The Ext2/Ext3 File Systems

Michael Dippery

CSCI 780

November 11, 2008

Why Ext2?

Ext2 is the *Second Extended File System*.

1. Linux originally used the Minix file system, but the Minix FS had some major limitations.
2. In 1992, the Minix FS was replaced with the *Extended File System*, but Ext was only an incremental improvement.
3. 1993 saw the release of two new file systems: Xia and the *Second Extended File System*. Xia was based on Minix and had only minor improvements; Ext2, on the other hand, had a lot of new features and was designed with evolution in mind.

Features

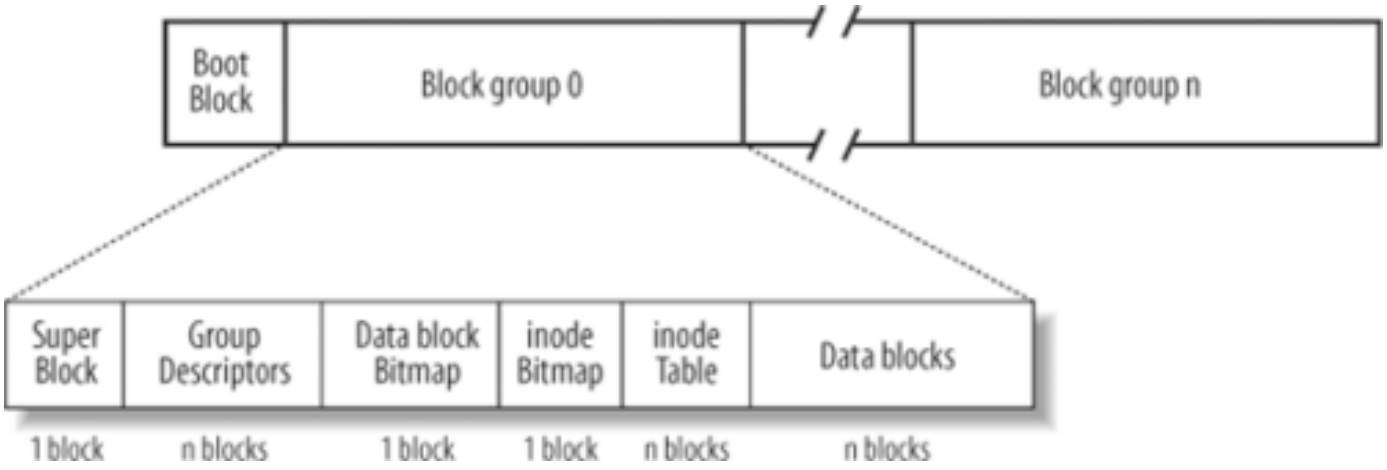
Ext2 supports the following features:

1. Variable block sizes
2. Variable number of inodes per partition
3. Disk blocks mounted into groups
4. Data blocks are preallocated to regular files
5. Fast symbolic links
6. Careful implementation of file-updating operations
7. Support for automatic consistency checks
8. Support for immutable and append-only files
9. Ability to reserve some blocks for the root user

Ext2 Disk Blocks

- Ext2 divides partitions into *block groups*, which contain *blocks* (and some other metadata).
- All block groups are the same size, and are stored sequentially on disk. Think of this like a C array: “indexing” into the list of block groups is an easy operation.
- The kernel tries to keep all of a file’s data blocks in one block group, which helps to reduce file fragmentation.

Block Groups



A block group contains:

- A copy of the file system's super block
- A copy of the block group descriptors
- A data block bitmap
- An inode bitmap
- An inode table
- Data (file) blocks

Block Groups

The number of block groups depends on the *partition size* and the *block size*.

- Block bitmap must be stored in a *single* block.
  - In each block group, there can be $8 \times b$ blocks.
- The number of blocks is $\approx \frac{s}{8 \times b}$.

Superblock Structure

linux/ext2_fs.h

```
337 struct ext2_super_block {
338         __le32  s_inodes_count;         /* Inodes count */
339         __le32  s_blocks_count;         /* Blocks count */
340         __le32  s_r_blocks_count;       /* Reserved blocks count */
341         __le32  s_free_blocks_count;    /* Free blocks count */
342         __le32  s_free_inodes_count;    /* Free inodes count */
343         __le32  s_first_data_block;     /* First Data Block */
344         __le32  s_log_block_size;       /* Block size */
345         __le32  s_log_frag_size;        /* Fragment size */
346         __le32  s_blocks_per_group;     /* # Blocks per group */
347         __le32  s_frags_per_group;      /* # Fragments per group */
348         __le32  s_inodes_per_group;     /* # Inodes per group */
349         __le32  s_mtime;                /* Mount time */
350         __le32  s_wtime;                /* Write time */
351         __le16  s_mnt_count;            /* Mount count */
352         __le16  s_max_mnt_count;        /* Maximal mount count */
353         __le16  s_magic;                /* Magic signature */
354         __le16  s_state;                /* File system state */
```

Integrity Checks

Field	Description
s_mnt_cnt	Number of mounts
s_max_mnt_count	Maximum number of mounts before check
s_lastcheck	Timestamp of last integrity check
s_checkinterval	Maximum time between automatic checks
s_state	Unmounted cleanly?

Group Descriptor

Each block group contains a *group descriptor*, which contains metadata about the group in a `struct ext2_group_desc` (defined in `linux/ext2_fs.h`):

```
136 struct ext2_group_desc
137 {
138         __le32  bg_block_bitmap;        /* Blocks bitmap block */
139         __le32  bg_inode_bitmap;        /* Inodes bitmap block */
140         __le32  bg_inode_table;         /* Inodes table block */
141         __le16  bg_free_blocks_count;   /* Free blocks count */
142         __le16  bg_free_inodes_count;   /* Free inodes count */
143         __le16  bg_used_dirs_count;     /* Directories count */
144         __le16  bg_pad;
145         __le32  bg_reserved[3];
146 };
```

Group Descriptor Bitmaps

- Bitmaps are sequences of bits in which each bit corresponds to an inode or datablock. The bit is set to 0 if the block is free, or 1 if it is used.
- Each bitmap must fit into a single block, and the block size must be 1024, 2048, or 4096 bytes; thus, a single bitmap may describe the state of *block size* $\times$ *8* blocks.

Inodes Table

The inodes table is a series of consecutive blocks that contain a static number of inodes.

How many blocks are occupied by the inode table?

$$\text{blocks occupied by inode table} = \frac{\texttt{s\_inodes\_per\_group}}{\text{inodes per block}}$$

Inodes

```
211 struct ext2_inode {
212         __le16  i_mode;         /* File mode */
213         __le16  i_uid;          /* Low 16 bits of Owner Uid */
214         __le32  i_size;         /* Size in bytes */
215         __le32  i_atime;        /* Access time */
216         __le32  i_ctime;        /* Creation time */
217         __le32  i_mtime;        /* Modification time */
218         __le32  i_dtime;        /* Deletion Time */
219         __le16  i_gid;          /* Low 16 bits of Group Id */
220         __le16  i_links_count;  /* Links count */
221         __le32  i_blocks;       /* Blocks count */
222         __le32  i_flags;        /* File flags */

234         __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
235         __le32  i_generation;   /* File version (for NFS) */
236         __le32  i_file_acl;     /* File ACL */
237         __le32  i_dir_acl;      /* Directory ACL */
238         __le32  i_faddr;        /* Fragment address */
```

Inode Numbers

The *inode number* (which must be unique in the VFS model) just acts as a key into block groups and inode tables:

Deriving an Inode Number

$$\text{block group} = \lfloor \frac{13021}{4096} \rfloor = 3$$

$$\text{inode} = 13021 \equiv 733 \pmod{4096}$$

File Size Limitations

An inode's `i_size` field is a 32-bit integer.

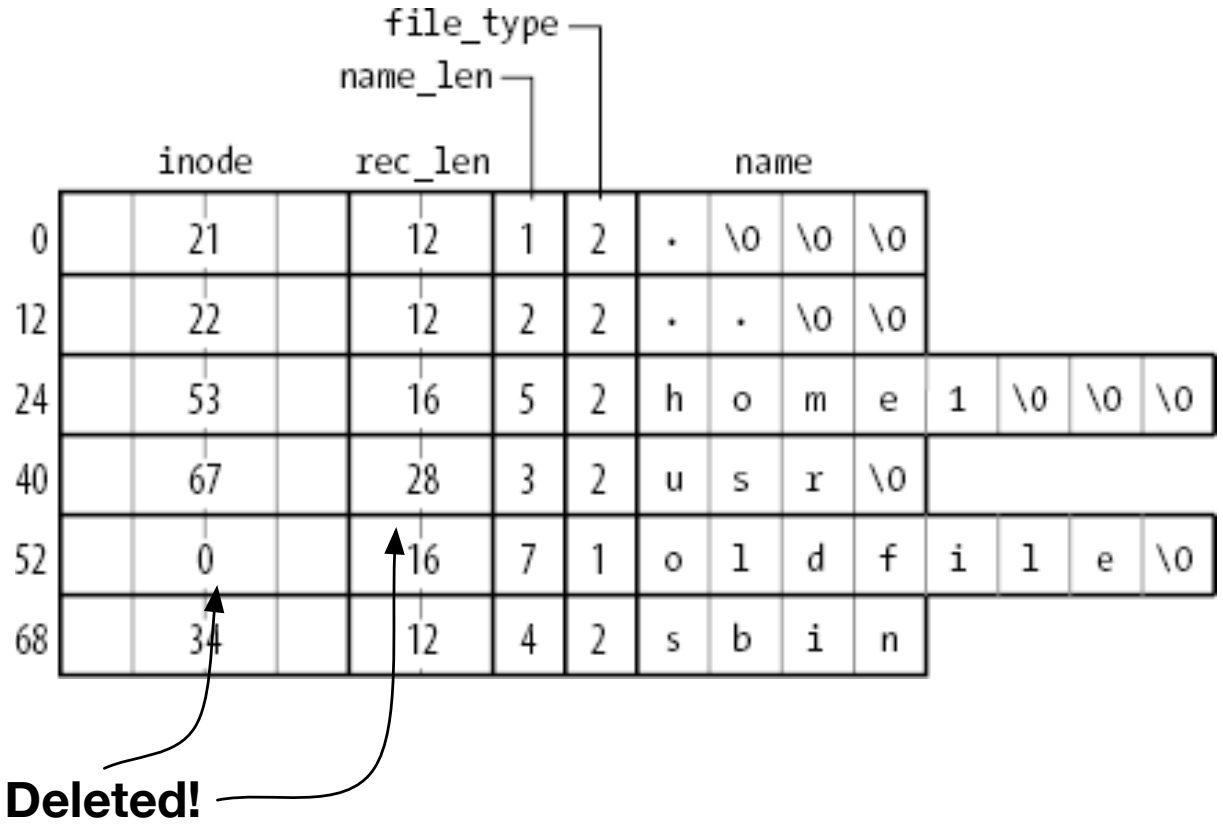
- This limits the file size to 4 GB.
- The highest-order bit is not used, so the maximum file size is really 2 GB.
- On 64-bit systems, the `i_dir_acl` field, which isn't normally used for inodes, is a 32-bit extension of `i_size`. This means the effective maximum file size is *a really big value* (2^{64} bytes).

Data Blocks and Directories

Directories are a special kind of file that store directory information in the file's data blocks.

```
506 struct ext2_dir_entry {
507         __le32  inode;                  /* Inode number */
508         __le16  rec_len;                /* Directory entry length */
509         __le16  name_len;               /* Name length */
510         char    name[EXT2_NAME_LEN];    /* File name */
511 };
```

Marking Files as Deleted



Ext2 and RAM

For efficiency's sake, most of Ext2's data structures are copied into RAM when the file system is mounted.

The kernel utilizes the *page cache* when working with the file system's disk data structures.

Disk-Memory Data Structure Mapping

Type	Disk	Memory
Superblock	`ext2_super_block`	`ext2_sb_info`
Group descriptor	`ext2_group_descriptor`	`ext2_group_desc`
Block bitmap	Bit array in block	Bit array in buffer
Inode bitmap	Bit array in block	Bit array in buffer
Inode	`ext2_inode`	`ext2_inode_info`
Data block	Array of bytes	VFS buffer
Free inode	`ext2_inode`	(none)
Free block	`Array of bytes`	(none)

Creating an Ext2 File System

1. Do a low-level format on the disk so the driver can read the blocks.
2. Creating the necessary disk data structures using the *mke2fs* utility.

Default Values

Attribute	**Default Value**
Block size	1024 bytes
Fragment size	Same as block size
Allocated inodes	1 node / 8192 bytes
Reserved blocks	5%

What Does *mke2fs* Do?

1. Initializes superblock and group descriptors.
2. Checks whether the partition contains “defective” blocks.
3. Reserves all the disk blocks needed to store metadata structures.
4. Zeroes inode bitmap and data bitmap.
5. Initializes inode table.
6. Creates `/` (the root directory).
7. Creates the `lost+found` directory.
8. Updates the inode bitmap and data block bitmap to record `/` and `/lost+found`.
9. Groups any defective blocks in `/lost+found`.

Managing Disk Space

Ext2 tries as best it can to make efficient use of disk space:

- Ext2 attempts to keep related blocks grouped sequentially, to minimize disk access time.
  - “File fragmentation”
- Allows the logical block number to be quickly determined, given a file offset.

Creating Inodes

fs/ext2/ialloc.c:ext2_new_inode()

```
463         sb = dir->i_sb;
464         inode = new_inode(sb);
465         if (!inode)
466                 return ERR_PTR(-ENOMEM);
467
468         ei = EXT2_I(inode);
469         sbi = EXT2_SB(sb);
470         es = sbi->s_es;
471         if (S_ISDIR(mode)) {
472                 if (test_opt(sb, OLDALLOC))
473                         group = find_group_dir(sb, dir);
474                 else
475                         group = find_group_orlov(sb, dir);
476         } else
477                 group = find_group_other(sb, dir);
478
479         if (group == -1) {
480                 if (group == -1) {
481                 goto fail;
482         }
483
484         for (i = 0; i < sbi->s_groups_count; i++) {
485                 gdp = ext2_get_group_desc(sb, group, &bh2);
486                 brelse(bitmap_bh);
487                 bitmap_bh = read_inode_bitmap(sb, group);
```

Creating Inodes

```
494 repeat_in_this_group:
495                 ino = ext2_find_next_zero_bit((unsigned long *)bitmap_bh->b_data,
496                                               EXT2_INODES_PER_GROUP(sb), ino);


530 got:
531         mark_buffer_dirty(bitmap_bh);
532         if (sb->s_flags & MS_SYNCHRONOUS)
533                 sync_dirty_buffer(bitmap_bh);
534         brelse(bitmap_bh);
535
536         ino += group * EXT2_INODES_PER_GROUP(sb) + 1;
537         if (ino < EXT2_FIRST_INO(sb) || ino > le32_to_cpu(es->s_inodes_count)) {
538                 ext2_error (sb, "ext2_new_inode",
539                             "reserved inode or inode > inodes count - "
540                             "block_group = %d,inode=%lu", group,
541                             (unsigned long) ino);
542                 err = -EIO;
543                 goto fail;
544         }
```

Creating Inodes

```
545
546         percpu_counter_mod(&sbi->s_freeinodes_counter, -1);
547         if (S_ISDIR(mode))
548                 percpu_counter_inc(&sbi->s_dirs_counter);
549
550         spin_lock(sb_bgl_lock(sbi, group));
551         gdp->bg_free_inodes_count =
552                 cpu_to_le16(le16_to_cpu(gdp->bg_free_inodes_count) - 1);
553         if (S_ISDIR(mode)) {
554                 if (sbi->s_debts[group] < 255)
555                         sbi->s_debts[group]++;
556                 gdp->bg_used_dirs_count =
557                         cpu_to_le16(le16_to_cpu(gdp->bg_used_dirs_count) + 1);
558         } else {
559                 if (sbi->s_debts[group])
560                         sbi->s_debts[group]--;
561         }
562         spin_unlock(sb_bgl_lock(sbi, group));
563
564         sb->s_dirt = 1;
565         mark_buffer_dirty(bh2);
```

Creating Inodes

```
605         insert_inode_hash(inode);
606
607         if (DQUOT_ALLOC_INODE(inode)) {
608                 DQUOT_DROP(inode);
609                 err = -ENOSPC;
610                 goto fail2;
611         }
612         err = ext2_init_acl(inode, dir);
613         if (err) {
614                 DQUOT_FREE_INODE(inode);
615                 goto fail2;
616         }
617         mark_inode_dirty(inode);
618         ext2_debug("allocating inode %lu\n", inode->i_ino);
619         ext2_preread_inode(inode);
620         return inode;
```

Deleting Inodes

fs/ext2/ialloc.c:ext2_free_inode()

```
134         /* Do this BEFORE marking the inode not in use or returning an error */
135         clear_inode (inode);
136
137         if (ino < EXT2_FIRST_INO(sb) ||
138             ino > le32_to_cpu(es->s_inodes_count)) {
139                 ext2_error (sb, "ext2_free_inode",
140                             "reserved or nonexistent inode %lu", ino);
141                 goto error_return;
142         }
143         block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
144         bit = (ino - 1) % EXT2_INODES_PER_GROUP(sb);
145         brelse(bitmap_bh);
146         bitmap_bh = read_inode_bitmap(sb, block_group);
147         if (!bitmap_bh)
148                 goto error_return;
149
150         /* Ok, now we can actually update the inode bitmaps.. */
151         if (!ext2_clear_bit_atomic(sb_bgl_lock(EXT2_SB(sb), block_group),
152                                    bit, (void *) bitmap_bh->b_data))
153                 ext2_error (sb, "ext2_free_inode",
154                             "bit already cleared for inode %lu", ino);
155         else
156                 ext2_release_inode(sb, block_group, is_directory);
157         mark_buffer_dirty(bitmap_bh);
158         if (sb->s_flags & MS_SYNCHRONOUS)
159                 sync_dirty_buffer(bitmap_bh);
```

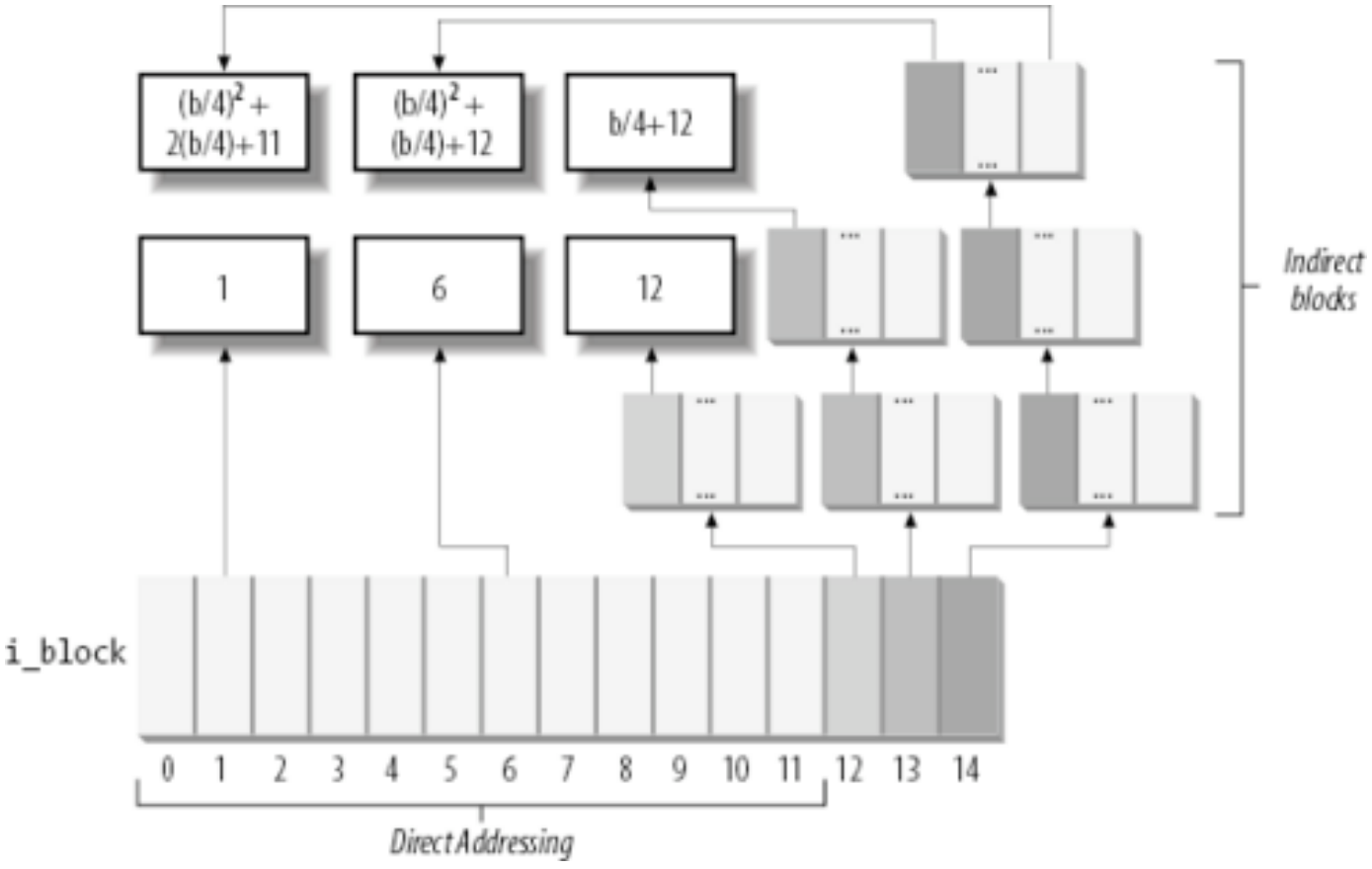
Deriving the Logical Block Number

To retrieve a file, the logical block number must be derived from the file offset f in a two-step process:

1. Derive from f the file block number.
2. Translate the file block number into the logical block number.

$$\text{file block number} = \lfloor \frac{f}{\text{block size}} \rfloor$$

Translating File Block Number



File Size Limits

Size Limitations as a Function of Block Size

Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1024	12 KB	268 KB	64.26 MB	16.06 GB
2048	24 KB	1.02 MB	513.02 MB	256.5 GB
4096	48 KB	4.04 MB	4 GB	$\approx$ 4 TB

Allocating a Data Block

fs/ext2/inode.c:ext2_alloc_block()

```
 94 static int ext2_alloc_block (struct inode * inode, unsigned long goal, int *err)
 95 {
 96 #ifdef EXT2FS_DEBUG
 97         static unsigned long alloc_hits, alloc_attempts;
 98 #endif
 99         unsigned long result;
100
101
102 #ifdef EXT2_PREALLOCATE
103         struct ext2_inode_info *ei = EXT2_I(inode);
104         write_lock(&ei->i_meta_lock);
105         if (ei->i_prealloc_count &&
106             (goal == ei->i_prealloc_block || goal + 1 == ei->i_prealloc_block))
107         {
108                 result = ei->i_prealloc_block++;
109                 ei->i_prealloc_count--;
110                 write_unlock(&ei->i_meta_lock);
111                 ext2_debug ("preallocation hit (%lu/%lu).\n",
112                             ++alloc_hits, ++alloc_attempts);
```

Allocating a Data Block

```
113         } else {
114                 write_unlock(&ei->i_meta_lock);
115                 ext2_discard_prealloc (inode);
116                 ext2_debug ("preallocation miss (%lu/%lu).\n",
117                             alloc_hits, ++alloc_attempts);
118                 if (S_ISREG(inode->i_mode))
119                         result = ext2_new_block (inode, goal,
120                                  &ei->i_prealloc_count,
121                                  &ei->i_prealloc_block, err);
122                 else
123                         result = ext2_new_block(inode, goal, NULL, NULL, err);
124         }
125 #else
126         result = ext2_new_block (inode, goal, 0, 0, err);
127 #endif
128         return result;
129 }
```

What's Missing From Ext2?

Block fragmentation Allows numerous small files to be stored in a single large block

Handling of compressed or encrypted files Allows users to transparently store compressed or encrypted files

Logical deletion Easily allow users to "undelete" files

Journaling Speedier consistency checks after a system crash

Ext3

The design of Ext3 had two goals:

1. Implement journaling
2. Remain backwards compatible with Ext2 “as much as possible”

An Overview of Journaling

A *journaled file system* records file system operations in a journal. This helps ensure the integrity of the file system, which means faster boot-up times if a file system has been unmounted uncleanly.

Journaling in Ext3: An Overview

1. A copy of the blocks to be written to disk is stored in the Ext3 *journal*.
2. The blocks are written to the file system.
3. The copies in the journal are removed.

Modes

Ext3 Journaling Modes

Mode	Metadata?	FS data?	Comments
Journal	Yes	Yes	Minimizes chance of corruption; requires more disk accesses
Ordered	Yes	No	Data blocks written *before* metadata
Writeback	Yes	No	Fastest

Journaling Block Device Layer

In the Linux kernel, journaling is abstracted through the *Journal Block Device* layer, or *JBD*.

Log records describe a *single* update of a disk block.

Atomic operation handles are the set of log records related to a *single* high-level file system operation—generally each system call has an associated atomic operation handle.

Transactions are collections of several atomic operation handles. An entire transaction must be committed to the file system from the journal for the file system to be in a consistent state.

A Call to `write()` in Ext3