xFS Transaction Mechanism

Adam Sweeney

1.0 Introduction

This document describes the transaction model to be used by xFS. It also specifies the interfaces exported to support this model. The model must support transactional updates of both file system meta data and user data. It should provide a clean, efficient interface to the logging and recovery mechanisms provided by the log management code. This mechanism must also be cleanly integrated with the system buffer cache code.

The xFS transaction mechanism will consist of a set of routines and data structures forming a layer above the buffer cache, inode management, and logging interfaces. It will manage the locking of resources and the necessary ordering of writing changed resources back to disk. This will be done with interfaces very similar to traditional buffer and inode managment interfaces as well as a few new, transaction-specific routines.

2.0 The Model

The xFS transaction model will support image and a limited form of operation logging. Operation logging, however, will be used only when image logging is insufficient or overly cumbersome for the required operation. The reason image logging will be the preferred mechanism is that it is simpler to implement and use. The transaction model is most easily described by a general description of how it is used. This is outlined below.

2.1 The Steps of a Transaction

Step 1 Allocate a Transaction

The first step in performing a transaction is to allocate a transaction structure and unique transaction identifier. This structure will be used to track all of the important events and information associated with an individual transaction. The transaction identifier will be used to tag all data in the on disk log which are related to a given transaction.

Step 2 Reserve Log Space

Once a transaction has been allocated, log space for that transaction must be reserved. Log space reservation is used to ensure that we never overrun the tail of the log. The transaction user must specify the maximum amount of log space that will be necessary to perform the given transaction. When each transaction ensures that there is enough log space for it to commit before locking any resources, we can avoid a very serious potential deadlock. The deadlock occurs when there is not enough log space for an active transaction to commit, but the tail of the log cannot be moved for-

ward because the dirty resource logged at the tail which must be flushed to move the tail is locked by the active transaction. By ensuring that each transaction will have enough space to commit before allowing it to lock any resources we avoid this deadlock.

If the amount of free log space needed by a transaction is not currently available, the transaction will sleep until it becomes available or give up. Space will be freed up as transactions commit and dirty resources are flushed out to disk. Determining the necessary amount of space for a given transaction may be difficult, but we will just have to do so by looking at what can be logged by a given transaction.

Step 3 Lock Resources

Actually, this step and the step below are intermingled by the transaction user. This step consists of calls to read in and lock resources such as buffers and inodes. Most transactions must follow two phase locking, meaning that no resource locked as part of a transaction can be unlocked until all resources to be locked as part of that transaction have been locked. That is, sequences such as lock A, unlock A, lock B are illegal. Also, no modified resource can be unlocked until the transaction has been committed to the in-core log. Once a transaction's modifications have been committed to the in-core log, all resources locked by that transaction can be released. This is because all changes are being serialized by the log. Any change made to a resource after a previous change to it has been logged will always appear to happen after the first, which is the correct behavior.

There are cases where an unmodified resource can be unlocked before some other resource is locked by a transaction, but these are specialized cases. One occurs in tree locking where in some cases a parent node may be unlocked once its child node is locked. The ability to take advantage of such cases will be provided by the transaction mechanism, but we must take care in using it.

Step 4 Modify Resources/Specify Operations

Once a resource is locked as part of a transaction it can be modified. The parts of a resource which are changed must be remembered so that they can be written to the log as part of the transaction commit. A modified resource cannot be unlocked until the transaction commits.

The transaction user at this stage can also create transaction operation structures for use in operation logging. These structures will describe the operation being performed for use by the log recovery code in the event of a system failure.

Step 5 Commit Transaction

Once all of the necessary resource changes have been completed, the transaction can be committed. Transaction commit will record all resource modifications and transaction operations in the in-core log, pin all resources in memory until the transaction makes it to the on-disk log, and unlock all resources associated with the transaction. At this point the transaction is not yet permament. It becomes permanent when the in-core log is written out to the on disk log. The user of the transaction mechanism at this point can either consider him or herself done and forget about the transaction or wait for the transaction to be written to disk.

Another wrinkle in transaction commit is that the transaction user will be able to specify whether or not a given locked resource should be unlocked. There are certain long running transactions which will actually be implemented as multiple transactions. These will require keeping some of their resources locked across all of the involved transaction commits.

Once a resource is unlocked by a transaction it is eligible for use by other transactions. This may result in the resource being associated with multiple transactions simultaneously. This will be handled cleanly by the transaction mechanism without any special intervention by the transaction mechanism user.

Step 6 Log Write Completes

At some point the in-core log will be written out to disk, and at that point the transaction has become permanent and its modified resources can be unpinned. This will be performed automatically by the transaction mechanism. The transaction structure will also be freed at this time.

All modified resources and operation records associated with the transaction will be placed in the Active Items List (AIL) at this time as well. This is a list used by the transaction and logging code to track the location of active data in the log. A modified resource recorded in the log is active until the copy recorded in the log is written out to the on disk image of the resource. Tracking the location of dirty items in the log allows us to prevent the overwriting of the tail of the log.

Step 7 Modified Resource is Flushed

When a resource is unpinned because the log write containing its modifications has completed, that resource is eligible to be flushed out to disk. This will happen when the resource is to be reused for something else, the tail of the log approaches the location of the resource in the log, or some cleaning entity such as the bdflush daemon pushes it out. Once the dirty in-core copy of a resource is flushed to its on disk image, the resource will be removed from the AIL. At that point the resource has no ties to the transaction system and the cycle is free to start over again.

3.0 Transaction Mechanism Interfaces

This section describes the function and structure interfaces exported for use by the transaction mechanism users. Interfaces for performing all of the steps described in the section above are specified here. They will be listed in an order comparable to that of the previous section.

3.1 Function Interfaces

This section describes the function interfaces to be used by file system code for performing atomic updates to file system data and meta-data. The design of these interfaces is driven by how they will be used. I have tried to encapsulate the common actions which will be performed by file system code in order to reduce redundant functionality throughout the file system. The result of this is that there are a few completely new transaction specific routines and many more extended buffer cache and inode management routines. While, as described in a following section, the

internal structure of the transaction layer is designed to be independent of the type of data being logged, the interface routines described here are quite type dependent. They will have knowledge of both the management of the objects they manipulate and the internals of the transaction mechanism.

3.1.1 xfs_trans_t *xfs_trans_alloc(struct mount *mp, uint type, uint reserve, uint flags)

This is the routine called to allocate a transaction structure and reserve log space for that transaction. If the flags parameter is set to TRANS_NOSLEEP, then the routine will return NULL if it cannot reserve the requested amount of log space rather than sleeping until it is available. The mount point structure should be that of the file system the transaction will be manipulating, and the type field should specify the transaction type. These types will be enumerated later.

3.1.2 void xfs_trans_callback(xfs_trans_t *trans, void(*)(xfs_trans_t*, void*) callback, void *arg)

This routine allows the transaction user to specify a routine to be called upon the completion of the log write which writes the transaction to the on disk log. The user is also allowed to specify a pointer which will be passed to the callback in addition to the transaction structure. This routine will be called before the normal transaction completion processing, but not in place of it. The transaction structure and its contents should not be modified by this routine.

3.1.3 buf_t *xfs_trans_getblk(xfs_trans_t *trans, dev_t dev, daddr_t blkno, int len)

This is called to assign a buffer to the specified block(s). If the buffer already exists and is locked by someone else, the routine will sleep until it becomes available. If the buffer is already locked within the given transaction it is just returned to the caller. If the buffer is not yet in the cache it will be allocated and returned to the user. The buffer is always locked upon return to the caller.

This routine will allocate an xfs_buf_log_item structure for the buffer if it does not yet have one. This structure (described in detail below) is what is used by the transaction code internals to manipulate buffers within a transaction.

3.1.4 buf_t *xfs_trans_bread(xfs_trans_t *trans, dev_t dev, daddr_t blkno, int len)

This routine is just like xfs_trans_getblk(), except that it ensures that the buffer is completely read in from disk before returning it to the caller.

3.1.5 buf_t *xfs_trans_getchunk(xfs_trans_t *trans, vnode_t *vp, struct bmapval *bmap, struct cred *cred)

This routine is just like xfs_trans_getblk(), except that the buffer specified is a part of the chunk cache rather than the block buffer cache. This is used for getting buffers associated with file data within a transaction.

3.1.6 buf_t *xfs_trans_chunkread(xfs_trans_t *trans, vnode_t *vp, struct bmapval *bmap,

struct cred *cred)

This routine is just like xfs_trans_getchunk(), except that it ensures that the buffer is read in before returning it to the caller. This routine differs from xfs_chunkread() in that only a single bmapval structure can be specified.

3.1.7 void xfs_trans_brelse(xfs_trans_t *trans, buf_t *bp)

This routine releases the given buffer which must have been allocated with one of the above xfs_trans_xxx() buffer allocation routines or added to the transaction with xfs_trans_bjoin(). This will decrement the lock recursion count on the given buffer. If the count goes to 0 the buffer will be unlocked and disassociated from the transaction. The buffer must not have been modified within this transaction, because we have no way to restore it to its previous state.

If the xfs_buf_log_item structure associated with the buffer is not needed because the buffer has no logged data, then it will be freed.

3.1.8 void xfs_trans_bjoin(xfs_trans_t *trans, buf_t *bp)

This routine is called to add an already locked buffer to the given transaction. Use of this routine is discouraged, but there may be some cases where it is necessary. If the buffer does not yet have an xfs_buf_log_item structure associated with it one will be allocated.

3.1.9 void xfs_trans_bhold(xfs_trans_t *trans, buf_t *bp)

This routine can be called with a buffer locked within the transaction to prevent the buffer from being unlocked when the transaction commits. It is the responsibility of the caller to track the buffer and unlock it eventually.

3.1.10 void xfs_trans_iget(xfs_trans_t *trans, xfs_ino_t ino, struct xfs_inode **ipp)

This routine will return the inode with number ino in the ipp parameter. The file system to which ino is relative is already known since it was specified in xfs_trans_alloc(), so it is not necessary to specify it here. The inode is returned locked to the caller, and if it is already locked within the transaction it is simply returned to the caller.

3.1.11 void xfs_trans_irelse(xfs_trans_t *trans, struct xfs_inode *ip)

This routine releases the given inode which must have been allocated with xfs_trans_iget() or added to the transaction with xfs_trans_ijoin(). The inode is unlocked and disassociated from the transaction. The inode must not have been modified within the scope of the transaction.

3.1.12 void xfs_trans_ijoin(xfs_trans_t *trans, struct xfs_inode *ip)

This routine is called to add an already locked inode to the given transaction. Use of this routine is discouraged, but there may be some cases where it is necessary. If the indoe does not yet have an xfs_inode_log_item structure associated with it one will be allocated.

3.1.13 void xfs_trans_ihold(xfs_trans_t *trans, struct xfs_inode *ip)

This routine can be called with an inode locked within the transaction to prevent the inode from being unlocked when the transaction commits. It is the responsibility of the call to track the inode and unlock it eventually.

3.1.14 void xfs_trans_log_buf(xfs_trans_t *trans, buf_t *bp, uint first, uint last)

This routine is called to notify the transaction that bytes first through last inclusive of the given buffer have been modified and must be logged at commit time. The transaction layer is free to log more bytes than those specified, but it must log at least bytes first through last.

3.1.15 void xfs_trans_log_inode(xfs_trans_t *trans, xfs_inode *ip, uint fieldmask)

This routine is called to notify the transaction that the fields indicated in the fieldmask bitmask have been modified and must be logged. The values to be specified in fieldmask will be defined later.

3.1.16 void xfs_trans_log_op(xfs_trans_t *trans, xfs_log_item_t *op)

This routine will add the log operation structure given to the list of things which need to be logged when the transaction commits.

3.1.17 trans_id_t xfs_trans_id(xfs_trans_t *trans)

This function returns the unique transaction id of given transaction structure. This is for use in log operations which require making references to other transactions.

3.1.18 void xfs_trans_commit(xfs_trans_t *trans, uint flags)

This routine moves all data which has been specified as needing to be logged to the incore log. All resources which are logged are pinned, and all resources are unlocked unless it has been specified that they should not be. If the flags specify that the log should be flushed immediately then it will be done before returning to the caller. The flags may also indicate that the caller cannot sleep (perhaps when calling from an interrupt handler), in which case the unlocking of resources and copy into the log will happen asynchronously. Finally, the flags can indicate that the caller wants to wait for the commit of the transaction to the on disk log before returning.

If nothing has been logged within the transaction, then almost none of the above will occur. Instead, all resources will be unlocked unless it has been specified that they should not be, any transaction completion function will be called, and the transaction structure will be freed.

3.1.19 void xfs_trans_cancel(xfs_trans_t *trans)

This is called to cancel a transaction by unlocking all of its resources. None of the buffers or inodes can be modified within the transaction when this is called, because there is no way to restore them to their previous states.

3.2 Exported Structures

3.2.1 xfs_log_item

In order to keep the transaction management code simple and modular, all logged structures will be manipulated through a common internal interface. This interface will allow the code to manage buffers, inodes, and log operation structures identically. The xfs_log_item is the common structure among all things which can be logged. It consists of a small amount of data and a vector of function pointers used to manipulate the log item. It is the function pointers which contain the type specific code for manipulating various types of log items. The xfs_log_item is defined as:

field name	type	comment
li_parent	struct xfs_log_item *	active item list pointers
li_left	struct xfs_log_item *	
li_right	struct xfs_log_item *	
li_type	unsigned int	item type label
li_lsn	xfs_lsn_t	log sequence number of the location of the item in the log
li_ops	xfs_item_ops_t *	item specific operation pointers

The xfs_item_ops structure is defined as:

field name	type	comment
iop_size	uint (*)(xfs_log_item*)	Return the amount of space needed to log the item
iop_format	void (*)(xfs_log_item*, caddr_t)	Write data to be logged into the given buffer
iop_pin	void (*)(xfs_log_item*)	Pin the item in memory
iop_unpin	void (*)(xfs_log_item*)	Unpin the item
iop_trylock	uint (*)(xfs_log_item*)	Lock the item
iop_unlock	void (*)(xfs_log_item*)	Unlock the item
iop_committed	xfs_lsn_t(*) (xfs_log_item*, xfs_lsn_t)	Notify item of its latest lsn, let it return new lsn if any for the AIL
iop_push	void (*)(xfs_log_item*)	Try to flush the item to its on disk image because its space in the log needs to be reused.

The behavior of each of the functions in the xfs_item_ops structure is described below.

3.2.1.1 uint iop_size(xfs_log_item_t *item)

This will be called by the transaction layer to determine how much space is needed to log the given item. This number should include the space for a header describing the item and its on disk location, information describing which parts of the item are being logged, and the actual data from the item.

3.2.1.2 void iop_format(xfs_log_item_t *item, caddr_t buf)

This will be called after a call to the iop_size routine to put the data to be logged into the in-core log. The buffer given to the routine will be aligned on at least a 32 byte boundary. The routine should record enough information to allow the recovery code to understand and restore the item being logged. This should include (as mentioned above): a header describing the item and its on disk location, information describing which parts of the item are being logged, and the actual data from the item.

The first 8 bytes of data written by the format function must consist of a 4 byte log item type field followed by a 4 byte unsigned integer field indicating the size of the entire log item record in the log.

3.2.1.3 void iop_pin(xfs_log_item_t *item)

This will be called to pin the given item in memory. The item is guaranteed to be locked when this routine is called. After this call, it must be impossible to flush the item to disk until the corresponding call to iop_unpin().

3.2.1.4 void iop_unpin(xfs_log_item_t *item)

This will be called to unpin a given item. It should release the item to be flushed to disk whenever it is convenient.

3.2.1.5 uint iop_trylock(xfs_log_item_t *item)

This routine is called to attempt to lock the given item. It should attempt to lock the item, but it is not allowed to sleep because it will be called holding a spin lock. The routine should return 1 on success and 0 on failure.

3.2.1.6 void iop_unlock(xfs_log_item_t *item)

This function will be called to unlock the item once its transaction has committed.

3.2.1.7 xfs_lsn_t iop_committed(xfs_log_item_t *item, xfs_lsn_t lsn)

This will be called when a log write completes to notify the log item that it has been committed to disk and to determine if the position of the log item in the AIL should change. It passes in the LSN of the log write which just completed. The routine should return the new LSN to be used to place the item in the AIL. This should be the LSN of the copy of the item in the log which is furthest back in the log but is still needed for recovery. This gives the item control over when its log images become inactive.

If the routine returns the value -1, then the position of the item in the AIL will not be updateed and the item will not be referenced again by the committed transaction. The same behavior will occur if the routine returns the same value that is in item->xli_lsn before the call. If the value returned is different, then the xli_lsn field of the item will be changed to the returned value and the location of the item in the AIL will be updated to reflect this value.

3.2.1.8 void iop_flush(xfs_log_item_t *item)

This routine will be called to asynchronously write the item out to disk. It is guaranteed that the item has already been successfully locked by a call to iop_trylock(). If the item is no longer dirty when this is called, then it should just return immediately. It is alright for this routine to sleep if it must, but this is discouraged.

4.0 Transaction Mechanism Internals

This section describes the internal structures, interfaces, and algorithms of the transaction layer.

4.1 Transaction Structure

The transaction structure is used to track all of the information relevant to a given transaction. The xfs_trans structure is described below:

field name	field type	comment
t_tid	xfs_trans_id_t	transaction id
t_reserve	uint	log reservation amount
t_type	uint	transaction type
t_forw	struct trans *	active transaction list pointer
t_back	struct trans *	active transaction list pointer
t_sema	sema_t	transaction commit completion semaphore
t_mountp	struct mount *	pointer to mount structure of file system
t_callback	void(*)(xfs_trans_t*, void*)	transaction completion callback function
t_item_descs_free	uint	count of free item descriptors
t_item_descs	log_item_chunk_t	first chunk of log item descriptors

The log items currently associated with a transaction are tracked by a chunk list of log item descriptor structures. These form a list of structures pointing to log item structures. We can’t link

through the log item structures because a single log item can be associated with multiple transactions simultaneously, although it can only be locked by one at a time. The chunk list reduces the number of allocations and deallocations we need to do. The t_item_descs_free field tracks the number of free log item descriptors in the chunks that have already been allocated. Tracking this number prevents us from searching for free descriptors which are not there. The xfs_log_item_-desc structure is defined as:

field name	field type	comment
lid_item	xfs_log_item_t *	log item pointer
lid_flags	uint	misc information

These log item descriptors are kept in a chunk list of the following xfs_log_item_chunk structure:

field name	field type	comment
lic_next	struct xfs_log_item_chunk*	next chunk
lic_free	uint	free descriptor mask
lic_descs	struct xfs_log_item_desc[15]	log item descriptors

When checking to see if an item is already locked by a transaction, we could search the log item descriptor list for the item. However, this would get really slow when the number of already locked items grows large. Instead, we'll be using the structures usually used to look up such items, for example the buffer cache hash table, and looking at the actual item to decide if we already have it locked. This will require an extra pointer to the transaction structure in each item, not in the log item structure, but it is required for adequate performance.

4.2 Active Item List

All log items which reside in the active portion of the on disk log, meaning that their changes have not yet been flushed to their on disk images, will be linked into the Active Item List (AIL). This list (it will probably actually be a balanced tree for performance reasons) will be sorted by the log sequence numbers of the items, and it will be used to track the tail of the log. When an item is flushed to its on disk image it will be removed from the AIL, and when it is re-logged it will be moved up in the AIL according to its new log sequence number. There will be one AIL per file system. Each will be guarded by a spinlock which must be held for moving, adding, or removing an item in the AIL.

The log manager will have a process responsible for pushing the tail of the log forward when the head of the log starts to approach the tail. This is what the iop_flush() routine of each log item is to be used for. The log manager process will do the following to push an item out of the AIL:

Obtain the AIL spinlock. Find the first log item in the AIL. Call item->iop_trylock() to attempt to lock the item. If we succeed, unlock the AIL spinlock and then call item->iop_flush(). If we fail then look for other items which need to be pushed. We will assure that if the item is locked, then the one holding the lock will flush the buffer when unlocking it. The items will be removed from the AIL by their I/O completion routines, because that is when we know that on disk log image of the item is no longer needed.

4.3 Buffer Log Item

The buffer log item is the log item structure expanded for the extra needs of logged buffers. To most of the transaction code it will look like a common log item, but it will contain buffer specific data for use by the buffer specific log item operations. The xfs_buf_log_item structure is defined as:

field name	field type	comment
bli_item	xfs_log_item_t	common item structure
bli_buf	buf_t *	real buffer pointer
bli_recur	uint	lock recursion count
bli_map_size	uint	size of the dirty bitmap in bytes
bli_dirty_map	uint[1]	variable sized bitmap of dirty 128 byte regions in the buffer to be logged

The buffer pointer points to the buffer described by this item structure, and the recur field is used to track the number of times the buffer has been locked within the current transaction. The dirty bitmap and its size field are used to track which bytes in a buffer need to be logged when the current transaction is committed.

For buffers, all logging will be cumulative. What this means is that all dirty data in a buffer which has not yet been flushed back to disk will be logged each time the buffer is logged, whether the dirty data belongs to the current transaction or not. This will allow us to move the buffer forward in the AIL each time it is logged, and we will only need to track a single LSN for each buffer.

4.3.1 Buffer Log Item Operations

4.3.1.1 void xfs_buf_item_format(xfs_log_item_t *buf_item, caddr_t buffer)

The image of the buffer log item will be laid out in the following xfs_buf_log_format structure:

field name	field type	field size
blf_type	uint	4 bytes
blf_size	uint	4 bytes
blf_dev	dev_t	4 bytes
blf_blkno	daddr_t	8 bytes
blf_map_size	uint	4 bytes
blf_data_map	uint[blf_map_size / 4]	multiple of 4 bytes
blf_padding	uint	pad to 32 byte boundary
blf_data	uint[variable]	128 byte blocks of data as indicated by blf_data_map bitmap

The type field will indicate that this is a buffer log item. The size field will indicate the total size of the log item. The device field indicates the number of the device this data is to be written to, and the block number field gives the starting block for the data described in this log item. The map size and data map fields describe which data for the given block are logged by this item, and the actual data is written in the data field in 128 byte chunks. The data field is padded to start on a 32 byte boundary so that data copies go faster.

4.3.1.2 uint xfs_buf_item_size(xfs_log_item_t *buf_item)

This just returns the amount of space needed for the structure described in the xfs_buf_item_format() description.

4.3.1.3 void xfs_buf_item_pin(xfs_log_item_t *buf_item)

This function will assert that the buffer is locked and then call bpin() on the buffer associated with the buffer log item.

4.3.1.4 void xfs_buf_item_unpin(xfs_log_item_t *buf_item)

This function will simply call bunpin() on the buffer associated with the buffer log item.

4.3.1.5 int xfs_buf_item_trylock(xfs_log_item_t *buf_item)

This will perform a cpsema() on the semaphore of the buffer associated with this buffer log item. If it can grab the buffer, it will remove it from the free list, claim the buffer, and return 1. If it cannot get the buffer semaphore, then it will return 0.

4.3.1.6 void xfs_buf_item_unlock(xfs_log_item_t *buf_item)

This will be called to unlock a buffer log item previously locked via a call to an xfs_trans “get buffer” routine (e.g. xfs_trans_bread()). It will do this by a call to xfs_brelse() with the buffer associated with the given buffer log item. If the buffer has no data in the active or in-core logs, then the xfs_buf_log_item structure is no longer needed, and it will be freed before releasing the buffer.

4.3.1.7 xfs_lsn_t xfs_buf_item_committed(xfs_log_item_t *buf_item, xfs_lsn_t lsn)

This routine will always return the lsn value passed in to it. This is because the buffer always relogs all dirty data in the buffer, so the most recent log version is the only one needed.

4.3.1.8 void xfs_buf_item_flush(xfs_log_item_t *buf_item)

This function will write the buffer out to its place on disk if it is still necessary. First it will check that the buffer is marked B_DELWRI. If this flag is not set then the routine will just unlock the buffer and return. If the flag is set, then it will call bawrite() with the buffer to write it out asynchronously.

The I/O completion routine for the buffer will remove the xfs_buf_log_item structure from the AIL, free it, and unlock the buffer.

4.4 Inode Log Item

The inode log item structure, like the buffer log item structure, will extend the common log item structure with inode specific data. The xfs_inode_log_item structure will look like:

field name	field type	comment
ili_item	xfs_log_item_t	real log item
ili_inode	xfs_inode_t *	real inode
ili_recur	uint	lock recursion count
ili_field_mask	uint	logged field mask

The inode field will point to the inode associated with this inode log item, and the recur field will count the number of times the inode has been locked recursively. The fieldmask field will be a bit-map of fields in the inode which have been dirtied and need to be logged.

As with buffers, inode logging will be cumulative. All dirty fields in the inode will be logged every time the inode is logged in order to keep inodes seeing heavy traffic moving forward in the log.

4.4.1 Inode Log Item Operations

4.4.1.1 void xfs_inode_item_format(xfs_log_item_t *inode_item, caddr_t buffer)

The inode log item will be laid out in log with the following xfs_inode_log_format structure:

field name	field type	field size	comment
ilf_type	uint	4 bytes	log item type indicator
ilf_size	uint	4 bytes	log item size indicator
ilf_ino	xfs_ino_t	8 bytes	inode number of inode
ilf_blkno	daddr_t	8 bytes	block number where inode lives
ilf_blkoff	uint	4 bytes	offset in disk block of inode
ilf_field_mask	uint	4 bytes	logged field mask
ilf_data	uint[variable]	n*4 bytes	logged inode data

The inode number field gives the unique identifier of the inode being logged mainly for sanity checking. The block number and block offset fields are used to determine where on disk the logged data needs to reside. The field mask is a bitmask indicating which fields in the inode have been logged, and the data field contains the data from the in core inode image which is logged. The in core inode image is used rather than an on disk image in order to push the work of translation onto the recovery process. This is simply an optimization for the common case, which is that we are logging data that is never looked at by the recovery code.

4.4.1.2 unit xfs_inode_item_size(xfs_log_item_t *inode_item)

Simply return the size of the structure described above.

4.4.1.3 void xfs_inode_item_pin(xfs_log_item_t *inode_item)

This will just call xfs_inode_pin() with the inode associated with the given inode log item.

4.4.1.4 void xfs_inode_item_unpin(xfs_log_item_t *inode_item)

This will just call xfs_inode_unpin() with the inode associated with the given inode log item.

4.4.1.5 void xfs_inode_item_trylock(xfs_log_item_t *inode_item)

This will perform a cpsema() on the semaphore of the inode associated with this inode log item. If it can grab the inode, it will return 1. If it cannot get the inode semaphore, then it will return 0.

4.4.1.6 void xfs_inode_item_unlock(xfs_log_item_t *inode_item)

This will simply release the semaphore of the inode associated with inode log item.

4.4.1.7 xfs_lsn_t xfs_inode_item_committed(xfs_log_item_t *inode_item, xfs_lsn_t lsn)

This routine will always return the lsn value passed in to it. This is because the inode always relogs all dirty data in the inode, so the most recent log version is the only one needed.

4.4.1.8 void xfs_inode_item_flush(xfs_log_item_t *inode_item)

This routine is called to flush the inode associated with the given inode log item out to its on disk image. The inode must already be locked when calling this routine. This will get the buffer for the disk block containing the inode, copy the latest image of the inode into it, and write out the disk block asynchronously. The inode log item will attach itself to the buffer via the fsdata pointer of the buffer, so buffers containing inode blocks cannot be logged since that would use the same pointer. There is no need to log the buffers, however, because all they contain is inodes which are logged independently. The iodone routine of the buffer will be set to xfs_inode_item_done() which will remove the inode log item from the AIL and release the buffer and inode locks.

This will require holding the inode lock across the disk write. We could go with a more complicated scheme which did not keep the inode locked, but the only inodes being flushed will be those which have not been used recently. Inodes which are being modified all the time will be moving forward in the log all the time and will not need to be flushed. For this reason I am going to go with the simple but less concurrent scheme for now. If this turns out to be a problem we will revisit it.

4.5 Block Zero Operation and Don’t Block Zero Operation Log Items

This section describes the use of operation log items to perform block zeroing of newly allocated blocks. It is meant to provide an example of how operation log items can be used to implement transactions in the proposed framework.

Block zeroing is used to prevent the user from finding trash in a file when space is allocated to a file but the system crashes before the allocated space can be initialized. In xFS blocks will be allocated immediately prior to the writing of those blocks, and this section describes a means by

which we can ensure that the blocks are zeroed in the event of a crash without actually zeroing them before allocating them. This will be a big performance win.

4.5.1 Block Zero Operation Log Item

The transaction which allocates additional blocks for a file will log a block zero (BZ) operation record along with the data for allocating the new blocks for the file. This record will be recognized by the recovery code to indicate that the blocks it specifies should be filled with zeros if a corresponding don't block zero (DBZ) operation record is not found further down in the log. The corresponding DBZ record will be recognized by the fact that it contains the xfs_trans_id of the BZ record it is paired with. The xfs_bz_log_item structure given to xfs_trans_log_op() will consist of:

field name	field type	comment
bzli_item	xfs_log_item_t	common log item
bzli_ex_count	uint	number of extents to zero
bzli_extents	xfs_extent_t[variable]	array of extents to zero

The extent count and extent array fields will be used to record all of the extents which need to be zeroed in the case of a crash. The block zero operation log item operations are described below.

4.5.1.1 void xfs_bz_item_format(xfs_log_item_t *bz_item, caddr_t format)

The image of the BZ operation log item will be laid out in the following xfs_bz_log_format structure:

field name	field type	field size
bzlf_type	uint	4 bytes
bzlf_size	uint	4bytes
bzlf_ex_count	uint	4 bytes
bzlf_extents	xfs_extent_t[]	bzlf_ex_count * sizeof(xfs_extent_t)

This fields here correspond directly to those in the xfs_bz_log_item structure described above.

4.5.1.2 uint xfs_bz_item_size(xfs_log_item_t *bz_item)

This will return the size of the structure to be written by xfs_bzop_item_format().

4.5.1.3 xfs_lsn_t xfs_bz_item_committed(xfs_log_item_t *bz_item, xfs_lsn_t lsn)

This routine will always return the lsn value passed in to it. The block zero item will only be logged once, so the value passed in is the only location of the item in the log.

All of the other operations of the xfs_bz_log_item structure will simply return without doing anything. The only one which might have something to do is the flush routine, but given that the data write will start immediately and the DBZ item will be logged as soon as the data write completes there does not seem to be much to do even from the flush routine.

4.5.2 Don’t Block Zero Log Item

The DBZ item will be logged upon the completion of the data write to the newly allocated blocks. Once this item makes it to the on disk log, the recovery manager will not zero the allocated blocks in the event of a crash. the xfs_dbz_log_item structure is defined as:

field name	field type	comment
dbzli_item	xfs_log_item_t	common log item
dbzli_bz_trans_id	xfs_trans_id	trans id of BZ transaction
dbzli_extent	xfs_extent_t	extent not to zero
dbzli_bz_item	xfs_bz_log_item_t *	pointer to BZ log item
dbzli_trans	xfs_trans_t *	DBZ preallocated transaction
dbzli_buf_item	xfs_log_item_t *	log item attached to buffer before the DBZ item was added
dbzli_buf_iodone	void(*)(struct buf *)	iodone function attached to the buffer before the DBZ item was added

The transaction id field identifies the transaction id of the block zero transaction. This will be placed in the log to indicate which BZ transaction is cancelled by the DBZ transaction. The extent field will describe the exact extent not to zero in the so that the case in which a single BZ operation describes multiple extents each can be handled cleanly. The block zero item pointer tracks the log item for the block zero operation. This pointer will be used to remove the block zero operation from the AIL once the DBZ operation is in the on disk log. The transaction pointer field points to a transaction structure to be used to log the DBZ operation. Finally, the log item pointer will be used to hold a pointer to any log item which was attached to the buffer before the DBZ item is added to the buffer and the iodone function pointer field will any corresponding iodone function. This will allow us to attach a DBZ item to a buffer which already has a buffer log item attached to it for some reason. The DBZ item operations are described below:

4.5.2.1 void xfs_dbz_item_format(xfs_log_item_t *dbz_item, caddr_t buffer)

The image of the DBZ operation log item will be laid out in the following xfs_dbz_log_format structure:

field name	field type	field size
dbzlf_type	uint	4 bytes
dbzlf_size	uint	4 bytes
dbzlf_trans_id	xfs_trans_id_t	8 bytes
dbzlf_extent	xfs_extent_t	9 bytes

The transaction id field identifies the BZ operation with which this DBZ operation is paired, and the extent field identifies the extent in the BZ operation which should no longer be zeroed.

4.5.2.2 void xfs_dbz_item_size(xfs_log_item_t *dbz_item)

Return the size of the structure described above in xfs_dbz_item_format.

4.5.2.3 xfs_lsn_t xfs_dbz_item_committed(xfs_log_item_t *dbz_item, xfs_lsn_t lsn)

This routine will always free the DBZ log item and return -1. The -1 return value will prevent the transaction code from making further references to the DBZ log item.

The rest of the log item operations for the xfs_dbz_log_item all just return without doing anything.

4.5.3 The BZ/DBZ Transaction

This section describes how the BZ and DBZ log items will be used to implement transactional block zeroing. The basic idea is that the BZ item must remain an active item in the log until all of the data for the newly allocated blocks has been written. This ensures that the blocks will be zeroed if the data does not make it to disk. Each data write will have a “reference” to the BZ item, and when each completes it can remove its reference. When the last reference is removed, meaning that all of the data has been written, the BZ item can be removed from the AIL and can thus be over-written in the log because it is no longer needed. At the completion of each data write, in addition to removing a reference to the BZ item, a DBZ item will be logged so that if the BZ item is not overwritten in the log the data just written will not be replaced by zeros by the recovery code. The steps involved in this rather extended transaction are outlined below:

First, perform a space allocation transaction which includes a block zero operation for all the allocated blocks and commit the transaction.

Then for each individual extent which was allocated: allocate a DBZ log item and a transaction with which to log it, allocate a buf_t to perform the data write if necessary, attach the transaction structure to the DBZ item and attach the DBZ item to the buffer, modify the iodone routine of the buffer to point to xfs_dbz_iodone(), and write the data asynchronously.

When the data write completes, xfs_dbz_iodone() will be called by the interrupt handler. Using the pointer to the BZ item contained in the DBZ item, the routine should decrement the number of extent pointers in the BZ item by 1. If this brings the count to 0, then the BZ item should be removed from the AIL and freed. Then the DBZ item should be asynchronously (since we can’t sleep from the interrupt handler) logged and committed using the transaction structure allocated earlier. Finally, if there was another log item attached to the buffer when the DBZ item was attached to it, the other log item should be reattached to the buffer, its iodone routine should be restored, and iodone() should be called on the buffer (recursively but that’s OK).

When the log write completes, xfs_dbz_item_committed() will be called. This will just free the DBZ item.

5.0 Transaction Layer - Log Manager Interfaces

TBD.

6.0 Transaction Layer - Buffer Cache Interactions

See the document “xFS Buffer Cache” for a description of buffer cache interfaces to be used by xFS and their interaction with the xFS transaction mechanism.