The Linux Journalling API

Overview

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 jbd2_journal_init_inode() call is for journals stored in filesystem inodes, or the jbd2_journal_init_dev() call can be used 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 jbd2_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. The journalling layer expects the space for the journal was already allocated and initialized properly by the userspace tools. When loading the journal you must call jbd2_journal_load() to process journal contents. If the client file system detects the journal contents does not need to be processed (or even need not have valid contents), it may call jbd2_journal_wipe() to clear the journal contents before calling jbd2_journal_load().

Note that jbd2_journal_wipe(..,0) calls jbd2_journal_skip_recovery() for you if it detects any outstanding transactions in the journal and similarly jbd2_journal_load() will call jbd2_journal_recover() if necessary. I would advise reading ext4_load_journal() in fs/ext4/super.c for examples on this stage.

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 buffers with calls to the journal layer, so it knows what the modifications you are actually making are. To do this use jbd2_journal_start() which returns a transaction handle.

jbd2_journal_start() and its counterpart jbd2_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 jbd2_journal_stop() the same number of times as jbd2_journal_start() before the transaction is completed (or more accurately leaves the update phase). Ext4/VFS makes use of this feature to simplify handling of inode dirtying, quota support, etc.

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 jbd2_journal_get_create_access() / jbd2_journal_get_write_access() / jbd2_journal_get_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 jbd2_journal_dirty_metadata(). 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 jbd2_journal_forget() in much the same way as you might have used bforget() in the past.

A jbd2_journal_flush() may be called at any time to commit and checkpoint all your transactions.

Then at umount time , in your put_super() you can then call jbd2_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 jbd2_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 ext4) may be modified in a later syscall.

The second case to bear in mind is that jbd2_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 jbd2_journal_stop(). So to avoid deadlocks you must treat jbd2_journal_start() / jbd2_journal_stop() as if they were semaphores and include them in your semaphore ordering rules to prevent deadlocks. Note that jbd2_journal_extend() has similar blocking behaviour to jbd2_journal_start() so you can deadlock here just as easily as on jbd2_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 ext4_jbd.h to see the basis on which ext4 uses to make these decisions.

Another wriggle to watch out for is your on-disk block allocation strategy. Why? Because, if you do a delete, you need to ensure you haven't reused any of the freed blocks until the transaction freeing these blocks commits. If you reused these blocks and crash happens, there is no way to restore the contents of the reallocated blocks at the end of the last fully committed transaction. One simple way of doing this is to mark blocks as free in internal in-memory block allocation structures only after the transaction freeing them commits. Ext4 uses journal commit callback for this purpose.

With journal commit callbacks you can ask the journalling layer to call a callback function when the transaction is finally committed to disk, so that you can do some of your own management. You ask the journalling layer for calling the callback by simply setting `journal->j_commit_callback` function pointer and that function is called after each transaction commit. You can also use `transaction->t_private_list` for attaching entries to a transaction that need processing when the transaction commits.

JBD2 also provides a way to block all transaction updates via jbd2_journal_lock_updates() / jbd2_journal_unlock_updates(). Ext4 uses this when it wants a window with a clean and stable fs for a moment. E.g.

```
jbd2_journal_lock_updates() //stop new stuff happening..
jbd2_journal_flush()        // checkpoint everything.
..do stuff on stable fs
jbd2_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.

Fast commits

JBD2 to also allows you to perform file-system specific delta commits known as fast commits. In order to use fast commits, you will need to set following callbacks that perform correspodning work:

journal->j_fc_cleanup_cb: Cleanup function called after every full commit and fast commit.

journal->j_fc_replay_cb: Replay function called for replay of fast commit blocks.

File system is free to perform fast commits as and when it wants as long as it gets permission from JBD2 to do so by calling the function :c:func:`jbd2_fc_begin_commit()`. Once a fast commit is done, the client file system should tell JBD2 about it by calling :c:func:`jbd2_fc_end_commit()`. If file system wants JBD2 to perform a full commit immediately after stopping the fast commit it can do so by calling :c:func:`jbd2_fc_end_commit_fallback()`. This is useful if fast commit operation fails for some reason and the only way to guarantee consistency is for JBD2 to perform the full traditional commit.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, **line 148**); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, **line 148**); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, **line 148**); *backlink*

Unknown interpreted text role "c:func".

JBD2 helper functions to manage fast commit buffers. File system can use :c:func:`jbd2_fc_get_buf()` and :c:func:`jbd2_fc_wait_bufs()` to allocate and wait on IO completion of fast commit buffers.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, **line 158**); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, **line 158**); *backlink*

Unknown interpreted text role "c:func".

Currently, only Ext4 implements fast commits. For details of its implementation of fast commits, please refer to the top level comments in fs/ext4/fast_commit.c.

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.

Data Types

The journalling layer uses typedefs to 'hide' the concrete definitions of the structures used. As a client of the JBD2 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'.

Structures

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, line 184)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/jbd2.h
   :internal:
```

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

Journal Level

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, line 196)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: fs/jbd2/journal.c
   :export:
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, line 199)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: fs/jbd2/recovery.c
   :internal:
```

Transasction Level

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]journalling.rst`, line 205)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: fs/jbd2/transaction.c
```

See also

Journaling the Linux ext2fs Filesystem, LinuxExpo 98, Stephen Tweedie

Ext3 Journalling FileSystem, OLS 2000, Dr. Stephen Tweedie