Journal (jbd2)

Introduced in ext3, the ext4 filesystem employs a journal to protect the filesystem against metadata inconsistencies in the case of a system crash. Up to 10,240,000 file system blocks (see man mke2fs(8) for more details on journal size limits) can be reserved inside the filesystem as a place to land “important” data writes on-disk as quickly as possible. Once the important data transaction is fully written to the disk and flushed from the disk write cache, a record of the data being committed is also written to the journal. At some later point in time, the journal code writes the transactions to their final locations on disk (this could involve a lot of seeking or a lot of small read-write-erases) before erasing the commit record. Should the system crash during the second slow write, the journal can be replayed all the way to the latest commit record, guaranteeing the atomicity of whatever gets written through the journal to the disk. The effect of this is to guarantee that the filesystem does not become stuck midway through a metadata update.

For performance reasons, ext4 by default only writes filesystem metadata through the journal. This means that file data blocks are /not/ guaranteed to be in any consistent state after a crash. If this default guarantee level (`data=ordered`) is not satisfactory, there is a mount option to control journal behavior. If `data=journal`, all data and metadata are written to disk through the journal. This is slower but safest. If `data=writeback`, dirty data blocks are not flushed to the disk before the metadata are written to disk through the journal.

In case of `data=ordered` mode, Ext4 also supports fast commits which help reduce commit latency significantly. The default `data=ordered` mode works by logging metadata blocks to the journal. In fast commit mode, Ext4 only stores the minimal delta needed to recreate the affected metadata in fast commit space that is shared with JBD2. Once the fast commit area fills in or if fast commit is not possible or if JBD2 commit timer goes off, Ext4 performs a traditional full commit. A full commit invalidates all the fast commits that happened before it and thus it makes the fast commit area empty for further fast commits. This feature needs to be enabled at mkfs time.

The journal inode is typically inode 8. The first 68 bytes of the journal inode are replicated in the ext4 superblock. The journal itself is normal (but hidden) file within the filesystem. The file usually consumes an entire block group, though mke2fs tries to put it in the middle of the disk.

All fields in jbd2 are written to disk in big-endian order. This is the opposite of ext4.

NOTE: Both ext4 and ocfs2 use jbd2.

The maximum size of a journal embedded in an ext4 filesystem is 2^32 blocks. jbd2 itself does not seem to care.

Layout

Generally speaking, the journal has this format:

Superblock	**descriptor_block (data_blocks or revocation_block) [more data or revocations] commmit_block**	**[more transactions...]**
	One transaction	

Notice that a transaction begins with either a descriptor and some data, or a block revocation list. A finished transaction always ends with a commit. If there is no commit record (or the checksums don't match), the transaction will be discarded during replay.

External Journal

Optionally, an ext4 filesystem can be created with an external journal device (as opposed to an internal journal, which uses a reserved inode). In this case, on the filesystem device, `s_journal_inum` should be zero and `s_journal_uuid` should be set. On the journal device there will be an ext4 super block in the usual place, with a matching UUID. The journal superblock will be in the next full block after the superblock.

1024 bytes of padding	**ext4 Superblock**	**Journal Superblock**	**descriptor_block (data_blocks or revocation_block) [more data or revocations] commmit_block**	**[more transactions...]**
			One transaction	

Block Header

Every block in the journal starts with a common 12-byte header `struct journal_header_s`:

Offset	Type	Name	Description
0x0	__be32	h_magic	jbd2 magic number, 0xC03B3998.
0x4	__be32	h_blocktype	Description of what this block contains. See the jbd2_blocktype table below.
0x8	__be32	h_sequence	The transaction ID that goes with this block.

The journal block type can be any one of:

Value	Description
1	Descriptor. This block precedes a series of data blocks that were written through the journal during a transaction.
2	Block commit record. This block signifies the completion of a transaction.
3	Journal superblock, v1.
4	Journal superblock, v2.
5	Block revocation records. This speeds up recovery by enabling the journal to skip writing blocks that were subsequently rewritten.

Super Block

The super block for the journal is much simpler as compared to ext4's. The key data kept within are size of the journal, and where to find the start of the log of transactions.

The journal superblock is recorded as `struct journal_superblock_s`, which is 1024 bytes long:

Offset	Type	Name	Description
			Static information describing the journal.
0x0	journal_header_t (12 bytes)	s_header	Common header identifying this as a superblock.
0xC	__be32	s_blocksize	Journal device block size.
0x10	__be32	s_maxlen	Total number of blocks in this journal.
0x14	__be32	s_first	First block of log information.
			Dynamic information describing the current state of the log.
0x18	__be32	s_sequence	First commit ID expected in log.
0x1C	__be32	s_start	Block number of the start of log. Contrary to the comments, this field being zero does not imply that the journal is clean!
0x20	__be32	s_errno	Error value, as set by jbd2_journal_abort().
			The remaining fields are only valid in a v2 superblock.
0x24	__be32	s_feature_compat;	Compatible feature set. See the table jbd2_compat below.
0x28	__be32	s_feature_incompat	Incompatible feature set. See the table jbd2_incompat below.
0x2C	__be32	s_feature_ro_compat	Read-only compatible feature set. There aren't any of these currently.
0x30	__u8	s_uuid[16]	128-bit uuid for journal. This is compared against the copy in the ext4 super block at mount time.
0x40	__be32	s_nr_users	Number of file systems sharing this journal.
0x44	__be32	s_dynsuper	Location of dynamic super block copy. (Not used?)
0x48	__be32	s_max_transaction	Limit of journal blocks per transaction. (Not used?)
0x4C	__be32	s_max_trans_data	Limit of data blocks per transaction. (Not used?)
0x50	__u8	s_checksum_type	Checksum algorithm used for the journal. See jbd2_checksum_type for more info.
0x51	__u8[3]	s_padding2	
0x54	__be32	s_num_fc_blocks	Number of fast commit blocks in the journal.
0x58	__u32	s_padding[42]	
0xFC	__be32	s_checksum	Checksum of the entire superblock, with this field set to zero.
0x100	__u8	s_users[16*48]	ids of all file systems sharing the log. e2fsprogs/Linux don't allow shared external journals, but I imagine Lustre (or ocfs2?), which use the jbd2 code, might.

The journal compat features are any combination of the following:

Value	Description
0x1	Journal maintains checksums on the data blocks. (JBD2_FEATURE_COMPAT_CHECKSUM)

The journal incompat features are any combination of the following:

Value	Description
0x1	Journal has block revocation records. (JBD2_FEATURE_INCOMPAT_REVOKE)
0x2	Journal can deal with 64-bit block numbers. (JBD2_FEATURE_INCOMPAT_64BIT)
0x4	Journal commits asynchronously. (JBD2_FEATURE_INCOMPAT_ASYNC_COMMIT)
0x8	This journal uses v2 of the checksum on-disk format. Each journal metadata block gets its own checksum, and the block tags in the descriptor table contain checksums for each of the data blocks in the journal. (JBD2_FEATURE_INCOMPAT_CSUM_V2)

Value	Description
0x10	This journal uses v3 of the checksum on-disk format. This is the same as v2, but the journal block tag size is fixed regardless of the size of block numbers. (JBD2_FEATURE_INCOMPAT_CSUM_V3)
0x20	Journal has fast commit blocks. (JBD2_FEATURE_INCOMPAT_FAST_COMMIT)

Journal checksum type codes are one of the following. crc32 or crc32c are the most likely choices.

Value	Description
1	CRC32
2	MD5
3	SHA1
4	CRC32C

Descriptor Block

The descriptor block contains an array of journal block tags that describe the final locations of the data blocks that follow in the journal. Descriptor blocks are open-coded instead of being completely described by a data structure, but here is the block structure anyway. Descriptor blocks consume at least 36 bytes, but use a full block:

Offset	Type	Name	Descriptor
0x0	journal_header_t	(open coded)	Common block header.
0xC	struct journal_block_tag_s	open coded array[]	Enough tags either to fill up the block or to describe all the data blocks that follow this descriptor block.

Journal block tags have any of the following formats, depending on which journal feature and block tag flags are set.

If JBD2_FEATURE_INCOMPAT_CSUM_V3 is set, the journal block tag is defined as `struct journal_block_tag3_s`, which looks like the following. The size is 16 or 32 bytes.

Offset	Type	Name	Descriptor
0x0	__be32	t_blocknr	Lower 32-bits of the location of where the corresponding data block should end up on disk.
0x4	__be32	t_flags	Flags that go with the descriptor. See the table jbd2_tag_flags for more info.
0x8	__be32	t_blocknr_high	Upper 32-bits of the location of where the corresponding data block should end up on disk. This is zero if JBD2_FEATURE_INCOMPAT_64BIT is not enabled.
0xC	__be32	t_checksum	Checksum of the journal UUID, the sequence number, and the data block.
			This field appears to be open coded. It always comes at the end of the tag, after t_checksum. This field is not present if the "same UUID" flag is set.
0x8 or 0xC	char	uuid[16]	A UUID to go with this tag. This field appears to be copied from the `j_uuid` field in `struct journal_s`, but only tune2fs touches that field.

The journal tag flags are any combination of the following:

Value	Description
0x1	On-disk block is escaped. The first four bytes of the data block just happened to match the jbd2 magic number.
0x2	This block has the same UUID as previous, therefore the UUID field is omitted.
0x4	The data block was deleted by the transaction. (Not used?)
0x8	This is the last tag in this descriptor block.

If JBD2_FEATURE_INCOMPAT_CSUM_V3 is NOT set, the journal block tag is defined as `struct journal_block_tag_s`, which looks like the following. The size is 8, 12, 24, or 28 bytes:

Offset	Type	Name	Descriptor
0x0	__be32	t_blocknr	Lower 32-bits of the location of where the corresponding data block should end up on disk.
0x4	__be16	t_checksum	Checksum of the journal UUID, the sequence number, and the data block. Note that only the lower 16 bits are stored.
0x6	__be16	t_flags	Flags that go with the descriptor. See the table jbd2_tag_flags for more info.
			This next field is only present if the super block indicates support for 64-bit block numbers.
0x8	__be32	t_blocknr_high	Upper 32-bits of the location of where the corresponding data block should end up on disk.

Offset	Type	Name	Descriptor
			This field appears to be open coded. It always comes at the end of the tag, after t_flags or t_blocknr_high. This field is not present if the "same UUID" flag is set.
0x8 or 0xC	char	uuid[16]	A UUID to go with this tag. This field appears to be copied from the `j_uuid` field in `struct journal_s`, but only tune2fs touches that field.

If JBD2_FEATURE_INCOMPAT_CSUM_V2 or JBD2_FEATURE_INCOMPAT_CSUM_V3 are set, the end of the block is a `struct jbd2_journal_block_tail`, which looks like this:

Offset	Type	Name	Descriptor
0x0	__be32	t_checksum	Checksum of the journal UUID + the descriptor block, with this field set to zero.

Data Block

In general, the data blocks being written to disk through the journal are written verbatim into the journal file after the descriptor block. However, if the first four bytes of the block match the jbd2 magic number then those four bytes are replaced with zeroes and the “escaped” flag is set in the descriptor block tag.

Revocation Block

A revocation block is used to prevent replay of a block in an earlier transaction. This is used to mark blocks that were journalled at one time but are no longer journalled. Typically this happens if a metadata block is freed and re-allocated as a file data block; in this case, a journal replay after the file block was written to disk will cause corruption.

NOTE: This mechanism is NOT used to express “this journal block is superseded by this other journal block”, as the author (djwong) mistakenly thought. Any block being added to a transaction will cause the removal of all existing revocation records for that block.

Revocation blocks are described in `struct jbd2_journal_revoke_header_s`, are at least 16 bytes in length, but use a full block:

Offset	Type	Name	Description
0x0	journal_header_t	r_header	Common block header.
0xC	__be32	r_count	Number of bytes used in this block.
0x10	__be32 or __be64	blocks[0]	Blocks to revoke.

After r_count is a linear array of block numbers that are effectively revoked by this transaction. The size of each block number is 8 bytes if the superblock advertises 64-bit block number support, or 4 bytes otherwise.

If JBD2_FEATURE_INCOMPAT_CSUM_V2 or JBD2_FEATURE_INCOMPAT_CSUM_V3 are set, the end of the revocation block is a `struct jbd2_journal_revoke_tail`, which has this format:

Offset	Type	Name	Description
0x0	__be32	r_checksum	Checksum of the journal UUID + revocation block

Commit Block

The commit block is a sentry that indicates that a transaction has been completely written to the journal. Once this commit block reaches the journal, the data stored with this transaction can be written to their final locations on disk.

The commit block is described by `struct commit_header`, which is 32 bytes long (but uses a full block):

Offset	Type	Name	Descriptor
0x0	journal_header_s	(open coded)	Common block header.
0xC	unsigned char	h_chksum_type	The type of checksum to use to verify the integrity of the data blocks in the transaction. See jbd2_checksum_type for more info.
0xD	unsigned char	h_chksum_size	The number of bytes used by the checksum. Most likely 4.
0xE	unsigned char	h_padding[2]	

Offset	Type	Name	Descriptor
0x10	__be32	h_chksum[JBD2_CHECKSUM_BYTES]	32 bytes of space to store checksums. If JBD2_FEATURE_INCOMPAT_CSUM_V2 or JBD2_FEATURE_INCOMPAT_CSUM_V3 are set, the first `__be32` is the checksum of the journal UUID and the entire commit block, with this field zeroed. If JBD2_FEATURE_COMPAT_CHECKSUM is set, the first `__be32` is the crc32 of all the blocks already written to the transaction.
0x30	__be64	h_commit_sec	The time that the transaction was committed, in seconds since the epoch.
0x38	__be32	h_commit_nsec	Nanoseconds component of the above timestamp.

Fast commits

Fast commit area is organized as a log of tag length values. Each TLV has a `struct ext4_fc_tl` in the beginning which stores the tag and the length of the entire field. It is followed by variable length tag specific value. Here is the list of supported tags and their meanings:

Tag	Meaning	Value struct	Description
EXT4_FC_TAG_HEAD	Fast commit area header	`struct ext4_fc_head`	Stores the TID of the transaction after which these fast commits should be applied.
EXT4_FC_TAG_ADD_RANGE	Add extent to inode	`struct ext4_fc_add_range`	Stores the inode number and extent to be added in this inode
EXT4_FC_TAG_DEL_RANGE	Remove logical offsets to inode	`struct ext4_fc_del_range`	Stores the inode number and the logical offset range that needs to be removed
EXT4_FC_TAG_CREAT	Create directory entry for a newly created file	`struct ext4_fc_dentry_info`	Stores the parent inode number, inode number and directory entry of the newly created file
EXT4_FC_TAG_LINK	Link a directory entry to an inode	`struct ext4_fc_dentry_info`	Stores the parent inode number, inode number and directory entry
EXT4_FC_TAG_UNLINK	Unlink a directory entry of an inode	`struct ext4_fc_dentry_info`	Stores the parent inode number, inode number and directory entry
EXT4_FC_TAG_PAD	Padding (unused area)	None	Unused bytes in the fast commit area.
EXT4_FC_TAG_TAIL	Mark the end of a fast commit	`struct ext4_fc_tail`	Stores the TID of the commit, CRC of the fast commit of which this tag represents the end of

Fast Commit Replay Idempotence

Fast commits tags are idempotent in nature provided the recovery code follows certain rules. The guiding principle that the commit path follows while committing is that it stores the result of a particular operation instead of storing the procedure.

Let's consider this rename operation: 'mv /a /b'. Let's assume dirent '/a' was associated with inode 10. During fast commit, instead of storing this operation as a procedure "rename a to b", we store the resulting file system state as a "series" of outcomes:

- Link dirent b to inode 10
- Unlink dirent a
- Inode 10 with valid refcount

Now when recovery code runs, it needs "enforce" this state on the file system. This is what guarantees idempotence of fast commit replay.

Let's take an example of a procedure that is not idempotent and see how fast commits make it idempotent. Consider following sequence of operations:

1. rm A
2. mv B A
3. read A

If we store this sequence of operations as is then the replay is not idempotent. Let's say while in replay, we crash after (2). During the second replay, file A (which was actually created as a result of "mv B A" operation) would get deleted. Thus, file named A would be absent when we try to read A. So, this sequence of operations is not idempotent. However, as mentioned above, instead of storing the procedure fast commits store the outcome of each procedure. Thus the fast commit log for above procedure would be as follows:

(Let's assume dirent A was linked to inode 10 and dirent B was linked to inode 11 before the replay)

1. Unlink A
2. Link A to inode 11
3. Unlink B
4. Inode 11

If we crash after (3) we will have file A linked to inode 11. During the second replay, we will remove file A (inode 11). But we will create it back and make it point to inode 11. We won't find B, so we'll just skip that step. At this point, the refcount for inode 11 is not reliable, but that gets fixed by the replay of last inode 11 tag. Thus, by converting a non-idempotent procedure into a series of idempotent outcomes, fast commits ensured idempotence during the replay.

Journal Checkpoint

Checkpointing the journal ensures all transactions and their associated buffers are submitted to the disk. In-progress transactions are waited upon and included in the checkpoint. Checkpointing is used internally during critical updates to the filesystem including journal recovery, filesystem resizing, and freeing of the journal_t structure.

A journal checkpoint can be triggered from userspace via the ioctl EXT4_IOC_CHECKPOINT. This ioctl takes a single, u64 argument for flags. Currently, three flags are supported. First, EXT4_IOC_CHECKPOINT_FLAG_DRY_RUN can be used to verify input to the ioctl. It returns error if there is any invalid input, otherwise it returns success without performing any checkpointing. This can be used to check whether the ioctl exists on a system and to verify there are no issues with arguments or flags. The other two flags are EXT4_IOC_CHECKPOINT_FLAG_DISCARD and EXT4_IOC_CHECKPOINT_FLAG_ZEROOUT. These flags cause the journal blocks to be discarded or zero-filled, respectively, after the journal checkpoint is complete. EXT4_IOC_CHECKPOINT_FLAG_DISCARD and EXT4_IOC_CHECKPOINT_FLAG_ZEROOUT cannot both be set. The ioctl may be useful when snapshotting a system or for complying with content deletion SLOs.