Ext4 Disk Layout

From Ext4

This document attempts to describe the on-disk format for ext4 filesystems. The same general ideas should apply to ext2/3 filesystems as well, though they do not support all the features that ext4 supports, and the fields will be shorter.

NOTE: This is a work in progress, based on notes that the author (djwong) made while picking apart a filesystem by hand. The data structure definitions should be current as of Linux 4.0 and e2fsprogs-1.42.13. All comments and corrections are welcome, since there is undoubtedly plenty of lore that might not be reflected in freshly created demonstration filesystems.

Contents

Terminology

ext4 divides a storage device into an array of logical blocks both to reduce bookkeeping overhead and to increase throughput by forcing larger transfer sizes. Generally, the block size will be 4KiB (the same size as pages on x86 and the block layer's default block size), though the actual size is calculated as 2 ^ (10 + `sb.s_log_block_size`) bytes. Throughout this document, disk locations are given in terms of these logical blocks, not raw LBAs, and not 1024-byte blocks. For the sake of convenience, the logical block size will be referred to as `$block_size` throughout the rest of the document.

When referenced in `preformatted text` blocks, `sb` refers to fields in the super block, and `inode` refers to fields in an inode table entry.

Overview

An ext4 file system is split into a series of block groups. To reduce performance difficulties due to fragmentation, the block allocator tries very hard to keep each file's blocks within the same group, thereby reducing seek times. The size of a block group is specified in `sb.s_blocks_per_group` blocks, though it can also calculated as 8 * `block_size_in_bytes`. With the default block size of 4KiB, each group will contain 32,768 blocks, for a length of 128MiB. The number of block groups is the size of the device divided by the size of a block group.

All fields in ext4 are written to disk in little-endian order. HOWEVER, all fields in jbd2 (the journal) are written to disk in big-endian order.

Blocks

ext4 allocates storage space in units of "blocks". A block is a group of sectors between 1KiB and 64KiB, and the number of sectors must be an integral power of 2. Blocks are in turn grouped into larger units called block groups. Block size is specified at mkfs time and typically is 4KiB. You may experience mounting problems if block size is greater than page size (i.e. 64KiB blocks on a i386 which only has 4KiB memory pages). By default a filesystem can contain 2^32 blocks; if the '64bit' feature is enabled, then a filesystem can have 2^64 blocks.

File System Maximums

	32-bit mode				64-bit mode			
Item	**1KiB**	**2KiB**	**4KiB**	**64KiB**	**1KiB**	**2KiB**	**4KiB**	**64KiB**
Blocks	2^32	2^32	2^32	2^32	2^64	2^64	2^64	2^64
Inodes	2^32	2^32	2^32	2^32	2^32	2^32	2^32	2^32
File System Size	4TiB	8TiB	16TiB	256PiB	16ZiB	32ZiB	64ZiB	1YiB

Blocks Per Block Group	8,192	16,384	32,768	524,288	8,192	16,384	32,768	524,288
Inodes Per Block Group	8,192	16,384	32,768	524,288	8,192	16,384	32,768	524,288
Block Group Size	8MiB	32MiB	128MiB	32GiB	8MiB	32MiB	128MiB	32GiB
Blocks Per File, Extents	2^32	2^32	2^32	2^32	2^32	2^32	2^32	2^32
Blocks Per File, Block Maps	16,843,020	134,480,396	1,074,791,436	4,398,314,962,956	16,843,020	134,480,396	1,074,791,436	4,398,314,962,956
File Size, Extents	4TiB	8TiB	16TiB	256TiB	4TiB	8TiB	16TiB	256TiB
File Size, Block Maps	16GiB	256GiB	4TiB	256PiB	16GiB	256GiB	4TiB	256PiB

Note: Files not using extents (i.e. files using block maps) must be placed in the first 2^32 blocks of a filesystem.

Layout

The layout of a standard block group is approximately as follows (each of these fields is discussed in a separate section below):

Group 0 Padding	ext4 Super Block	Group Descriptors	Reserved GDT Blocks	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1024 bytes	1 block	many blocks	many blocks	1 block	1 block	many blocks	many more blocks

For the special case of block group 0, the first 1024 bytes are unused, to allow for the installation of x86 boot sectors and other oddities. The superblock will start at offset 1024 bytes, whichever block that happens to be (usually 0). However, if for some reason the block size = 1024, then block 0 is marked in use and the superblock goes in block 1. For all other block groups, there is no padding.

The ext4 driver primarily works with the superblock and the group descriptors that are found in block group 0. Redundant copies of the superblock and group descriptors are written to some of the block groups across the disk in case the beginning of the disk gets trashed, though not all block groups necessarily host a redundant copy (see following paragraph for more details). If the group does not have a redundant copy, the block group begins with the data block bitmap. Note also that when the filesystem is freshly formatted, mkfs will allocate "reserve GDT block" space after the block group descriptors and before the start of the block bitmaps to allow for future expansion of the filesystem. By default, a filesystem is allowed to increase in size by a factor of 1024x over the

original filesystem size.

The location of the inode table is given by `grp.bg_inode_table_*`. It is continuous range of blocks large enough to contain `sb.s_inodes_per_group * sb.s_inode_size` bytes.

As for the ordering of items in a block group, it is generally established that the super block and the group descriptor table, if present, will be at the beginning of the block group. The bitmaps and the inode table can be anywhere, and it is quite possible for the bitmaps to come after the inode table, or for both to be in different groups (flex_bg). Leftover space is used for file data blocks, indirect block maps, extent tree blocks, and extended attributes.

Flexible Block Groups

Starting in ext4, there is a new feature called flexible block groups (flex_bg). In a flex_bg, several block groups are tied together as one logical block group; the bitmap spaces and the inode table space in the first block group of the flex_bg are expanded to include the bitmaps and inode tables of all other block groups in the flex_bg. For example, if the flex_bg size is 4, then group 0 will contain (in order) the superblock, group descriptors, data block bitmaps for groups 0-3, inode bitmaps for groups 0-3, inode tables for groups 0-3, and the remaining space in group 0 is for file data. The effect of this is to group the block metadata close together for faster loading, and to enable large files to be continuous on disk. Backup copies of the superblock and group descriptors are always at the beginning of block groups, even if flex_bg is enabled. The number of block groups that make up a flex_bg is given by 2 ^ `sb.s_log_groups_per_flex`.

Meta Block Groups

Without the option META_BG, for safety concerns, all block group descriptors copies are kept in the first block group. Given the default 128MiB(2^27 bytes) block group size and 64-byte group descriptors, ext4 can have at most 2^27/64 = 2^21 block groups. This limits the entire filesystem size to 2^21 ∗ 2^27 = 2^48bytes or 256TiB.

The solution to this problem is to use the metablock group feature (META_BG), which is already in ext3 for all 2.6 releases. With the META_BG feature, ext4 filesystems are partitioned into many metablock groups. Each metablock group is a cluster of block groups whose group descriptor structures can be stored in a single disk block. For ext4 filesystems with 4 KB block size, a single metablock group partition includes 64 block groups, or 8 GiB of disk space. The metablock group feature moves the location of the group descriptors from the congested first block group of the whole filesystem into the first group of each metablock group itself. The backups are in the second and last group of each metablock group. This increases the 2^21 maximum block groups limit to the hard limit 2^32, allowing support for a 512PiB filesystem.

The change in the filesystem format replaces the current scheme where the superblock is followed by a variable-length set of block group descriptors. Instead, the superblock and a single block group descriptor block is placed at the beginning of the first, second, and last block groups in a meta-block group. A meta-block group is a collection of block groups which can be described by a single block group descriptor block. Since the size of the block group descriptor structure is 32 bytes, a meta-block group contains 32 block groups for filesystems with a 1KB block size, and 128 block groups for filesystems with a 4KB blocksize. Filesystems can either be created using this new block group descriptor layout, or existing filesystems can be resized on-line, and the field s_first_meta_bg in the superblock will indicate the first block group using this new layout.

Please see an important note about `BLOCK_UNINIT` in the section about block and inode bitmaps.

Lazy Block Group Initialization

A new feature for ext4 are three block group descriptor flags that enable mkfs to skip initializing other parts of the block group metadata. Specifically, the INODE_UNINIT and BLOCK_UNINIT flags mean that the inode and block bitmaps for that group can be calculated and therefore the on-disk bitmap blocks are not initialized. This is generally the case for an empty block group or a block group containing only fixed-location block group metadata. The INODE_ZEROED flag means that the inode table has been initialized; mkfs will unset this flag and rely on the kernel to initialize the inode tables in the background.

By not writing zeroes to the bitmaps and inode table, mkfs time is reduced considerably. Note the feature flag is RO_COMPAT_GDT_CSUM, but the dumpe2fs output prints this as "uninit_bg". They are the same thing.

Special inodes

ext4 reserves some inode for special features, as follows:

inode	Purpose
0	Doesn't exist; there is no inode 0.
1	List of defective blocks.
2	Root directory.
3	User quota.
4	Group quota.
5	Boot loader.
6	Undelete directory.
7	Reserved group descriptors inode. ("resize inode")
8	Journal inode.
9	The "exclude" inode, for snapshots(?)
10	Replica inode, used for some non-upstream feature?
11	Traditional first non-reserved inode. Usually this is the lost+found directory. See s_first_ino in the superblock.

Block and Inode Allocation Policy

ext4 recognizes (better than ext3, anyway) that data locality is generally a desirably quality of a filesystem. On a spinning disk, keeping related blocks near each other reduces the amount of movement that the head actuator and disk must perform to access a data block, thus speeding up disk IO. On an SSD there of course are no moving parts, but locality can increase the size of each transfer request while reducing the total number of requests. This locality may also have the effect of concentrating writes on a single erase block, which can speed up file rewrites significantly. Therefore, it is useful to reduce fragmentation whenever possible.

The first tool that ext4 uses to combat fragmentation is the multi-block allocator. When a file is first created, the block allocator speculatively allocates 8KiB of disk space to the file on the assumption that the space will get written soon. When the file is closed, the unused speculative allocations are of course freed, but if the speculation is correct (typically the case for full writes of small files) then the file data gets written out in a single multi-block extent. A second related trick that ext4 uses is delayed allocation. Under this scheme, when a file needs more blocks to absorb file writes, the filesystem defers deciding the exact placement on the disk until all the dirty buffers are being written out to disk. By not committing to a particular placement until it's absolutely necessary (the commit timeout is hit, or sync() is called, or the kernel runs out of memory), the hope is that the filesystem can make better location decisions.

The third trick that ext4 (and ext3) uses is that it tries to keep a file's data blocks in the same block group as its inode. This cuts down on the seek penalty when the filesystem first has to read a file's inode to learn where the file's data blocks live and then seek over to the file's data blocks to begin I/O operations.

The fourth trick is that all the inodes in a directory are placed in the same block group as the directory, when feasible. The working assumption here is that all the files in a directory might be related, therefore it is useful to try to keep them all together.

The fifth trick is that the disk volume is cut up into 128MB block groups; these mini-containers are used as outlined above to try to maintain data locality. However, there is a deliberate quirk -- when a directory is created in the root directory, the inode allocator scans the block groups and puts that directory into the least heavily loaded block group that it can find. This encourages directories to spread out over a disk; as the top-level directory/file blobs fill up one block group, the allocators simply move on to the next block group. Allegedly this scheme evens out the loading on the block groups, though the author suspects that the directories which are so unlucky as to land towards the end of a spinning drive get a raw deal performance-wise.

Of course if all of these mechanisms fail, one can always use e4defrag to defragment files.

Checksums

Starting in early 2012, metadata checksums were added to all major ext4 and jbd2 data structures. The associated feature flag is metadata_csum. The desired checksum algorithm is indicated in the superblock, though as of October 2012 the only supported algorithm is crc32c. Some data structures did not have space to fit a full 32-bit checksum, so only the lower 16 bits are stored. Enabling the 64bit feature increases the data structure size so that full 32-bit checksums can be stored for many data structures. However, existing 32-bit filesystems cannot be extended to enable 64bit mode, at least not without the experimental resize2fs patches to do so.

Existing filesystems can have checksumming added by running `tune2fs -O metadata_csum` against the underlying device. If tune2fs encounters directory blocks that lack sufficient empty space to add a checksum, it will request that you run `e2fsck -D` to have the directories rebuilt with checksums. This has the added benefit of removing slack space from the directory files and rebalancing the htree indexes. If you _ignore_ this step, your directories will not be protected by a checksum!

The following table describes the data elements that go into each type of checksum. The checksum function is whatever the superblock describes (crc32c as of October 2013) unless noted otherwise.

Metadata	Length	Ingredients
Superblock	__le32	The entire superblock up to the checksum field. The UUID lives inside the superblock.
MMP	__le32	UUID + the entire MMP block up to the checksum field.
Extended Attributes	__le32	UUID + the entire extended attribute block. The checksum field is set to zero.
Directory Entries	__le32	UUID + inode number + inode generation + the directory block up to the fake entry enclosing the checksum field.
HTREE Nodes	__le32	UUID + inode number + inode generation + all valid extents + HTREE tail. The checksum field is set to zero.
Extents	__le32	UUID + inode number + inode generation + the entire extent block up to the checksum field.
Bitmaps	__le32 or __le16	UUID + the entire bitmap. Checksums are stored in the group descriptor, and truncated if the group descriptor size is 32 bytes (i.e. ^64bit)
Inodes	__le32	UUID + inode number + inode generation + the entire inode. The checksum field is set to zero. Each inode has its own checksum.
Group Descriptors	__le16	If metadata_csum, then UUID + group number + the entire descriptor; else if gdt_csum, then crc16(UUID + group number + the entire descriptor). In all cases, only the lower 16 bits are stored.

Bigalloc

At the moment, the default size of a block is 4KiB, which is a commonly supported page size on most MMU-capable hardware. This is fortunate, as ext4 code is not prepared to handle the case where the block size exceeds the page size. However, for a filesystem of mostly huge files, it is desirable to be able to allocate disk blocks in units of multiple blocks to reduce both fragmentation and metadata overhead. The bigalloc feature provides exactly this ability. The administrator can set a block cluster size at mkfs time (which is stored in the s_log_cluster_size field in the superblock); from then on, the block bitmaps track clusters, not individual blocks. This means that block groups can be several gigabytes in size (instead of just 128MiB); however, the minimum allocation unit becomes a cluster, not a block, even for directories. TaoBao had a patchset to extend the "use units of clusters instead of blocks" to the extent tree, though it is not clear where those patches went-- they eventually morphed into "extent tree v2" but that code has not landed as of May 2015.

Inline Data

The inline data feature was designed to handle the case that a file's data is so tiny that it readily fits inside the inode, which (theoretically) reduces disk block consumption and reduces seeks. If the file is smaller than 60 bytes, then the data are stored inline

in `inode.i_block`. If the rest of the file would fit inside the extended attribute space, then it might be found as an extended attribute "system.data" within the inode body ("ibody EA"). This of course constrains the amount of extended attributes one can attach to an inode. If the data size increases beyond i_block + ibody EA, a regular block is allocated and the contents moved to that block.

Inline Directories

The first four bytes of i_block are the inode number of the parent directory. Following that is a 56-byte space for an array of directory entries; see `struct ext4_dir_entry`. If there is a "system.data" attribute in the inode body, the EA value is an array of `struct ext4_dir_entry` as well. Note that for inline directories, the i_block and EA space are treated as separate dirent blocks; directory entries cannot span the two.

Inline directory entries are not checksummed, as the inode checksum should protect all inline data contents.

Orphan File [PROPOSED]

Proposed by Jan Kara in early 2015, the orphan file feature aims to reduce locking contention during delete operations by replacing the singly linked orphan inode list (and lock) with a file containing multiple blocks. Each CPU ought to be able to claim its own block, which implies that the orphan list can be updated locklessly. Each block contains a list of orphaned inodes; recovery involves iterating all blocks of the orphan file looking for non-zero inode numbers to erase. This feature will come with a rocompat feature flag to indicate the ability to use an orphan file and a compat flag indicating that the orphan file actually contains orphaned inode records. The format of an orphan file block is as follows:

Field	Length	Description
inodes	__le32[]	A list of orphaned inodes. 0 means the entry is empty. The length of the array is the size of the block minus the tail.
magic	__le32	Magic number of orphan file blocks, 0x0B10CA04.
checksum	__le32	Checksum of the UUID + inode number + inode generation + the orphan block up to but not including the footer.

The inode number of the orphan file itself has not been settled; as of May 2015 the test patches re-use inode #9.

The Super Block

The superblock records various information about the enclosing filesystem, such as block counts, inode counts, supported features, maintenance information, and more.

If the sparse_super feature flag is set, redundant copies of the superblock and group descriptors are kept only in the groups whose group number is either 0 or a power of 3, 5, or 7. If the flag is not set, redundant copies are kept in all groups.

The superblock checksum is calculated against the superblock structure, which includes the FS UUID.

The ext4 superblock is laid out as follows in `struct ext4_super_block`:

Offset	Size	Name	Description
0x0	__le32	s_inodes_count	Total inode count.
0x4	__le32	s_blocks_count_lo	Total block count.
0x8	__le32	s_r_blocks_count_lo	This number of blocks can only be allocated by the super-user.
0xC	__le32	s_free_blocks_count_lo	Free block count.
0x10	__le32	s_free_inodes_count	Free inode count.

0x14	__le32	s_first_data_block	First data block. This must be at least 1 for 1k-block filesystems and is typically 0 for all other block sizes.
0x18	__le32	s_log_block_size	Block size is 2 ^ (10 + s_log_block_size).
0x1C	__le32	s_log_cluster_size	Cluster size is (2 ^ s_log_cluster_size) blocks if bigalloc is enabled, zero otherwise.
0x20	__le32	s_blocks_per_group	Blocks per group.
0x24	__le32	s_clusters_per_group	Clusters per group, if bigalloc is enabled.
0x28	__le32	s_inodes_per_group	Inodes per group.
0x2C	__le32	s_mtime	Mount time, in seconds since the epoch.
0x30	__le32	s_wtime	Write time, in seconds since the epoch.
0x34	__le16	s_mnt_count	Number of mounts since the last fsck.
0x36	__le16	s_max_mnt_count	Number of mounts beyond which a fsck is needed.
0x38	__le16	s_magic	Magic signature, 0xEF53
0x3A	__le16	s_state	File system state. Valid values are: 0x0001 Cleanly umounted 0x0002 Errors detected 0x0004 Orphans being recovered
0x3C	__le16	s_errors	Behaviour when detecting errors. One of: 1 Continue 2 Remount read-only 3 Panic
0x3E	__le16	s_minor_rev_level	Minor revision level.
0x40	__le32	s_lastcheck	Time of last check, in seconds since the epoch.
0x44	__le32	s_checkinterval	Maximum time between checks, in seconds.
0x48	__le32	s_creator_os	OS. One of: 0 Linux 1 Hurd 2 Masix 3 FreeBSD 4 Lites
0x4C	__le32	s_rev_level	Revision level. One of: 0 Original format 1 v2 format w/ dynamic inode sizes
0x50	__le16	s_def_resuid	Default uid for reserved blocks.
0x52	__le16	s_def_resgid	Default gid for reserved blocks.
These fields are for EXT4_DYNAMIC_REV superblocks only. Note: the difference between the compatible feature set and the incompatible feature set is that if there is a bit set in the incompatible feature set that the kernel doesn't know about, it should refuse to mount the filesystem. e2fsck's requirements are more strict; if it doesn't know about a feature in either the compatible or incompatible feature set, it must abort and not try to meddle with things it doesn't understand...			
0x54	__le32	s_first_ino	First non-reserved inode.

0x58	__le16	s_inode_size	Size of inode structure, in bytes.
0x5A	__le16	s_block_group_nr	Block group # of this superblock.
0x5C	__le32	s_feature_compat	Compatible feature set flags. Kernel can still read/write this fs even if it doesn't understand a flag; fsck should not do that. Any of: 0x1 Directory preallocation (COMPAT_DIR_PREALLOC). 0x2 "imagic inodes". Not clear from the code what this does (COMPAT_IMAGIC_INODES). 0x4 Has a journal (COMPAT_HAS_JOURNAL). 0x8 Supports extended attributes (COMPAT_EXT_ATTR). 0x10 Has reserved GDT blocks for filesystem expansion (COMPAT_RESIZE_INODE). 0x20 Has directory indices (COMPAT_DIR_INDEX). 0x40 "Lazy BG". Not in Linux kernel, seems to have been for uninitialized block groups? (COMPAT_LAZY_BG) 0x80 "Exclude inode". Not used. (COMPAT_EXCLUDE_INODE). 0x100 "Exclude bitmap". Seems to be used to indicate the presence of snapshot-related exclude bitmaps? Not defined in kernel or used in e2fsprogs (COMPAT_EXCLUDE_BITMAP). 0x200 Sparse Super Block, v2. If this flag is set, the SB field s_backup_bgs points to the two block groups that contain backup superblocks (COMPAT_SPARSE_SUPER2).
0x60	__le32	s_feature_incompat	Incompatible feature set. If the kernel or fsck doesn't understand one of these bits, it should stop. Any of: 0x1 Compression (INCOMPAT_COMPRESSION). 0x2 Directory entries record the file type. See ext4_dir_entry_2 below (INCOMPAT_FILETYPE). 0x4 Filesystem needs recovery (INCOMPAT_RECOVER). 0x8 Filesystem has a separate journal device (INCOMPAT_JOURNAL_DEV). 0x10 Meta block groups. See the earlier discussion of this feature (INCOMPAT_META_BG). 0x40 Files in this filesystem use extents (INCOMPAT_EXTENTS). 0x80 Enable a filesystem size of 2^64 blocks (INCOMPAT_64BIT). 0x100 Multiple mount protection. Not implemented (INCOMPAT_MMP). 0x200 Flexible block groups. See the earlier discussion of this feature (INCOMPAT_FLEX_BG). 0x400 Inodes can be used for large extended attributes (INCOMPAT_EA_INODE). (Not implemented?) 0x1000 Data in directory entry (INCOMPAT_DIRDATA). (Not implemented?) 0x2000 Never used (INCOMPAT_BG_USE_META_CSUM). Available for use. 0x4000 Large directory >2GB or 3-level htree (INCOMPAT_LARGEDIR). 0x8000 Data in inode (INCOMPAT_INLINE_DATA). 0x10000 Encrypted inodes are present on the filesystem. (INCOMPAT_ENCRYPT).
0x64	__le32	s_feature_ro_compat	Readonly-compatible feature set. If the kernel doesn't understand one of these bits, it can still mount read-only. Any of: 0x1 Sparse superblocks. See the earlier discussion of this feature

			(RO_COMPAT_SPARSE_SUPER). 0x2 This filesystem has been used to store a file greater than 2GiB (RO_COMPAT_LARGE_FILE). 0x4 Not used in kernel or e2fsprogs (RO_COMPAT_BTREE_DIR). 0x8 This filesystem has files whose sizes are represented in units of logical blocks, not 512-byte sectors. This implies a very large file indeed! (RO_COMPAT_HUGE_FILE) 0x10 Group descriptors have checksums. In addition to detecting corruption, this is useful for lazy formatting with uninitialized groups (RO_COMPAT_GDT_CSUM). 0x20 Indicates that the old ext3 32,000 subdirectory limit no longer applies (RO_COMPAT_DIR_NLINK). 0x40 Indicates that large inodes exist on this filesystem (RO_COMPAT_EXTRA_ISIZE). 0x80 This filesystem has a snapshot (RO_COMPAT_HAS_SNAPSHOT). 0x100 Quota (RO_COMPAT_QUOTA). 0x200 This filesystem supports "bigalloc", which means that file extents are tracked in units of clusters (of blocks) instead of blocks (RO_COMPAT_BIGALLOC). 0x400 This filesystem supports metadata checksumming. (RO_COMPAT_METADATA_CSUM; implies RO_COMPAT_GDT_CSUM, though GDT_CSUM must not be set) 0x800 Filesystem supports replicas. This feature is neither in the kernel nor e2fsprogs. (RO_COMPAT_REPLICA) 0x1000 Read-only filesystem image; the kernel will not mount this image read-write and most tools will refuse to write to the image. (RO_COMPAT_READONLY)
0x68	__u8	s_uuid[16]	128-bit UUID for volume.
0x78	char	s_volume_name[16]	Volume label.
0x88	char	s_last_mounted[64]	Directory where filesystem was last mounted.
0xC8	__le32	s_algorithm_usage_bitmap	For compression (Not used in e2fsprogs/Linux)
Performance hints. Directory preallocation should only happen if the EXT4_FEATURE_COMPAT_DIR_PREALLOC flag is on.			
0xCC	__u8	s_prealloc_blocks	# of blocks to try to preallocate for ... files? (Not used in e2fsprogs/Linux)
0xCD	__u8	s_prealloc_dir_blocks	# of blocks to preallocate for directories. (Not used in e2fsprogs/Linux)
0xCE	__le16	s_reserved_gdt_blocks	Number of reserved GDT entries for future filesystem expansion.
Journaling support valid if EXT4_FEATURE_COMPAT_HAS_JOURNAL set.			
0xD0	__u8	s_journal_uuid[16]	UUID of journal superblock
0xE0	__le32	s_journal_inum	inode number of journal file.
0xE4	__le32	s_journal_dev	Device number of journal file, if the external journal feature flag is set.
0xE8	__le32	s_last_orphan	Start of list of orphaned inodes to delete.
0xEC	__le32	s_hash_seed[4]	HTREE hash seed.
0xFC	__u8	s_def_hash_version	Default hash algorithm to use for directory hashes. One of: 0x0 Legacy. 0x1 Half MD4.

			0x2 Tea. 0x3 Legacy, unsigned. 0x4 Half MD4, unsigned. 0x5 Tea, unsigned.
0xFD	__u8	s_jnl_backup_type	If this value is 0 or EXT3_JNL_BACKUP_BLOCKS (1), then the `s_jnl_blocks` field contains a duplicate copy of the inode's `i_block[]` array and `i_size`.
0xFE	__le16	s_desc_size	Size of group descriptors, in bytes, if the 64bit incompat feature flag is set.
0x100	__le32	s_default_mount_opts	Default mount options. Any of: 0x0001 Print debugging info upon (re)mount. (EXT4_DEFM_DEBUG) 0x0002 New files take the gid of the containing directory (instead of the fsgid of the current process). (EXT4_DEFM_BSDGROUPS) 0x0004 Support userspace-provided extended attributes. (EXT4_DEFM_XATTR_USER) 0x0008 Support POSIX access control lists (ACLs). (EXT4_DEFM_ACL) 0x0010 Do not support 32-bit UIDs. (EXT4_DEFM_UID16) 0x0020 All data and metadata are commited to the journal. (EXT4_DEFM_JMODE_DATA) 0x0040 All data are flushed to the disk before metadata are committed to the journal. (EXT4_DEFM_JMODE_ORDERED) 0x0060 Data ordering is not preserved; data may be written after the metadata has been written. (EXT4_DEFM_JMODE_WBACK) 0x0100 Disable write flushes. (EXT4_DEFM_NOBARRIER) 0x0200 Track which blocks in a filesystem are metadata and therefore should not be used as data blocks. This option will be enabled by default on 3.18, hopefully. (EXT4_DEFM_BLOCK_VALIDITY) 0x0400 Enable DISCARD support, where the storage device is told about blocks becoming unused. (EXT4_DEFM_DISCARD) 0x0800 Disable delayed allocation. (EXT4_DEFM_NODELALLOC)
0x104	__le32	s_first_meta_bg	First metablock block group, if the meta_bg feature is enabled.
0x108	__le32	s_mkfs_time	When the filesystem was created, in seconds since the epoch.
0x10C	__le32	s_jnl_blocks[17]	Backup copy of the journal inode's `i_block[]` array in the first 15 elements and i_size_high and i_size in the 16th and 17th elements, respectively.
64bit support valid if EXT4_FEATURE_COMPAT_64BIT			
0x150	__le32	s_blocks_count_hi	High 32-bits of the block count.
0x154	__le32	s_r_blocks_count_hi	High 32-bits of the reserved block count.
0x158	__le32	s_free_blocks_count_hi	High 32-bits of the free block count.
0x15C	__le16	s_min_extra_isize	All inodes have at least # bytes.
0x15E	__le16	s_want_extra_isize	New inodes should reserve # bytes.
0x160	__le32	s_flags	Miscellaneous flags. Any of: 0x0001 Signed directory hash in use. 0x0002 Unsigned directory hash in use. 0x0004 To test development code.

0x164	__le16	s_raid_stride	RAID stride. This is the number of logical blocks read from or written to the disk before moving to the next disk. This affects the placement of filesystem metadata, which will hopefully make RAID storage faster.
0x166	__le16	s_mmp_interval	# seconds to wait in multi-mount prevention (MMP) checking. In theory, MMP is a mechanism to record in the superblock which host and device have mounted the filesystem, in order to prevent multiple mounts. This feature does not seem to be implemented...
0x168	__le64	s_mmp_block	Block # for multi-mount protection data.
0x170	__le32	s_raid_stripe_width	RAID stripe width. This is the number of logical blocks read from or written to the disk before coming back to the current disk. This is used by the block allocator to try to reduce the number of read-modify-write operations in a RAID5/6.
0x174	__u8	s_log_groups_per_flex	Size of a flexible block group is 2 ^ `s_log_groups_per_flex`.
0x175	__u8	s_checksum_type	Metadata checksum algorithm type. The only valid value is 1 (crc32c).
0x176	__le16	s_reserved_pad	
0x178	__le64	s_kbytes_written	Number of KiB written to this filesystem over its lifetime.
0x180	__le32	s_snapshot_inum	inode number of active snapshot. (Not used in e2fsprogs/Linux.)
0x184	__le32	s_snapshot_id	Sequential ID of active snapshot. (Not used in e2fsprogs/Linux.)
0x188	__le64	s_snapshot_r_blocks_count	Number of blocks reserved for active snapshot's future use. (Not used in e2fsprogs/Linux.)
0x190	__le32	s_snapshot_list	inode number of the head of the on-disk snapshot list. (Not used in e2fsprogs/Linux.)
0x194	__le32	s_error_count	Number of errors seen.
0x198	__le32	s_first_error_time	First time an error happened, in seconds since the epoch.
0x19C	__le32	s_first_error_ino	inode involved in first error.
0x1A0	__le64	s_first_error_block	Number of block involved of first error.
0x1A8	__u8	s_first_error_func[32]	Name of function where the error happened.
0x1C8	__le32	s_first_error_line	Line number where error happened.
0x1CC	__le32	s_last_error_time	Time of most recent error, in seconds since the epoch.
0x1D0	__le32	s_last_error_ino	inode involved in most recent error.
0x1D4	__le32	s_last_error_line	Line number where most recent error happened.
0x1D8	__le64	s_last_error_block	Number of block involved in most recent error.
0x1E0	__u8	s_last_error_func[32]	Name of function where the most recent error happened.
0x200	__u8	s_mount_opts[64]	ASCIIZ string of mount options.
0x240	__le32	s_usr_quota_inum	Inode number of user quota file.
0x244	__le32	s_grp_quota_inum	Inode number of group quota file.
0x248	__le32	s_overhead_blocks	Overhead blocks/clusters in fs. (Huh? This field is always zero, which means that the kernel calculates it dynamically.)
0x24C	__le32	s_backup_bgs[2]	Block groups containing superblock backups (if sparse_super2)
0x24E	__le32	s_encrypt_algos[4]	Encryption algorithms in use. There can be up to four algorithms in use at any time; valid algorithm codes are given below: 0 Invalid algorithm (ENCRYPTION_MODE_INVALID). 1 256-bit AES in XTS mode (ENCRYPTION_MODE_AES_256_XTS). 2 256-bit AES in GCM mode (ENCRYPTION_MODE_AES_256_GCM).

			3 256-bit AES in CBC mode (ENCRYPTION_MODE_AES_256_CBC).
0x252	__le32	s_reserved[105]	Padding to the end of the block.
0x3FC	__le32	s_checksum	Superblock checksum.

Total size is 1024 bytes.

Block Group Descriptors

Each block group on the filesystem has one of these descriptors associated with it. As noted in the Layout section above, the group descriptors (if present) are the second item in the block group. The standard configuration is for each block group to contain a full copy of the block group descriptor table unless the sparse_super feature flag is set.

Notice how the group descriptor records the location of both bitmaps and the inode table (i.e. they can float). This means that within a block group, the only data structures with fixed locations are the superblock and the group descriptor table. The flex_bg mechanism uses this property to group several block groups into a flex group and lay out all of the groups' bitmaps and inode tables into one long run in the first group of the flex group.

If the meta_bg feature flag is set, then several block groups are grouped together into a meta group. Note that in the meta_bg case, however, the first and last two block groups within the larger meta group contain only group descriptors for the groups inside the meta group.

flex_bg and meta_bg do not appear to be mutually exclusive features.

In ext2, ext3, and ext4 (when the 64bit feature is not enabled), the block group descriptor was only 32 bytes long and therefore ends at bg_checksum. On an ext4 filesystem with the 64bit feature enabled, the block group descriptor expands to at least the 64 bytes described below; the size is stored in the superblock.

If gdt_csum is set and metadata_csum is not set, the block group checksum is the crc16 of the FS UUID, the group number, and the group descriptor structure. If metadata_csum is set, then the block group checksum is the lower 16 bits of the checksum of the FS UUID, the group number, and the group descriptor structure. Both block and inode bitmap checksums are calculated against the FS UUID, the group number, and the entire bitmap.

The block group descriptor is laid out in `struct ext4_group_desc`.

Offset	Size	Name	Description
0x0	__le32	bg_block_bitmap_lo	Lower 32-bits of location of block bitmap.
0x4	__le32	bg_inode_bitmap_lo	Lower 32-bits of location of inode bitmap.
0x8	__le32	bg_inode_table_lo	Lower 32-bits of location of inode table.
0xC	__le16	bg_free_blocks_count_lo	Lower 16-bits of free block count.
0xE	__le16	bg_free_inodes_count_lo	Lower 16-bits of free inode count.
0x10	__le16	bg_used_dirs_count_lo	Lower 16-bits of directory count.
0x12	__le16	bg_flags	Block group flags. Any of: 0x1 inode table and bitmap are not initialized (EXT4_BG_INODE_UNINIT). 0x2 block bitmap is not initialized (EXT4_BG_BLOCK_UNINIT). 0x4 inode table is zeroed (EXT4_BG_INODE_ZEROED).
0x14	__le32	bg_exclude_bitmap_lo	Lower 32-bits of location of snapshot exclusion bitmap.
0x18	__le16	bg_block_bitmap_csum_lo	Lower 16-bits of the block bitmap checksum.
0x1A	__le16	bg_inode_bitmap_csum_lo	Lower 16-bits of the inode bitmap checksum.

0x1C	__le16	bg_itable_unused_lo	Lower 16-bits of unused inode count. If set, we needn't scan past the `(sb.s_inodes_per_group - gdt.bg_itable_unused)` th entry in the inode table for this group.
0x1E	__le16	bg_checksum	Group descriptor checksum; crc16(sb_uuid+group+desc) if the RO_COMPAT_GDT_CSUM feature is set, or crc32c(sb_uuid+group_desc) & 0xFFFF if the RO_COMPAT_METADATA_CSUM feature is set.
These fields only exist if the 64bit feature is enabled and s_desc_size > 32.			
0x20	__le32	bg_block_bitmap_hi	Upper 32-bits of location of block bitmap.
0x24	__le32	bg_inode_bitmap_hi	Upper 32-bits of location of inodes bitmap.
0x28	__le32	bg_inode_table_hi	Upper 32-bits of location of inodes table.
0x2C	__le16	bg_free_blocks_count_hi	Upper 16-bits of free block count.
0x2E	__le16	bg_free_inodes_count_hi	Upper 16-bits of free inode count.
0x30	__le16	bg_used_dirs_count_hi	Upper 16-bits of directory count.
0x32	__le16	bg_itable_unused_hi	Upper 16-bits of unused inode count.
0x34	__le32	bg_exclude_bitmap_hi	Upper 32-bits of location of snapshot exclusion bitmap.
0x38	__le16	bg_block_bitmap_csum_hi	Upper 16-bits of the block bitmap checksum.
0x3A	__le16	bg_inode_bitmap_csum_hi	Upper 16-bits of the inode bitmap checksum.
0x3C	__u32	bg_reserved	Padding to 64 bytes.

Total size is 64 bytes.

Block and inode Bitmaps

The data block bitmap tracks the usage of data blocks within the block group.

The inode bitmap records which entries in the inode table are in use.

As with most bitmaps, one bit represents the usage status of one data block or inode table entry. This implies a block group size of 8 * number_of_bytes_in_a_logical_block.

NOTE: If `BLOCK_UNINIT` is set for a given block group, various parts of the kernel and e2fsprogs code pretends that the block bitmap contains zeros (i.e. all blocks in the group are free). However, it is not necessarily the case that no blocks are in use -- if `meta_bg` is set, the bitmaps and group descriptor live inside the group. Unfortunately, ext2fs_test_block_bitmap2() will return '0' for those locations, which produces confusing debugfs output.

Inode Table

In a regular UNIX filesystem, the inode stores all the metadata pertaining to the file (time stamps, block maps, extended attributes, etc), not the directory entry. To find the information associated with a file, one must traverse the directory files to find the directory entry associated with a file, then load the inode to find the metadata for that file. ext4 appears to cheat (for performance reasons) a little bit by storing a copy of the file type (normally stored in the inode) in the directory entry. (Compare all this to FAT, which stores all the file information directly in the directory entry, but does not support hard links and is in general more seek-happy than ext4 due to its simpler block allocator and extensive use of linked lists.)

The inode table is a linear array of `struct ext4_inode`. The table is sized to have enough blocks to store at least `sb.s_inode_size * sb.s_inodes_per_group` bytes. The number of the block group containing an inode can be calculated as `(inode_number - 1) / sb.s_inodes_per_group`, and the offset into the group's table is `(inode_number - 1) % sb.s_inodes_per_group`. There is no inode 0.

The inode checksum is calculated against the FS UUID, the inode number, and the inode structure itself.

The inode table entry is laid out in `struct ext4_inode`.

Offset	Size	Name	Description
0x0	__le16	i_mode	File mode. Any of: 0x1 S_IXOTH (Others may execute) 0x2 S_IWOTH (Others may write) 0x4 S_IROTH (Others may read) 0x8 S_IXGRP (Group members may execute) 0x10 S_IWGRP (Group members may write) 0x20 S_IRGRP (Group members may read) 0x40 S_IXUSR (Owner may execute) 0x80 S_IWUSR (Owner may write) 0x100 S_IRUSR (Owner may read) 0x200 S_ISVTX (Sticky bit) 0x400 S_ISGID (Set GID) 0x800 S_ISUID (Set UID) These are mutually-exclusive file types: 0x1000 S_IFIFO (FIFO) 0x2000 S_IFCHR (Character device) 0x4000 S_IFDIR (Directory) 0x6000 S_IFBLK (Block device) 0x8000 S_IFREG (Regular file) 0xA000 S_IFLNK (Symbolic link) 0xC000 S_IFSOCK (Socket)
0x2	__le16	i_uid	Lower 16-bits of Owner UID.
0x4	__le32	i_size_lo	Lower 32-bits of size in bytes.
0x8	__le32	i_atime	Last access time, in seconds since the epoch.
0xC	__le32	i_ctime	Last inode change time, in seconds since the epoch.
0x10	__le32	i_mtime	Last data modification time, in seconds since the epoch.
0x14	__le32	i_dtime	Deletion Time, in seconds since the epoch.
0x18	__le16	i_gid	Lower 16-bits of GID.
0x1A	__le16	i_links_count	Hard link count.
0x1C	__le32	i_blocks_lo	Lower 32-bits of "block" count. If the huge_file feature flag is not set on the filesystem, the file consumes `i_blocks_lo` 512-byte blocks on disk. If huge_file is set and EXT4_HUGE_FILE_FL is NOT set in `inode.i_flags`, then the file consumes `i_blocks_lo + (i_blocks_hi << 32)` 512-byte blocks on disk. If huge_file is set and EXT4_HUGE_FILE_FL IS set in `inode.i_flags`, then this file consumes (`i_blocks_lo + i_blocks_hi` << 32) filesystem blocks on disk.
0x20	__le32	i_flags	Inode flags. Any of: 0x1 This file requires secure deletion (EXT4_SECRM_FL). (not implemented)

0x2	This file should be preserved, should undeletion be desired (EXT4_UNRM_FL). (not implemented)
0x4	File is compressed (EXT4_COMPR_FL). (not really implemented)
0x8	All writes to the file must be synchronous (EXT4_SYNC_FL).
0x10	File is immutable (EXT4_IMMUTABLE_FL).
0x20	File can only be appended (EXT4_APPEND_FL).
0x40	The dump(1) utility should not dump this file (EXT4_NODUMP_FL).
0x80	Do not update access time (EXT4_NOATIME_FL).
0x100	Dirty compressed file (EXT4_DIRTY_FL). (not used)
0x200	File has one or more compressed clusters (EXT4_COMPRBLK_FL). (not used)
0x400	Do not compress file (EXT4_NOCOMPR_FL). (not used)
0x800	Encrypted inode (EXT4_ENCRYPT_FL). This bit value previously was EXT4_ECOMPR_FL (compression error), which was never used.
0x1000	Directory has hashed indexes (EXT4_INDEX_FL).
0x2000	AFS magic directory (EXT4_IMAGIC_FL).
0x4000	File data must always be written through the journal (EXT4_JOURNAL_DATA_FL).
0x8000	File tail should not be merged (EXT4_NOTAIL_FL). (not used by ext4)
0x10000	All directory entry data should be written synchronously (see `dirsync`) (EXT4_DIRSYNC_FL).
0x20000	Top of directory hierarchy (EXT4_TOPDIR_FL).
0x40000	This is a huge file (EXT4_HUGE_FILE_FL).
0x80000	Inode uses extents (EXT4_EXTENTS_FL).
0x200000	Inode used for a large extended attribute (EXT4_EA_INODE_FL).
0x400000	This file has blocks allocated past EOF (EXT4_EOFBLOCKS_FL). (deprecated)
0x01000000	Inode is a snapshot (`EXT4_SNAPFILE_FL`). (not in mainline)
0x04000000	Snapshot is being deleted (`EXT4_SNAPFILE_DELETED_FL`). (not in mainline)
0x08000000	Snapshot shrink has completed (`EXT4_SNAPFILE_SHRUNK_FL`). (not in mainline)
0x10000000	Inode has inline data (EXT4_INLINE_DATA_FL).
0x80000000	Reserved for ext4 library (EXT4_RESERVED_FL).
Aggregate flags:	
0x4BDFFF	User-visible flags.
0x4B80FF	User-modifiable flags. Note that while EXT4_JOURNAL_DATA_FL and EXT4_EXTENTS_FL can be set with setattr, they are not in the kernel's EXT4_FL_USER_MODIFIABLE mask, since it needs to handle the setting of these flags in a special manner and they are masked

<table>
<tr><td></td><td></td><td></td><td>out of the set of flags that are saved directly to i_flags.</td></tr>
<tr><td>0x24</td><td>4 bytes</td><td colspan="2">Union osd1:
<table>
<tr><th>Tag</th><th>Contents</th></tr>
<tr><td>linux1</td><td>Offset Size Name Description
0x0 __le32 l_i_version Version (High 32-bits of the i_generation field?)</td></tr>
<tr><td>hurd1</td><td>Offset Size Name Description
0x0 __le32 h_i_translator ??</td></tr>
<tr><td>masix1</td><td>Offset Size Name Description
0x0 __le32 m_i_reserved ??</td></tr>
</table></td></tr>
<tr><td>0x28</td><td>60 bytes</td><td>i_block[EXT4_N_BLOCKS=15]</td><td>Block map or extent tree. See the section "The Contents of inode.i_block".</td></tr>
<tr><td>0x64</td><td>__le32</td><td>i_generation</td><td>File version (for NFS).</td></tr>
<tr><td>0x68</td><td>__le32</td><td>i_file_acl_lo</td><td>Lower 32-bits of extended attribute block. ACLs are of course one of many possible extended attributes; I think the name of this field is a result of the first use of extended attributes being for ACLs.</td></tr>
<tr><td>0x6C</td><td>__le32</td><td>i_size_high / i_dir_acl</td><td>Upper 32-bits of file size. In ext2/3 this field was named i_dir_acl, though it was usually set to zero and never used.</td></tr>
<tr><td>0x70</td><td>__le32</td><td>i_obso_faddr</td><td>(Obsolete) fragment address.</td></tr>
<tr><td>0x74</td><td>12 bytes</td><td colspan="2">Union osd2:
<table>
<tr><th>Tag</th><th>Contents</th></tr>
<tr><td>linux2</td><td>Offset Size Name Description
0x0 __le16 l_i_blocks_high Upper 16-bits of the block count. Please see the note attached to i_blocks_lo.
0x2 __le16 l_i_file_acl_high Upper 16-bits of the extended attribute block (historically, the file ACL location). See the Extended Attributes section below.
0x4 __le16 l_i_uid_high Upper 16-bits of the Owner UID.
0x6 __le16 l_i_gid_high Upper 16-bits of the GID.
0x8 __le16 l_i_checksum_lo Lower 16-bits of the inode checksum.
0xA __le16 l_i_reserved Unused.</td></tr>
<tr><td>hurd2</td><td>Offset Size Name Description
0x0 __le16 h_i_reserved1 ??
0x2 __u16 h_i_mode_high Upper 16-bits of the file mode.
0x4 __le16 h_i_uid_high Upper 16-bits of the Owner UID.
0x6 __le16 h_i_gid_high Upper 16-bits of the GID.
0x8 __u32 h_i_author Author code?</td></tr>
<tr><td>masix2</td><td>Offset Size Name Description
0x0 __le16 h_i_reserved1 ??</td></tr>
</table></td></tr>
</table>

			0x2 __u16 m_i_file_acl_high	Upper 16-bits of the extended attribute block (historically, the file ACL location).
			0x4 __u32 m_i_reserved2[2]	??
0x80	__le16	i_extra_isize		Size of this inode - 128. Alternately, the size of the extended inode fields beyond the original ext2 inode, including this field.
0x82	__le16	i_checksum_hi		Upper 16-bits of the inode checksum.
0x84	__le32	i_ctime_extra		Extra change time bits. This provides sub-second precision. See Inode Timestamps section.
0x88	__le32	i_mtime_extra		Extra modification time bits. This provides sub-second precision.
0x8C	__le32	i_atime_extra		Extra access time bits. This provides sub-second precision.
0x90	__le32	i_crtime		File creation time, in seconds since the epoch.
0x94	__le32	i_crtime_extra		Extra file creation time bits. This provides sub-second precision.
0x98	__le32	i_version_hi		Upper 32-bits for version number.

Inode Size

In ext2 and ext3, the inode structure size was fixed at 128 bytes (`EXT2_GOOD_OLD_INODE_SIZE`) and each inode had a disk record size of 128 bytes. Starting with ext4, it is possible to allocate a larger on-disk inode at format time for all inodes in the filesystem to provide space beyond the end of the original ext2 inode. The on-disk inode record size is recorded in the superblock as `s_inode_size`. The number of bytes actually used by struct ext4_inode beyond the original 128-byte ext2 inode is recorded in the `i_extra_isize` field for each inode, which allows struct ext4_inode to grow for a new kernel without having to upgrade all of the on-disk inodes. Access to fields beyond EXT2_GOOD_OLD_INODE_SIZE should be verified to be within `i_extra_isize`. By default, ext4 inode records are 256 bytes, and (as of October 2013) the inode structure is 156 bytes (`i_extra_isize = 28`). The extra space between the end of the inode structure and the end of the inode record can be used to store extended attributes. Each inode record can be as large as the filesystem block size, though this is not terribly efficient.

Finding an Inode

Each block group contains `sb->s_inodes_per_group` inodes. Because inode 0 is defined not to exist, this formula can be used to find the block group that an inode lives in: `bg = (inode_num - 1) / sb->s_inodes_per_group`. The particular inode can be found within the block group's inode table at `index = (inode_num - 1) % sb->s_inodes_per_group`. To get the byte address within the inode table, use `offset = index * sb->s_inode_size`.

Inode Timestamps

Four timestamps are recorded in the lower 128 bytes of the inode structure -- inode change time (ctime), access time (atime), data modification time (mtime), and deletion time (dtime). The four fields are 32-bit signed integers that represent seconds since the Unix epoch (1970-01-01 00:00:00 GMT), which means that the fields will overflow in January 2038. For inodes that are not linked from any directory but are still open (orphan inodes), the dtime field is overloaded for use with the orphan list. The superblock field `s_last_orphan` points to the first inode in the orphan list; dtime is then the number of the next orphaned inode, or zero if there are no more orphans.

If the inode structure size `sb->s_inode_size` is larger than 128 bytes and the `i_inode_extra` field is large enough to encompass the respective `i_[cma]time_extra` field, the ctime, atime, and mtime inode fields are widened to 64 bits. Within this "extra" 32-bit field, the lower two bits are used to extend the 32-bit seconds field to be 34 bit wide; the upper 30 bits are used to provide nanosecond timestamp accuracy. Therefore, timestamps should not overflow until May 2446. dtime was not widened. There is also a fifth timestamp to record inode creation time (crtime); this field is 64-bits wide and decoded in the same manner as 64-bit [cma]time. Neither crtime nor dtime are accessible through the regular stat() interface, though debugfs will report them.

We use the 32-bit signed time value plus (2^32 * (extra epoch bits)). In other words:

Extra epoch bits	MSB of 32-bit time	Adjustment for signed 32-bit to 64-bit tv_sec	Decoded 64-bit tv_sec	valid time range
0 0	1	0	-0x80000000 - -0x00000001	1901-12-13 to 1969-12-31
0 0	0	0	0x000000000 - 0x07fffffff	1970-01-01 to 2038-01-19
0 1	1	0x100000000	0x080000000 - 0x0ffffffff	2038-01-19 to 2106-02-07
0 1	0	0x100000000	0x100000000 - 0x17fffffff	2106-02-07 to 2174-02-25
1 0	1	0x200000000	0x180000000 - 0x1ffffffff	2174-02-25 to 2242-03-16
1 0	0	0x200000000	0x200000000 - 0x27fffffff	2242-03-16 to 2310-04-04
1 1	1	0x300000000	0x280000000 - 0x2ffffffff	2310-04-04 to 2378-04-22
1 1	0	0x300000000	0x300000000 - 0x37fffffff	2378-04-22 to 2446-05-10

This is a somewhat odd encoding since there are effectively seven times as many positive values as negative values. There have also been long-standing bugs decoding and encoding dates beyond 2038, which don't seem to be fixed as of kernel 3.12 and e2fsprogs 1.42.8. 64-bit kernels incorrectly use the extra epoch bits 1,1 for dates between 1901 and 1970. At some point the kernel will be fixed and e2fsck will fix this situation, assuming that it is run before 2310.

The Contents of inode.i_block

Depending on the type of file an inode describes, the 60 bytes of storage in `inode.i_block` can be used in different ways. In general, regular files and directories will use it for file block indexing information, and special files will use it for special purposes.

Symbolic Links

The target of a symbolic link will be stored in this field if the target string is less than 60 bytes long. Otherwise, either extents or block maps will be used to allocate data blocks to store the link target.

Direct/Indirect Block Addressing

In ext2/3, file block numbers were mapped to logical block numbers by means of an (up to) three level 1-1 block map. To find the logical block that stores a particular file block, the code would navigate through this increasingly complicated structure. Notice that there is neither a magic number nor a checksum to provide any level of confidence that the block isn't full of garbage.

i.i_block Offset	Where It Points
0 to 11	Direct map to file blocks 0 to 11.
12	Indirect block: (file blocks 12 to (`$block_size` / 4) + 11, or 12 to 1035 if 4KiB blocks)

Indirect Block Offset	Where It Points

<table>
<tr><td></td><td>
<table>
<tr><td>0 to ($block_size / 4)</td><td>Direct map to ($block_size / 4) blocks (1024 if 4KiB blocks)</td></tr>
</table>
</td></tr>
<tr><td>13</td><td>Double-indirect block: (file blocks $block_size/4 + 12 to ($block_size / 4) ^ 2 + ($block_size / 4) + 11, or 1036 to 1049611 if 4KiB blocks)
<table>
<tr><th>Double Indirect Block Offset</th><th>Where It Points</th></tr>
<tr><td>0 to ($block_size / 4)</td><td>Map to ($block_size / 4) indirect blocks (1024 if 4KiB blocks)
<table>
<tr><th>Indirect Block Offset</th><th>Where It Points</th></tr>
<tr><td>0 to ($block_size / 4)</td><td>Direct map to ($block_size / 4) blocks (1024 if 4KiB blocks)</td></tr>
</table>
</td></tr>
</table>
</td></tr>
<tr><td>14</td><td>Triple-indirect block: (file blocks ($block_size / 4) ^ 2 + ($block_size / 4) + 12 to ($block_size / 4) ^ 3 + ($block_size / 4) ^ 2 + ($block_size / 4) + 12, or 1049612 to 1074791436 if 4KiB blocks)
<table>
<tr><th>Triple Indirect Block Offset</th><th>Where It Points</th></tr>
<tr><td>0 to ($block_size / 4)</td><td>Map to ($block_size / 4) double indirect blocks (1024 if 4KiB blocks)
<table>
<tr><th>Double Indirect Block Offset</th><th>Where It Points</th></tr>
<tr><td>0 to ($block_size / 4)</td><td>Map to ($block_size / 4) indirect blocks (1024 if 4KiB blocks)
<table>
<tr><th>Indirect Block Offset</th><th>Where It Points</th></tr>
<tr><td>0 to ($block_size / 4)</td><td>Direct map to ($block_size / 4) blocks (1024 if 4KiB blocks)</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>

Note that with this block mapping scheme, it is necessary to fill out a lot of mapping data even for a large contiguous file! This inefficiency led to the creation of the extent mapping scheme, discussed below.

Notice also that a file using this mapping scheme cannot be placed higher than 2^32 blocks.

Extent Tree

In ext4, the file to logical block map has been replaced with an extent tree. Under the old scheme, allocating a contiguous run of 1,000 blocks requires an indirect block to map all 1,000 entries; with extents, the mapping is reduced to a single `struct ext4_extent` with `ee_len = 1000`. If flex_bg is enabled, it is possible to allocate very large files with a single extent, at a considerable reduction in metadata block use, and some improvement in disk efficiency. The inode must have the extents flag (0x80000) flag set for this feature to be in use.

Extents are arranged as a tree. Each node of the tree begins with a `struct ext4_extent_header`. If the node is an interior node (`eh.eh_depth` > 0), the header is followed by `eh.eh_entries` instances of `struct ext4_extent_idx`; each of these index entries points to a block containing more nodes in the extent tree. If the node is a leaf node (`eh.eh_depth == 0`), then the header is followed by `eh.eh_entries` instances of `struct ext4_extent`; these instances point to the file's data blocks. The root node of the extent tree is stored in `inode.i_block`, which allows for the first four extents to be recorded without the use of extra metadata blocks.

The extent tree header is recorded in `struct ext4_extent_header`, which is 12 bytes long:

Offset	Size	Name	Description
0x0	__le16	eh_magic	Magic number, 0xF30A.

0x2	__le16	eh_entries	Number of valid entries following the header.
0x4	__le16	eh_max	Maximum number of entries that could follow the header.
0x6	__le16	eh_depth	Depth of this extent node in the extent tree. 0 = this extent node points to data blocks; otherwise, this extent node points to other extent nodes. The extent tree can be at most 5 levels deep: a logical block number can be at most `2^32`, and the smallest `n` that satisfies `4*(((blocksize - 12)/12)^n) >= 2^32` is 5.
0x8	__le32	eh_generation	Generation of the tree. (Used by Lustre, but not standard ext4).

Internal nodes of the extent tree, also known as index nodes, are recorded as `struct ext4_extent_idx`, and are 12 bytes long:

Offset	Size	Name	Description
0x0	__le32	ei_block	This index node covers file blocks from 'block' onward.
0x4	__le32	ei_leaf_lo	Lower 32-bits of the block number of the extent node that is the next level lower in the tree. The tree node pointed to can be either another internal node or a leaf node, described below.
0x8	__le16	ei_leaf_hi	Upper 16-bits of the previous field.
0xA	__u16	ei_unused	

Leaf nodes of the extent tree are recorded as `struct ext4_extent`, and are also 12 bytes long:

Offset	Size	Name	Description
0x0	__le32	ee_block	First file block number that this extent covers.
0x4	__le16	ee_len	Number of blocks covered by extent. If the value of this field is <= 32768, the extent is initialized. If the value of the field is > 32768, the extent is uninitialized and the actual extent length is `ee_len` - 32768. Therefore, the maximum length of a initialized extent is 32768 blocks, and the maximum length of an uninitialized extent is 32767.
0x6	__le16	ee_start_hi	Upper 16-bits of the block number to which this extent points.
0x8	__le32	ee_start_lo	Lower 32-bits of the block number to which this extent points.

Prior to the introduction of metadata checksums, the extent header + extent entries always left at least 4 bytes of unallocated space at the end of each extent tree data block (because (2^x % 12) >= 4). Therefore, the 32-bit checksum is inserted into this space. The 4 extents in the inode do not need checksumming, since the inode is already checksummed. The checksum is calculated against the FS UUID, the inode number, the inode generation, and the entire extent block leading up to (but not including) the checksum itself.

`struct ext4_extent_tail` is 4 bytes long:

Offset	Size	Name	Description
0x0	__le32	eb_checksum	Checksum of the extent block, crc32c(uuid+inum+igeneration+extentblock)

Inline Data

If the inline data feature is enabled for the filesystem and the flag is set for the inode, it is possible that the first 60 bytes of the file data are stored here.

Directory Entries

In an ext4 filesystem, a directory is more or less a flat file that maps an arbitrary byte string (usually ASCII) to an inode number on

the filesystem. There can be many directory entries across the filesystem that reference the same inode number--these are known as hard links, and that is why hard links cannot reference files on other filesystems. As such, directory entries are found by reading the data block(s) associated with a directory file for the particular directory entry that is desired.

Linear (Classic) Directories

By default, each directory lists its entries in an "almost-linear" array. I write "almost" because it's not a linear array in the memory sense because directory entries are not split across filesystem blocks. Therefore, it is more accurate to say that a directory is a series of data blocks and that each block contains a linear array of directory entries. The end of each per-block array is signified by reaching the end of the block; the last entry in the block has a record length that takes it all the way to the end of the block. The end of the entire directory is of course signified by reaching the end of the file. Unused directory entries are signified by inode = 0. By default the filesystem uses `struct ext4_dir_entry_2` for directory entries unless the "filetype" feature flag is not set, in which case it uses `struct ext4_dir_entry`.

The original directory entry format is `struct ext4_dir_entry`, which is at most 263 bytes long, though on disk you'll need to reference `dirent.rec_len` to know for sure.

Offset	Size	Name	Description
0x0	__le32	inode	Number of the inode that this directory entry points to.
0x4	__le16	rec_len	Length of this directory entry.
0x6	__le16	name_len	Length of the file name.
0x8	char	name[EXT4_NAME_LEN]	File name.

Since file names cannot be longer than 255 bytes, the new directory entry format shortens the rec_len field and uses the space for a file type flag, probably to avoid having to load every inode during directory tree traversal. This format is `ext4_dir_entry_2`, which is at most 263 bytes long, though on disk you'll need to reference `dirent.rec_len` to know for sure.

Offset	Size	Name	Description
0x0	__le32	inode	Number of the inode that this directory entry points to.
0x4	__le16	rec_len	Length of this directory entry.
0x6	__u8	name_len	Length of the file name.
0x7	__u8	file_type	File type code, one of: 0x0 Unknown. 0x1 Regular file. 0x2 Directory. 0x3 Character device file. 0x4 Block device file. 0x5 FIFO. 0x6 Socket. 0x7 Symbolic link.
0x8	char	name[EXT4_NAME_LEN]	File name.

In order to add checksums to these classic directory blocks, a phony `struct ext4_dir_entry` is placed at the end of each leaf block to hold the checksum. The directory entry is 12 bytes long. The inode number and name_len fields are set to zero to fool old software into ignoring an apparently empty directory entry, and the checksum is stored in the place where the name normally goes. The structure is `struct ext4_dir_entry_tail`:

Offset	Size	Name	Description

0x0	__le32	det_reserved_zero1	Inode number, which must be zero.
0x4	__le16	det_rec_len	Length of this directory entry, which must be 12.
0x6	__u8	det_reserved_zero2	Length of the file name, which must be zero.
0x7	__u8	det_reserved_ft	File type, which must be 0xDE.
0x8	__le32	det_checksum	Directory leaf block checksum.

The leaf directory block checksum is calculated against the FS UUID, the directory's inode number, the directory's inode generation number, and the entire directory entry block up to (but not including) the fake directory entry.

Hash Tree Directories

A linear array of directory entries isn't great for performance, so a new feature was added to ext3 to provide a faster (but peculiar) balanced tree keyed off a hash of the directory entry name. If the EXT4_INDEX_FL (0x1000) flag is set in the inode, this directory uses a hashed btree (htree) to organize and find directory entries. For backwards read-only compatibility with ext2, this tree is actually hidden inside the directory file, masquerading as "empty" directory data blocks! It was stated previously that the end of the linear directory entry table was signified with an entry pointing to inode 0; this is (ab)used to fool the old linear-scan algorithm into thinking that the rest of the directory block is empty so that it moves on.

The root of the tree always lives in the first data block of the directory. By ext2 custom, the '.' and '..' entries must appear at the beginning of this first block, so they are put here as two `struct ext4_dir_entry_2`s and not stored in the tree. The rest of the root node contains metadata about the tree and finally a hash->block map to find nodes that are lower in the htree. If `dx_root.info.indirect_levels` is non-zero then the htree has two levels; the data block pointed to by the root node's map is an interior node, which is indexed by a minor hash. Interior nodes in this tree contains a zeroed out `struct ext4_dir_entry_2` followed by a minor_hash->block map to find leafe nodes. Leaf nodes contain a linear array of all `struct ext4_dir_entry_2`; all of these entries (presumably) hash to the same value. If there is an overflow, the entries simply overflow into the next leaf node, and the least-significant bit of the hash (in the interior node map) that gets us to this next leaf node is set.

To traverse the directory as a htree, the code calculates the hash of the desired file name and uses it to find the corresponding block number. If the tree is flat, the block is a linear array of directory entries that can be searched; otherwise, the minor hash of the file name is computed and used against this second block to find the corresponding third block number. That third block number will be a linear array of directory entries.

To traverse the directory as a linear array (such as the old code does), the code simply reads every data block in the directory. The blocks used for the htree will appear to have no entries (aside from '.' and '..') and so only the leaf nodes will appear to have any interesting content.

The root of the htree is in `struct dx_root`, which is the full length of a data block:

Offset	Type	Name	Description
0x0	__le32	dot.inode	inode number of this directory.
0x4	__le16	dot.rec_len	Length of this record, 12.
0x6	u8	dot.name_len	Length of the name, 1.
0x7	u8	dot.file_type	File type of this entry, 0x2 (directory) (if the feature flag is set).
0x8	char	dot.name[4]	".\0\0\0"
0xC	__le32	dotdot.inode	inode number of parent directory.
0x10	__le16	dotdot.rec_len	block_size - 12. The record length is long enough to cover all htree data.
0x12	u8	dotdot.name_len	Length of the name, 2.
0x13	u8	dotdot.file_type	File type of this entry, 0x2 (directory) (if the feature flag is set).

0x14	char	dotdot_name[4]	"..\0\0"
0x18	__le32	struct dx_root_info.reserved_zero	Zero.
0x1C	u8	struct dx_root_info.hash_version	Hash version, one of: 0x0 Legacy. 0x1 Half MD4. 0x2 Tea. 0x3 Legacy, unsigned. 0x4 Half MD4, unsigned. 0x5 Tea, unsigned.
0x1D	u8	struct dx_root_info.info_length	Length of the tree information, 0x8.
0x1E	u8	struct dx_root_info.indirect_levels	Depth of the htree.
0x1F	u8	struct dx_root_info.unused_flags	
0x20	__le16	limit	Maximum number of dx_entries that can follow this header.
0x22	__le16	count	Actual number of dx_entries that follow this header.
0x24	__le32	block	The block number (within the directory file) that goes with hash=0.
0x28	struct dx_entry	entries[0]	As many 8-byte `struct dx_entry` as fits in the rest of the data block.

Interior nodes of an htree are recorded as `struct dx_node`, which is also the full length of a data block:

Offset	Type	Name	Description
0x0	__le32	fake.inode	Zero, to make it look like this entry is not in use.
0x4	__le16	fake.rec_len	The size of the block, in order to hide all of the dx_node data.
0x6	u8	name_len	Zero. There is no name for this "unused" directory entry.
0x7	u8	file_type	Zero. There is no file type for this "unused" directory entry.
0x8	__le16	limit	Maximum number of dx_entries that can follow this header.
0xA	__le16	count	Actual number of dx_entries that follow this header.
0xE	__le32	block	The block number (within the directory file) that goes with the lowest hash value of this block. This value is stored in the parent block.
0x12	struct dx_entry	entries[0]	As many 8-byte `struct dx_entry` as fits in the rest of the data block.

The hash maps that exist in both `struct dx_root` and `struct dx_node` are recorded as `struct dx_entry`, which is 8 bytes long:

Offset	Type	Name	Description
0x0	__le32	hash	Hash code.
0x4	__le32	block	Block number (within the directory file, not filesystem blocks) of the next node in the htree.

(If you think this is all quite clever and peculiar, so does the author.)

If metadata checksums are enabled, the last 8 bytes of the directory block (precisely the length of one dx_entry) are used to store a `struct dx_tail`, which contains the checksum. The `limit` and `count` entries in the dx_root/dx_node structures are adjusted as necessary to fit the dx_tail into the block. If there is no space for the dx_tail, the user is notified to run e2fsck -D to rebuild the

directory index (which will ensure that there's space for the checksum. The dx_tail structure is 8 bytes long and looks like this:

Offset	Type	Name	Description
0x0	u32	dt_reserved	
0x4	__le32	dt_checksum	Checksum of the htree directory block.

The checksum is calculated against the FS UUID, the htree index header (dx_root or dx_node), all of the htree indices (dx_entry) that are in use, and the tail block (dx_tail).

Extended Attributes

Extended attributes (xattrs) are typically stored in a separate data block on the disk and referenced from inodes via `inode.i_file_acl*`. The first use of extended attributes seems to have been for storing file ACLs and other security data (selinux). With the `user_xattr` mount option it is possible for users to store extended attributes so long as all attribute names begin with "user"; this restriction seems to have disappeared as of Linux 3.0.

There are two places where extended attributes can be found. The first place is between the end of each inode entry and the beginning of the next inode entry. For example, if inode.i_extra_isize = 28 and sb.inode_size = 256, then there are 256 - (128 + 28) = 100 bytes available for in-inode extended attribute storage. The second place where extended attributes can be found is in the block pointed to by `inode.i_file_acl`. As of Linux 3.11, it is not possible for this block to contain a pointer to a second extended attribute block (or even the remaining blocks of a cluster). In theory it is possible for each attribute's value to be stored in a separate data block, though as of Linux 3.11 the code does not permit this.

Keys are generally assumed to be ASCIIZ strings, whereas values can be strings or binary data.

Extended attributes, when stored after the inode, have a header `ext4_xattr_ibody_header` that is 4 bytes long:

Offset	Type	Name	Description
0x0	__le32	h_magic	Magic number for identification, 0xEA020000. This value is set by the Linux driver, though e2fsprogs doesn't seem to check it(?)

The beginning of an extended attribute block is in `struct ext4_xattr_header`, which is 32 bytes long:

Offset	Type	Name	Description
0x0	__le32	h_magic	Magic number for identification, 0xEA020000.
0x4	__le32	h_refcount	Reference count.
0x8	__le32	h_blocks	Number of disk blocks used.
0xC	__le32	h_hash	Hash value of all attributes.
0x10	__le32	h_checksum	Checksum of the extended attribute block.
0x14	__u32	h_reserved[2]	

The checksum is calculated against the FS UUID, the 64-bit block number of the extended attribute block, and the entire block (header + entries).

Following the `struct ext4_xattr_header` or `struct ext4_xattr_ibody_header` is an array of `struct ext4_xattr_entry`; each of these entries is at least 16 bytes long.

Offset	Type	Name	Description
0x0	__u8	e_name_len	Length of name.

0x1	__u8	e_name_index	Attribute name index. There is a discussion of this below.
0x2	__le16	e_value_offs	Location of this attribute's value on the disk block where it is stored. Multiple attributes can share the same value. For an inode attribute this value is relative to the start of the first entry; for a block this value is relative to the start of the block (i.e. the header).
0x4	__le32	e_value_block	The disk block where the value is stored. Zero indicates the value is in the same block as this entry. As of August 2013, this feature does not seem to be implemented.
0x8	__le32	e_value_size	Length of attribute value.
0xC	__le32	e_hash	Hash value of attribute name and attribute value. The kernel doesn't update the hash for in-inode attributes, so for that case this value must be zero, because e2fsck validates any non-zero hash regardless of where the xattr lives.
0x10	char	e_name[e_name_len]	Attribute name. Does not include trailing NULL.

Attribute values can follow the end of the entry table. There appears to be a requirement that they be aligned to 4-byte boundaries. The values are stored starting at the end of the block and grow towards the xattr_header/xattr_entry table. When the two collide, the overflow is put into a separate disk block. If the disk block fills up, the filesystem returns -ENOSPC.

The first four fields of the `ext4_xattr_entry` are set to zero to mark the end of the key list.

Attribute Name Indices

Logically speaking, extended attributes are a series of key=value pairs. The keys are assumed to be NULL-terminated strings. To reduce the amount of on-disk space that the keys consume, the beginning of the key string is matched against the attribute name index. If a match is found, the attribute name index field is set, and matching string is removed from the key name. Here is a map of name index values to key prefixes:

Name Index	Key Prefix
0	(no prefix)
1	"user."
2	"system.posix_acl_access"
3	"system.posix_acl_default"
4	"trusted."
6	"security."
7	"system." (inline_data only?)
8	"system.richacl" (SuSE kernels only?)

For example, if the attribute key is "user.fubar", the attribute name index is set to 1 and the "fubar" name is recorded on disk.

POSIX ACLs

POSIX ACLs are stored in a reduced version of the Linux kernel (and libacl's) internal ACL format. The key difference is that the version number is different (1) and the `e_id` field is only stored for named user and group ACLs.

Multiple Mount Protection

Multiple mount protection (MMP) is a feature that protects the filesystem against multiple hosts trying to use the filesystem simultaneously. When a filesystem is opened (for mounting, or fsck, etc.), the MMP code running on the node (call it node A) checks a sequence number. If the sequence number is EXT4_MMP_SEQ_CLEAN, the open continues. If the sequence number is EXT4_MMP_SEQ_FSCK, then fsck is (hopefully) running, and open fails immediately. Otherwise, the open code will wait for

twice the specified MMP check interval and check the sequence number again. If the sequence number has changed, then the filesystem is active on another machine and the open fails. If the MMP code passes all of those checks, a new MMP sequence number is generated and written to the MMP block, and the mount proceeds.

While the filesystem is live, the kernel sets up a timer to re-check the MMP block at the specified MMP check interval. To perform the re-check, the MMP sequence number is re-read; if it does not match the in-memory MMP sequence number, then another node (node B) has mounted the filesystem, and node A remounts the filesystem read-only. If the sequence numbers match, the sequence number is incremented both in memory and on disk, and the re-check is complete.

The hostname and device filename are written into the MMP block whenever an open operation succeeds. The MMP code does not use these values; they are provided purely for informational purposes.

The checksum is calculated against the FS UUID and the MMP structure. The MMP structure (`struct mmp_struct`) is as follows:

Offset	Type	Name	Description
0x0	__le32	mmp_magic	Magic number for MMP, 0x004D4D50 ("MMP").
0x4	__le32	mmp_seq	Sequence number, updated periodically.
0x8	__le64	mmp_time	Time that the MMP block was last updated.
0x10	char[64]	mmp_nodename	Hostname of the node that opened the filesystem.
0x50	char[32]	mmp_bdevname	Block device name of the filesystem.
0x70	__le16	mmp_check_interval	The MMP re-check interval, in seconds.
0x72	__le16	mmp_pad1	
0x74	__le32[226]	mmp_pad2	
0x3FC	__le32	mmp_checksum	Checksum of the MMP block.

Journal (jbd2)

Introduced in ext3, the ext4 filesystem employs a journal to protect the filesystem against corruption in the case of a system crash. A small continuous region of disk (default 128MiB) is 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.

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. One of: 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.
0x8	__be32	h_sequence	The transaction ID that goes with this block.

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
0x0	journal_header_t (12 bytes)	s_header	Common header identifying this as a superblock.

Static information describing the journal.			
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 version 2 superblock.			
0x24	__be32	s_feature_compat;	Compatible feature set. Any of: 0x1 Journal maintains checksums on the data blocks. (JBD2_FEATURE_COMPAT_CHECKSUM)
0x28	__be32	s_feature_incompat	Incompatible feature set. Any of: 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) 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)
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. 1 = crc32, 2 = md5, 3 = sha1, 4 = crc32c. 1 or 4 are the most likely choices.
0x51	__u8[3]	s_padding2	
0x54	__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.

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. Any of: 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.
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.

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. Any of: 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.
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.

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. One of: 1 CRC32 2 MD5 3 SHA1 4 CRC32C
0xD	unsigned char	h_chksum_size	The number of bytes used by the checksum. Most likely 4.
0xE	unsigned char	h_padding[2]	
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.

Areas in Need of Work

New patchsets to track with regards to changes in on-disk formats (in no particular order):

- Amir's ext4 snapshot work (dead as of Oct. 2013?)

Other References

Also see http://www.nongnu.org/ext2-doc/ for quite a collection of information about ext2/3. Here's another old reference: http://wiki.osdev.org/Ext2

Retrieved from "https://ext4.wiki.kernel.org/index.php?title=Ext4_Disk_Layout&oldid=9009"

- This page was last modified on 22 May 2015, at 18:06.