the increased disk capacity, bandwidth, and parallelism available on its workstations. Applications in film and video, supercomputing, and huge databases all required performance and capacities beyond what EFS, with a design similar to the Berkeley Fast File System [1], could provide. EFS limitations were similar to those found recently in Linux file systems: small file system sizes (8 gigabytes), small file sizes (2 gigabytes), and slow recovery times using fsck.

1.1 XFS Features

In response, SGI began the development of XFS [2], [3], a completely new file system designed to support the following requirements:

- fast crash recovery
- large file systems
- large sparse files
- large contiguous files
- large directories
- large numbers of files

File systems without journaling must run an fsck [4] (the file system checker) over the entire file system; instead, XFS uses database recovery techniques that recover a consistent file system state following a crash in less than a second. XFS meets the requirements for large files systems, files, and directories through the following mechanisms:

- B+ tree indices on all file system data structures [5], [6]
- tight integration with the kernel, including use of advanced page/buffer cache features, the directory name lookup cache, and the dynamic vnode cache
- dynamic allocation of disk blocks to inodes
- sophisticated space management techniques which exploit contiguity, parallelism, and fast logging.

XFS uses B+ trees extensively in place of traditional linear file system structures. B+ trees provide an efficient indexing method that is used to rapidly locate free space, to index directory entries, to manage file extents, and to keep track of the locations of file index information within the file system.

XFS is a fully 64-bit file system. Most of the global counters in the system are 64-bits in length, as are the addresses used for each disk block and the unique number assigned to each file (the inode number). A single file system can theoretically be as large as 18 million terabytes. The file system is partitioned into regions called Allocation Groups (AG). Like UFS cylinder groups, each

AG manages its own free space and inodes. The primary purpose of Allocation Groups is to provide scalability and parallelism within the file system. This partitioning also limits the size of the structures needed to track this information and allows the internal pointers to be 32-bits. AGs typically range in size from 0.5 to 4GB. Files and directories are not limited to allocating space within a single AG.

Other related file system work in Linux includes [7],[8],[9], [10],[11], [12],[13].

1.2 The XFS Architecture

The high level structure of XFS is similar to a conventional file system with the addition of a transaction manager and a volume manager. XFS supports all of the standard Unix file interfaces and is entirely POSIX and XPG4-compliant. It sits below the vnode interface [14] in the IRIX kernel and takes full advantage of services provided by the kernel, including the buffer/page cache, the directory name lookup cache, and the dynamic vnode cache.

XFS is modularized into several parts, each of which is responsible for a separate piece of the file system's functionality. The central and most important piece of the file system is the space manager. This module manages the file system free space, the allocation of inodes, and the allocation of space within individual files. The I/O manager is responsible for satisfying file I/O requests and depends on the space manager for allocating and keeping track of space for files. The directory manager implements the XFS file system name space. The buffer cache is used by all of these pieces to cache the contents of frequently accessed blocks from the underlying volume in memory. It is an integrated page and file cache shared by all file systems in the kernel.

The transaction manager is used by the other pieces of the file system to make all updates to the metadata of the file system atomic. This enables the quick recovery of the file system after a crash. While the XFS implementation is modular, it is also large and complex. The current implementation is over 110,00 lines of C code (not including the buffer cache or vnode code, or user-level XFS utilities); in contrast, the EFS implementation is approximately 12,000 lines.

The volume manager used by XFS, known as XLV, provides a layer of abstraction between XFS and its underlying disk devices. XLV provides all of the disk striping, concatenation, and mirroring used by XFS. XFS itself knows nothing of the layout of the devices upon which it is stored. This separation of disk management from the file system simplifies the file system implementation, its application interfaces, and the management of the file system itself.

1.3 Support Features

XFS has a variety of sophisticated support utilities to enhance its usability. These include fast mkfs (make a file system), dump and restore utilities for backup, xfsdb (XFS debug), xfscheck (XFS check), and xfsrepair to perform file system checking and repairing. The xfs_fsr utility defragments existing XFS file systems. The xfs_bmap utility can be used to interpret the metadata layouts for an XFS file system. The growfs utility allows XFS file systems to be enlarged on-line.

1.4 Journaling

XFS journals metadata updates by first writing them to an in-core log buffer, then asynchronously writing log buffers to the on-disk log. The on-disk log is a circular buffer: new log entries are written to the head of the log, and old log entries are removed from the tail once the in-place metadata updates occur. After a crash, the on-disk log is read by the recovery code which is called during a mount operation.

XFS metadata modifications use transactions: create, remove, link, unlink, allocate, truncate, and rename operations all require transactions. This means the operation, from the standpoint of the file system on-disk metadata, either never starts or always completes. These operations are never partially completed on-disk: they either happened or they didn't. Transactional semantics are required for databases, but until recently have not been considered necessary for file systems. This is likely to change, as huge disks and file systems require the fast recovery and good performance journaling can provide.

An important aspect of journaling is write-ahead logging: metadata objects are pinned in kernel memory while the transaction is being committed to the on-disk log. The metadata is unpinned once the in-core log has been written to the on-disk log.

Note that multiple transactions may be in each in-core log buffer. Multiple in-core log buffers allow for transactions when another buffer is being written. Each transaction requires space reservation from the log system (i.e., the maximum number of blocks this transaction may need to write.) All metadata objects modified by an operation, e.g., create, must be contained in one transaction.

2 The vnode/vfs Interface in IRIX

The vnode/vfs file system interface was developed in the mid-80s [14], [15] to allow the UNIX kernel to support multiple file systems simultaneously. Up to that time, UNIX kernels typically supported a single file system that was bolted directly into the kernel internals. With the advent of local area networks in the mid-80s, file sharing across networks became possible, and it was necessary to allow multiple file system types to be installed into the kernel. The vnode/vfs interface separates the file-system-independent vnode from the file-system-dependent inode. This separation allows new file systems to re-use existing file-system-independent code, and, at least in theory, to be developed indepently of the internal kernel data structures.

IRIX and XFS use the following major structures to interface between the file system and the rest of the IRIX OS components:

- vfs – Virtual File System structure.
- vnode – Virtual node (as opposed to inode)
- bhv_desc – behaviors are used for file system stacking
- uio – I/O parameters (primarily for read and write).
- buf – used as an interface to store data in memory (to and from disk)
- xfs_mount – top-level per XFS file system structure
- xfs_inode – top-level per XFS file structure.

2.1 vfs

Figure 1 depicts the vfs, bhv_desc, xfs_mount, and xfs_vfsops structures and their relationship in IRIX.

The vfs structure is the highest-level structure in the file system. It contains fields such as:

- the mounted device
- native block size
- file system type
- pointer to first file system (bhv_desc)
- flags

In IRIX, the vfs object points to a behavior (bhv_desc) structure which is used to construct layered file systems for this vfs. The bhv_desc structure has the following fields:

- data (file-system-dependent data, xfs_mount in figure 1)
- vobj (file-system-independent data, vfs in figure 1)
- ops (pointer to file-system-dependent functions)
- next (next bhv_desc in list of file systems for this vfs).

In the example, note that we have one file system layer since the bhv_desc structure's next pointer is NULL.

If some other layered file system was added above XFS, a new bhv_desc would be added in front of the existing bhv_desc for XFS.

2.2 vnode

The IRIX vnode structure is similar to the IRIX vfs structure as can be seen in figure 2.

The vnode structure points at the first behavior in the chain of file systems handling the file associated with this vnode. In figure 2, there is one behavior only: the XFS inode itself. The behavior also points to the function vector, xfs_vnodeops, which contains all the file-system-specific routines at the file level. In IRIX, the vnodeops contains more than 57 routines which can be invoked on a "file". These routines cover many functions such as create, remove, read, write, open, close, and others.

2.3 uio

The uio structure in IRIX is used to pass I/O parameters between the OS and the file system. This structure can be seen in figure 3.

The uio structure can be used to point at multiple different buffers per I/O operation. This can be used to create a scatter/gather I/O interface allowing users to fill different, discontinous memory areas with one system call [15].

The uio structure on IRIX has various other fields that are used by the Virtual Memory (VM) system to communicate with the file system. One such field is uio_segflg, which indicates the different types of memory involved in transfers such as user space, system space, or instruction space. This information is used by the file system when determining how to move data to and from the uio's associated memory.

There are several other fields in the uio which are used to communicate between the file system and the rest of the kernel, including:

- uio_fmode – file mode flags
- uio_offset – file offset

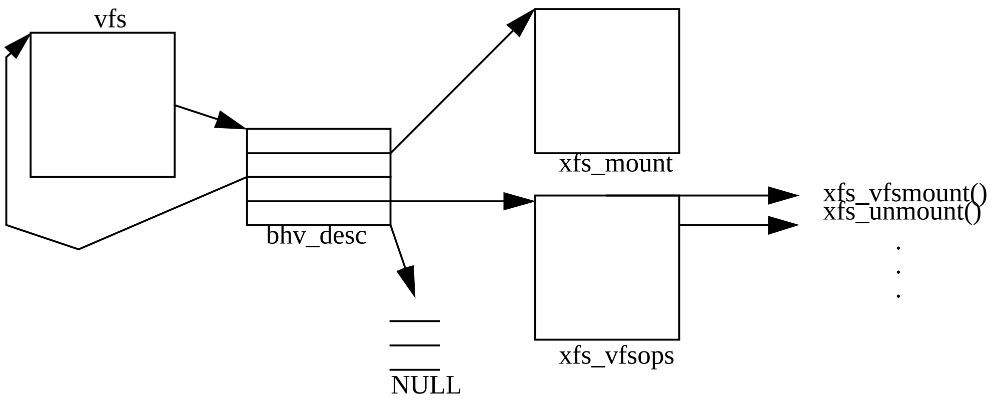
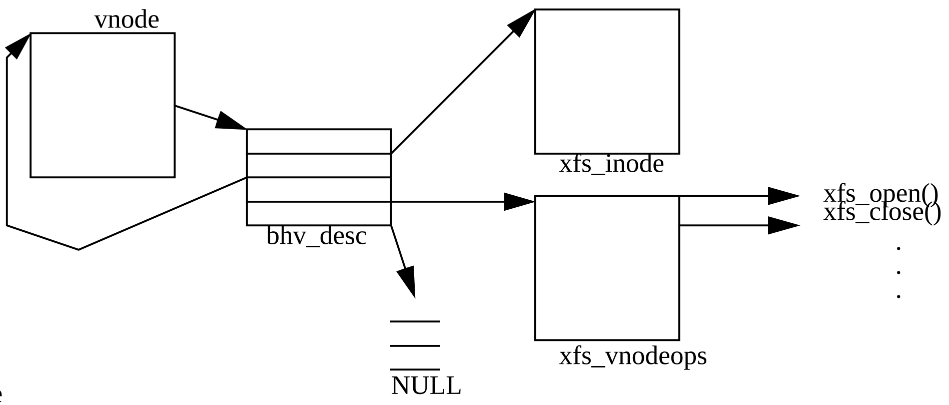


Figure 1: vfs, bhv_desc, and XFS mount relationship.



centerline

Figure 2: vnode, bhv_desc, and XFS inode relationship.

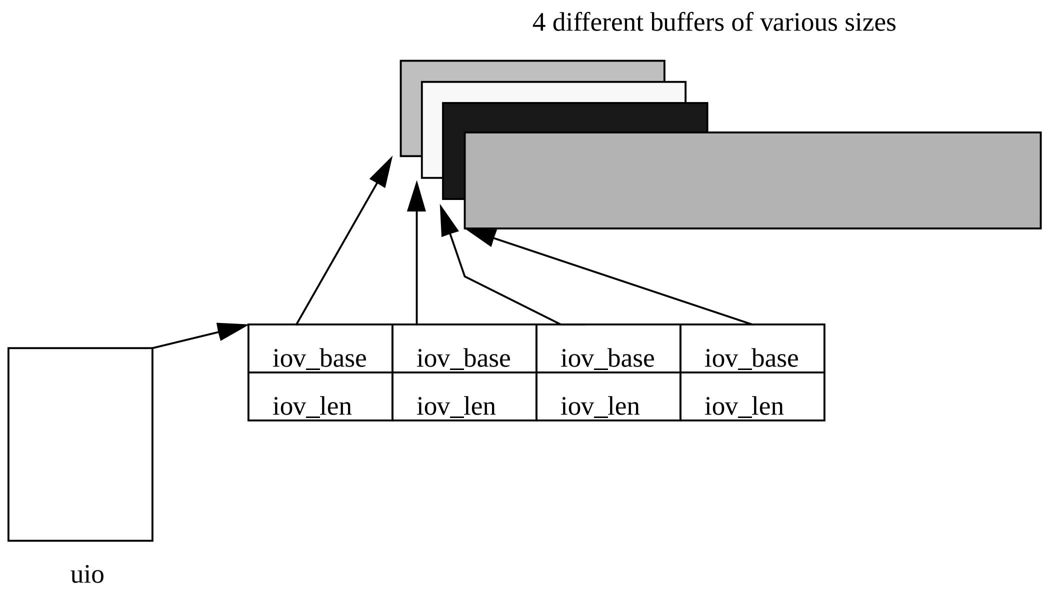


Figure 3: uio structure with 4 distinct memory areas.

- uio_resid – residual count (set by the file system after the I/O is done)
- uio_limit – u-limit (maximum byte offset)
- uio_fp – file pointer

2.4 layered vnode/vfs

SGI has created a clustered version of XFS called CXFS. This file system lets multiple machines share the same XFS file system, much like NFS clients share the NFS server's local file system. The major difference between NFS and CXFS is that the data for I/O goes directly from the disk devices to each machine instead of going through a server like NFS.

The CXFS file system uses behaviors to layer on top of XFS, as shown in figure 4. The dsvn structure is the CXFS layer's inode. It contains information kept private to the CXFS layer, such as which nodes in the cluster are using the file. The dsvnops contains pointers to the file-system-dependent routines in CXFS.

In most cases, each dsvn routine does some work before calls to the next behavior in the chain, in this case, XFS. Theoretically, other layers could be inserted between CXFS and XFS, or above CXFS.

Whenever a vnode needs to have a new layer inserted, a lock is obtained to prevent any operations from "crossing" the behavior. If an operation is currently active, e.g. xfs_read, the insertion must wait until it completes. Some dsvn routines are simply pass through. They just call the next layer in the behavior chain.

2.5 buffer cache/buf structure

One of the fundamental components of IRIX, XFS, and modern file systems is the buffer cache. The buffer cache is main memory maintained by the operating system which contains data being transferred to and from disk. Disk data is cached to help prevent I/O between memory and disks. The basic idea is to keep parts of disk in memory since disk data is often extensively re-used.

The top level data structure for the buffer cache in IRIX is struct buf. This structure contains information such as:

- pointers to the actual memory
- the device and block number with which the buffer is associated
- various flags indicating the state of the memory (dirty, locked, already read, busy, etc.)
- pointers to other bufs
- error state
- size of data
- pinned status
- device-specific information

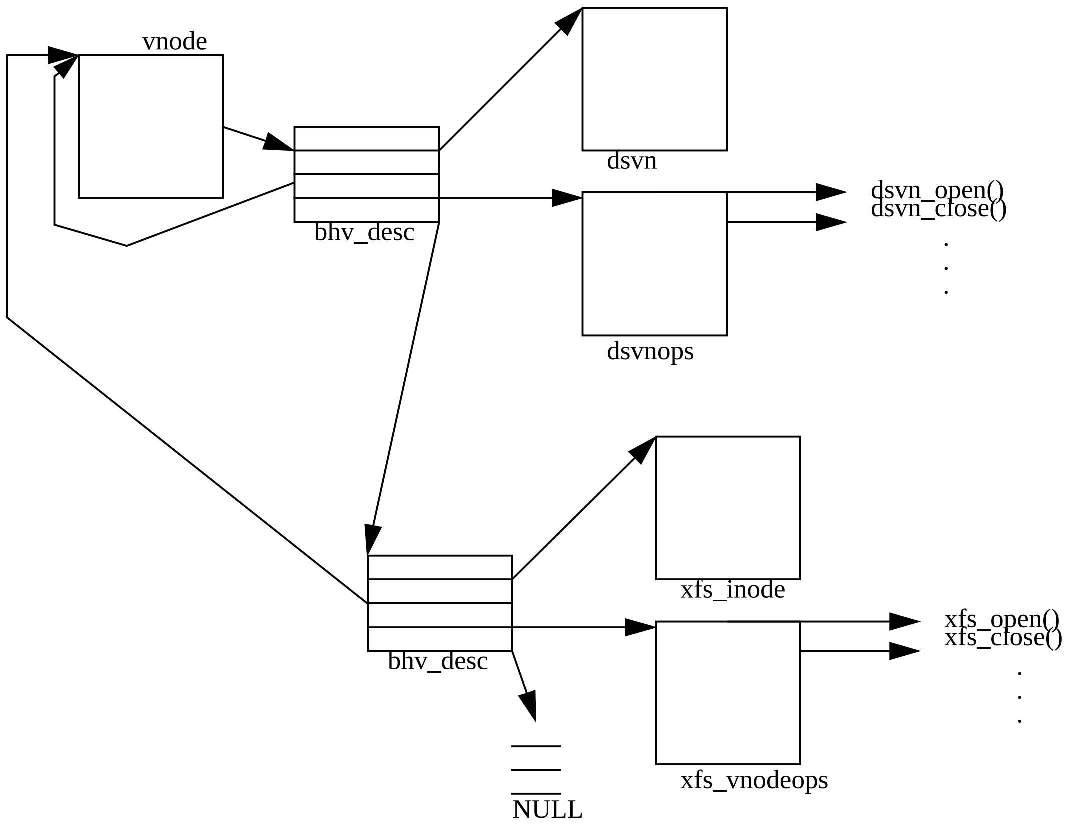


Figure 4: CXFS layered over XFS.

- file-system specific information
- pointer to the vnode

Each buf structure is associated with a device and block number. A major interface routine for the buffer cache is get_buf. It is called to associate a piece of memory and the buf structure with a block number and device, and return the buf structure "locked". If the buf structure already exists, we have a cache hit. If it doesn't exist, a buffer (associated with a different device and block) may need to be flushed from memory and re-associated with the block requested by get_buf. get_buf does not read the disk; instead, bread or breada call get_buf and then read the disk (if the data isn't already there). XFS uses get_buf extensively to associate buffers with meta-data such as inodes, the super block, and the XFS log.

There are various other routines that are used by XFS to interface with the buffer cache including:

- getblk – same as get_buf but with no flags
- bread – get_buf and read the disk if the data is not already there
- breada – bread, but don't wait for the I/O to complete
- bwrite – release the buf, write it, and wait for I/O completion
- bdwrite – release the buf, it will be written before it is reassociated
- bawrite – release the buf, start the write, don't wait
- brelse – release a previously locked buffer (get_buf, getblk, ... lock)
- bpin – pin the buffer (don't let it be written) to disk until unpinned
- bunpin – unpin the buffer and wakeup processes waiting on the buffer.

IRIX was enhanced to have buffer "clusters" above the standard buffer cache to improve write performance [16]. This allows logically contiguous buffers (with non-contiguous memory) to be handled as if they were all physically contiguous. The buf structure continues to be the basic data structure even for clustered bufs. Another structure, bmapval, is used to specify the cluster.

An important capability of the buffer clusters is delayed allocation. Actual allocation of disk blocks for user writes can be postponed or completely eliminated with this functionality. The block number of a buf which is "under delayed allocation" is -1. If the file is truncated before the actual disk allocation, the data never touches the disk. Disk allocation is caused either by a get_buf reassociating a buffer to another device and block number, or by a background daemon which periodically flushes out a percentage of dirty buffers (which can include delayed allocation buffers).

The interface routines for clustering buffers are:

- chunkread – return a buffer cluster, start a readahead, wait for I/O
- getchunk – return a clustered buffer associated with a vnode
- chunkpush – push out buffers in a range for the given vnode.

More details on IRIX clustering and the buffer cache operation are given in later sections.

3 The Linux VFS Layer

In this section we describe the Linux VFS layer [17].

3.1 struct file_system_type

The *struct file_system_type* structure is used to register a particular file system type with Linux. Its elements are:

name The name of the filesystem (ext2, nfs, xfs, ...)

flags These flags describe the type of file system that this structure represents. The most commonly used flag is FS_REQUIRES_DEV which tells Linux that this filesystem must mount a block device. (Networked filesystems such as NFS shouldn't set this flag.)

read super function This is the function that is called to mount a file system of this type.

When a file system module is initialized, it calls the function *register_filesystem* with a pointer to this structure. From that point on, any attempts to mount a file system with that name find that structure. The structure's *read_super* function is called to mount the file system.

The *unregister_filesystem* function is called when the file system module is unloaded from memory.

3.2 struct super

The super block structure, *struct super*, contains fields and operations having to do with the whole filesystem. The major fields in the superblock include:

device name The device structure that describes the device the file system is mounted on. The device may be zero for some distributed file systems.

file system blocksize The Linux VFS knows what size buffers make up the file system.

root dcache entry The pointer to the root Dcache entry. This means that the Linux VFS layer always has a handle on the root inode of the file system. vnode/vfs style operating systems have a vfs operation that returns it.

file system private data The *struct super* contains a C union which is the union of the private data of all the different Linux file systems. This means that memory for the *struct super* and the private data are allocated all at once to reduce memory fragmentation. Installable filesystems can also use a void data pointer in the union, so the kernel doesn't have to know about every filesystem when it is compiled.

super block operations The superblock defines operations that operate on that file system. Some of the operations found here are typical of those associated with a vfs layer, including:

- report block and inode usage information about the file system
- remount the file system
- unmount the file system.

Linux's super block operations are different from the SVR4 vfs operations [15] in that they also include operations to deal with the reading and writing inodes. There are operations to:

- read in an inode given the file system it belongs to and its inode number.
- remove the inode from memory
- deallocte the inode
- modify the "stat"-type information about the inode.

3.3 struct dentry

The Linux VFS layer also exploits the directory cache (dcache). The dcache is a cache of directories and the inodes contained in them. Its function is to cache directory lookups and inodes in memory in order to make directory manipulation operations faster.

When the VFS wants to find a particular filename in a directory, it first does a search of the dcache. If it finds the entry in the dcache, it just uses that. If the entry is not in the dcache, it issues a lookup call to the file system. The new inode is then added to the dcache.

Since hard links are allowed in UNIX file systems, there may be more than one dcache entry pointing to a given inode. Each dcache entry represents a particular path to that inode.

The dcache is made up of *struct dentry* structure. The major members of this structure are:

inode pointer the inode that this dcache entry represents.

parent directory a pointer the the directory containing this dcache entry.

name the name of this inode in the parent directory.

list of subdirectories If this dcache entry represents a directory, there is a partial list of its subdirectories.

private data and dcache operations The *struct dentry* contains a pointer to private data and a set of operations that each specific file system can implement. They are used by the file system to make sure the dcache is current as other clients of a distributed file system manipulate the directory tree.

The most important dcache operation is revalidate. When the Linux VFS uses the data in the dcache to avoid doing a lookup, it calls the revalidate operation. The revalidate asks the underlying file system to make sure the dcache entry is still valid.

3.4 struct inode

The *struct inode* represents inodes in the Linux VFS layer. The important members of the structure are:

inode number the Linux inode knows the number of the inode it's representing

"stat" information the inode contains the file size, permissions, ownership, timestamps, and link count

list of Dcache entries a list of all the Dcache entries that point to the inode is kept. This enables the Linux VFS to find full path names for each inode in memory

inode private data there is a union similar to the one stored in the *struct super* that holds data about the inode that is private to the file system

inode operations These are the operations that act on each inode, including lookup, create, symlink, readlink, rename, mkdir, rmdir, unlink, mknod, and bmap. Some of the notable operations that are missing in this structure are:

- operations that read in and throw away inodes are in the super block operations
- readdir, read, write, ioctl, and fsync are part of the file operations.

One operation that is unique to the linux inode structure is the revalidate operation. The Linux inode contains "stat" information about the inode. For a local file system, the information in the *struct inode* is always at least as current as the information that the underlying file system has on disk. This isn't true for distributed file systems. The Linux VFS layer needs the revalidate operation to ask the underlying file system to refresh the information stored in the *struct inode*. The Linux VFS can then act on this current information.

3.5 struct file

Linux's *struct file* contains operations that have to do a particular file open. The important members of *struct file* are:

dcache pointer this points to the dcache entry (and thus the inode) that this *struct file* represents

file position the offset in the file where the next read or write will take place

file modes the mode the file was opened for (O_RDONLY, O_WRONLY, O_RDWR, ...)

file operations a switch table of the operations that can happen on a file. These include read, write, poll, ioctl, mmap, open, close (called release), lseek, and fsync. This structure is also used for block and character devices, so these operations are things that you would want to do on a device or file in general. Readdir is also a file operation for some strange reason.

3.6 The Linux Buffer Cache

The Linux buffer cache is made up of a set of structures called the *struct buffer_head*. Each buffer contains a block of data from disk. All buffers for a given device are of the same size, although different devices can have different sized blocks.

Some of the interesting buffer cache operations are:

set_blocksize this function sets the fundamental blocksize for a given device

getblk getblk creates a buffer for a given block on the underlying device

brelse brelse frees a buffer

ll_rw_block starts I/O on a given block

4 Key Issues in Porting XFS to Linux

In this section, we describe several additional features required in the Linux kernel to maximize the performance achievable with XFS. In addition, we describe alternative porting strategies for moving XFS to Linux.

4.1 Integrating the Linux Buffer and Page Cache with XFS

4.1.1 XFS requirements for the buffer and page cache

The IRIX implementation of XFS depends on the buffer cache for several key facilities. First, the buffer cache allows XFS to store file data which has been written by an application without first allocating space on disk. The routines which flush delayed writes are prepared to call back into XFS, when necessary, to get XFS to assign disk addresses to such blocks when it is time to flush the blocks to disk. Since delayed allocation means that XFS can see if a large number of blocks have been written before it allocates space, XFS is able to allocate large extents for large files, without have to reallocate or fragment storage when writing small files. This facility allows XFS to optimize transfer sizes for writes, so that writes can proceed at close to the maximum speed of the disk, even if the application does its write operations in small blocks.

Second, the buffer cache provides a reservation scheme, so that blocks with delayed allocation will not take so much of the available memory that XFS would deadlock on memory when trying to do metadata reads and writes in the course of allocating space for delayed allocation blocks.

Third, the buffer cache and the interface to disk drivers support the use of a single buffer object to refer to as much as an entire disk extent, even if the extent is very large and the buffered pages in memory are not contiguous. This is important for high performance, since allocating, initializing, and processing a control block for each disk block in, for example, a 7 MB HDTV video frame, would represent a large amount of processor overhead, particularly when one considers the cost of cache misses on modern processors. XFS has been able to deliver 7 GB/second from a single file on an SGI Origin 2000 system, so the overhead of processing millions of control blocks per second is of practical significance.

Fourth, the buffer cache supports "pinning" buffered storage in memory, which means that the affected buffers will not be forced to disk until they have been "unpinned". XFS relies on this capability to keep metadata updates from being written to disk until after the log entries for those updates have been written to disk. That is, XFS keeps just one version of the metadata on disk (not counting any copies in the log), and requiring that the log be written before the metadata updates are written back means that recovery can simply apply after-images from the log to make the metadata consistent.

4.1.2 Mapping the XFS view of the buffer and page cache to Linux

With Linux 2.3, the intent is that most file system data will be buffered in the page cache, but I/O requests are still issued one block at a time, with a separate buffer_head for each disk block and multiple buffer_head objects for each page (if the disk block size is smaller than the page size). As in Linux 2.2, drivers may freely aggregate requests for adjacent disk blocks to reduce controller overhead, but they must discover any possibilities for aggregation by scanning the buffer_head structures on the disk queue.

Our plan for porting XFS is to build a layered buffer cache module on top of the Linux page cache, which allows XFS to act on extent-sized aggregates, as in IRIX, even if the actual I/O operations are performed by creating a list of buffer_head structures to send to the disk drivers. We will also explore how to extend the Linux driver interface to support queueing aggregate buffers directly to the drivers, at least for any drivers which support the extended interface. If the extension is optional, then perhaps only the SCSI driver need be changed to support it.

A key goal for the layered buffer cache module is that its objects be strictly temporary, so that they are discarded when released by the file system, with all persistent data held purely in the page cache. This will require storing a little more information in each mem_map_t, but it will avoid creating yet another class of permanent system object, with separate locking and resource management issues. The IRIX buffer cache is about 11,000 lines of very complex code. By relying purely on the page cache for buffering, we expect to avoid most of the complexity, particularly in regard to locking and resource management, at the cost of having to pay careful attention to efficient algorithms for assembling large buffers from pages.

4.2 Issues for the aggregate buffer cache module

4.2.1 Partial Page Mappings

In general, disk extents will not align with page boundaries. This means that a given page in the page cache may map to several different disk extents, depending on the block size and the page size, which means that several different aggregate buffers may address the same page. Moreover, for efficient I/O, it is desirable to read or write entire extents, so a given page may be only partially valid when an aggregate buffer referencing it is released. This implies that the mem_map_t needs to include a bit map of which blocks within the page are valid. On a virtual memory fault for such a page the virtual memory system must force the missing parts of the page to be read (which might, as a side-effect, cause other partially-read pages to be created in the page cache).

4.2.2 Partial Aggregate Buffers

In general, not all of the pages in a given aggregate buffer will be in the page cache when the file system requests the buffer. The aggregate buffer module will supply several interfaces to obtain buffers. One interface will return the buffer with empty pages, marked not valid, supplied for the "holes". Another will force the empty pages to be read in from disk. XFS makes use of both interfaces, since in some cases (such as a write which covers an entire extent), the old value of the missing pages is not needed.

4.2.3 Efficient Assembly of Buffers

At present, pages are both entered in a hash table, based on the inode and offset, and on a page list associated with the inode. This means that one must probe the hash table for each page in the range of a buffer when assembling the buffer. If the list of pages for an inode were kept sorted, then one could simply find one page and walk the list to find the rest. Better yet, if the pages were on an AVL tree associated with the inode, and not in the hash table at all, then one could easily search the tree to find the first valid page, and immediately know that prior pages were not valid.

4.3 Metadata Buffers

In order to have just one way to do I/O for XFS, the aggregate buffer cache will use the page cache to store XFS metadata. The metadata pages will be associated with the device inode on which the file system (and the log, if separate) is located.

4.4 Direct I/O

For files which are referenced multiple times, and particularly for small files, saving a copy of the file contents in the page cache is very desirable. For very large data files, such as streaming video files, this can be worse than useless, since caching such data will force useful data out of the cache. Also, for very large files transferred at high rates, the processor overhead of copying all of the data is very high. XFS supports doing the file system equivalent of “raw I/O”, called direct I/O, where file data moves directly between the file system and the user buffers (whether reading or writing). This has proved sufficiently efficient that even large databases may be efficiently stored in the file system, thereby simplifying system administration.

Direct I/O shares with raw I/O the need to lock the user buffer pages in memory during the I/O transfer, since the disk driver will be asked to transfer directly to or from those pages. For consistency and simplicity of interfaces, it is highly desirable, therefore, that the aggregate buffer cache module allow XFS to bind a buffer object to a range of user memory (suitably locked), and then do I/O on the buffer object in the usual way.

4.5 Alternative Porting Strategies

We are considering three principal strategies for porting XFS to Linux:

1. change Linux to directly support an IRIX-like vnode/vfs interface, thereby minimizing the changes required in XFS and enhances Linux to more easily support other vnode/vfs file systems

2. change XFS so that it can be directly integrated into the existing Linux VFS, thereby minimizing the changes required in Linux

3. introduce a layer between XFS code and the Linux VFS interface that translates Linux VFS calls into the equivalent IRIX vnode/vfs operations.

The first strategy would require a new vnode/vfs layer that would parallel (but probably not replace) the existing Linux VFS layer. Because the vnode/vfs interface is not standardized across the different UNIX implementations [15], it is unlikely that the Linux vnode interface created for XFS would directly support file systems from other vendors, but it would make porting vnode/vfs-based file systems easier. This might be turned into an opportunity to develop an open source standard vnode/vfs interface in the highest volume UNIX implementation, thereby creating a de-facto vnode/vfs interface standard in code. In any case, this would require major changes to the existing Linux kernel. Also, at this time it is unclear how the Linux community in general and Linus in particular would react to these proposed changes.

Changing XFS to fit directly into Linux VFS interface would require significant changes to nearly every XFS routine. The current source code organization would need to be significantly changed. In addition, XFS uses the UNIX uio structure to describe the I/O transfer required at the system call level, and the uio structure is embedded throughout the XFS code. XFS consists of a lot of sophisticated code. Some commercial journaled file systems we are aware of consist of more than 150,000 lines of C code.

The third alternative is to integrate the XFS vnode and XFS vfs object as private file-system-dependent data in the *struct inode* and *struct super_block* data in Linux.

This approach introduces a translation layer between the XFS code and the Linux VFS interface. This layer will translate Linux VFS calls into the equivalent XFS vnode operations. The XFS vnode itself would be attached to the private data area of the Linux inode, while the XFS vfs object would be attached to the private data area of the Linux superblock. As an example, a create request to the file system would get mapped to the XFS create via this pointer to the XFS vnode, which includes the vnode operation for create. Similarly, a mount operation (on Linux, the read_super vfs call) would result in a call to the XFS-specific mount operation available through the Linux superblock’s pointer to the vfs object.

This approach is shown in figure 5 and figure 6.

Currently, we are focusing on the third alternative as the fastest way of getting the port to Linux completed. The overhead introduced by the translation layer should

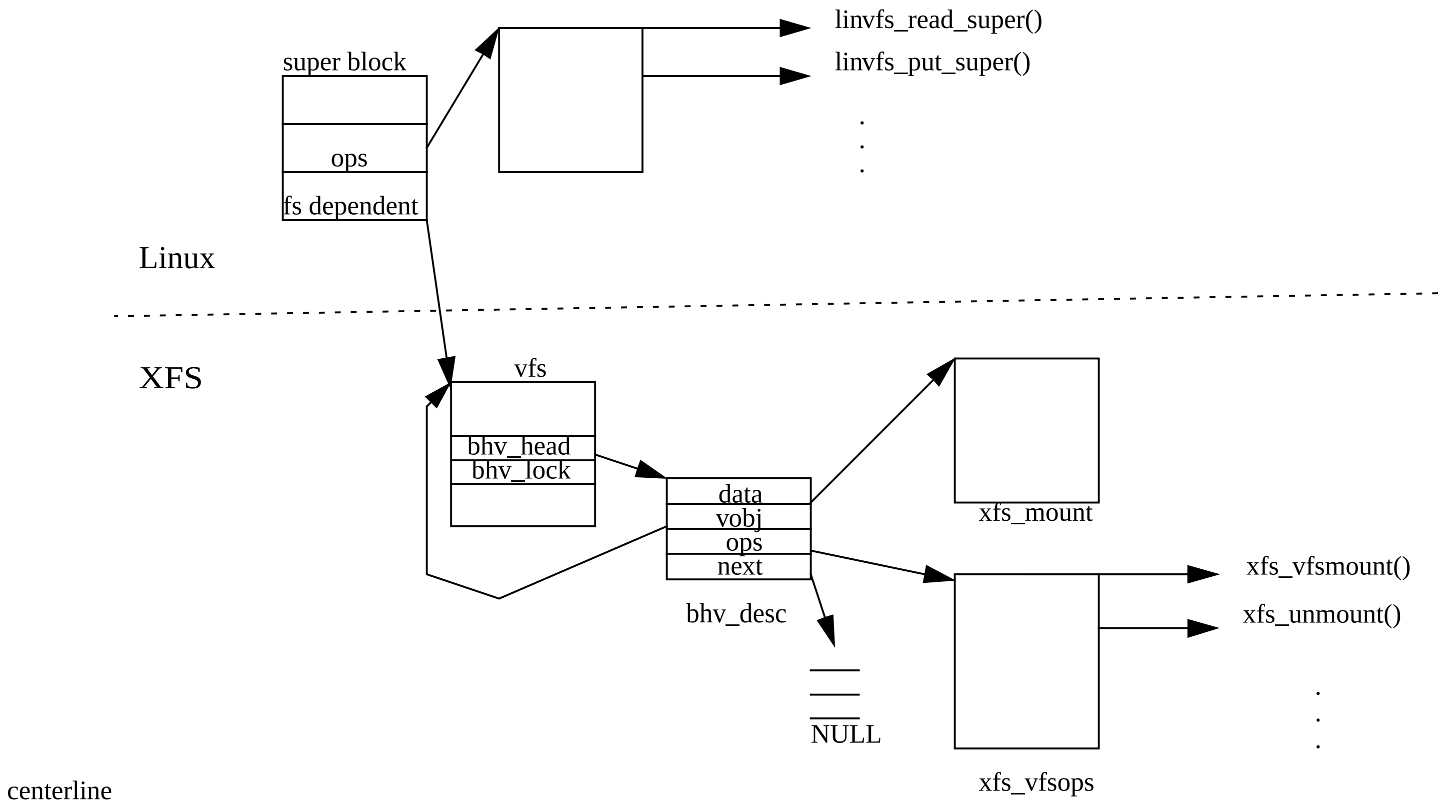


Figure 5: Converting an Linux VFS operation to an XFS vfs operation.

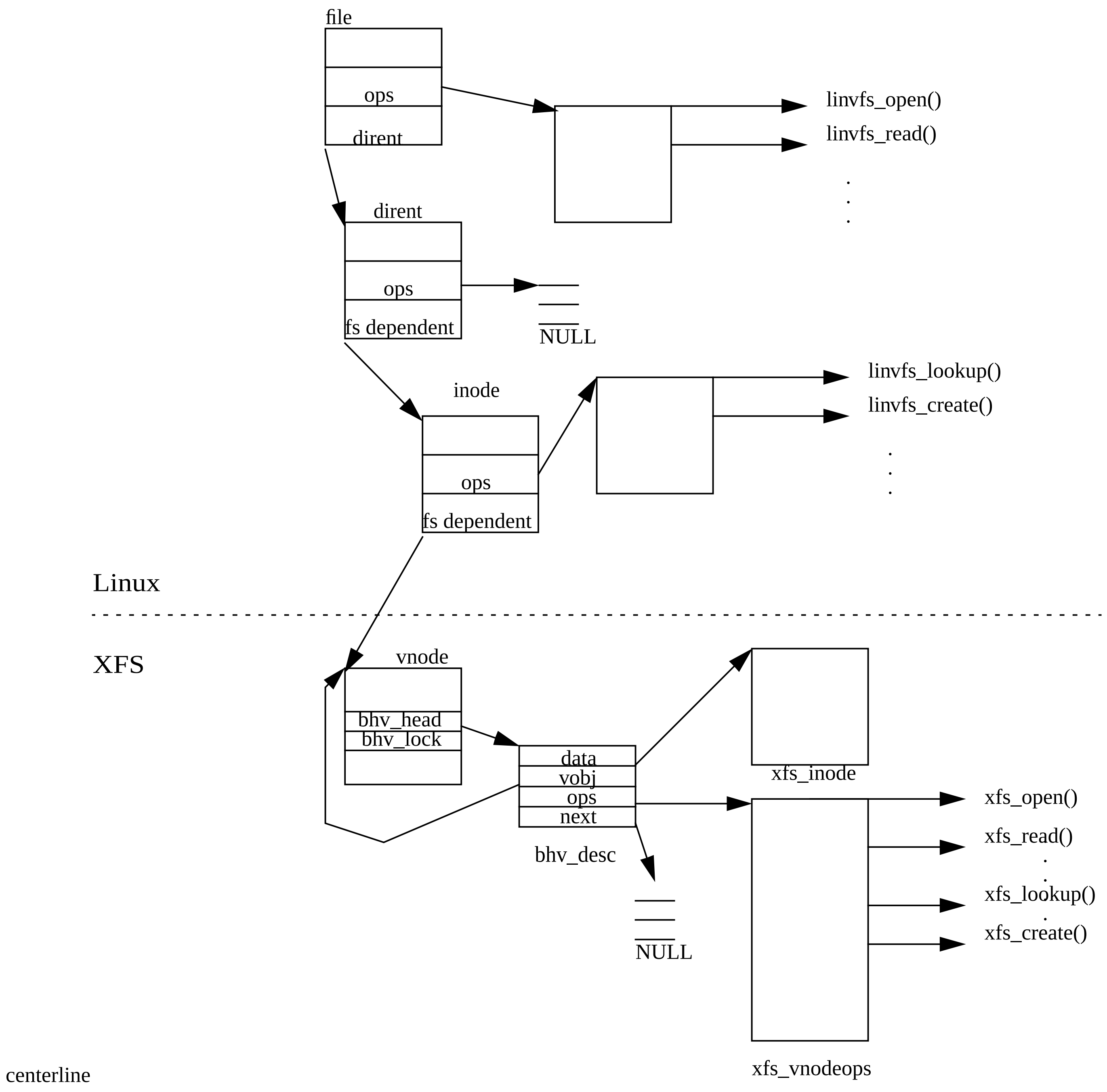


Figure 6: Converting an Linux VFS operation to an XFS vfs operation.

be relatively small: the wrapfs stackable file system layer [11] was found to yield about 7-10% additional overhead. We expect the XFS translation layer overheads to be less than 5Changing XFS to fit directly into the Linux XFS interface is still an option for this project in the future. The initial XFS port will be to the 2.2 kernel as a module.

4.6 Volume Management

XFS depends on a volume manager for providing an integrated block interface to a set of disk drives. The current XFS implementation relies on xlv, a relatively simple logical volume manager developed by SGI to support XFS. There are two volume managers available in Linux today: Linux lvm [13] and md [18]. MD focuses on software RAID support, whereas Linux lvm is a more traditional logical volume management layer modeled after the HP-UX design, which itself followed the OSF/1 model.

Linux lvm adds an additional layer between the physical peripherals and the i/o interface in the kernel to get a logical view of disks. Unlike current partition schemes where disks are divided into fixed-sized sections, lvm allows the user to consider disks, also known as physical volumes (PV), as a pool (or volume) of data storage, consisting of equal-sized extents.

An lvm volume consists of arbitrary groups of physical volumes, organized into volume groups (VG). A volume group can consist of one or more physical volumes. There can be more than one volume group in the system. Once created, the volume group, and not the disk, is the basic unit of data storage.

The pool of disk space that is represented by a volume group can be divided into virtual partitions (called logical volumes – LV) of various sizes. A logical volume can span a number of physical volumes or represent only a portion of one physical volume. The size of a logical volume is determined by the number of extents it contains. Once created, logical volumes can be used like regular disk partitions to create a file system or as a swap device.

Currently, we plan to use Linux lvm to support XFS logical volume manager requirements. However, SGI's new XVM volume manager will become available on Linux in the near future, and XFS will exploit its advanced features.

5 Summary

As the XFS port to Linux proceeds, source code and progress reports will be posted to *http://oss.sgi.com/projects/xfs*. The porting team will be presenting its work at various Linux conferences over the coming year, and we look forward to working more closely with the Linux developer community as the source code becomes available. At the present time, the source code is being reviewed to insure that it can be GPL'ed without any restrictions. Once this code review is complete, the XFS source code will be made available at the web site.

6 Acknowledgments

The original XFS team, led by Geoff Peck, made XFS possible. The team included Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, and Mike Nishimoto. We would also like to thank Stephen Tweedie, Ted Ts'o, and Peter Braam for their feedback on this work.

References

[1] Marshall Kirk McKusick, Keith Bostic, Michael Karels, and John Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[2] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[3] Geoff Peck et al. Original xfs design documents. http://oss.sgi.com/projects/xfs/, 1993.

[4] J. Kent Peacock and et al. Fast consistency checking for the solaris file system. In *Proceedings of the USENIX Annual Technical Conference*, pages 77–89, June 1998.

[5] Michael J. Folk, Bill Zoellick, and Greg Riccardi. *File Structures*. Addison-Wesley, March 1998.

[6] Douglas Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.

[7] Stephen C. Tweedie. A journaled file system for linux. In *Proceedings of the 4th Annual Linux Expo*, Raleigh, North Carolina, May 1998.

[8] Theodore Tsó. Introducing b-trees into the second extended filesystem. In *Proceedings of the 4th Annual Linux Expo*, Raleigh, North Carolina, May 1998.

[9] Peter Braam and Philip Nelson. Removing bottlenecks in distributed filesystems. In *Proceedings of the 5th Annual Linux Expo*, pages 131–139, Raleigh, North Carolina, May 1999.

[10] Kenneth Preslan et al. A 64-bit, shared disk file system for linux. In *Proceedings of the 5th Annual Linux Expo*, pages 111–130, Raleigh, North Carolina, May 1999.

[11] Erez Zadok and Ion Badulescu. A stackable file system interface for linux. In *Proceedings of the 5th Annual Linux Expo*, pages 141–151, Raleigh, North Carolina, May 1999.

[12] Hans Reiser. Reiserfs file system. http://idiom.com/b̃everly/reiserfs.html, July 1999.

[13] Heinz Mauelshagen. Linux logical volume manager (lvm), version 0.7. http://linux.msede.com/lvm/, July 1999.

[14] S.R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 238–247, June 1986.

[15] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice-Hall, 1996.

[16] L. McVoy and S. Kleiman. Extent-like performance from a unix file system. In *Proceedings of the 1991 Winter USENIX Conference*, pages 33–43, Dallas, TX, June 1991.

[17] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, second edition, 1998.

[18] Jakob Ostergaard. The Software-RAID HOWTO. http://ostenfeld.dk/j̃akob/Software-RAID.HOWTO/Software-RAID.HOWTO.html.