xFS In-core Inode Management

Adam Sweeney

1.0 Introduction

This document describes the creation, manipulation, and destruction of in-core inodes in xFS. It does not deal with the management of on-disk inodes. It explains in-core inode allocation, locking, transactional update, and field management. All of this is subject to change, but this is where things stand at this time.

2.0 Inode Data Structure

The xFS in-core inode structure is defined as:

```
typedef struct xfs_inode {
   struct xfs_ihash       *i_hash;                 /* pointer to hash header */
   struct xfs_inode       *i_next;                 /* inode hash link forw */
   struct xfs_inode       **i_prevp;               /* ptr to prev i_next */
   struct xfs_mount       *i_mount;                /* fs mount struct ptr */
   struct xfs_inode       *i_mnext;                /* next inode in mount’s list */
   struct xfs_inode       **i_mprevp;              /* ptr to prev i_mnext */
   struct vnode           *i_vnode;                /* ptr to associated vnode */
   dev_t                  i_dev;                   /* dev containing this inode */
   xfs_ino_t              i_ino;                   /* inode number (agno/agino) */
   xfs_agblock_t          i_bno;                   /* ag block # of inode */
   int                    i_index;                 /* which inode in block */
   xfs_trans_t            *i_transp;               /* ptr to owning transaction */
   xfs_inode_log_item_t   i_item;                  /* logging information */
   mrlock_t               i_lock;                  /* inode lock */
   sema_t                 i_flock;                 /* inode flush lock */
   unsigned int           i_pincount;              /* # of times inode is pinned */
   sema_t                 i_pinsema;               /* inode pin sema */
   ushort                 i_flags;                 /* misc state */
   ulong                  i_vcode;                 /* version code token (RFS) */
   ulong                  i_mapcnt;                /* count of mapped pages */
   ulong                  i_update_core;           /* inode timestamp dirty flag */
   size_t                 i_bytes;                 /* bytes in i_u1 */
   union {
      xfs_bmbt_rec_t      *iu_extents;             /* linear map of file extents */
      char                *iu_data;                /* inline file data */
   } i_u1;
   xfs_btree_block_t      *i_broot;                /* file’s in-core b-tree root */
   size_t                 i_broot_bytes;           /* bytes allocated for root */
   union {
      xfs_bmbt_rec_t      iu_inline_ext[2];        /* very small file extents */
      char                iu_inline_data[32];      /* very small file data */
      dev_t               iu_rdev;                 /* dev number if special */
      xfs_uuid_t          iu_uuid;                 /* mount point value */
   } i_u2;
   ushort                 i_abytes;                /* bytes in i_u3 */
   union {
        xfs_bmbt_rec_t      *iu_aextents;           /* map of attribute extents */
        char                *iu_adata;              /* inline attribute data */
    } i_u3;
    xs_dinode_core_t        i_d;                    /* most of the on-disk inode */
} xfs_inode_t;
```

3.0 Inode Life Cycle

This section sketches the life cycle of an in-core inode. It follows the inode from the time it is read in from the disk until the time the in-core structure is returned to the kernel heap.

Step 1 Allocate an in-core inode

The first thing the user of in-core inodes needs to do is to get hold of one. This is done via a call to xfs_iget() or xfs_trans_iget(). The caller specifies the inode number of the desired inode and whether the inode should be locked exclusively or shared, and the function returns a pointer to the initialized in-core inode. This is usually done as part of file lookup. The inode is returned locked with the reference count of the inode's vnode incremented. Others may have references to the same inode, and the inode lock is used to synchronize accesses to the structure.

Step 2 Look at the inode

Once a process gets a hold of an inode it looks at it. If the process intends to modify the inode it should be locked exclusively, but if it is only reading the inode it should be locked shared. Actually, since inodes are usually found by the lookupname() routine, their user usually gets the inode/vnode with a reference but without a lock. The user then explicitly locks the inode with a call to xfs_ilock(). Once the inode is locked, the process holding it can read any of its fields. The inode can only be modified, however, if it has been brought into the context of a transaction.

Step 3 Modifying the inode

If the inode is to be modified it must be locked exclusively. If the caller did not obtain the inode with a call to xfs_trans_iget(), which is just like xfs_iget() except that it takes a transaction pointer, then it should be locked exclusively with a call to xfs_ilock() and added to the transaction with a call to xfs_trans_ijoin(). Once this is done the inode can be modified. When all changes to the inode have been made, the transaction mechanism needs to be told what within the inode has been changed and therefore needs to be logged.

Step 4 Logging the inode changes

The changes made to an inode can be logged as part of a transaction using the xfs_trans_log_inode() function. This function takes flags indicating which parts of the inode have been modified and therefore need to be logged. These flags are all defined in xfs_inode_item.h and will be described in more detail later on.

Step 5 Releasing the inode

Once a process is through looking at an inode or modifying an inode, the inode needs to be unlocked and released. If the inode was not used as part of a transaction, then the process can simply call xfs_iput() which will unlock the inode and release the reference to the inode's vnode. If the inode is being used as part of a transaction, then when the process calls xfs_trans_commit() the inode will be unlocked and its reference released. If the process wants to hold onto the inode even after the commit, then it needs to call xfs_trans_ihold() before committing the transaction. This tells the transaction code not to unlock or release the inode when the transaction commits.

Step 6 Writing back inode changes

When an inode is modified as part of a transaction, the dirty inode structure will remain in memory and the changes will be written into the on-disk log. At some point the inode will be written back to its on-disk home by either bdflush() calling xfs_sync() or by the inode structure being reclaimed for use as another inode. In either case, all modifications to the inode will at this point be written back to the disk and the on-disk log copy will no longer be needed for file system recovery.

Step 7 Destroying the inode structure

As mentioned above, the inode structure may at some point be recycled for use as another inode. This is the end of the in-core inode's life cycle as its memory is freed and reused for something else.

4.0 Inode Allocation

In-core inodes are allocated by calls to xfs_iget(). The function prototype for xfs_iget() is:

```
xfs_inode_t*xfs_iget(xfs_mount_t *mp, xfs_trans_t *tp, xfs_ino_t ino, uint flags).
```

The caller gives a pointer to the file system's mount structure, a transaction pointer if executing within a transaction, the inode number of the desired inode, and flags indicating whether the inode should be locked in shared or exclusive mode. The function returns a pointer to the in-core version of the desired inode. The inode is returned to the caller locked in the requested mode. The fields of the inode will be filled in according to the format of the on-disk inode.

If the inode is in LOCAL format, meaning that the file's data fits entirely within the on-disk inode, then i_u1.iu_data will point to an in memory array containing the contents of the file and i_bytes will contain the number of bytes in the array. This array will either be the i_u2.iu_inline_data array if the file data is less than or equal to 32 bytes or it will be an array allocated from the kernel heap. If the file is of length 0, then i_u1.iu_data will be NULL and i_bytes will be 0.

If the inode is in EXTENTS format, meaning that the file's data will not fit in the on-disk inode but the extent descriptors for the inode will, then i_u1.iu_extents will point to an in memory array containing the extent descriptors of the file and i_bytes will contain the number of bytes in the array. This array will be either the i_u1.iu_inline_ext array if there are only 1 or 2 extents or it will be an array allocated from the kernel heap. If the file is of length 0, then i_u1.iu_extents will be NULL and i_bytes will be 0. The XFS_IEXTENTS flag will be set in i_flags. This flag indicates that all of the file's extent descriptors have been read in and are in the i_u1.iu_extents array.

If the inode is in BTREE format, meaning that the file has too many extent descriptors to fit into the on-disk inode, then i_broot will point to an in memory array containing the file's extent b-tree root and i_broot_bytes will contain the number of bytes in the array. If the XFS_IEXTENTS flag is set in i_flags, then i_u1.iu_extents will point to an array containing all of the extents of the file as in the case above. If the flag is not set, then the extents have not yet been read in and should be read in with a call to xfs_ireadindir() if they are needed. Lazily reading in all of the extents for files with a large number of extents ensures that simple stat()s of the file can be done efficiently.

The returned inode will be hashed into a per file system in-core inode hash table. We are switching from the traditional single hash table for all file systems to per file system hash tables to improve the scalability of the system. Calls to xfs_iget() first look for the desired inode in this hash table before bringing in the inode from disk. An in-core inode is only removed from the hash table when it is recycled. The inode will also be placed on a list attached to the file system's mount structure. This list is used for traversal of all the in-core inodes in routines such as xfs_sync().

5.0 Inode in-line data/extents/b-tree root

This section describes the management of the iu_data, iu_extents, and i_broot fields. These fields point to arrays whose size must change as the size of the file changes.

5.1 iu_data

As described in the section on inode allocation, this field points to an array containing the in-line data of an inode in LOCAL format. When the size of a file in this format changes, the process changing the size should call xfs_idata_realloc() to resize the in-core array. This function takes the delta in the number of bytes needed. If the delta is positive then more memory will be allocated for the array, and if it is negative the array will be made smaller. If the size goes to 0 then iu_data will be made NULL. If the size is to become greater than will fit in the on-disk inode, then it is the responsibility of the process changing the size to perform a transaction to change the inode from LOCAL to EXTENTS format. As a part of that transaction the in-line data should be logged, the iu_data array should be freed with a call to xfs_idata_realloc(), and an array for the inode's extents should be allocated in iu_extents with a call to xfs_iext_realloc().

5.2 iu_extents

When an inode is in EXTENTS or BTREE format, this field points to an array containing all of the extent descriptors for that inode. If the file is in EXTENTS format, then this array is guaranteed to be there unless the file is of length 0. If the file is in BTREE format, then this array is read in when it is first needed and its presence is signified by the XFS_IEXTENTS flag in the inode's i_flags field. When the number of extents in the file changes, the process changing the number of extents should call xfs_iext_realloc() to resize the in-core array. This function takes the delta in the number of extents needed. It allocates and frees memory for the array as needed. If the number of extents goes to 0 then iu_extents will be set to NULL. If the number of extents exceeds some threshold which is yet to be determined, then the array will stop growing and the block mapping code will have to do slower access through its b-tree in the buffer cache.

The iu_extents array contains all of the extents of the inode sorted by file offset. It will be used by the block mapping code to quickly find the location of file disk blocks. This will be done by binary search of the array, possibly enhanced with a single entry cache for improving the efficiency of sequential access lookups.

5.3 iu_broot

When an inode is in BTREE format, this field points to an in-core copy of the on-disk inode's b-tree root. Like the other arrays mentioned above, this array only takes enough memory to hold the used portion of the b-tree root and must be resized dynamically as the root grows and shrinks. This is done by calls to xfs_iroot_realloc(), which takes the change in the number of b-tree records needed. This routine understands the format of the b-tree root, and it moves existing information within the root as is appropriate when the size changes. This means that when the size of the root increases by some number of records the pointers for the records are shifted towards the end of the data structure, and when the size shrinks the pointers are moved forward.

When the number of records in the b-tree root goes to 0, the data structure is not freed because it still contains the b-tree block header. If this is no longer needed, the process should call xfs_iroot_free() to release the memory used to contain the root. This should only be done if the inode is no longer in BTREE format.

5.4 i_bytes and i_broot_bytes

These two counters track the used bytes in the arrays pointed to by the corresponding iu_data/iu_extents and i_broot fields. For now the arrays are exactly the size that is currently needed, except when using the iu_inline_data/iu_inline_ext array. This implies that we must call kmem_-realloc() or something similar each time one of the xfs_i***_realloc() routines is called. We could keep more memory than we are currently using in an attempt to reduce the number of kmem calls, but this is currently being traded in favor of better memory utilization. If this turns out to be a high overhead decision in terms of cpu cycles, we can pretty easily change it.

6.0 Inode locking

As mentioned above, inodes can be locked in either shared or exlusive mode. This means we can have multiple readers of the same inode simutaneously. This should allow multiple readers and non-allocating writers of the file to work in parallel. Simultaneous file access is especially important for async I/O and for directories. Our async I/O implementation is based on threads, so allowing multiple threads to access the file at the same time can increase the pipeline of I/O requests to high throughput devices such as large, striped volumes. Directories are read doing path searches far more often than they are written, so allowing parallel access to popular directories should increase our pathname resolution performance. This has been a significant bottleneck in other file systems, so this should be a big win.

The inode lock will need to be held exclusively for anything updating the contents of the inode. The contents includes all fields which are contained in the on-disk inode and others as needed.

7.0 Inode transactions and logging

Almost all changes to an inode which will be reflected in the on-disk inode must be done within the context of a transaction and must logged within that transaction. The only exceptions to this may be the access, change, and modify times on the inode which are discussed in the next section. Once an inode has been modified, the transaction mechanism should be notified of the change with a call to xfs_trans_log_inode(). This function takes a set of flags indicating which parts of the inode have been changed. The flags are:

- XFS_ILOG_META -- This should be specified if any of the fields in the i_d sub-structure have been modified.
- XFS_ILOG_DATA -- This should be specified if the inline data of the inode has been changed.
- XFS_ILOG_EXT -- This should be specified if the iu_extents array has been modified and the file is in EXTENTS format.
- XFS_ILOG_BROOT -- This should be specified if the file is in BTREE format and the contents of the i_broot array have been modified.
- XFS_ILOG_DEV -- This should be specified if the i_u2.iu_rdev field has been modified.

These flags tell the transaction code which parts of the inode need to be logged when the current transaction commits. Each specified section is logged in its entirety, so specifying XFS_ILOG_META will log the entire xfs_dinode_core structure embedded in the in-core inode and specifying XFS_ILOG_BROOT will log the entire b-tree root. We will not be logging little, tiny pieces of the inode, because the overhead for tracking such pieces is as high as the overhead for copying them into the log.

When a transaction manipulating an inode commits, the inode is unlocked and the reference to the inode is released. In order to prevent dirty inodes from being reclaimed, which could become a performance problem if they are reclaimed while they are still pinned by a transaction, when a clean inode is logged an additional reference to the inode will be taken by the transaction code. This reference will be released when the inode is flushed back to disk to clean it.

8.0 Inode timestamp updates

The access, modify, and change timestamps within the inode are updated on almost every file system operation. Requiring the inode to be locked exlusively and updating the timestamps within transactions would be overly expensive and would eliminate the benefits we hope to gain in using the multi-reader inode lock. Instead, inode timestamps will be updated outside the protection of even the inode lock.The scheme will work as follows.

Whenever a process needs to update a timestamp or timestamps, it will simply store the current time into the timestamps. **After** updating the timestamps it will set the i_update_core field in the inode to 1 to indicate the the inode is dirty. This field is only cleared from xfs_iflush(), which does so **before** copying the timestamps from the in-core inode to the on-disk inode. This ordering of the i_update_core updates, given that our machines support a strongly ordered memory model, ensures that we will never miss an update and consider a dirty inode to be clean.

The timestamp fields themselves will be 64 bit fields of which we will currently only use the upper 32 bits to store the time in seconds. The other 32 bits will be left unused in case we need finer grade timestamps in the future. Since 32 bit stores are atomic, each individual timestamp will be consistent at all times. However, there is no locking around multiple timestamp updates, so updates to multiple timestamps at once will not be atomic. In the normal case anyone updating the timestamps is putting in the current time so this should not matter, but this is not the case for the utimes(2) system call. This system call is used to set the timestamps to arbitrary values. Since we will have no locking it will be possible to mix the results of a utimes(2) system call and a normal path timestamp update. In the name of performance, since fixing this would require a lock, we are not going to worry about this.

9.0 Inode flushing

Dirty inodes are flushed to disk either by calls to efs_sync() from the bdflush daemon or by the transaction management code when an inode's log image is too far back in the log. The inode must be locked in shared mode to prevent other processes from modifying the inode but still allow other processes to look at the inode while it is being written to disk. Since multiple processes could attempt to flush the inode simultaneously, the i_flock will be used to synchronize the flushing of an inode. This is necessary both for performance and for correctness. For performance we don't wan't to do unnecessary work. For correctness we must make sure that the one reference on the inode taken by the transaction management code is not used or released by multiple processes. Once the reference is released, it will be possible for the inode to be recycled. Thus only one process can assume that the reference will protect it.

The routine to perform the actual flushing of the inode is xfs_iflush(). This routine will obtain the buffer for the disk block of the inode from the buffer cache, copy the inode into that buffer, and write the buffer back to the disk synchronously, asynchronously, or delayed write. If the inode is pinned in memory (because it is a part of a transaction which has not yet been committed to disk) this routine will sleep until it is unpinned. When the inode is not pinned the routine will attach the function xfs_iflush_done() and the inode's log item to the buffer's b_iodone function and the b_fsprivate pointer.[1]This routine will be executed when the write completes. Its purpose is to remove the inode from the file system's Active Item List, mark the inode clean, and release the reference to the inode taken by the transaction code. Finally, the inode will be marked clean, unlocked and the buffer write initiated.

Since the buffer is unlocked in xfs_iflush(), it is possible for the inode to be dirtied again before the write completes and xfs_iflush_done() executes. In this case the inode may even be moved forward in the AIL, meaning that the flush being completed by xfs_iflush_done() does not have the right to remove the inode from the AIL. In order to coordinate this, xfs_iflush_done() will do the following:

1. These will be attached using the xfs_buf_attach_iodone() function. This will chain the inode log item onto a chain of log items rooted at the buffer's b_fsprivate field. This will require the buf log item code to handle having the buf log item not being pointed to directly by the buf's b_fsprivate field. It will get us, however, a clean, generic mechanism for attaching log item's to the io completion of buffers.

- First look at the inode’s LSN without obtaining the AIL lock (which protects this field). If it has changed then the inode has moved or is moving in the AIL and we should not bother with it.
- If the value has not changed then get the AIL lock and if the value has still not changed then remove the inode from the AIL.
- Next, release the inode’s i_flock.
- Finally, release the reference to the inode.

Once the reference is released, we can no longer manipulate or look at the inode and we are done. Note that we are not doing anything special with the inode reference for the case where the inode is re-dirtied while the disk write is taking place. Since the inode will be marked clean in xfs_i-flush() before releasing the lock, any process modifying the inode during the write will obtain another reference for the transaction code. This allows us to just drop the one that we are using.

10.0 Inode recycling

At some point a call to xfs_reclaim() will want to recycle an inode which is not being referenced. The inode is guaranteed to have no references at this point, so we know that it is not dirty. All we need to do is flush any dirty file data associated with the inode, remove the inode from the mount structure’s list of in-core inodes, and free all memory associated with the inode.