xFS Superblock Management

Adam Sweeney

1.0 Introduction

In the xFS file system, the superblock is a centralized resource which is modified by most transactions. This makes its potential for becoming a bottleneck very high. The reason for this is that once a transaction modifies a resource, that resource cannot be made visible to other transactions until the first is committed. If a resource such as the superblock is accessed by many transactions and each holds the resource for a significant period of time, the transactions will become bottlenecked waiting for access to the resource. In order to prevent this problem, in xFS the superblock will be modified through routines designed to minimize the amount of time the superblock is kept locked.

The reason the superblock is modified so much is that it contains counters of the total number of inodes in the file system, the total number of free inodes in the file system, and the total number of free blocks in the file system. It is these counters that will be modified most of the time, so it is updates to them that will be optimized. This is consistent with the general rule of optimizing for the common case.

While updates to the superblock counters are the most common, other fields of the superblock will need to be modified within transactions occasionally. The interfaces for this uncommon path must work with those for the fast path without breaking anything. All of this must also be coordinated with access to the in-core copy of the superblock.

This document describes the mechanisms to be used in xFS to solve these problems. The solutions chosen and the programming interfaces derived from those solutions are described below.

2.0 Superblock management

This section gives a brief description of the schemes chosen for superblock managment in xFS.

2.1 The superblock buffer

Instead of being kept in a common buffer cache buffer accessed through the normal getblk()/bread() path, the xFS superblock will be kept in a buffer private to the file system. This way the buffer cache code, such as that in bdflush(), will never mess with our superblock buffer. This means, of course, that we are entirely responsible for making sure that the superblock is flushed to disk when necessary. Since the xfs_sync() routine will be called periodically anyway, flushing the superblock from there when necessary should be more than enough.

The superblock buffer will simply be pointed to by a pointer kept in the mount structure. Access to the buffer will be through the xfs_getsb() and xfs_trans_getsb() routines described in detail below. These routines are responsible for properly synchronizing access to the superblock buffer. The buffer will only be read in from disk at mount time, and from then on the buffer will be kept in sync with the on-disk copy of the superblock. This ensures that accesses to the superblock are never delayed to do I/O because some other resource forced the buffer to be recycled.

2.2 Superblock modifications

In order to minimize the amount of time the superblock buffer is kept locked during a transaction, the superblock will not actually be locked and changed until just before the transaction commits. This will ensure that the buffer is not held locked while waiting for other resources to be read in from disk or to be released by other transactions. This works fine for the counters which are modified in the common case, as their updates consist of amounts to add or subtract from counters rather than absolute numbers. These updates can be delayed until the end of the transaction without violating the correctness of the transactions. The user of a transaction will indicate that a change needs to be applied to a superblock counter with a call to xfs_trans_mod_sb(). This routine will be responsible for recording the requested change and making sure that it is applied to the superblock as part of the transaction.

When fields of the superblock other than the counters need to be modified, the superblock buffer can be accessed just as any other buffer except that it must be obtained with a call to xfs_trans_getsb() rather than xfs_trans_getblk()/xfs_trans_bread(). Calls to xfs_trans_log_buf() using the superblock buffer will work fine and can even be intermixed with the use of xfs_trans_mod_sb().

2.3 In-core superblock

The in-core superblock can be used to look up information about the file system which doesn't change as well as system summary information. The superblock buffer should only be accessed when something is being changed. The fields in the in-core superblock which do change are protected by the m_sb_lock spin lock in the mount structure. This lock can be used to ensure that what is being looked at is consistent. The code that modifies the in-core superblock after a transaction which modified the superblock commits uses this lock to protect its updates.

3.0 Superblock managment interfaces

3.1 xfs_trans_mod_sb()

xfs_trans_mod_sb() is used to apply changes to the counters in the superblock without immediately locking the superblock buffer. The prototype for the function is:

```
void   xfs_trans_mod_sb(xfs_trans_t *tp, uint field, int delta);
```

The field argument specifies to which of the counters the number passed in the delta parameter should be added. To subtract from a given counter a negative value should be passed in delta. The valid values for the field parameter are:

```
       XFS_SB_ICOUNT -- Apply the delta to the sb_icount field
       XFS_SB_IFREE -- Apply the delta to the sb_ifree field
```

```
      XFS_SB_FDBLOCKS -- Apply the delta to the sb_fdblocks field
      XFS_SB_FREXTENTS -- Apply the delta to the sb_frextents field
```

When xfs_trans_commit() is called the superblock buffer will be locked by the transaction and all specified deltas will be applied to it. The deltas are commulative, so the same field may be specified in multiple calls to xfs_trans_mod_sb() within a given transaction. Once the transaction commits, the deltas will also be applied to the in-core copy of the superblock.

3.2 xfs_trans_getsb()

xfs_trans_getsb() is used to lock the superblock buffer within a transaction. This should only be used when the transaction needs to modify fields in the superblock other than those that can be modified by xfs_trans_mod_sb(). The results of this function are just like those of xfs_trans_-bread(), but it is only used to obtain the superblock. The function prototype is:

```
buf_t  *xfs_trans_getsb(xfs_trans_t *tp);
```

The buffer can be released with a call to xfs_trans_brelse(). No special call is necessary.

3.3 xfs_getsb()

xfs_getsb() is just like xfs_trans_getsb() except that it can be used outside of a transaction. It returns a pointer to the locked superblock buffer. This buffer should never be modified, however, as the superblock can only be updated within transactions. Most of the time the information needed should also be available from the in-core superblock, so use of this function is discouraged. The prototype for the function is:

```
buf_t  *xfs_getsb(xfs_mount_t *mp);
```

3.4 xfs_mod_incore_sb()

xfs_mod_incore_sb() is used to modify the in-core copy of the superblock. Its prototype is:

```
int    xfs_mod_incore_sb(xfs_mount_t *mp, uint field, int delta);
```

The field parameter indicates the field of the superblock to which to apply the given delta. This routine takes care of acquiring the spin lock protecting the in-core superblock. It currently only supports updates to the fields available through xfs_trans_mod_sb() specified above, but this can be expanded as necessary.

This routine enforces the assumption that the counters in the superblock never go below 0. If a delta is specified that would cause such a condition, then the delta will not be applied and the routine will return EINVAL.

3.5 xfs_mod_incore_sb_batch()

xfs_mod_incore_sb_batch() is used to apply multiple deltas to multiple fields in the superblock. It takes an array of xfs_mod_sb_t structures each of which specify a field and a delta for that field. By allowing the caller to specify multiple deltas to be applied to the superblock, this routine allows multiple updates to be atomically applied and reduces the locking overhead necessary for multiple calls to xfs_mod_incore_sb(). The prototype for the function and the xfs_mod_sb_t definition are:

```
typedef struct xfs_mod_sb {
   uint      msb_field;          /* the field to which to apply msb_delta */
   int       msb_delta;          /* the amount to add to the specified field */
xfs_mod_sb_t;

int    xfs_mod_incore_sb_batch(xfs_mount_t *mp, xfs_mod_sb_t *msb, uint nmsb);
```

Like xfs_mod_incore_sb(), this routine handles the locking protecting the superblock and enforces the restriction that no counter in the superblock may go below 0. If any of the specified deltas would cause such a condition, then none of the deltas will be applied and the function will return EINVAL.