

The Linux Journalling API

Roger Gammans

`rgammans@computer-surgery.co.uk`

Stephen Tweedie

`sct@redhat.com`

The Linux Journalling API

by Roger Gammans

by Stephen Tweedie

Copyright © 2002 Roger Gammans

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

Table of Contents

1. Overview	1
1.1. Details	1
1.2. Summary	3
2. Data Types	5
2.1. Structures	5
typedef handle_t.....	5
typedef journal_t.....	5
struct handle_s	6
struct journal_s.....	7
3. Functions.....	13
3.1. Journal Level.....	13
journal_init_dev	13
journal_init_inode.....	14
journal_create.....	14
journal_update_superblock.....	15
journal_load	16
journal_destroy	17
journal_check_used_features.....	18
journal_check_available_features.....	18
journal_set_features	19
journal_update_format.....	20
journal_flush	21
journal_wipe	22
journal_abort.....	23
journal_errno.....	24
journal_clear_err.....	25
journal_ack_err	26
journal_recover	27
3.2. Transasction Level	27
journal_start	28
journal_extend.....	28
journal_restart	29
journal_lock_updates	30
journal_unlock_updates	31
journal_get_write_access.....	32
journal_get_create_access.....	33
journal_get_undo_access	34
journal_dirty_data.....	36
journal_dirty_metadata	37
journal_forget.....	38

journal_stop.....	39
journal_try_to_free_buffers	39
journal_invalidatepage	41
4. See also	43

Chapter 1. Overview

1.1. Details

The journalling layer is easy to use. You need to first of all create a `journal_t` data structure. There are two calls to do this dependent on how you decide to allocate the physical media on which the journal resides. The `journal_init_inode()` call is for journals stored in filesystem inodes, or the `journal_init_dev()` call can be use for journal stored on a raw device (in a continuous range of blocks). A `journal_t` is a typedef for a struct pointer, so when you are finally finished make sure you call `journal_destroy()` on it to free up any used kernel memory.

Once you have got your `journal_t` object you need to 'mount' or load the journal file, unless of course you haven't initialised it yet - in which case you need to call `journal_create()`.

Most of the time however your journal file will already have been created, but before you load it you must call `journal_wipe()` to empty the journal file. Hang on, you say , what if the filesystem wasn't cleanly umount()'d . Well, it is the job of the client file system to detect this and skip the call to `journal_wipe()`.

In either case the next call should be to `journal_load()` which prepares the journal file for use. Note that `journal_wipe(..,0)` calls `journal_skip_recovery()` for you if it detects any outstanding transactions in the journal and similarly `journal_load()` will call `journal_recover()` if necessary. I would advise reading `fs/ext3/super.c` for examples on this stage. [RGG: Why is the `journal_wipe()` call necessary - doesn't this needlessly complicate the API. Or isn't a good idea for the journal layer to hide dirty mounts from the client fs]

Now you can go ahead and start modifying the underlying filesystem. Almost.

You still need to actually journal your filesystem changes, this is done by wrapping them into transactions. Additionally you also need to wrap the modification of each of the the buffers with calls to the journal layer, so it knows what the modifications you are actually making are. To do this use `journal_start()` which returns a transaction handle.

`journal_start()` and its counterpart `journal_stop()`, which indicates the end of a transaction are nestable calls, so you can reenter a transaction if necessary, but remember you must call `journal_stop()` the same number of times as `journal_start()` before the transaction is completed (or more accurately leaves the the update phase). Ext3/VFS makes use of this feature to simplify quota support.

Inside each transaction you need to wrap the modifications to the individual buffers (blocks). Before you start to modify a buffer you need to call

`journal_get_{create,write,undo}_access()` as appropriate, this allows the journalling layer to copy the unmodified data if it needs to. After all the buffer may be part of a previously uncommitted transaction. At this point you are at last ready to modify a buffer, and once you are have done so you need to call `journal_dirty_{meta,}data()`. Or if you've asked for access to a buffer you now know is now longer required to be pushed back on the device you can call `journal_forget()` in much the same way as you might have used `bforget()` in the past.

A `journal_flush()` may be called at any time to commit and checkpoint all your transactions.

Then at umount time , in your `put_super()` (2.4) or `write_super()` (2.5) you can then call `journal_destroy()` to clean up your in-core journal object.

Unfortunately there a couple of ways the journal layer can cause a deadlock. The first thing to note is that each task can only have a single outstanding transaction at any one time, remember nothing commits until the outermost `journal_stop()`. This means you must complete the transaction at the end of each file/inode/address etc. operation you perform, so that the journalling system isn't re-entered on another journal. Since transactions can't be nested/batched across differing journals, and another filesystem other than yours (say ext3) may be modified in a later syscall.

The second case to bear in mind is that `journal_start()` can block if there isn't enough space in the journal for your transaction (based on the passed `nblocks` param) - when it blocks it merely(!) needs to wait for transactions to complete and be committed from other tasks, so essentially we are waiting for `journal_stop()`. So to avoid deadlocks you must treat `journal_start/stop()` as if they were semaphores and include them in your semaphore ordering rules to prevent deadlocks. Note that `journal_extend()` has similar blocking behaviour to `journal_start()` so you can deadlock here just as easily as on `journal_start()`.

Try to reserve the right number of blocks the first time. ;-). This will be the maximum number of blocks you are going to touch in this transaction. I advise having a look at at least `ext3_jbd.h` to see the basis on which ext3 uses to make these decisions.

Another wriggle to watch out for is your on-disk block allocation strategy. why? Because, if you undo a delete, you need to ensure you haven't reused any of the freed blocks in a later transaction. One simple way of doing this is make sure any blocks you allocate only have checkpointed transactions listed against them. Ext3 does this in `ext3_test_allocatable()`.

Lock is also providing through `journal_{un,}lock_updates()`, ext3 uses this when it wants a window with a clean and stable fs for a moment. eg.

```
journal_lock_updates() //stop new stuff happening..  
journal_flush()        // checkpoint everything..  
..do stuff on stable fs
```

```
journal_unlock_updates() // carry on with filesystem use.
```

The opportunities for abuse and DOS attacks with this should be obvious, if you allow unprivileged userspace to trigger codepaths containing these calls.

A new feature of jbd since 2.5.25 is commit callbacks with the new `journal_callback_set()` function you can now ask the journalling layer to call you back when the transaction is finally committed to disk, so that you can do some of your own management. The key to this is the `journal_callback` struct, this maintains the internal callback information but you can extend it like this:-

```
struct myfs_callback_s {
    //Data structure element required by jbd..
    struct journal_callback for_jbd;
    // Stuff for myfs allocated together.
    myfs_inode*      i_committed;
}
```

this would be useful if you needed to know when data was committed to a particular inode.

1.2. Summary

Using the journal is a matter of wrapping the different context changes, being each mount, each modification (transaction) and each changed buffer to tell the journalling layer about them.

Here is a some pseudo code to give you an idea of how it works, as an example.

```
journal_t* my_jnrl = journal_create();
journal_init_{dev,inode}(jnrl,...)
if (clean) journal_wipe();
journal_load();

foreach(transaction) { /*transactions must be
                        completed before
                        a syscall returns to
                        userspace*/

    handle_t * xct=journal_start(my_jnrl);
    foreach(bh) {
        journal_get_{create,write,undo}_access(xact,bh);
        if ( myfs_modify(bh) ) { /* returns true
                                if makes changes */
            journal_dirty_{meta,}data(xact,bh);
```

Chapter 1. Overview

```
        } else {  
            journal_forget(bh);  
        }  
    }  
    journal_stop(xct);  
}  
journal_destroy(my_jrnl);
```


Chapter 2. Data Types

The journalling layer uses typedefs to 'hide' the concrete definitions of the structures used. As a client of the JBD layer you can just rely on the using the pointer as a magic cookie of some sort. Obviously the hiding is not enforced as this is 'C'.

2.1. Structures

typedef handle_t

Name

`typedef handle_t` — The `handle_t` type represents a single atomic update being performed by some process.

Synopsis

```
typedef handle_t;
```

Description

All filesystem modifications made by the process go through this handle. Recursive operations (such as quota operations) are gathered into a single update.

The buffer credits field is used to account for journaled buffers being modified by the running process. To ensure that there is enough log space for all outstanding operations, we need to limit the number of outstanding buffers possible at any time. When the operation completes, any buffer credits not used are credited back to the transaction, so that at all times we know how many buffers the outstanding updates on a transaction might possibly touch.

This is an opaque datatype.

typedef journal_t

Name

`typedef journal_t` — The `journal_t` maintains all of the journaling state information for a single filesystem.

Synopsis

```
typedef journal_t;
```

Description

`journal_t` is linked to from the fs superblock structure.

We use the `journal_t` to keep track of all outstanding transaction activity on the filesystem, and to manage the state of the log writing process.

This is an opaque datatype.

struct handle_s

Name

`struct handle_s` — The `handle_s` type is the concrete type associated with

Synopsis

```
struct handle_s {  
    transaction_t * h_transaction;  
    int h_buffer_credits;  
    int h_ref;  
    int h_err;  
    unsigned int h_sync:1;  
    unsigned int h_jdata:1;  
};
```

```
    unsigned int h_aborted:1;
};
```

Members

`h_transaction`

Which compound transaction is this update a part of?

`h_buffer_credits`

Number of remaining buffers we are allowed to dirty.

`h_ref`

Reference count on this handle

`h_err`

Field for caller's use to track errors through large fs operations

`h_sync`

flag for sync-on-close

`h_jdata`

flag to force data journaling

`h_aborted`

flag indicating fatal error on handle

Description

`handle_t`.

struct journal_s

Name

struct journal_s — The journal_s type is the concrete type associated with

Synopsis

```
struct journal_s {
    unsigned long j_flags;
    int j_errno;
    struct buffer_head * j_sb_buffer;
    journal_superblock_t * j_superblock;
    int j_format_version;
    int j_barrier_count;
    struct semaphore j_barrier;
    transaction_t * j_running_transaction;
    transaction_t * j_committing_transaction;
    transaction_t * j_checkpoint_transactions;
    wait_queue_head_t j_wait_transaction_locked;
    wait_queue_head_t j_wait_logspace;
    wait_queue_head_t j_wait_done_commit;
    wait_queue_head_t j_wait_checkpoint;
    wait_queue_head_t j_wait_commit;
    wait_queue_head_t j_wait_updates;
    struct semaphore j_checkpoint_sem;
    unsigned long j_head;
    unsigned long j_tail;
    unsigned long j_free;
    unsigned long j_first;
    unsigned long j_last;
    struct block_device * j_dev;
    int j_blocksize;
    unsigned int j_blk_offset;
    struct block_device * j_fs_dev;
    unsigned int j_maxlen;
    struct inode * j_inode;
    tid_t j_tail_sequence;
    tid_t j_transaction_sequence;
    tid_t j_commit_sequence;
    tid_t j_commit_request;
    __u8 j_uuid[16];
    struct task_struct * j_task;
    int j_max_transaction_buffers;
```

```
unsigned long j_commit_interval;  
struct timer_list * j_commit_timer;  
struct jbd_revoke_table_s * j_revoke;  
};
```

Members

`j_flags`

General journaling state flags

`j_errno`

Is there an outstanding uncleared error on the journal (from a prior abort)?

`j_sb_buffer`

First part of superblock buffer

`j_superblock`

Second part of superblock buffer

`j_format_version`

Version of the superblock format

`j_barrier_count`

Number of processes waiting to create a barrier lock

`j_barrier`

The barrier lock itself

`j_running_transaction`

The current running transaction..

`j_committing_transaction`

the transaction we are pushing to disk

`j_checkpoint_transactions`

a linked circular list of all transactions waiting for checkpointing

Chapter 2. Data Types

`j_wait_transaction_locked`

Wait queue for waiting for a locked transaction to start committing, or for a barrier lock to be released

`j_wait_logspace`

Wait queue for waiting for checkpointing to complete

`j_wait_done_commit`

Wait queue for waiting for commit to complete

`j_wait_checkpoint`

Wait queue to trigger checkpointing

`j_wait_commit`

Wait queue to trigger commit

`j_wait_updates`

Wait queue to wait for updates to complete

`j_checkpoint_sem`

Semaphore for locking against concurrent checkpoints

`j_head`

Journal head - identifies the first unused block in the journal

`j_tail`

Journal tail - identifies the oldest still-used block in the journal.

`j_free`

Journal free - how many free blocks are there in the journal?

`j_first`

The block number of the first usable block

`j_last`

The block number one beyond the last usable block

`j_dev`

Device where we store the journal

`j_blocksize`

blocksize for the location where we store the journal.

`j_blk_offset`

starting block offset for into the device where we store the journal

`j_fs_dev`

Device which holds the client fs. For internal journal this will be equal to `j_dev`

`j_maxlen`

Total maximum capacity of the journal region on disk.

`j_inode`

Optional inode where we store the journal. If present, all journal block numbers are mapped into this inode via `bmap`.

`j_tail_sequence`

Sequence number of the oldest transaction in the log

`j_transaction_sequence`

Sequence number of the next transaction to grant

`j_commit_sequence`

Sequence number of the most recently committed transaction

`j_commit_request`

Sequence number of the most recent transaction wanting commit

`j_uuid[16]`

Uuid of client object.

`j_task`

Pointer to the current commit thread for this journal

`j_max_transaction_buffers`

Maximum number of metadata buffers to allow in a single compound commit transaction

`j_commit_interval`

What is the maximum transaction lifetime before we begin a commit?

Chapter 2. Data Types

j_commit_timer

The timer used to wakeup the commit thread

j_revoke

The revoke table - maintains the list of revoked blocks in the current transaction.

Description

journal_t.

Chapter 3. Functions

The functions here are split into two groups those that affect a journal as a whole, and those which are used to manage transactions

3.1. Journal Level

journal_init_dev

Name

`journal_init_dev` — creates an initialises a journal structure

Synopsis

```
journal_t * journal_init_dev (struct block_device * bdev,  
struct block_device * fs_dev, int start, int len, int  
blocksize);
```

Arguments

bdev

Block device on which to create the journal

fs_dev

Device which hold journalled filesystem for this journal.

start

Block nr Start of journal.

len

Lenght of the journal in blocks.

blocksize

blocksize of journalling device

Description

`journal_init_dev` creates a journal which maps a fixed contiguous range of blocks on an arbitrary block device.

journal_init_inode

Name

`journal_init_inode` — creates a journal which maps to a inode.

Synopsis

```
journal_t * journal_init_inode (struct inode * inode);
```

Arguments

inode

An inode to create the journal in

Description

`journal_init_inode` creates a journal which maps an on-disk inode as the journal. The inode must exist already, must support `bmap` and must have all data blocks preallocated.

journal_create

Name

`journal_create` — Initialise the new journal file

Synopsis

```
int journal_create (journal_t * journal);
```

Arguments

journal

Journal to create. This structure must have been initialised

Description

Given a `journal_t` structure which tells us which disk blocks we can use, create a new journal superblock and initialise all of the journal fields from scratch.

journal_update_superblock

Name

`journal_update_superblock` — Update journal sb on disk.

Synopsis

```
void journal_update_superblock (journal_t * journal, int  
wait);
```

Arguments

journal

The journal to update.

wait

Set to '0' if you don't want to wait for IO completion.

Description

Update a journal's dynamic superblock fields and write it to disk, optionally waiting for the IO to complete.

journal_load

Name

`journal_load` — Read journal from disk.

Synopsis

```
int journal_load (journal_t * journal);
```

Arguments

journal

Journal to act on.

Description

Given a `journal_t` structure which tells us which disk blocks contain a journal, read the journal from disk to initialise the in-memory structures.

journal_destroy

Name

`journal_destroy` — Release a `journal_t` structure.

Synopsis

```
void journal_destroy (journal_t * journal);
```

Arguments

journal

Journal to act on.

Description

Release a `journal_t` structure once it is no longer in use by the journaled object.

journal_check_used_features

Name

`journal_check_used_features` — Check if features specified are used.

Synopsis

```
int journal_check_used_features (journal_t * journal, unsigned
long compat, unsigned long ro, unsigned long incompat);
```

Arguments

journal

-- undescribed --

compat

-- undescribed --

ro

-- undescribed --

incompat

-- undescribed --

Description

Check whether the journal uses all of a given set of features. Return true (non-zero) if it does.

journal_check_available_features

Name

`journal_check_available_features` — Check feature set in journalling layer

Synopsis

```
int journal_check_available_features (journal_t * journal,  
unsigned long compat, unsigned long ro, unsigned long  
incompat);
```

Arguments

journal

-- undescrbed --

compat

-- undescrbed --

ro

-- undescrbed --

incompat

-- undescrbed --

Description

Check whether the journaling code supports the use of all of a given set of features on this journal. Return true

journal_set_features

Name

`journal_set_features` — Mark a given journal feature in the superblock

Synopsis

```
int journal_set_features (journal_t * journal, unsigned long
compat, unsigned long ro, unsigned long incompat);
```

Arguments

journal

-- undescribed --

compat

-- undescribed --

ro

-- undescribed --

incompat

-- undescribed --

Description

Mark a given journal feature as present on the superblock. Returns true if the requested features could be set.

journal_update_format

Name

`journal_update_format` — Update on-disk journal structure.

Synopsis

```
int journal_update_format (journal_t * journal);
```

Arguments

journal

-- undescribed --

Description

Given an initialised but unloaded journal struct, poke about in the on-disk structure to update it to the most recent supported version.

journal_flush

Name

`journal_flush` — Flush journal

Synopsis

```
int journal_flush (journal_t * journal);
```

Arguments

journal

Journal to act on.

Description

Flush all data for a given journal to disk and empty the journal. Filesystems can use this when remounting readonly to ensure that recovery does not need to happen on remount.

journal_wipe

Name

`journal_wipe` — Wipe journal contents

Synopsis

```
int journal_wipe (journal_t * journal, int write);
```

Arguments

journal

Journal to act on.

write

flag (see below)

Description

Wipe out all of the contents of a journal, safely. This will produce a warning if the journal contains any valid recovery information. Must be called between `journal_init_*`() and `journal_load`.

If 'write' is non-zero, then we wipe out the journal on disk; otherwise we merely suppress recovery.

journal_abort

Name

`journal_abort` — Shutdown the journal immediately.

Synopsis

```
void journal_abort (journal_t * journal, int errno);
```

Arguments

journal

the journal to shutdown.

errno

an error number to record in the journal indicating the reason for the shutdown.

Description

Perform a complete, immediate shutdown of the ENTIRE journal (not of a single transaction). This operation cannot be undone without closing and reopening the journal.

The `journal_abort` function is intended to support higher level error recovery mechanisms such as the `ext2/ext3` remount-readonly error mode.

Journal abort has very specific semantics. Any existing dirty, unjournalized buffers in the main filesystem will still be written to disk by `bdfuse`, but the journaling mechanism will be suspended immediately and no further transaction commits will be honoured.

Any dirty, journaled buffers will be written back to disk without hitting the journal. Atomicity cannot be guaranteed on an aborted filesystem, but we `_do_` attempt to leave as much data as possible behind for `fsck` to use for cleanup.

Any attempt to get a new transaction handle on a journal which is in `ABORT` state will just result in an `-EROFS` error return. A `journal_stop` on an existing handle will return `-EIO` if we have entered abort state during the update.

Recursive transactions are not disturbed by journal abort until the final `journal_stop`, which will receive the `-EIO` error.

Finally, the `journal_abort` call allows the caller to supply an `errno` which will be recorded (if possible) in the journal superblock. This allows a client to record failure conditions in the middle of a transaction without having to complete the transaction to record the failure to disk. `ext3_error`, for example, now uses this functionality.

Errors which originate from within the journaling layer will NOT supply an `errno`; a null `errno` implies that absolutely no further writes are done to the journal (unless there are any already in progress).

journal_errno

Name

`journal_errno` — returns the journal's error state.

Synopsis

```
int journal_errno (journal_t * journal);
```

Arguments

journal

journal to examine.

Description

This is the errno numbet set with `journal_abort`, the last time the journal was mounted - if the journal was stopped without calling abort this will be 0.

If the journal has been aborted on this mount time -EROFS will be returned.

journal_clear_err

Name

`journal_clear_err` — clears the journal's error state

Synopsis

```
int journal_clear_err (journal_t * journal);
```

Arguments

journal
-- undescribed --

Description

An error must be cleared or Acked to take a FS out of readonly mode.

journal_ack_err

Name

`journal_ack_err` — Ack journal err.

Synopsis

```
void journal_ack_err (journal_t * journal);
```

Arguments

journal
-- undescribed --

Description

An error must be cleared or Acked to take a FS out of readonly mode.

journal_recover

Name

`journal_recover` — recovers a on-disk journal

Synopsis

```
int journal_recover (journal_t * journal);
```

Arguments

journal

the journal to recover

Description

The primary function for recovering the log contents when mounting a journaled device.

Recovery is done in three passes. In the first pass, we look for the end of the log. In the second, we assemble the list of revoke blocks. In the third and final pass, we replay any un-revoked blocks in the log.

3.2. Transasction Level

journal_start

Name

`journal_start` — Obtain a new handle.

Synopsis

```
handle_t * journal_start (journal_t * journal, int nblocks);
```

Arguments

journal

Journal to start transaction on.

nblocks

number of block buffer we might modify

Description

We make sure that the transaction can guarantee at least `nblocks` of modified buffers in the log. We block until the log can guarantee that much space.

This function is visible to journal users (like `ext3fs`), so is not called with the journal already locked.

Return a pointer to a newly allocated handle, or `NULL` on failure

journal_extend

Name

`journal_extend` — extend buffer credits.

Synopsis

```
int journal_extend (handle_t * handle, int nblocks);
```

Arguments

handle

handle to 'extend'

nblocks

nr blocks to try to extend by.

Description

Some transactions, such as large extends and truncates, can be done atomically all at once or in several stages. The operation requests a credit for a number of buffer modifications in advance, but can extend its credit if it needs more.

`journal_extend` tries to give the running handle more buffer credits. It does not guarantee that allocation - this is a best-effort only. The calling process **MUST** be able to deal cleanly with a failure to extend here.

Return 0 on success, non-zero on failure.

return code < 0 implies an error return code > 0 implies normal transaction-full status.

journal_restart

Name

`journal_restart` — restart a handle .

Synopsis

```
int journal_restart (handle_t * handle, int nblocks);
```

Arguments

handle

handle to restart

nblocks

nr credits requested

Description

Restart a handle for a multi-transaction filesystem operation.

If the `journal_extend` call above fails to grant new buffer credits to a running handle, a call to `journal_restart` will commit the handle's transaction so far and reattach the handle to a new transaction capable of guaranteeing the requested number of credits.

journal_lock_updates

Name

`journal_lock_updates` — establish a transaction barrier.

Synopsis

```
void journal_lock_updates (journal_t * journal);
```

Arguments

journal

Journal to establish a barrier on.

Description

This locks out any further updates from being started, and blocks until all existing updates have completed, returning only once the journal is in a quiescent state with no updates running.

The journal lock should not be held on entry.

journal_unlock_updates

Name

`journal_unlock_updates` — release barrier

Synopsis

```
void journal_unlock_updates (journal_t * journal);
```

Arguments

journal

Journal to release the barrier on.

Description

Release a transaction barrier obtained with `journal_lock_updates`.

Should be called without the journal lock held.

journal_get_write_access

Name

`journal_get_write_access` — notify intent to modify a buffer for metadata (not data) update.

Synopsis

```
int journal_get_write_access (handle_t * handle, struct  
buffer_head * bh, int * credits);
```

Arguments

handle

transaction to add buffer modifications to

bh

bh to be used for metadata writes

credits

-- undescribed --

Description

Returns an error code or 0 on success.

In full data journalling mode the buffer may be of type BJ_AsyncData, because we're writing a buffer which is also part of a shared mapping.

journal_get_create_access

Name

journal_get_create_access — notify intent to use newly created bh

Synopsis

```
int journal_get_create_access (handle_t * handle, struct
buffer_head * bh);
```

Arguments

handle

transaction to new buffer to

bh

new buffer.

Description

Call this if you create a new bh.

journal_get_undo_access

Name

`journal_get_undo_access` — Notify intent to modify metadata with

Synopsis

```
int journal_get_undo_access (handle_t * handle, struct  
buffer_head * bh, int * credits);
```

Arguments

handle

transaction

bh

buffer to undo

credits

store the number of taken credits here (if not NULL)

Description

Sometimes there is a need to distinguish between metadata which has been committed to disk and that which has not. The ext3fs code uses this for freeing and allocating space, we have to make sure that we do not reuse freed space until the deallocation has been committed, since if we overwrote that space we would make the delete un-rewindable in case of a crash.

To deal with that, `journal_get_undo_access` requests write access to a buffer for parts of non-rewindable operations such as delete operations on the bitmaps. The journaling code must keep a copy of the buffer's contents prior to the `undo_access` call until such time as we know that the buffer has definitely been committed to disk.

We never need to know which transaction the committed data is part of, buffers touched here are guaranteed to be dirtied later and so will be committed to a new transaction in due course, at which point we can discard the old committed data pointer.

Returns error number or 0 on success.

Description

Sometimes there is a need to distinguish between metadata which has been committed to disk and that which has not. The ext3fs code uses this for freeing and allocating space, we have to make sure that we do not reuse freed space until the deallocation has been committed, since if we overwrote that space we would make the delete un-rewindable in case of a crash.

To deal with that, `journal_get_undo_access` requests write access to a buffer for parts of non-rewindable operations such as delete operations on the bitmaps. The journaling code must keep a copy of the buffer's contents prior to the `undo_access` call until such time as we know that the buffer has definitely been committed to disk.

We never need to know which transaction the committed data is part of, buffers touched here are guaranteed to be dirtied later and so will be committed to a new transaction in due course, at which point we can discard the old committed data pointer.

Returns error number or 0 on success.

journal_dirty_data

Name

`journal_dirty_data` — mark a buffer as containing dirty data which

Synopsis

```
int journal_dirty_data (handle_t * handle, struct buffer_head  
* bh);
```

Arguments

handle

transaction

bh

bufferhead to mark

Description

The buffer is placed on the transaction's data list and is marked as belonging to the transaction.

Returns error number or 0 on success.

`journal_dirty_data` can be called via `page_launder->ext3_writepage` by `kswapd`.

Description

The buffer is placed on the transaction's data list and is marked as belonging to the transaction.

Returns error number or 0 on success.

`journal_dirty_data` can be called via `page_laundry->ext3_writepage` by `kswapd`.

journal_dirty_metadata

Name

`journal_dirty_metadata` — mark a buffer as containing dirty metadata

Synopsis

```
int journal_dirty_metadata (handle_t * handle, struct
buffer_head * bh);
```

Arguments

handle

transaction to add buffer to.

bh

buffer to mark

Description

mark dirty metadata which needs to be journaled as part of the current transaction.

The buffer is placed on the transaction's metadata list and is marked as belonging to the transaction.

Returns error number or 0 on success.

Special care needs to be taken if the buffer already belongs to the current committing transaction (in which case we should have frozen data present for that commit). In that case, we don't relink the

buffer

that only gets done when the old transaction finally completes its commit.

journal_forget

Name

`journal_forget` — `bforget` for potentially-journaled buffers.

Synopsis

```
int journal_forget (handle_t * handle, struct buffer_head *  
bh);
```

Arguments

handle

transaction handle

bh

bh to 'forget'

Description

We can only do the `bforget` if there are no commits pending against the buffer. If the buffer is dirty in the current running transaction we can safely unlink it.

`bh` may not be a journalled buffer at all - it may be a non-JBD buffer which came off the hashtable. Check for this.

Decrements `bh->b_count` by one.

Allow this call even if the handle has aborted --- it may be part of the caller's cleanup after an abort.

journal_stop

Name

`journal_stop` — complete a transaction

Synopsis

```
int journal_stop (handle_t * handle);
```

Arguments

handle

transaction to complete.

Description

All done for a particular handle.

There is not much action needed here. We just return any remaining buffer credits to the transaction and remove the handle. The only complication is that we need to start a commit operation if the filesystem is marked for synchronous update.

`journal_stop` itself will not usually return an error, but it may do so in unusual circumstances. In particular, expect it to return `-EIO` if a `journal_abort` has been executed since the transaction began.

journal_try_to_free_buffers

Name

`journal_try_to_free_buffers` — try to free page buffers.

Synopsis

```
int journal_try_to_free_buffers (journal_t * journal, struct  
page * page, int unused_gfp_mask);
```

Arguments

journal

journal for operation

page

to try and free

unused_gfp_mask

-- undescribed --

Description

For all the buffers on this page, if they are fully written out ordered data, move them onto BUF_CLEAN so `try_to_free_buffers` can reap them.

This function returns non-zero if we wish `try_to_free_buffers` to be called. We do this if the page is releasable by `try_to_free_buffers`. We also do it if the page has locked or dirty buffers and the caller wants us to perform sync or async writeout.

This complicates JBD locking somewhat. We aren't protected by the BKL here. We wish to remove the buffer from its committing or running transaction's `->t_datalist` via `__journal_unfile_buffer`.

This may **change** the value of `transaction_t->t_datalist`, so anyone who looks at `t_datalist` needs to lock against this function.

Even worse, someone may be doing a `journal_dirty_data` on this buffer. So we need to lock against that. `journal_dirty_data` will come out of the lock with the buffer dirty, which makes it ineligible for release here.

Who else is affected by this? hmm... Really the only contender is `do_get_write_access` - it could be looking at the buffer while `journal_try_to_free_buffer` is changing its state. But that cannot happen because we never reallocate freed data as metadata while the data is part of a transaction. Yes?

journal_invalidatepage

Name

`journal_invalidatepage` —

Synopsis

```
int journal_invalidatepage (journal_t * journal, struct page *
page, unsigned long offset);
```

Arguments

journal

journal to use for flush...

page

page to flush

offset

length of page to invalidate.

Description

Reap page buffers containing data after offset in page.

Return non-zero if the page's buffers were successfully reaped.

Chapter 4. See also

[Journaling the Linux ext2fs Filesystem, LinuxExpo 98, Stephen Tweedie
(<ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/journal-design.ps.gz>)]

[Ext3 Journalling FileSystem , OLS 2000, Dr. Stephen Tweedie
(<http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html>)]

Chapter 4. See also