Proceedings of FREENIX Track:
2000 USENIX Annual Technical Conference
San Diego, California, USA, June 18–23, 2000

PORTING THE SGI XFS FILE SYSTEM TO LINUX

Jim Mostek, Bill Earl, Steven Levine, Steve Lord, Russell Cattelan,
Ken McDonell, Ted Kline, Brian Gaffey, and Rajagopal Ananthanarayanan

Porting the SGI XFS File System to Linux

Jim Mostek, Bill Earl, Steven Levine, Steve Lord, Russell Cattelan, Ken McDonell, Ted Kline, Brian Gaffey, Rajagopal Ananthanarayanan

SGI

Abstract

The limitations of traditional Linux file systems are becoming evident as new application demands for Linux file systems arise. SGI has ported the XFS file system to the Linux operating system to address these constraints. This paper describes the major technical areas that were addressed in this port, specifically regarding the file system interface to the operating system, buffer caching in XFS, and volume management layers. In addition, this paper describes some of the legal issues surrounding the porting of the XFS file system, and the encumbrance review process that SGI performed.

1. Introduction

In the early 1990s, SGI realized its existing file system, 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 systems. 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, 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), statically allocated metadata, and slow recovery times using fsck.

To address these issues in EFS, in 1994 SGI released an advanced, journaled file system on IRIX[1]; this file system was called XFS[1]. 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.

To help address these same issues in Linux as well as to demonstrate commitment to the open source community, SGI has made XFS technology available as Open Source XFS[2], an open source journaling file system. Open Source XFS is available as free software for Linux, licensed with the GNU General Public License (GPL).

As part of our port of XFS, we have made two major additions to Linux. The first is *linvfs*, which is a porting layer we created to map the Linux VFS to the VFS layer in IRIX. The second is *pagebuf*, a cache and I/O layer which provides most of the advantages of the cache layer in IRIX. These additions to Linux are described in this paper.

2. The File System Interface

The XFS file system on IRIX was designed and implemented to the vnode/VFS interface[2]. In addition, the IRIX XFS implementation was augmented to include layered file systems using structures called “behaviors”. Behaviors are used primarily for CXFS, which is a clustered version of XFS. CXFS is also being ported to Linux. Much of XFS contains references to vnode and behavior interfaces.

On Linux, the file system interface occurs at 2 major levels: the file and inode. The file has operations such as open() and read() while the inode has operations such as lookup() and create(). On IRIX, these are all at one level, vnode operations. This can be seen in figure 1, which shows the mapping of Linux file system operations to vnode operations such as XFS uses.

In order to ease the port to Linux and maintain the structure of XFS we created a Linux VFS to IRIX VFS mapping layer (linvfs).

2.1 The VFS Mapping Layer (linvfs)

For the most part, the XFS port to Linux maintained the vnode/VFS and behavior interfaces[3]. Translation from file/inodes in Linux to vnodes/behaviors in XFS is performed through the linvfs layer. The linvfs layer maps all of the file and inode operations to vnode operations.

1. SGI’s System-V-derived version of UNIX
2. http://oss.sgi.com/projects/xfs

Figure 1 shows the mapping of Linux VFS to IRIX VFS. In this figure, the Linux file system interface is shown above the dotted line. The bottom half of the figure shows the file system dependent code, which resides below the inode.

The two major levels of the Linux file system interface, file and inode, are shown in the figure. Each of these levels has a set of operations associated with it. The dirent level, also shown in the figure, has operations as well, but XFS and most other file systems do not provide file system specific calls.

The linvfs layer is invoked through the file and inode operations. This layer then locates the vnode and implements the VFS/vnode interface calls using the semantics that XFS expects.

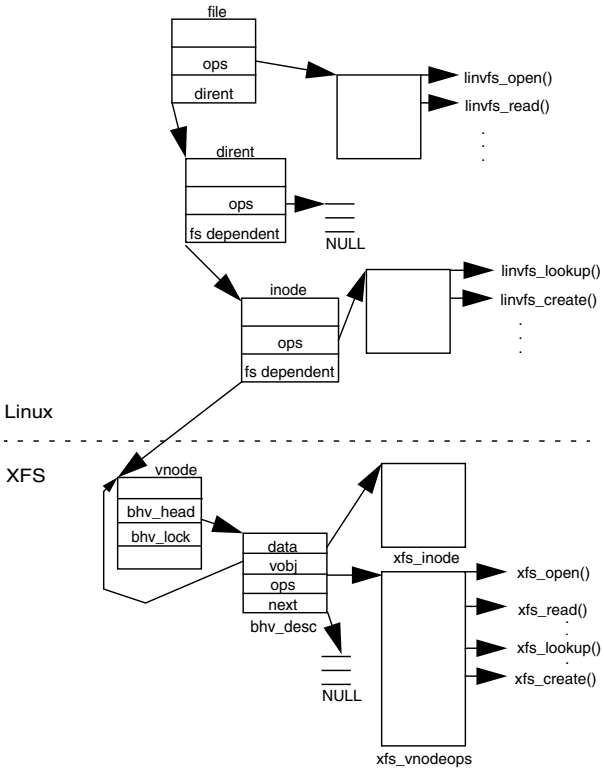

Figure 1: Mapping a Linux VFS Operation to an IRIX VFS Operation.

Linux has three separate types of file and inode operations: directory, symlink, and regular file. This helps split up the functionality and semantics. If the file system does not provide a specific operation, a default action is taken.

Information on the XFS Linux I/O path itself is provided in section 3.8, File I/O.

The linvfs layer is a porting device to get XFS to work in Linux. linvfs allows other VFS/vnode-based file systems to be ported to Linux.

2.2 linvfs Operation Examples

The following examples show how three operations are performed in the linvfs layer.

Example 1: The lookup operation

The lookup operation is performed to convert a file name into an inode. It makes sense to do a lookup only in a directory, so the symlink and regular file operation tables have no operation for lookup. The directory operation for XFS is linvfs_lookup:

```
struct dentry * linvfs_lookup(struct inode
*dir, struct dentry *dentry)
{
```

First, get the vnode from the Linux inode.

```
vp = LINVFS_GET_VP(dir);
```

Now, initialize vnode interface structures and pointers from the Linux values:

```
/*
 * Initialize a pathname_t to pass down.
 */
bzero(pnp, sizeof(pathname_t));
pnp->pn_complen = dentry->d_name.len;
pnp->pn_hash = dentry->d_name.hash;
pnp->pn_path = (char *)dentry->d_name.name;

cvp = NULL;

VOP_LOOKUP(vp, (char *)dentry->d_name.name,
    &cvp, pnp, 0, NULL, &cred, error);
```

If the lookup succeeds, linvfs_lookup gets the inode number from the vnode. The inode number is needed to get a new Linux inode. XFS was modified to set this new field, v_nodeid, for Linux.

```
if (!error) {
        ASSERT(cvp);
        ino = cvp->v_nodeid;
        ASSERT(ino);
        ip = iget(dir->i_sb, ino);
        if (!ip) {
                VN_RELE(cvp);
                return ERR_PTR(-EACCES);
        }
}
```

In all cases of linvfs_lookup, an entry is added to the Linux dcache.

```
/* Negative entry goes in if ip is NULL */
d_add(dentry, ip);
```

If the lookup fails, ip will be NULL and a negative cache entry is added, which is an entry that will return an indication of not found on subsequent lookups. Subsequent lookups of the same pathname will not reach linvfs_lookup since the dentry will already be initialized. If the pathname is modified by an operation such as create, rename, remove, or rmdir, the dcache is modified to remove the dentry for this pathname.

Most of the functionality of the lookup operation occurs below VOP_LOOKUP and iget(). VOP_LOOKUP may read the disk to search the directory, allocate an xfs_inode, and more.

iget() is a Linux routine that eventually calls the file system specific read_inode() super operation. This routine required a new VOP, VOP_GET_VNODE, which simply returns the already allocated vnode so the inode can be initialized and point to the vnode. The vnode is actually allocated and initialized by VOP_LOOKUP.

The VOP_LOOKUP functionality is much broader than what is expected in the Linux lookup operation. For now, the XFS port keeps the VOP_LOOKUP functionality and just requires the file system to provide a new VOP_GET_VNODE interface where the vnode can be found and linked to the inode. In the future, this could be split such that the xfs_iget() code could be moved into linvfs_read_inode().

Example 2: The linvfs_open operation

An example of a file system operation is open(). For XFS, the operation is currently linvfs_open() for files and directories and is as follows:

```
static int linvfs_open(
        struct inode *inode,
        struct file *filp)
{
        vnode_t *vp = LINVFS_GET_VP(inode);
        vnode_t *newvp;
        int     error;

        VOP_OPEN(vp, &newvp, 0,
            get_current_cred(), error);

        if (error)
                return -error;

        return 0;
}
```

This is a very simple routine. XFS never returns a newvp and this vnode functionality needs more work if additional file systems are added that exploit newvp. For performance, XFS starts a read-ahead if the inode is a directory and the directory is large enough. For all cases, xfs_open checks to see if the file system has been shutdown and fails the open. The shutdown check provides important functionality to avoid panics and protect the integrity of the file system when errors occur such as permanent disk errors.

Example 3: The linvfs_permission routine

The final example is linvfs_permission. On linux, this routine is used to check permissions and this maps to the IRIX VOP_ACCESS() call as follows:

```
int linvfs_permission(struct inode *ip, int
mode)
{
       cred_t  cred;
       vnode_t *vp;
       int     error;

       /* convert from linux to xfs */
       /* access bits               */
       mode <<= 6;
       vp = LINVFS_GET_VP(ip);
       VOP_ACCESS(vp, mode, &cred, error);

       return error ? -error : 0;
}
```

2.3 linvfs Overhead

The overhead of the linvfs_permission operation has been computed. These numbers were generated by averaging 4 runs of 1000000 access() system calls after an initial run to warm the cache. This was performed on a 2 CPU 450 MHz PIII box.

The implementation of VOP_ACCESS in XFS is fundamentally the same as the permissions function code executed by ext2.

Table 1 shows the access system call numbers for this test. For this test, we turned off our locking code since XFS without locks more closely resembles the ext2 code. Our goal was to measure the overhead of the linvfs layer, not to measure the overhead of locking.

Table 1: access() Timings

File system	Microseconds per call
ext2	3.54
xfs	3.89 (with locking commented out)

The linvfs layer overhead is around 0.35 micro-seconds for this simple call.

With locking turned on, XFS took 5.14 microseconds per call.

2.4 fcntl Versus ioctl in IRIX and Linux

In IRIX, XFS supports approximately 20 special fcntl interfaces used for space pre-allocation, extent retrieval, extended file information, etc. In addition, IRIX XFS has about a dozen special system call interfaces, all implemented via the special *syssgi* system call. These interfaces are used for operations such as growing the file system or retrieving internal file system information.

The Linux file system interface has no fcntl operation. The only supported fcntl calls on Linux are file locking calls. We proposed to the Linux community that a fnctl file operation be added. After extensive discussion, it was decided to use the existing ioctl operation, linvfs_ioctl, and we are in the process of converting all of the fcntl and syssgi usages into ioctls. A shortcoming to the ioctl approach is in the semantics of an ioctl to block or character special devices which reside within the file system: In these cases, the device driver's ioctl routine will be used rather than the file system's. Outside of that, porting the fcntl and syssgi interfaces to ioctl's has been straightforward.

2.5 IRIX XFS creds and Linux

In older UNIX systems, the file system code used the current process's data structures to determine the user's credentials such as uid, gid, capabilities, etc. The VFS/vnode interface removed this code and introduced a cred structure which is passed to certain file system operations such as create and lookup. The file system uses this information to determine permissions and ownership.

XFS was written using the VOP/vnode interface, so it regularly uses cred structures. One of the more prevalent cred usages on IRIX XFS is get_current_cred, which returns this structure for the current process.

Linux is similar to older UNIX implementations in that file systems are expected to look directly at the task structure to determine the current process's credentials. Linux does not utilize a cred structure.

In porting XFS to Linux, we first attempted to map the various cred fields onto the corresponding task fields. This had the undesired side-effect of producing code that utilized a cred pointer that in actuality was pointing at a task. This was determined to be unacceptable.

We then considered implementing a complete cred infrastructure, which would include a pool of active creds, cred setup, teardown, lookup, etc. It was determined that this would require too much overhead.

In looking at the Linux code, we saw that all of the access/permission work occurs above the file system dependent code, so having a cred is important only on creation. We then examined our own internal usage of cred fields in XFS, and found that more often than not, a cred was passed down through a VOP_, and never used. The few places that did use a cred field were changed to use the current task structure in Linux.

We still pass a cred address on the VOP_s, but it is not used. In the future, as part of the port cleanup, we may change the VOP_ macros, or more likely simply pass NULL as the cred address.

In addition to these cred changes, we have removed many access checks from the XFS code since these are now performed at a higher layer and are redundant in the file system dependent code.

3. XFS Caching and I/O

When XFS was first implemented within IRIX, the buffer cache was enhanced in a number of ways to better support XFS, both for better file I/O performance and for better journaling performance. The IRIX implementation of XFS depends on this buffer cache functionality 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 for such blocks when it is time to flush the blocks to disk. Since delayed allocation means that XFS can determine if a large number of blocks have been written before it allocates space, XFS is able to allocate large extents for large files, without having to reallocate or fragment storage when writing small files. This facility allows XFS to optimize transfer sizes for writes, so that writes can pro-

ceed at close to the maximum speed of the disk, even if the application does its write operations in small blocks. In addition, if a file is removed and its written data is still in delayed allocation extents, the data can be discarded without ever allocating disk space.

Second, the buffer cache provides a reservation scheme, so that blocks with delayed allocation will not result in deadlock. If too much of the available memory is used for delayed allocation, a deadlock on the memory occurs when trying to do conversion from delayed to real allocations. The deadlock can occur since the conversion requires metadata reads and writes which need available memory.

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 written to disk until they have been "unpinned". XFS uses a write-ahead log protocol for metadata writes, which means XFS writes a log entry containing the desired after-image before updating the actual on disk metadata. On recovery, XFS just applies after-images from the log (in case some of the metadata writes were not completed). In order to avoid having to force the log before updating metadata, XFS "pins" modified metadata pages in memory. Such pages must count against the memory reservation (just as do delayed allocation pages). XFS pins a metadata page before updating it, logs the updates, and then unpins the page when the relevant log entries have been written to disk. Since the log is usually written lazily, this in effect provides group commit of metadata updates.

With Linux 2.3 and later releases, the intent is that most file system data will be buffered in the page cache, but the 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.

3.1 The pagebuf Module

Our approach to porting XFS has included adding pagebuf, a layered buffer cache module on top of the Linux page cache. This allows XFS to act on extent-sized aggregates. Key to this approach is the pagebuf structure, which is the major structure of the pagebuf layer. The pagebuf objects implemented by this module include a *kiovec* (a Linux data structure which describes one or more lists of physical pages) to describe the set of pages (or parts of pages) associated with pagebuf, plus the file and device information needed to perform I/O. We are experimenting with a new device request interface, so that we can queue one of these pagebuf objects directly to a device, rather than having to create and queue a large number of single-block buffer_head objects for each logical I/O request. For backward compatibility, we support a routine which does create and queue buffer_head objects to perform the I/O for pagebuf.

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 requires storing a little more information in each mem_map_t (page frame descriptor), but it avoids creating yet another class of permanent system object, with separate locking and resource management issues. The IRIX buffer cache implementation has about 11000 lines of very complex code. By relying purely on the page cache for buffering, we avoid most of the complexity, particularly in regard to locking and resource management, of hybrid page and buffer caches, at the cost of having to pay careful attention to efficient algorithms for assembling large buffers from pages.

3.2 Partial Aggregate Buffers

Pages within an extent may be missing from memory, so a buffer for an extent may not find all pages present. If the buffer is needed for writing, empty (invalid) pages may be used to fill the holes, in cases where the intended write will overwrite the missing pages. If only part of a page will be modified, pagebuf will read in the missing page or pages. If the buffer is needed for reading or writing just part of the extent, missing pages need not be read if all pages to be read are present.

XFS extents are typically large. However, small extents are still possible, and hence a page could be mapped by

more than one extent. In such cases, it would be desirable to avoid having to seek to multiple locations when the I/O is to a part of the page mapped by one extent. Therefore, we added a map (block_map) of valid disk blocks for each page. When the entire page is read or written, all blocks of the page are marked valid. On a file write, if a part of the page is modified, only the corresponding bits for the modified blocks are marked valid. Similar actions are performed during a read of a partial page. Note that similar information is maintained by Linux if buffers are mapped (or not mapped) to parts of the page. block_map provides this functionality without having to attach buffer_head's to the page. Finally, if the page and the disk block sizes are the same, the block_map is equivalent to (and is replaced by) the PG_uptodate flag of the page.

3.3 Locking for pagebufs

As noted above, the basic model for pagebufs is that each pagebuf structure is independent, and that it is just a way of describing a set of pages. In this view, there may be multiple pagebufs referring to a given page, just as a given page may be mapped into multiple address spaces. This works well for file data, and allows pagebufs in the kernel to be operationally equivalent to memory mapped file pages in user mode. For metadata, however, XFS assumes that its buffer abstraction allows for sleeping locks on metadata buffers, and that metadata buffers are unique. That is, no two metadata buffers will share a given byte of memory. On the other hand, two metadata buffers may well occupy disjoint portions of a single page.

To support this model, we implemented a module layered above the basic pagebuf module, which keeps an ordered list (currently an AVL tree) of active metadata buffers for a given mounted file system. This module ensures uniqueness for such buffers, assures that they do not overlap, and allows locking of such buffers. Since we were unsure whether this facility would be of use for other file systems, we have kept it separate from the basic pagebuf module, to keep the basic module as clean and fast as possible.

3.4 Delayed Allocation of Disk Space for Cached Writes

Allocating space when appending to a file slows down writes, since reliable metadata updates (to record extent allocations) result in extra writes. Also, incremental allocations can produce too-small extents, if new extents are allocated each time a small amount of data is appended to a file (as when many processes append to a log file). Delayed allocation reserves disk space for a write but does not allocate any particular space; it simply buffers the write in the page cache. Later, when pages are flushed to disk, the page writing path must ask the file system to do the actual allocation. Also, to allow for optimal extent allocations and optimal write performance, the page writing path must collect adjacent dirty pages ("cluster" them) and write them as a unit.

Since allocation of disk space may be required in the page writing path when delayed allocation is present, and such allocation may require the use of temporary storage for metadata I/O operations, some provision must be made to avoid memory deadlocks. The delayed allocation path for writes must make use of a main memory reservation system, which will limit the aggregate amount of memory used for dirty pages for which disk space has not been allocated, so that there will always be some minimum amount of space free to allow allocations to proceed. Any other non-preemptible memory allocations, such as kernel working storage pages, must be counted against the reservation limit, so that the remaining memory is genuinely available. Based on experience with IRIX, limiting dirty delayed allocation pages and other non-preemptible uses to 80% of available main memory is sufficient to allow allocations to proceed reliably and efficiently.

Page flushing should flush enough delayed allocation pages to keep some memory available for reservation, even if there is free memory. That is, the page flushing daemon must attempt both to have free memory available and have free reservable memory available. The reservation system must, of course, allow threads to wait for space.

3.5 Page Cleaning

At present in Linux, a dirty page is represented either by being mapped into an address space with the "dirty" bit set in the PTE, or by having buffer_head objects pointing to it queued on the buffer cache delayed write queue, or both. Such dirty pages are then eventually written to disk by the bdflush daemon. This has several drawbacks for a high-performance file system. First, having multiple buffer_head objects to represent, in effect, one bit of state for page is inefficient in both space and time. Second, when dirty pages are cleaned, there is no particular reason to expect that all adjacent dirty pages within an extent will be written at once, so the disk is used less efficiently. Third, modifications to pages made via mmap() may be indefinitely delayed in being written to disk, so many updates may be lost on a power failure, unless msync() is used frequently. That is, either one

slowly and synchronously updates the disk, or one gives up any expectation of timely updating of the disk. Fourth, disk write traffic is not managed for smoothness, so there is considerable burstiness in the rate of disk write requests. This in turn reduces utilization of the drive and increases the variability of read latencies.

For the XFS port, we are implementing a page cleaner based on a "clock" algorithm. If the system is configured to move a given update to disk with in K seconds, and if there are N dirty pages out of M total pages, the page cleaner will visit enough pages every second to clean N/K pages. That is, the page cleaner will advance its clock hand through the page frame table (mem_map) until it has caused sufficient pages to be queued to the disk to meet its target. Since the page write path will cluster dirty pages, visiting a given dirty page may cause multiple pages to be written, so the page cleaner may not directly visit as many dirty pages as are written.

Note that the page cleaner does not simply run once a second. Rather, it runs often enough (multiple times per second) to maintain a queue of disk writes, but without creating an excessively long queue (which would increase the variability of read latency). Moreover, on a ccNUMA system, there will be a page cleaner thread per node, so that page cleaning performance will scale with the size of the system.

Since the page cleaner must steadily clean pages, it must not block for I/O. For clusters of pages where disk space has already been allocated, the page cleaner can simply queue a pagebuf to write the cluster to disk. For cases where the allocation has been delayed, however, the allocation of disk space by the file system will in general require waiting to read in metadata from disk and may also require waiting for a metadata log write, if the log happens to be full. The page cleaner has a number of writeback daemon threads to handle writing pages for which disk allocation has been deferred. (This could also be used by any file system where a write might block.) The file system pagebuf_iostart routine has a flags argument, and a flag, PBF_DONTBLOCK, which the page cleaner uses to indicate that the request should not block. If the pagebuf_iostart routine returns an EWOULDBLOCK error, the page cleaner queues the cluster pagebuf to its writeback deamon threads, one of which then processes it by calling the file system pagebuf_iostart routine without the PBF_DONTBLOCK flag.

3.6 Pinning memory for a buffer

In the course of the XFS port, it became clear that metadata objects would in many cases be smaller than pages. (This will be even more common on systems with larger page sizes.) This means that we cannot actually "pin" the entire page, since doing so might keep the page from being cleaned indefinitely: The different metadata objects sharing the page might be logged at various times, and at least one might be pinned at any given time. We decided to treat metadata pages specially in regard to page cleaning. The page cleaning routine associated with a metadata page (via the address_space operations vector) looks up any metadata pagebufs for the page, instead of simply writing out the page. Then, for each such metadata pagebuf which is not "pinned", the page cleaning routine initiates writing the pagebuf to disk. The page cleaning routine simply skips any pinned pagebufs (which will then be revisiting on the next page cleaning cycle).

Since only metadata pages are pinned, we do not need a separate mechanism to pin regular file data pages. If, however, it were desirable to integrate transactional updates for file data into a future file system, it would be possible to simply add a "pin" count to the mem_map_t. We have not done this, however, since XFS does not support transactional update of file data.

3.7 Efficient Assembly of Buffers

We would like the overhead for finding all valid pages within an extent to be low. At present, we simply probe the hash table to find the relevant pages. If this proves to be a performance problem for large I/O requests, we could modify the address_space object to record pages in some sorted fashion, such as an AVL tree, so that we could quickly locate all of the pages in a range, at a very low and constant cost after the first one.

3.8 File I/O

The Linux XFS file read and write routines are provided by the default Linux I/O path with calls into page_buf, when appropriate.

Buffered File Reading

The read routine first loops through the user's I/O requests, searching for pages by probing the page cache. If a page is present, the read routine copies the data from the page cache to the user buffer. If all of the pages are present, the read is then complete.

When a page is not found, what the read routine will do depends on the remaining I/O size. If the remaining I/O

size is less then the page cache size times 4 (which is 16K for IA 32 systems, which have a 4K page size) the default Linux kernel do_generic_file_read is performed, just as for an ext2 file system.

If the remaining I/O size when a page is not found is greater than 4 times the page cache size, the read routine calls back into the file system via a pagebuf_bmap routine to obtain an extent map covering the portion of the file to be read which is not in the cache. The extent map is composed of one or more extent descriptions. Each can be a real extent (with a disk address and a length), a hole (with no disk storage assigned or reserved), or a delayed allocation (with disk storage reserved but not assigned).

Unlike the conventional Linux bmap file system callback routine, the pagebuf_bmap routine can return a list of extentmaps, not simply a single disk block address. Each entry in the list of extentmaps can represent many file system blocks.

One or more instances of an extent map are returned by pagebuf_bmap(). Table 2 shows the fields that are contained in an extent map.

Table 2: Contents of Extent Map

Field	Meaning
bn	starting block number of map
offset	byte offset from bn (block number) of user's request.
bsize	size of mapping in bytes
flags	option flags for mapping

Table 3 shows the option flags that are current used in an extent map.

Table 3: Option Flags for Mapping

Flag	Meaning
HOLE	mapping covers a hole
DELAY	mapping covers dealloc region
NEW	mapping was just allocated
no value	mapping is actual disk extent

For non-delayed allocations, the pages might or might not be present in the cache. For a hole, any cached pages would have been created by mmap access or by small reads. The pagebuf read path does not allocate pages for holes. Rather, it zeros out the user's buffer.

If a read returns a delayed allocation, the pages must exist and are copied to the user's buffer.

If the map is not HOLE or Delay, the read routine fills in the disk block addresses, allocates empty pages and enters them in the page cache, attaches them to the pagebufs, and queues the read requests to the disk driver. A page may already be marked as being up to date, however, in which case it is not necessary to read that page from disk.

After the requests have completed, the read routine marks the pages valid, releases the pagebufs, and finishes copying the data to the user buffer. If a single pagebuf is not sufficient, the read path waits for the data from the first pagebuf read requests to be copied to the user buffer before getting the next pagebuf.

Buffered File Writing

The write routine, like the read routine, first loops through the file page offsets covering the desired portion of the file, probing the page cache. If a page is present, the write routine copies the data from the user buffer to the page cache, and marks the pages dirty. If O_SYNC is not set, the write is then complete. If O_SYNC is set, the write routine writes the pages synchronously to disk before returning.

If a page is not found, the write routine calls back into the file system to obtain the extent map that covers the portion of the file to be written. The call to the file system is for the offset and length of the user's I/O, rounded to page boundaries.

The file system pagebuf_bmap routine accepts some optional flags in a flag argument. One, PBF_WRITE, specifies that a write is intended. In this case, any hole will be converted to a real or delayed allocation extent. A second flag, PBF_ALLOCATE, specifies that a real extent is required. In this case, any delayed allocation extent will be converted to a real extent. This latter flag is used when O_SYNC is set, and is also used when cleaning pages or when doing direct I/O.

The NEW flag indicates when a mapping has just been allocated, and should not be read from disk. This can occur when a hole is converted to allocated space, or you are allocating new space at the end of a file. Empty pages are then allocated and hashed for any pages not

present in the cache. Any pages covering a new map which will be partially overwritten are then initialized to zero, at least insofar as they will not be overwritten. If the NEW flag is not set, the space already existed in the file. In this case, any pages which will only be partially overwritten must be synchronously read from disk before modification with the new user's data.

In both the NEW and already existing cases, the pages are marked as up to date but dirty, indicating that they need to be written to disk.

Preliminary I/O Performance Testing

We have tested the read performance for XFS on Linux, using the lmdd program of the lmbench toolkit to time the I/O. We used lmdd I/O request sizes from 1K to 1024K for the following file system configurations. Each file system was created with a 4K block size.

- XFS file system that does not use pagebufs
- XFS file system with 16K cutover point for using pagebufs
- XFS file system with 32K cutover point for using pagebufs
- ext2 file system

What we found for this specific test and configuration was that for XFS file systems, currently the I/O size makes only a small difference in performance. We also have not yet seen a significant difference in performance when switching the pagebuf cutoff size between 16K (4 pages) and 32K (8 pages). This was because the test was disk bound. At the time of this writing, we have done only minimal performance testing on a single thread without read-ahead. We expect to see better performance with multiple threads and faster disks.

For XFS and the specific configuration tested, the file system I/O rate generally fluctuated between approximately 19.01 and 19.09 MB/sec. For an ext2 file system using the same test, the file system I/O rate fluctuated between approximately 18.8 and 18.9 MB/sec.

3.9 Direct I/O

Small files which are frequently referenced are best kept in cache. Huge files, such as image and streaming media files and scientific data files, are best not cached, since blocks will always be replaced before being reused. Direct I/O is raw I/O for files: I/O directly to or from user buffers, with no data copying. The page cache must cooperate with direct I/O, so that any pages, which are cached and are modified, are read from memory, and so that writes update cached pages.

Direct I/O and raw I/O avoid copying, by addressing user pages directly. The application promises not to change the buffer during a write. The physical pages are locked in place for the duration of the I/O, via the Linux kiobuf routines. The read and write paths for the pagebuf module treat direct I/O requests much as they do regular requests, except for identifying the pages to be addressed in the pagebuf. That is, for direct I/O, the user buffer is referenced via a kiovec, which is then associated with the pagebuf, instead of locating page cache pages and addressing them via a kiovec.

Any dirty pages in the page cache must be flushed to disk before issuing direct I/O. The normal case will find no pages in the cache, and this can be efficiently tested by checking the inode. Once the pagebuf is assembled, the I/O path is largely common with the normal file I/O path, except that the write is never delayed and allocation is never delayed.

Direct I/O is indicated at open() time by using the O_DIRECT flag. Usually the needed space for the file is pre-allocated using an XFS ioctl call to insure maximum performance.

4. Volume Management Layers

The integration of existing Linux volume managers with the XFS file system has created some issues for the XFS port to Linux.

Traditional Linux file systems have been written to account for the requirements of the block device interface, ll_rw_block(). ll_rw_block accepts a list of fixed size I/O requests. For any given block device on a system, the basic unit of I/O operation is set when the device is opened. This size is then a fixed length of I/O for that device. The current implementations of Linux volume managers have keyed off this fixed size I/O and utilize an I/O dispatcher algorithm.

By using a fixed I/O length, the amount of "math" that is needed is significantly less than what it would be if the I/O length were not fixed. All I/O requests from a file system will be of the same size, as both metadata and user data is of fixed size. Therefore, all underlying devices of a logical volume must accept I/O requests of the same size. All that the volume manager needs to do for any I/O request is to determine which device in the logical volume the I/O should go to and recalculate the start block of the new device. Each I/O request is directed wholly to a new device.

The XFS file system, however, does not assume fixed size I/O. In an XFS file system, metadata can be anywhere from 512 bytes to over 8 Kbytes. The basic minimum I/O size for user data is set at file system creation time, with a typical installation using 4 Kbytes. One of the XFS design goals was to aggregate I/O together, creating large sequential I/O.

This feature of XFS creates a problem for current Linux volume managers, since the XFS file system can hand an I/O request off to a block device driver specifying the start position and length, which is not always fixed. A logical volume manager is just another block device to XFS, and a logical volume manager working in conjunction with XFS needs to be able to handle whatever size I/O request XFS desires, to some reasonable limit.

One of the options to address this problem in XFS is to change the on disk format of the file system to use a fixed size. This would render the Linux version of XFS incompatible with the current IRIX implementations, however, and so it was deemed unacceptable, just as making different versions of NFS would be unacceptable.

Currently, XFS for Linux is addressing the problem of variable I/O request size by opening a device with the minimum I/O size needed: 512 bytes. Any request calling ll_rw_block directly must be of that basic size. User data requests use a different version of ll_rw_block, that has been modified to accept a larger size.

Since all Linux volume managers use a call out from ll_rw_block it is clear that XFS currently will not work with current implementations.

In the long run, the solution to the problem of using XFS with currently available Linux volume managers is not clear. Changing interfaces in the kernel that other file systems rely on is not an easy thing to do. It requires agreement of all the current file-system maintainers to change their interface to the kernel.

One thing that is clear is that it will be necessary to develop an additional layer above ll_rw_block that accepts I/O requests of variable size. This interface would either be a direct connection to a device driver, an interface to a logical volume manager, or the fall back case of just calling ll_rw_block for compatibility.

The logical volume interface may need to "split" the I/O request into multiple sub-I/O requests if the I/O is large enough to span multiple devices. This not a difficult thing to implement, but it must be done in agreement of all the logical volume maintainers.

Since it is becoming apparent that Linux is growing up and moving to larger and higher performance hardware. The high bandwidth I/O that XFS offers will be needed. High performance logical volumes will be an integral part of this.

5. Moving XFS to Open Source

For XFS to be a viable alternative file system for the open source community, it was deemed essential that XFS be released with a license at least compatible with the GNU Public License (GPL).

The IRIX operating system in which XFS was originally developed has evolved over a long period of time, and includes assorted code bases with a variety of associated third party license agreements. For the most part these agreements are in conflict with the terms and conditions of the GNU Public License.

The initial XFS project was an SGI initiative that started with a top-to-bottom file system design rather than an extension of an existing file system. Based upon the assertions of the original developers and the unique features of XFS, there was a priori a low probability of overlap between the XFS code and the portions of IRIX to which third-party licenses might apply. However it was still necessary to establish that the XFS source code to be open sourced was free of all encumbrances, including any associated with terms and conditions of third party licenses applying to parts of IRIX.

SGI's objectives were:

- to ensure the absence of any intellectual property infringements
- to establish the likely derivation history to ensure the absence of any code subject to third party terms and conditions

This was a major undertaking; as the initial release of buildable XFS open source contained some 400 files and 199,000 lines of source. The process was long, but relatively straightforward, and encumbrance relief was usually by removal of code.

The encumbrance review was a combined effort for SGI's Legal and Engineering organizations. The comments here will be confined to the technical issues and techniques used by the engineers.

5.1 The Encumbrance Review Process

We were faced with making comparisons across several large code bases, and in particular UNIX System V Release 4.2-MP, BSD4.3 NET/2, BSD4.4-lite and the open source version of XFS. We performed the following tasks:

1. Historical survey

   We contacted as many as possible of the original XFS developers and subsequent significant maintainers, and asked a series of questions. This information was most useful as guideposts or to corroborate conclusions from the other parts of the review.

2. Keyword search (all case insensitive)

   In each of the non-XFS code bases, we searched for keywords associated with unique XFS concepts or technologies (e.g. journal, transaction, etc.). In the XFS code base, we searched for keywords associated with ownership, concepts and technologies in the non-XFS code bases (e.g. at&t, berkeley, etc.).

3. Literal copy check

   Using a specially built tool, we compared every line of each XFS source file against all of the source in the non-XFS code bases. The comparison ignored white space, and filtered out some commonly occurring strings (e.g. matching "i++;" is never going to be helpful).

4. Symbol matching

   We developed tools to post-process the ASCII format databases from cscope to generate lists of symbols and their associated generic type (function, global identifier, macro, struct, union, enum, struct/union/enum member, typedef, etc.). In each XFS source file the symbols were extracted and compared against all symbols found in all the non-XFS code bases. A match occurred when the same symbol name and type was found in two different source files. Some post-processing of the symbols was done to include plausible name transformations, e.g. adding an "xfs_" prefix, or removal of all underscores, etc.

5. Prototype matching

   Starting with a variant of the mkproto tool, we scanned the source code to extract ANSI C prototypes. Based on some equivalence classes, "similar" types were mapped to a smaller number of base types, and then the prototypes compared. A match occurred when the type of the function and the number and type of the arguments agreed.

6. Check for similarity of function, design, concept or implementation.

   This process is based upon an understanding, and a study, of the source code. In the XFS code, for each source file, or feature implemented in a source file, or group of source files implementing a feature, it was necessary to conduct a review of the implementation of any similar source file or feature in each of the non-XFS code bases. The objective of this review is to determine if an issue of potential encumbrance arises as a consequence of similarity in the function, implementation with respect to algorithms, source code structure, etc.

7. Check for evidence of license agreements.

   We examined the XFS code (especially in comments) to identify any references to relevant copyrights or license agreements.

In all of the steps above, the outcome was a list of *possible* matches. For each match, it was necessary to establish in the context of the matches (in one or more files), if there was a real encumbrance issue.

We used a modified version of the tkdiff tool to graphically highlight the areas of the "match" without the visual confusion of all of the minutiae of the line-by-line differences. However, the classification of the matches was ultimately a manual process, based on the professional and technical skills of the engineers.

5.2 Encumbrance Relief

Especially in view of the size of the XFS source, a very small number of real encumbrance issues were identified.

In all cases the relief was relatively straightforward, with removal of code required for IRIX, but not for Linux, being the most common technique.

6. Summary

Porting the XFS file system to Linux required that we address a variety of technical and legal issues. In this paper, we have summarized how the file and inode operations of the XFS vnode interface were translated to Linux by means of the linvfs layer. We have also described how XFS caching was implemented in Linux, and how this caching is used for I/O operations. Finally,

we provided a brief overview of the volume management layer of XFS, and how this has been implemented in Linux.

In addition to these technical concerns we have summarized the technical aspects of the encumbrance review process we went through to ensure that XFS could be released to the open source community without legal ramifications.

We continue to evaluate our use of the linvfs porting layer, especially as regards performance. Our new buffer layer has addressed many of the performance issues and generally extended Linux functionality. XFS on Linux is evolving and will meet the demands of applications moving from traditional UNIX platforms to Linux.

7. Availability

We have completed our initial port of XFS to Linux and the open source process for that port and we are currently in the process of finishing the work on the code. The current version of XFS for Linux is available for downloading at http://oss.sgi.com/projects/xfs.

References

[1] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto and Geoff Peck. *Scalability in the XFS File System.* http://www.usenix.org/publications/library/proceedings/sd96/sweeney.html, January 1996.

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

[3] Jim Mostek, William Earl, Dan Koren, Russell Cattelan, Kenneth Preslan, and Matthew O'Keefe. *Porting the SGI XFS File System to Linux*. http://oss.sgi.com/projects/xfs/papers/als/als.ps, October 1999.