

The Ext2 and Ext3 Filesystems



In this chapter, we finish our extensive discussion of I/O and filesystems by taking a look at the details the kernel has to take care of when interacting with a specific filesystem. Because the Second Extended Filesystem (Ext2) is native to Linux and is used on virtually every Linux system, it is a natural choice for this discussion. Furthermore, Ext2 illustrates a lot of good practices in its support for modern filesystem features with fast performance. To be sure, other filesystems supported by Linux include many interesting features, but we have no room to examine all of them.

After introducing Ext2 in the section “General Characteristics of Ext2,” we describe the data structures needed, just as in other chapters. Because we are looking at a specific way to store data on disk, we have to consider two versions of the same data structures. The section “Ext2 Disk Data Structures” shows the data structures stored by Ext2 on disk, while “Ext2 Memory Data Structures” shows the corresponding versions in memory.

Then we get to the operations performed on the filesystem. In the section “Creating the Ext2 Filesystem,” we discuss how Ext2 is created in a disk partition. The next sections describe the kernel activities performed whenever the disk is used. Most of these are relatively low-level activities dealing with the allocation of disk space to inodes and data blocks.

In the last section, we give a short description of the Ext3 filesystem, which is the next step in the evolution of the Ext2 filesystem.

General Characteristics of Ext2

Unix-like operating systems use several types of filesystems. Although the files of all such filesystems have a common subset of attributes required by a few POSIX APIs such as `stat()`, each filesystem is implemented in a different way.

The first versions of Linux were based on the MINIX filesystem. As Linux matured, the *Extended Filesystem (Ext FS)* was introduced; it included several significant

extensions, but offered unsatisfactory performance. The *Second Extended Filesystem* (Ext2) was introduced in 1994; besides including several new features, it is quite efficient and robust and is, together with its offspring Ext3, the most widely used Linux filesystem.

The following features contribute to the efficiency of Ext2:

- When creating an Ext2 filesystem, the system administrator may choose the optimal block size (from 1,024 to 4,096 bytes), depending on the expected average file length. For instance, a 1,024-block size is preferable when the average file length is smaller than a few thousand bytes because this leads to less internal fragmentation—that is, less of a mismatch between the file length and the portion of the disk that stores it (see the section “Memory Area Management” in Chapter 8, where internal fragmentation for dynamic memory was discussed). On the other hand, larger block sizes are usually preferable for files greater than a few thousand bytes because this leads to fewer disk transfers, thus reducing system overhead.
- When creating an Ext2 filesystem, the system administrator may choose how many inodes to allow for a partition of a given size, depending on the expected number of files to be stored on it. This maximizes the effectively usable disk space.
- The filesystem partitions disk blocks into groups. Each group includes data blocks and inodes stored in adjacent tracks. Thanks to this structure, files stored in a single block group can be accessed with a lower average disk seek time.
- The filesystem *preallocates* disk data blocks to regular files before they are actually used. Thus, when the file increases in size, several blocks are already reserved at physically adjacent positions, reducing file fragmentation.
- Fast symbolic links (see the section “Hard and Soft Links” in Chapter 1) are supported. If the symbolic link represents a short pathname (at most 60 characters), it can be stored in the inode and can thus be translated without reading a data block.

Moreover, the Second Extended Filesystem includes other features that make it both robust and flexible:

- A careful implementation of file-updating that minimizes the impact of system crashes. For instance, when creating a new hard link for a file, the counter of hard links in the disk inode is increased first, and the new name is added into the proper directory next. In this way, if a hardware failure occurs after the inode update but before the directory can be changed, the directory is consistent, even if the inode’s hard link counter is wrong. Deleting the file does not lead to catastrophic results, although the file’s data blocks cannot be automatically reclaimed. If the reverse were done (changing the directory before updating the inode), the same hardware failure would produce a dangerous inconsistency: deleting the original hard link would remove its data blocks from disk, yet the new directory entry would refer to an inode that no longer exists. If that inode

number were used later for another file, writing into the stale directory entry would corrupt the new file.

- Support for automatic consistency checks on the filesystem status at boot time. The checks are performed by the *e2fsck* external program, which may be activated not only after a system crash, but also after a predefined number of filesystem mounts (a counter is increased after each mount operation) or after a predefined amount of time has elapsed since the most recent check.
- Support for *immutable* files (they cannot be modified, deleted, or renamed) and for *append-only* files (data can be added only to the end of them).
- Compatibility with both the Unix System V Release 4 and the BSD semantics of the user group ID for a new file. In SVR4, the new file assumes the user group ID of the process that creates it; in BSD, the new file inherits the user group ID of the directory containing it. Ext2 includes a mount option that specifies which semantic to use.

Even if the Ext2 filesystem is a mature, stable program, several additional features have been considered for inclusion. Some of them have already been coded and are available as external patches. Others are just planned, but in some cases, fields have already been introduced in the Ext2 inode for them. The most significant features being considered are:

Block fragmentation

System administrators usually choose large block sizes for accessing disks, because computer applications often deal with large files. As a result, small files stored in large blocks waste a lot of disk space. This problem can be solved by allowing several files to be stored in different fragments of the same block.

Handling of transparently compressed and encrypted files

These new options, which must be specified when creating a file, allow users to transparently store compressed and/or encrypted versions of their files on disk.

Logical deletion

An *undelete* option allows users to easily recover, if needed, the contents of a previously removed file.

Journaling

Journaling avoids the time-consuming check that is automatically performed on a filesystem when it is abruptly unmounted—for instance, as a consequence of a system crash.

In practice, none of these features has been officially included in the Ext2 filesystem. One might say that Ext2 is victim of its success; it has been the preferred filesystem adopted by most Linux distribution companies until a few years ago, and the millions of users who relied on it every day would have looked suspiciously at any attempt to replace Ext2 with some other filesystem.

The most compelling feature missing from Ext2 is journaling, which is required by high-availability servers. To provide for a smooth transition, journaling has not been introduced in the Ext2 filesystem; rather, as we'll discuss in the later section "The Ext3 Filesystem," a more recent filesystem that is fully compatible with Ext2 has been created, which also offers journaling. Users who do not really require journaling may continue to use the good old Ext2 filesystem, while the others will likely adopt the new filesystem. Nowadays, most distributions adopt Ext3 as the standard filesystem.

Ext2 Disk Data Structures

The first block in each Ext2 partition is never managed by the Ext2 filesystem, because it is reserved for the partition boot sector (see Appendix A). The rest of the Ext2 partition is split into *block groups*, each of which has the layout shown in Figure 18-1. As you will notice from the figure, some data structures must fit in exactly one block, while others may require more than one block. All the block groups in the filesystem have the same size and are stored sequentially, thus the kernel can derive the location of a block group in a disk simply from its integer index.

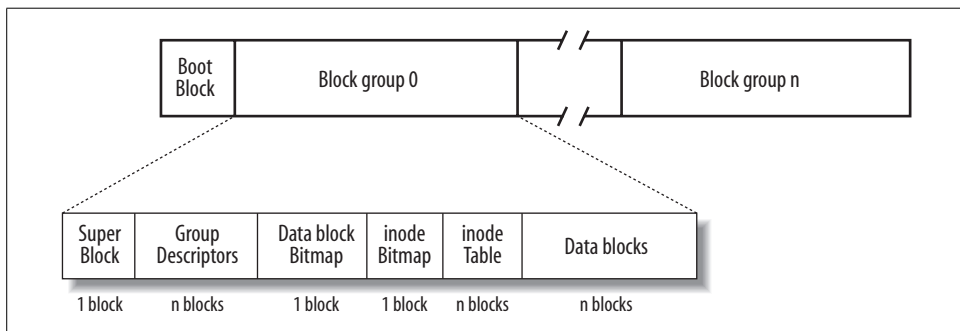


Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group

Block groups reduce file fragmentation, because the kernel tries to keep the data blocks belonging to a file in the same block group, if possible. Each block in a block group contains one of the following pieces of information:

- A copy of the filesystem's superblock
- A copy of the group of block group descriptors
- A data block bitmap
- An inode bitmap
- A table of inodes
- A chunk of data that belongs to a file; i.e., data blocks

If a block does not contain any meaningful information, it is said to be free.

As you can see from Figure 18-1, both the superblock and the group descriptors are duplicated in each block group. Only the superblock and the group descriptors included in block group 0 are used by the kernel, while the remaining superblocks and group descriptors are left unchanged; in fact, the kernel doesn't even look at them. When the *e2fsck* program executes a consistency check on the filesystem status, it refers to the superblock and the group descriptors stored in block group 0, and then copies them into all other block groups. If data corruption occurs and the main superblock or the main group descriptors in block group 0 become invalid, the system administrator can instruct *e2fsck* to refer to the old copies of the superblock and the group descriptors stored in a block groups other than the first. Usually, the redundant copies store enough information to allow *e2fsck* to bring the Ext2 partition back to a consistent state.

How many block groups are there? Well, that depends both on the partition size and the block size. The main constraint is that the block bitmap, which is used to identify the blocks that are used and free inside a group, must be stored in a single block. Therefore, in each block group, there can be at most $8 \times b$ blocks, where b is the block size in bytes. Thus, the total number of block groups is roughly $s/(8 \times b)$, where s is the partition size in blocks.

For example, let's consider a 32-GB Ext2 partition with a 4-KB block size. In this case, each 4-KB block bitmap describes 32K data blocks—that is, 128 MB. Therefore, at most 256 block groups are needed. Clearly, the smaller the block size, the larger the number of block groups.

Superblock

An Ext2 disk superblock is stored in an `ext2_super_block` structure, whose fields are listed in Table 18-1.* The `__u8`, `__u16`, and `__u32` data types denote unsigned numbers of length 8, 16, and 32 bits respectively, while the `__s8`, `__s16`, `__s32` data types denote signed numbers of length 8, 16, and 32 bits. To explicitly specify the order in which the bytes of a word or double-word are stored on disk, the kernel also makes use of the `__le16`, `__le32`, `__be16`, and `__be32` data types; the former two types denote the *little-endian ordering* for words and double-words (the least significant byte is stored at the highest address), respectively, while the latter two types denote the *big-endian ordering* (the most significant byte is stored at the highest address).

Table 18-1. The fields of the Ext2 superblock

Type	Field	Description
<code>__le32</code>	<code>s_inodes_count</code>	Total number of inodes
<code>__le32</code>	<code>s_blocks_count</code>	Filesystem size in blocks

* To ensure compatibility between the Ext2 and Ext3 filesystems, the `ext2_super_block` data structure includes some Ext3-specific fields, which are not shown in Table 18-1.

Table 18-1. The fields of the Ext2 superblock (continued)

Type	Field	Description
__le32	s_r_blocks_count	Number of reserved blocks
__le32	s_free_blocks_count	Free blocks counter
__le32	s_free_inodes_count	Free inodes counter
__le32	s_first_data_block	Number of first useful block (always 1)
__le32	s_log_block_size	Block size
__le32	s_log_frag_size	Fragment size
__le32	s_blocks_per_group	Number of blocks per group
__le32	s_frags_per_group	Number of fragments per group
__le32	s_inodes_per_group	Number of inodes per group
__le32	s_mtime	Time of last mount operation
__le32	s_wtime	Time of last write operation
__le16	s_mnt_count	Mount operations counter
__le16	s_max_mnt_count	Number of mount operations before check
__le16	s_magic	Magic signature
__le16	s_state	Status flag
__le16	s_errors	Behavior when detecting errors
__le16	s_minor_rev_level	Minor revision level
__le32	s_lastcheck	Time of last check
__le32	s_checkinterval	Time between checks
__le32	s_creator_os	OS where filesystem was created
__le32	s_rev_level	Revision level of the filesystem
__le16	s_def_resuid	Default UID for reserved blocks
__le16	s_def_resgid	Default user group ID for reserved blocks
__le32	s_first_ino	Number of first nonreserved inode
__le16	s_inode_size	Size of on-disk inode structure
__le16	s_block_group_nr	Block group number of this superblock
__le32	s_feature_compat	Compatible features bitmap
__le32	s_feature_incompat	Incompatible features bitmap
__le32	s_feature_ro_compat	Read-only compatible features bitmap
__u8 [16]	s_uuid	128-bit filesystem identifier
char [16]	s_volume_name	Volume name
char [64]	s_last_mounted	Pathname of last mount point
__le32	s_algorithm_usage_bitmap	Used for compression
__u8	s_prealloc_blocks	Number of blocks to preallocate
__u8	s_prealloc_dir_blocks	Number of blocks to preallocate for directories
__u16	s_padding1	Alignment to word
__u32 [204]	s_reserved	Nulls to pad out 1,024 bytes

The `s_inodes_count` field stores the number of inodes, while the `s_blocks_count` field stores the number of blocks in the Ext2 filesystem.

The `s_log_block_size` field expresses the block size as a power of 2, using 1,024 bytes as the unit. Thus, 0 denotes 1,024-byte blocks, 1 denotes 2,048-byte blocks, and so on. The `s_log_frag_size` field is currently equal to `s_log_block_size`, because block fragmentation is not yet implemented.

The `s_blocks_per_group`, `s_frags_per_group`, and `s_inodes_per_group` fields store the number of blocks, fragments, and inodes in each block group, respectively.

Some disk blocks are reserved to the superuser (or to some other user or group of users selected by the `s_def_resuid` and `s_def_resgid` fields). These blocks allow the system administrator to continue to use the filesystem even when no more free blocks are available for normal users.

The `s_mnt_count`, `s_max_mnt_count`, `s_lastcheck`, and `s_checkinterval` fields set up the Ext2 filesystem to be checked automatically at boot time. These fields cause `e2fsck` to run after a predefined number of mount operations has been performed, or when a predefined amount of time has elapsed since the last consistency check. (Both kinds of checks can be used together.) The consistency check is also enforced at boot time if the filesystem has not been cleanly unmounted (for instance, after a system crash) or when the kernel discovers some errors in it. The `s_state` field stores the value 0 if the filesystem is mounted or was not cleanly unmounted, 1 if it was cleanly unmounted, and 2 if it contains errors.

Group Descriptor and Bitmap

Each block group has its own group descriptor, an `ext2_group_desc` structure whose fields are illustrated in Table 18-2.

Table 18-2. The fields of the Ext2 group descriptor

Type	Field	Description
__le32	<code>bg_block_bitmap</code>	Block number of block bitmap
__le32	<code>bg_inode_bitmap</code>	Block number of inode bitmap
__le32	<code>bg_inode_table</code>	Block number of first inode table block
__le16	<code>bg_free_blocks_count</code>	Number of free blocks in the group
__le16	<code>bg_free_inodes_count</code>	Number of free inodes in the group
__le16	<code>bg_used_dirs_count</code>	Number of directories in the group
__le16	<code>bg_pad</code>	Alignment to word
__le32 [3]	<code>bg_reserved</code>	Nulls to pad out 24 bytes

The `bg_free_blocks_count`, `bg_free_inodes_count`, and `bg_used_dirs_count` fields are used when allocating new inodes and data blocks. These fields determine the most suitable block in which to allocate each data structure. The bitmaps are sequences of

bits, where the value 0 specifies that the corresponding inode or data block is free and the value 1 specifies that it is used. Because each bitmap must be stored inside a single block and because the block size can be 1,024, 2,048, or 4,096 bytes, a single bitmap describes the state of 8,192, 16,384, or 32,768 blocks.

Inode Table

The inode table consists of a series of consecutive blocks, each of which contains a predefined number of inodes. The block number of the first block of the inode table is stored in the `bg_inode_table` field of the group descriptor.

All inodes have the same size: 128 bytes. A 1,024-byte block contains 8 inodes, while a 4,096-byte block contains 32 inodes. To figure out how many blocks are occupied by the inode table, divide the total number of inodes in a group (stored in the `s_inodes_per_group` field of the superblock) by the number of inodes per block.

Each Ext2 inode is an `ext2_inode` structure whose fields are illustrated in Table 18-3.

Table 18-3. The fields of an Ext2 disk inode

Type	Field	Description
__le16	<code>i_mode</code>	File type and access rights
__le16	<code>i_uid</code>	Owner identifier
__le32	<code>i_size</code>	File length in bytes
__le32	<code>i_atime</code>	Time of last file access
__le32	<code>i_ctime</code>	Time that inode last changed
__le32	<code>i_mtime</code>	Time that file contents last changed
__le32	<code>i_dtime</code>	Time of file deletion
__le16	<code>i_gid</code>	User group identifier
__le16	<code>i_links_count</code>	Hard links counter
__le32	<code>i_blocks</code>	Number of data blocks of the file
__le32	<code>i_flags</code>	File flags
union	<code>osd1</code>	Specific operating system information
__le32 [EXT2_N_BLOCKS]	<code>i_block</code>	Pointers to data blocks
__le32	<code>i_generation</code>	File version (used when the file is accessed by a network filesystem)
__le32	<code>i_file_acl</code>	File access control list
__le32	<code>i_dir_acl</code>	Directory access control list
__le32	<code>i_faddr</code>	Fragment address
union	<code>osd2</code>	Specific operating system information

Many fields related to POSIX specifications are similar to the corresponding fields of the VFS's inode object and have already been discussed in the section “Inode

Objects” in Chapter 12. The remaining ones refer to the Ext2-specific implementation and deal mostly with block allocation.

In particular, the `i_size` field stores the effective length of the file in bytes, while the `i_blocks` field stores the number of data blocks (in units of 512 bytes) that have been allocated to the file.

The values of `i_size` and `i_blocks` are not necessarily related. Because a file is always stored in an integer number of blocks, a nonempty file receives at least one data block (since fragmentation is not yet implemented) and `i_size` may be smaller than $512 \times i_blocks$. On the other hand, as we’ll see in the section “File Holes” later in this chapter, a file may contain holes. In that case, `i_size` may be greater than $512 \times i_blocks$.

The `i_block` field is an array of `EXT2_N_BLOCKS` (usually 15) pointers to blocks used to identify the data blocks allocated to the file (see the section “Data Blocks Addressing” later in this chapter).

The 32 bits reserved for the `i_size` field limit the file size to 4 GB. Actually, the highest-order bit of the `i_size` field is not used, so the maximum file size is limited to 2 GB. However, the Ext2 filesystem includes a “dirty trick” that allows larger files on systems that sport a 64-bit processor such as AMD’s Opteron or IBM’s PowerPC G5. Essentially, the `i_dir_acl` field of the inode, which is not used for regular files, represents a 32-bit extension of the `i_size` field. Therefore, the file size is stored in the inode as a 64-bit integer. The 64-bit version of the Ext2 filesystem is somewhat compatible with the 32-bit version because an Ext2 filesystem created on a 64-bit architecture may be mounted on a 32-bit architecture, and vice versa. On a 32-bit architecture, a large file cannot be accessed, unless opening the file with the `O_LARGEFILE` flag set (see the section “The `open()` System Call” in Chapter 12).

Recall that the VFS model requires each file to have a different inode number. In Ext2, there is no need to store on disk a mapping between an inode number and the corresponding block number because the latter value can be derived from the block group number and the relative position inside the inode table. For example, suppose that each block group contains 4,096 inodes and that we want to know the address on disk of inode 13,021. In this case, the inode belongs to the third block group and its disk address is stored in the 733rd entry of the corresponding inode table. As you can see, the inode number is just a key used by the Ext2 routines to retrieve the proper inode descriptor on disk quickly.

Extended Attributes of an Inode

The Ext2 inode format is a kind of straitjacket for filesystem designers. The length of an inode must be a power of 2 to avoid internal fragmentation in the blocks that store the inode table. Actually, most of the 128 characters of an Ext2 inode are currently packed with information, and there is little room left for additional fields. On the other hand, expanding the inode length to 256 would be quite wasteful, besides

introducing compatibility problems between Ext2 filesystems that use different inode lengths.

Extended attributes have been introduced to overcome the above limitation. These attributes are stored on a disk block allocated outside of any inode. The `i_file_acl` field of an inode points to the block containing the extended attributes. Different inodes that have the same set of extended attributes may share the same block.

Each extended attribute has a name and a value. Both of them are encoded as variable length arrays of characters, as specified by the `ext2_xattr_entry` descriptor. Figure 18-2 shows the layout in Ext2 of the extended attributes inside a block. Each attribute is split in two parts: the `ext2_xattr_entry` descriptor together with the name of the attribute are placed at the beginning of the block, while the value of the attribute is placed at the end of the block. The entries at the beginning of the block are ordered according to the attribute names, while the positions of the values are fixed, because they are determined by the allocation order of the attributes.

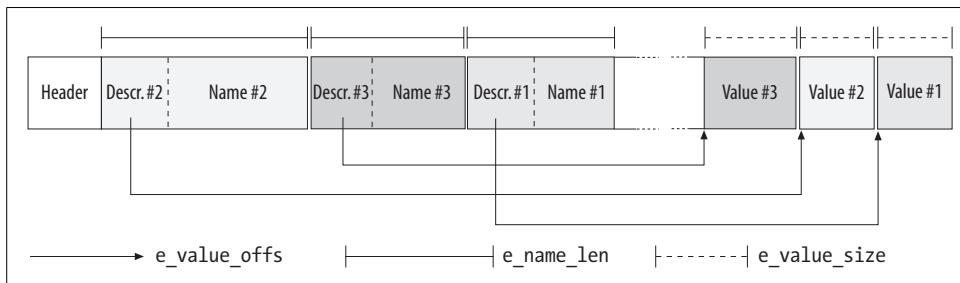


Figure 18-2. Layout of a block containing extended attributes

There are many system calls used to set, retrieve, list, and remove the extended attributes of a file. The `setxattr()`, `lsetxattr()`, and `fsetxattr()` system calls set an extended attribute of a file; essentially, they differ in how symbolic links are handled, and in how the file is specified (either passing a pathname or a file descriptor). Similarly, the `getxattr()`, `lgetxattr()`, and `fgetxattr()` system calls return the value of an extended attribute. The `listxattr()`, `llistxattr()`, and `flistxattr()` list all extended attributes of a file. Finally, the `removexattr()`, `lremovexattr()`, and `fremovexattr()` system calls remove an extended attribute from a file.

Access Control Lists

Access control lists were proposed a long time ago to improve the file protection mechanism in Unix filesystems. Instead of classifying the users of a file under three classes—owner, group, and others—an *access control list* (ACL) can be associated with each file. Thanks to this kind of list, a user may specify for each of his files the names of specific users (or groups of users) and the privileges to be given to these users.

Linux 2.6 fully supports ACLs by making use of inode extended attributes. As a matter of fact, extended attributes have been introduced mainly to support ACLs. Therefore, the `chacl()`, `setfacl()`, and `getfacl()` library functions, which allow you to manipulate the ACLs of a file, rely essentially upon the `setxattr()` and `getxattr()` system calls introduced in the previous section.

Unfortunately, the outcome of a working group that defined security extensions within the POSIX 1003.1 family of standards has never been formalized as a new POSIX standard. As a result, ACLs are supported nowadays on different filesystem types on many UNIX-like systems, albeit with a number of subtle differences among the different implementations.

How Various File Types Use Disk Blocks

The different types of files recognized by Ext2 (regular files, pipes, etc.) use data blocks in different ways. Some files store no data and therefore need no data blocks at all. This section discusses the storage requirements for each type, which are listed in Table 18-4.

Table 18-4. Ext2 file types

File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Regular file

Regular files are the most common case and receive almost all the attention in this chapter. But a regular file needs data blocks only when it starts to have data. When first created, a regular file is empty and needs no data blocks; it can also be emptied by the `truncate()` or `open()` system calls. Both situations are common; for instance, when you issue a shell command that includes the string *>filename*, the shell creates an empty file or truncates an existing one.

Directory

Ext2 implements directories as a special kind of file whose data blocks store file-names together with the corresponding inode numbers. In particular, such data blocks contain structures of type `ext2_dir_entry_2`. The fields of that structure are

shown in Table 18-5. The structure has a variable length, because the last name field is a variable length array of up to EXT2_NAME_LEN characters (usually 255). Moreover, for reasons of efficiency, the length of a directory entry is always a multiple of 4 and, therefore, null characters (\0) are added for padding at the end of the filename, if necessary. The name_len field stores the actual filename length (see Figure 18-3).

Table 18-5. The fields of an Ext2 directory entry

Type	Field	Description
__le32	inode	Inode number
__le16	rec_len	Directory entry length
__u8	name_len	Filename length
__u8	file_type	File type
char [EXT2_NAME_LEN]	name	Filename

The file_type field stores a value that specifies the file type (see Table 18-4). The rec_len field may be interpreted as a pointer to the next valid directory entry: it is the offset to be added to the starting address of the directory entry to get the starting address of the next valid directory entry. To delete a directory entry, it is sufficient to set its inode field to 0 and suitably increment the value of the rec_len field of the previous valid entry. Read the rec_len field of Figure 18-3 carefully; you'll see that the *oldfile* entry was deleted because the rec_len field of *usr* is set to 12+16 (the lengths of the *usr* and *oldfile* entries).

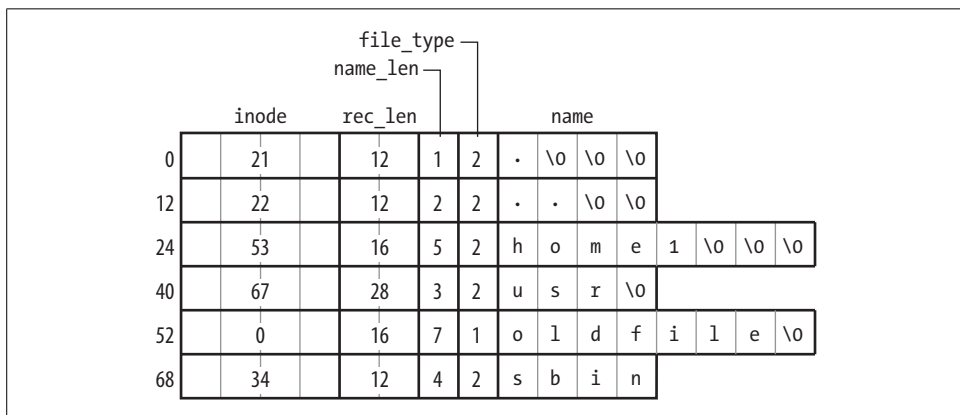


Figure 18-3. An example of the Ext2 directory

Symbolic link

As stated before, if the pathname of a symbolic link has up to 60 characters, it is stored in the i_block field of the inode, which consists of an array of 15 4-byte integers; no data block is therefore required. If the pathname is longer than 60 characters, however, a single data block is required.

Device file, pipe, and socket

No data blocks are required for these kinds of files. All the necessary information is stored in the inode.

Ext2 Memory Data Structures

For the sake of efficiency, most information stored in the disk data structures of an Ext2 partition are copied into RAM when the filesystem is mounted, thus allowing the kernel to avoid many subsequent disk read operations. To get an idea of how often some data structures change, consider some fundamental operations:

- When a new file is created, the values of the `s_free_inodes_count` field in the Ext2 superblock and of the `bg_free_inodes_count` field in the proper group descriptor must be decreased.
- If the kernel appends some data to an existing file so that the number of data blocks allocated for it increases, the values of the `s_free_blocks_count` field in the Ext2 superblock and of the `bg_free_blocks_count` field in the group descriptor must be modified.
- Even just rewriting a portion of an existing file involves an update of the `s_wtime` field of the Ext2 superblock.

Because all Ext2 disk data structures are stored in blocks of the Ext2 partition, the kernel uses the page cache to keep them up-to-date (see the section “Writing Dirty Pages to Disk” in Chapter 15).

Table 18-6 specifies, for each type of data related to Ext2 filesystems and files, the data structure used on the disk to represent its data, the data structure used by the kernel in memory, and a rule of thumb used to determine how much caching is used. Data that is updated very frequently is always cached; that is, the data is permanently stored in memory and included in the page cache until the corresponding Ext2 partition is unmounted. The kernel gets this result by keeping the page’s usage counter greater than 0 at all times.

Table 18-6. VFS images of Ext2 data structures

Type	Disk data structure	Memory data structure	Caching mode
Superblock	<code>ext2_super_block</code>	<code>ext2_sb_info</code>	Always cached
Group descriptor	<code>ext2_group_desc</code>	<code>ext2_group_desc</code>	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	<code>ext2_inode</code>	<code>ext2_inode_info</code>	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	<code>ext2_inode</code>	None	Never
Free block	Array of bytes	None	Never

The never-cached data is not kept in any cache because it does not represent meaningful information. Conversely, the always-cached data is always present in RAM, thus it is never necessary to read the data from disk (periodically, however, the data must be written back to disk). In between these extremes lies the *dynamic* mode. In this mode, the data is kept in a cache as long as the associated object (inode, data block, or bitmap) is in use; when the file is closed or the data block is deleted, the page frame reclaiming algorithm may remove the associated data from the cache.

It is interesting to observe that inode and block bitmaps are not kept permanently in memory; rather, they are read from disk when needed. Actually, many disk reads are avoided thanks to the page cache, which keeps in memory the most recently used disk blocks (see the section “Storing Blocks in the Page Cache” in Chapter 15).*

The Ext2 Superblock Object

As stated in the section “Superblock Objects” in Chapter 12, the `s_fs_info` field of the VFS superblock points to a structure containing filesystem-specific data. In the case of Ext2, this field points to a structure of type `ext2_sb_info`, which includes the following information:

- Most of the disk superblock fields
- An `s_sbh` pointer to the buffer head of the buffer containing the disk superblock
- An `s_es` pointer to the buffer containing the disk superblock
- The number of group descriptors, `s_desc_per_block`, that can be packed in a block
- An `s_group_desc` pointer to an array of buffer heads of buffers containing the group descriptors (usually, a single entry is sufficient)
- Other data related to mount state, mount options, and so on

Figure 18-4 shows the links between the `ext2_sb_info` data structures and the buffers and buffer heads relative to the Ext2 superblock and to the group descriptors.

When the kernel mounts an Ext2 filesystem, it invokes the `ext2_fill_super()` function to allocate space for the data structures and to fill them with data read from disk (see the section “Mounting a Generic Filesystem” in Chapter 12). This is a simplified description of the function, which emphasizes the memory allocations for buffers and descriptors:

1. Allocates an `ext2_sb_info` descriptor and stores its address in the `s_fs_info` field of the superblock object passed as the parameter.

* In Linux 2.4 and earlier versions, the most recently used inode and block bitmaps were stored in ad-hoc caches of bounded size.

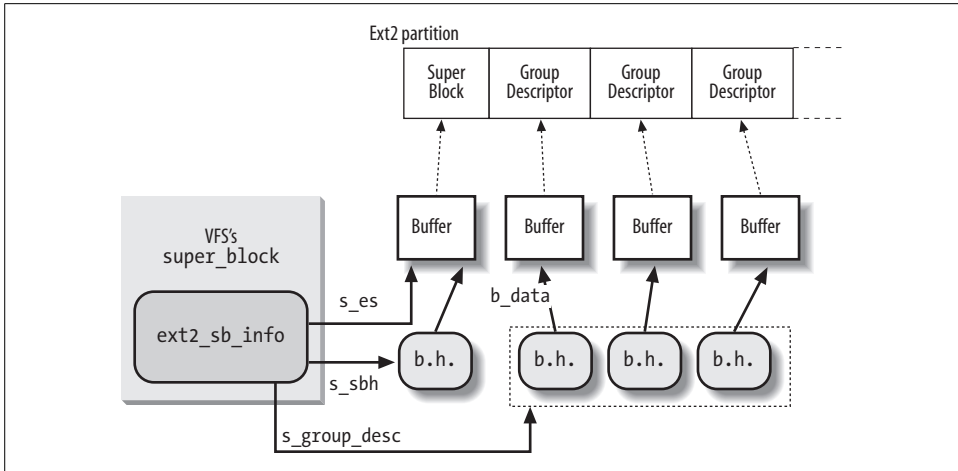


Figure 18-4. The `ext2_sb_info` data structure

2. Invokes `__bread()` to allocate a buffer in a buffer page together with the corresponding buffer head, and to read the superblock from disk into the buffer; as discussed in the section “Searching Blocks in the Page Cache” in Chapter 15, no allocation is performed if the block is already stored in a buffer page in the page cache and it is up-to-date. Stores the buffer head address in the `s_sb` field of the Ext2 superblock object.
3. Allocates an array of bytes—one byte for each group—and stores its address in the `s_debts` field of the `ext2_sb_info` descriptor (see the section “Creating inodes” later in this chapter).
4. Allocates an array of pointers to buffer heads, one for each group descriptor, and stores the address of the array in the `s_group_desc` field of the `ext2_sb_info` descriptor.
5. Invokes repeatedly `__bread()` to allocate buffers and to read from disk the blocks containing the Ext2 group descriptors; stores the addresses of the buffer heads in the `s_group_desc` array allocated in the previous step.
6. Allocates an inode and a dentry object for the root directory, and sets up a few fields of the superblock object so that it will be possible to read the root inode from disk.

Clearly, all the data structures allocated by `ext2_fill_super()` are kept in memory after the function returns; they will be released only when the Ext2 filesystem will be unmounted. When the kernel must modify a field in the Ext2 superblock, it simply writes the new value in the proper position of the corresponding buffer and then marks the buffer as dirty.

The Ext2 inode Object

When opening a file, a pathname lookup is performed. For each component of the pathname that is not already in the dentry cache, a new dentry object and a new inode object are created (see the section “Standard Pathname Lookup” in Chapter 12). When the VFS accesses an Ext2 disk inode, it creates a corresponding *inode descriptor* of type `ext2_inode_info`. This descriptor includes the following information:

- The whole VFS inode object (see Table 12-3 in Chapter 12) stored in the field `vfs_inode`
- Most of the fields found in the disk’s inode structure that are not kept in the VFS inode
- The `i_block_group` block group index at which the inode belongs (see the section “Ext2 Disk Data Structures” earlier in this chapter)
- The `i_next_alloc_block` and `i_next_alloc_goal` fields, which store the logical block number and the physical block number of the disk block that was most recently allocated to the file, respectively
- The `i_prealloc_block` and `i_prealloc_count` fields, which are used for data block preallocation (see the section “Allocating a Data Block” later in this chapter)
- The `xattr_sem` field, a read/write semaphore that allows extended attributes to be read concurrently with the file data
- The `i_acl` and `i_default_acl` fields, which point to the ACLs of the file

When dealing with Ext2 files, the `alloc_inode` superblock method is implemented by means of the `ext2_alloc_inode()` function. It gets first an `ext2_inode_info` descriptor from the `ext2_inode_cache` slab allocator cache, then it returns the address of the inode object embedded in the new `ext2_inode_info` descriptor.

Creating the Ext2 Filesystem

There are generally two stages to creating a filesystem on a disk. The first step is to format it so that the disk driver can read and write blocks on it. Modern hard disks come preformatted from the factory and need not be reformatted; floppy disks may be formatted on Linux using a utility program such as *superformat* or *fdformat*. The second step involves creating a filesystem, which means setting up the structures described in detail earlier in this chapter.

Ext2 filesystems are created by the *mke2fs* utility program; it assumes the following default options, which may be modified by the user with flags on the command line:

- Block size: 1,024 bytes (default value for a small filesystem)
- Fragment size: block size (block fragmentation is not implemented)
- Number of allocated inodes: 1 inode for each 8,192 bytes
- Percentage of reserved blocks: 5 percent

The program performs the following actions:

1. Initializes the superblock and the group descriptors.
2. Optionally, checks whether the partition contains defective blocks; if so, it creates a list of defective blocks.
3. For each block group, reserves all the disk blocks needed to store the superblock, the group descriptors, the inode table, and the two bitmaps.
4. Initializes the inode bitmap and the data map bitmap of each block group to 0.
5. Initializes the inode table of each block group.
6. Creates the */root* directory.
7. Creates the *lost+found* directory, which is used by *e2fsck* to link the lost and found defective blocks.
8. Updates the inode bitmap and the data block bitmap of the block group in which the two previous directories have been created.
9. Groups the defective blocks (if any) in the *lost+found* directory.

Let's consider how an Ext2 1.44 MB floppy disk is initialized by *mke2fs* with the default options.

Once mounted, it appears to the VFS as a volume consisting of 1,412 blocks; each one is 1,024 bytes in length. To examine the disk's contents, we can execute the Unix command:

```
$ dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

to get a file containing the hexadecimal dump of the floppy disk contents in the */tmp* directory.*

By looking at that file, we can see that, due to the limited capacity of the disk, a single group descriptor is sufficient. We also notice that the number of reserved blocks is set to 72 (5 percent of 1,440) and, according to the default option, the inode table must include 1 inode for each 8,192 bytes—that is, 184 inodes stored in 23 blocks.

Table 18-7 summarizes how the Ext2 filesystem is created on a floppy disk when the default options are selected.

Table 18-7. Ext2 block allocation for a floppy disk

Block	Content
0	Boot block
1	Superblock
2	Block containing a single block group descriptor
3	Data block bitmap

* Most information on an Ext2 filesystem could also be obtained by using the *dumpe2fs* and *debugfs* utility programs.

Table 18-7. Ext2 block allocation for a floppy disk (continued)

Block	Content
4	inode bitmap
5–27	inode table: inodes up to 10: reserved (inode 2 is the root); inode 11: <i>lost+found</i> ; inodes 12–184: free
28	Root directory (includes <i>.</i> , <i>..</i> , and <i>lost+found</i>)
29	<i>lost+found</i> directory (includes <i>.</i> and <i>..</i>)
30–40	Reserved blocks preallocated for <i>lost+found</i> directory
41–1439	Free blocks

Ext2 Methods

Many of the VFS methods described in Chapter 12 have a corresponding Ext2 implementation. Because it would take a whole book to describe all of them, we limit ourselves to briefly reviewing the methods implemented in Ext2. Once the disk and the memory data structures are clearly understood, the reader should be able to follow the code of the Ext2 functions that implement them.

Ext2 Superblock Operations

Many VFS superblock operations have a specific implementation in Ext2, namely `alloc_inode`, `destroy_inode`, `read_inode`, `write_inode`, `delete_inode`, `put_super`, `write_super`, `statfs`, `remount_fs`, and `clear_inode`. The addresses of the superblock methods are stored in the `ext2_sops` array of pointers.

Ext2 inode Operations

Some of the VFS inode operations have a specific implementation in Ext2, which depends on the type of the file to which the inode refers.

The inode operations for Ext2 regular files and Ext2 directories are shown in Table 18-8; the purpose of each method is described in the section “Inode Objects” in Chapter 12. The table does not show the methods that are undefined (a NULL pointer) for both regular files and directories; recall that if a method is undefined, the VFS either invokes a generic function or does nothing at all. The addresses of the Ext2 methods for regular files and directories are stored in the `ext2_file_inode_operations` and `ext2_dir_inode_operations` tables, respectively.

Table 18-8. Ext2 inode operations for regular files and directories

VFS inode operation	Regular file	Directory
create	NULL	<code>ext2_create()</code>
lookup	NULL	<code>ext2_lookup()</code>
link	NULL	<code>ext2_link()</code>
unlink	NULL	<code>ext2_unlink()</code>

Table 18-8. Ext2 inode operations for regular files and directories (continued)

VFS inode operation	Regular file	Directory
symlink	NULL	ext2_symlink()
mkdir	NULL	ext2_mkdir()
rmdir	NULL	ext2_rmdir()
mknod	NULL	ext2_mknod()
rename	NULL	ext2_rename()
truncate	ext2_truncate()	NULL
permission	ext2_permission()	ext2_permission()
setattr	ext2_setattr()	ext2_setattr()
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

The inode operations for Ext2 symbolic links are shown in Table 18-9 (undefined methods have been omitted). Actually, there are two types of symbolic links: the fast symbolic links represent pathnames that can be fully stored inside the inodes, while the regular symbolic links represent longer pathnames. Accordingly, there are two sets of inode operations, which are stored in the `ext2_fast_symlink_inode_operations` and `ext2_symlink_inode_operations` tables, respectively.

Table 18-9. Ext2 inode operations for fast and regular symbolic links

VFS inode operation	Fast symbolic link	Regular symbolic link
readlink	generic_readlink()	generic_readlink()
follow_link	ext2_follow_link()	page_follow_link_light()
put_link	NULL	page_put_link()
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

If the inode refers to a character device file, to a block device file, or to a named pipe (see “FIFOs” in Chapter 19), the inode operations do not depend on the filesystem. They are specified in the `chrdev_inode_operations`, `blkdev_inode_operations`, and `fifo_inode_operations` tables, respectively.

Ext2 File Operations

The file operations specific to the Ext2 filesystem are listed in Table 18-10. As you can see, several VFS methods are implemented by generic functions that are common

to many filesystems. The addresses of these methods are stored in the `ext2_file_operations` table.

Table 18-10. Ext2 file operations

VFS file operation	Ext2 method
<code>llseek</code>	<code>generic_file_llseek()</code>
<code>read</code>	<code>generic_file_read()</code>
<code>write</code>	<code>generic_file_write()</code>
<code>aio_read</code>	<code>generic_file_aio_read()</code>
<code>aio_write</code>	<code>generic_file_aio_write()</code>
<code>ioctl</code>	<code>ext2_ioctl()</code>
<code>mmap</code>	<code>generic_file_mmap()</code>
<code>open</code>	<code>generic_file_open()</code>
<code>release</code>	<code>ext2_release_file()</code>
<code>fsync</code>	<code>ext2_sync_file()</code>
<code>readv</code>	<code>generic_file_readv()</code>
<code>writv</code>	<code>generic_file_writev()</code>
<code>sendfile</code>	<code>generic_file_sendfile()</code>

Notice that the Ext2's read and write methods are implemented by the `generic_file_read()` and `generic_file_write()` functions, respectively. These are described in the sections “Reading from a File” and “Writing to a File” in Chapter 16.

Managing Ext2 Disk Space

The storage of a file on disk differs from the view the programmer has of the file in two ways: blocks can be scattered around the disk (although the filesystem tries hard to keep blocks sequential to improve access time), and files may appear to a programmer to be bigger than they really are because a program can introduce holes into them (through the `lseek()` system call).

In this section, we explain how the Ext2 filesystem manages the disk space—how it allocates and deallocates inodes and data blocks. Two main problems must be addressed:

- Space management must make every effort to avoid *file fragmentation*—the physical storage of a file in several, small pieces located in non-adjacent disk blocks. File fragmentation increases the average time of sequential read operations on the files, because the disk heads must be frequently repositioned during

the read operation.* This problem is similar to the external fragmentation of RAM discussed in the section “The Buddy System Algorithm” in Chapter 8.

- Space management must be time-efficient; that is, the kernel should be able to quickly derive from a file offset the corresponding logical block number in the Ext2 partition. In doing so, the kernel should limit as much as possible the number of accesses to addressing tables stored on disk, because each such intermediate access considerably increases the average file access time.

Creating inodes

The `ext2_new_inode()` function creates an Ext2 disk inode, returning the address of the corresponding inode object (or `NULL`, in case of failure). The function carefully selects the block group that contains the new inode; this is done to spread unrelated directories among different groups and, at the same time, to put files into the same group as their parent directories. To balance the number of regular files and directories in a block group, Ext2 introduces a “debt” parameter for every block group.

The function acts on two parameters: the address `dir` of the inode object that refers to the directory into which the new inode must be inserted and a `mode` that indicates the type of inode being created. The latter argument also includes the `MS_SYNCHRONOUS` mount flag (see the section “Mounting a Generic Filesystem” in Chapter 12) that requires the current process to be suspended until the inode is allocated. The function performs the following actions:

1. Invokes `new_inode()` to allocate a new VFS inode object; initializes its `i_sb` field to the superblock address stored in `dir->i_sb`, and adds it to the in-use inode list and to the superblock’s list (see the section “Inode Objects” in Chapter 12).
2. If the new inode is a directory, the function invokes `find_group_orlov()` to find a suitable block group for the directory.[†] This function implements the following heuristics:
 - a. Directories having as parent the filesystem root should be spread among all block groups. Thus, the function searches the block groups looking for a group having a number of free inodes and a number of free blocks above the average. If there is no such group, it jumps to step 2c.
 - b. Nested directories—not having the filesystem root as parent—should be put in the group of the parent if it satisfies the following rules:
 - The group does not contain too many directories
 - The group has a sufficient number of free inodes left

* Please note that fragmenting a file across block groups (A Bad Thing) is quite different from the not-yet-implemented fragmentation of blocks to store many files in one block (A Good Thing).

[†] The Ext2 filesystem may also be mounted with an option flag that forces the kernel to make use of a simpler, older allocation strategy, which is implemented by the `find_group_dir()` function.

- The group has a small “debt” (the debt of a block group is stored in the array of counters pointed to by the `s_debts` field of the `ext2_sb_info` descriptor; the debt is increased each time a new directory is added and decreased each time another type of file is added)

If the parent’s group does not satisfy these rules, it picks the first group that satisfies them. If no such group exists, it jumps to step 2c.

- c. This is the “fallback” rule, to be used if no good group has been found. The function starts with the block group containing the parent directory and selects the first block group that has more free inodes than the average number of free inodes per block group.
3. If the new inode is not a directory, it invokes `find_group_other()` to allocate it in a block group having a free inode. This function selects the group by starting from the one that contains the parent directory and moving farther away from it; to be precise:
 - a. Performs a quick logarithmic search starting from the block group that includes the parent directory `dir`. The algorithm searches $\log(n)$ block groups, where n is the total number of block groups. The algorithm jumps further ahead until it finds an available block group—for example, if we call the number of the starting block group i , the algorithm considers block groups $i \bmod (n)$, $i+1 \bmod (n)$, $i+1+2 \bmod (n)$, $i+1+2+4 \bmod (n)$, etc.
 - b. If the logarithmic search failed in finding a block group with a free inode, the function performs an exhaustive linear search starting from the block group that includes the parent directory `dir`.
 4. Invokes `read_inode_bitmap()` to get the inode bitmap of the selected block group and searches for the first null bit into it, thus obtaining the number of the first free disk inode.
 5. Allocates the disk inode: sets the corresponding bit in the inode bitmap and marks the buffer containing the bitmap as dirty. Moreover, if the filesystem has been mounted specifying the `MS_SYNCHRONOUS` flag (see the section “Mounting a Generic Filesystem” in Chapter 12), the function invokes `sync_dirty_buffer()` to start the I/O write operation and waits until the operation terminates.
 6. Decreases the `bg_free_inodes_count` field of the group descriptor. If the new inode is a directory, the function increases the `bg_used_dirs_count` field and marks the buffer containing the group descriptor as dirty.
 7. Increases or decreases the group’s counter in the `s_debts` array of the superblock, according to whether the inode refers to a regular file or a directory.
 8. Decreases the `s_freeinodes_counter` field of the `ext2_sb_info` data structure; moreover, if the new inode is a directory, it increases the `s_dirs_counter` field in the `ext2_sb_info` data structure.
 9. Sets the `s_dirt` flag of the superblock to 1, and marks the buffer that contains it to as dirty.

10. Sets the `s_dirt` field of the VFS's superblock object to 1.
11. Initializes the fields of the inode object. In particular, it sets the inode number `i_no` and copies the value of `xtime.tv_sec` into `i_atime`, `i_mtime`, and `i_ctime`. Also loads the `i_block_group` field in the `ext2_inode_info` structure with the block group index. Refer to Table 18-3 for the meaning of these fields.
12. Initializes the ACLs of the inode.
13. Inserts the new inode object into the hash table `inode_hashtable` and invokes `mark_inode_dirty()` to move the inode object into the superblock's dirty inode list (see the section "Inode Objects" in Chapter 12).
14. Invokes `ext2_preread_inode()` to read from disk the block containing the inode and to put the block in the page cache. This type of read-ahead is done because it is likely that a recently created inode will be written back soon.
15. Returns the address of the new inode object.

Deleting inodes

The `ext2_free_inode()` function deletes a disk inode, which is identified by an inode object whose address `inode` is passed as the parameter. The kernel should invoke the function after a series of cleanup operations involving internal data structures and the data in the file itself. It should come after the inode object has been removed from the inode hash table, after the last hard link referring to that inode has been deleted from the proper directory and after the file is truncated to 0 length to reclaim all its data blocks (see the section "Releasing a Data Block" later in this chapter). It performs the following actions:

1. Invokes `clear_inode()`, which in turn executes the following operations:
 - a. Removes any dirty "indirect" buffer associated with the inode (see the later section "Data Blocks Addressing"); they are collected in the list headed at the `private_list` field of the `address_space` object `inode->i_data` (see the section "The `address_space` Object" in Chapter 15).
 - b. If the `I_LOCK` flag of the inode is set, some of the inode's buffers are involved in I/O data transfers; the function suspends the current process until these I/O data transfers terminate.
 - c. Invokes the `clear_inode` method of the superblock object, if defined; the Ext2 filesystem does not define it.
 - d. If the inode refers to a device file, it removes the inode object from the device's list of inodes; this list is rooted either in the `list` field of the `cdev` character device descriptor (see the section "Character Device Drivers" in Chapter 13) or in the `bd_inodes` field of the `block_device` block device descriptor (see the section "Block Devices" in Chapter 14).
 - e. Sets the state of the inode to `I_CLEAR` (the inode object contents are no longer meaningful).

2. Computes the index of the block group containing the disk inode from the inode number and the number of inodes in each block group.
3. Invokes `read_inode_bitmap()` to get the inode bitmap.
4. Increases the `bg_free_inodes_count()` field of the group descriptor. If the deleted inode is a directory, it decreases the `bg_used_dirs_count` field. Marks the buffer that contains the group descriptor as dirty.
5. If the deleted inode is a directory, it decreases the `s_dirs_counter` field in the `ext2_sb_info` data structure, sets the `s_dirty` flag of the superblock to 1, and marks the buffer that contains it as dirty.
6. Clears the bit corresponding to the disk inode in the inode bitmap and marks the buffer that contains the bitmap as dirty. Moreover, if the filesystem has been mounted with the `MS_SYNCHRONIZE` flag, it invokes `sync_dirty_buffer()` to wait until the write operation on the bitmap's buffer terminates.

Data Blocks Addressing

Each nonempty regular file consists of a group of data blocks. Such blocks may be referred to either by their relative position inside the file—their file block number—or by their position inside the disk partition—their logical block number (see the section “Block Devices Handling” in Chapter 14).

Deriving the logical block number of the corresponding data block from an offset f inside a file is a two-step process:

1. Derive from the offset f the file block number—the index of the block that contains the character at offset f .
2. Translate the file block number to the corresponding logical block number.

Because Unix files do not include any control characters, it is quite easy to derive the file block number containing the f^{th} character of a file: simply take the quotient of f and the filesystem's block size and round down to the nearest integer.

For instance, let's assume a block size of 4 KB. If f is smaller than 4,096, the character is contained in the first data block of the file, which has file block number 0. If f is equal to or greater than 4,096 and less than 8,192, the character is contained in the data block that has file block number 1, and so on.

This is fine as far as file block numbers are concerned. However, translating a file block number into the corresponding logical block number is not nearly as straightforward, because the data blocks of an Ext2 file are not necessarily adjacent on disk.

The Ext2 filesystem must therefore provide a method to store the connection between each file block number and the corresponding logical block number on disk. This mapping, which goes back to early versions of Unix from AT&T, is implemented partly inside the inode. It also involves some specialized blocks that contain extra pointers, which are an inode extension used to handle large files.

The `i_block` field in the disk inode is an array of `EXT2_N_BLOCKS` components that contain logical block numbers. In the following discussion, we assume that `EXT2_N_BLOCKS` has the default value, namely 15. The array represents the initial part of a larger data structure, which is illustrated in Figure 18-5. As can be seen in the figure, the 15 components of the array are of 4 different types:

- The first 12 components yield the logical block numbers corresponding to the first 12 blocks of the file—to the blocks that have file block numbers from 0 to 11.
- The component at index 12 contains the logical block number of a block, called *indirect block*, that represents a second-order array of logical block numbers. They correspond to the file block numbers ranging from 12 to $b/4+11$, where b is the filesystem's block size (each logical block number is stored in 4 bytes, so we divide by 4 in the formula). Therefore, the kernel must look in this component for a pointer to a block, and then look in that block for another pointer to the ultimate block that contains the file contents.
- The component at index 13 contains the logical block number of an indirect block containing a second-order array of logical block numbers; in turn, the entries of this second-order array point to third-order arrays, which store the logical block numbers that correspond to the file block numbers ranging from $b/4+12$ to $(b/4)^2+(b/4)+11$.
- Finally, the component at index 14 uses triple indirection: the fourth-order arrays store the logical block numbers corresponding to the file block numbers ranging from $(b/4)^2+(b/4)+12$ to $(b/4)^3+(b/4)^2+(b/4)+11$.

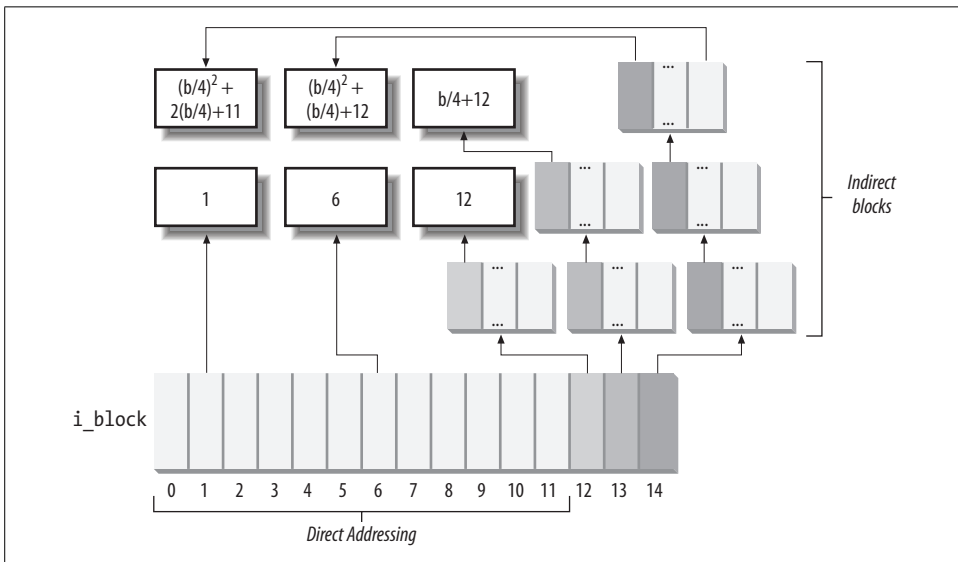


Figure 18-5. Data structures used to address the file's data blocks

In Figure 18-5, the number inside a block represents the corresponding file block number. The arrows, which represent logical block numbers stored in array components,

show how the kernel finds its way through indirect blocks to reach the block that contains the actual contents of the file.

Notice how this mechanism favors small files. If the file does not require more than 12 data blocks, every data can be retrieved in two disk accesses: one to read a component in the `i_block` array of the disk inode and the other to read the requested data block. For larger files, however, three or even four consecutive disk accesses may be needed to access the required block. In practice, this is a worst-case estimate, because dentry, inode, and page caches contribute significantly to reduce the number of real disk accesses.

Notice also how the block size of the filesystem affects the addressing mechanism, because a larger block size allows the Ext2 to store more logical block numbers inside a single block. Table 18-11 shows the upper limit placed on a file's size for each block size and each addressing mode. For instance, if the block size is 1,024 bytes and the file contains up to 268 kilobytes of data, the first 12 KB of a file can be accessed through direct mapping and the remaining 13–268 KB can be addressed through simple indirection. Files larger than 2 GB must be opened on 32-bit architectures by specifying the `O_LARGEFILE` opening flag.

Table 18-11. File-size upper limits for data block addressing

Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1,024	12 KB	268 KB	64.26 MB	16.06 GB
2,048	24 KB	1.02 MB	513.02 MB	256.5 GB
4,096	48 KB	4.04 MB	4 GB	~ 4 TB

File Holes

A *file hole* is a portion of a regular file that contains null characters and is not stored in any data block on disk. Holes are a long-standing feature of Unix files. For instance, the following Unix command creates a file in which the first bytes are a hole:

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

Now `/tmp/hole` has 6,145 characters (6,144 null characters plus an X character), yet the file occupies just one data block on disk.

File holes were introduced to avoid wasting disk space. They are used extensively by database applications and, more generally, by all applications that perform hashing on files.

The Ext2 implementation of file holes is based on dynamic data block allocation: a block is actually assigned to a file only when the process needs to write data into it. The `i_size` field of each inode defines the size of the file as seen by the program, including the holes, while the `i_blocks` field stores the number of data blocks effectively assigned to the file (in units of 512 bytes).

In our earlier example of the `dd` command, suppose the `/tmp/hole` file was created on an Ext2 partition that has blocks of size 4,096. The `i_size` field of the corresponding disk inode stores the number 6,144, while the `i_blocks` field stores the number 8 (because each 4,096-byte block includes eight 512-byte blocks). The second element of the `i_block` array (corresponding to the block having file block number 1) stores the logical block number of the allocated block, while all other elements in the array are null (see Figure 18-6).

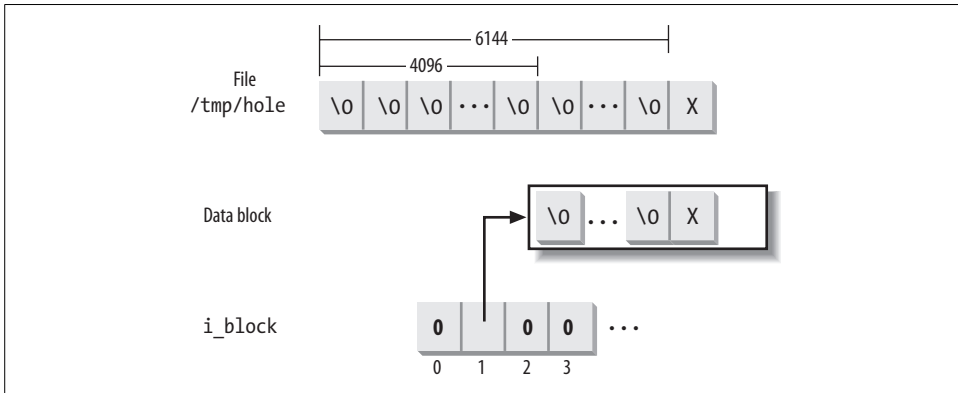


Figure 18-6. A file with an initial hole

Allocating a Data Block

When the kernel has to locate a block holding data for an Ext2 regular file, it invokes the `ext2_get_block()` function. If the block does not exist, the function automatically allocates the block to the file. Remember that this function may be invoked every time the kernel issues a read or write operation on an Ext2 regular file (see the sections “Reading from a File” and “Writing to a File” in Chapter 16); clearly, this function is invoked only if the affected block is not included in the page cache.

The `ext2_get_block()` function handles the data structures already described in the section “Data Blocks Addressing,” and when necessary, invokes the `ext2_alloc_block()` function to actually search for a free block in the Ext2 partition. If necessary, the function also allocates the blocks used for indirect addressing (see Figure 18-5).

To reduce file fragmentation, the Ext2 filesystem tries to get a new block for a file near the last block already allocated for the file. Failing that, the filesystem searches for a new block in the block group that includes the file’s inode. As a last resort, the free block is taken from one of the other block groups.

The Ext2 filesystem uses preallocation of data blocks. The file does not get only the requested block, but rather a group of up to eight adjacent blocks. The `i_prealloc_count` field in the `ext2_inode_info` structure stores the number of data blocks preallocated to a file that are still unused, and the `i_prealloc_block` field stores the logical

block number of the next preallocated block to be used. All preallocated blocks that remain unused are freed when the file is closed, when it is truncated, or when a write operation is not sequential with respect to the write operation that triggered the block preallocation.

The `ext2_alloc_block()` function receives as its parameters a pointer to an inode object, a *goal*, and the address of a variable that will store an error code. The goal is a logical block number that represents the preferred position of the new block. The `ext2_get_block()` function sets the goal parameter according to the following heuristic:

1. If the block that is being allocated and the previously allocated block have consecutive file block numbers, the goal is the logical block number of the previous block plus 1; it makes sense that consecutive blocks as seen by a program should be adjacent on disk.
2. If the first rule does not apply and at least one block has been previously allocated to the file, the goal is one of these blocks' logical block numbers. More precisely, it is the logical block number of the already allocated block that precedes the block to be allocated in the file.
3. If the preceding rules do not apply, the goal is the logical block number of the first block (not necessarily free) in the block group that contains the file's inode.

The `ext2_alloc_block()` function checks whether the goal refers to one of the preallocated blocks of the file. If so, it allocates the corresponding block and returns its logical block number; otherwise, the function discards all remaining preallocated blocks and invokes `ext2_new_block()`.

This latter function searches for a free block inside the Ext2 partition with the following strategy:

1. If the preferred block passed to `ext2_alloc_block()`—the block that is the goal—is free, the function allocates the block.
2. If the goal is busy, the function checks whether one of the next blocks after the preferred block is free.
3. If no free block is found in the near vicinity of the preferred block, the function considers all block groups, starting from the one including the goal. For each block group, the function does the following:
 - a. Looks for a group of at least eight adjacent free blocks.
 - b. If no such group is found, looks for a single free block.

The search ends as soon as a free block is found. Before terminating, the `ext2_new_block()` function also tries to preallocate up to eight free blocks adjacent to the free block found and sets the `i_prealloc_block` and `i_prealloc_count` fields of the disk inode to the proper block location and number of blocks.

Releasing a Data Block

When a process deletes a file or truncates it to 0 length, all its data blocks must be reclaimed. This is done by `ext2_truncate()`, which receives the address of the file's inode object as its parameter. The function essentially scans the disk inode's `i_block` array to locate all data blocks and all blocks used for the indirect addressing. These blocks are then released by repeatedly invoking `ext2_free_blocks()`.

The `ext2_free_blocks()` function releases a group of one or more adjacent data blocks. Besides its use by `ext2_truncate()`, the function is invoked mainly when discarding the preallocated blocks of a file (see the earlier section “Allocating a Data Block”). Its parameters are:

`inode`

The address of the inode object that describes the file

`block`

The logical block number of the first block to be released

`count`

The number of adjacent blocks to be released

The function performs the following actions for each block to be released:

1. Gets the block bitmap of the block group that includes the block to be released
2. Clears the bit in the block bitmap that corresponds to the block to be released and marks the buffer that contains the bitmap as dirty.
3. Increases the `bg_free_blocks_count` field in the block group descriptor and marks the corresponding buffer as dirty.
4. Increases the `s_free_blocks_count` field of the disk superblock, marks the corresponding buffer as dirty, and sets the `s_dirt` flag of the superblock object.
5. If the filesystem has been mounted with the `MS_SYNCHRONOUS` flag set, it invokes `sync_dirty_buffer()` and waits until the write operation on the bitmap's buffer terminates.

The Ext3 Filesystem

In this section we'll briefly describe the enhanced filesystem that has evolved from Ext2, named *Ext3*. The new filesystem has been designed with two simple concepts in mind:

- To be a journaling filesystem (see the next section)
- To be, as much as possible, compatible with the old Ext2 filesystem

Ext3 achieves both the goals very well. In particular, it is largely based on Ext2, so its data structures on disk are essentially identical to those of an Ext2 filesystem. As a

matter of fact, if an Ext3 filesystem has been cleanly unmounted, it can be remounted as an Ext2 filesystem; conversely, creating a journal of an Ext2 filesystem and remounting it as an Ext3 filesystem is a simple, fast operation.

Thanks to the compatibility between Ext3 and Ext2, most descriptions in the previous sections of this chapter apply to Ext3 as well. Therefore, in this section, we focus on the new feature offered by Ext3—“the journal.”

Journaling Filesystems

As disks became larger, one design choice of traditional Unix filesystems (such as Ext2) turns out to be inappropriate. As we know from Chapter 14, updates to filesystem blocks might be kept in dynamic memory for long period of time before being flushed to disk. A dramatic event such as a power-down failure or a system crash might thus leave the filesystem in an inconsistent state. To overcome this problem, each traditional Unix filesystem is checked before being mounted; if it has not been properly unmounted, then a specific program executes an exhaustive, time-consuming check and fixes all the filesystem’s data structures on disk.

For instance, the Ext2 filesystem status is stored in the `s_mount_state` field of the superblock on disk. The `e2fsck` utility program is invoked by the boot script to check the value stored in this field; if it is not equal to `EXT2_VALID_FS`, the filesystem was not properly unmounted, and therefore `e2fsck` starts checking all disk data structures of the filesystem.

Clearly, the time spent checking the consistency of a filesystem depends mainly on the number of files and directories to be examined; therefore, it also depends on the disk size. Nowadays, with filesystems reaching hundreds of gigabytes, a single consistency check may take hours. The involved downtime is unacceptable for every production environment or high-availability server.

The goal of a *journaling filesystem* is to avoid running time-consuming consistency checks on the whole filesystem by looking instead in a special disk area that contains the most recent disk write operations named *journal*. Remounting a journaling filesystem after a system failure is a matter of a few seconds.

The Ext3 Journaling Filesystem

The idea behind Ext3 journaling is to perform each high-level change to the filesystem in two steps. First, a copy of the blocks to be written is stored in the journal; then, when the I/O data transfer to the journal is completed (in short, data is *committed to the journal*), the blocks are written in the filesystem. When the I/O data transfer to the filesystem terminates (data is *committed to the filesystem*), the copies of the blocks in the journal are discarded.

While recovering after a system failure, the *e2fsck* program distinguishes the following two cases:

The system failure occurred before a commit to the journal. Either the copies of the blocks relative to the high-level change are missing from the journal or they are incomplete; in both cases, *e2fsck* ignores them.

The system failure occurred after a commit to the journal. The copies of the blocks are valid, and *e2fsck* writes them into the filesystem.

In the first case, the high-level change to the filesystem is lost, but the filesystem state is still consistent. In the second case, *e2fsck* applies the whole high-level change, thus fixing every inconsistency due to unfinished I/O data transfers into the filesystem.

Don't expect too much from a journaling filesystem; it ensures consistency only at the system call level. For instance, a system failure that occurs while you are copying a large file by issuing several `write()` system calls will interrupt the copy operation, thus the duplicated file will be shorter than the original one.

Furthermore, journaling filesystems do not usually copy all blocks into the journal. In fact, each filesystem consists of two kinds of blocks: those containing the so-called *metadata* and those containing regular data. In the case of Ext2 and Ext3, there are six kinds of metadata: superblocks, group block descriptors, inodes, blocks used for indirect addressing (indirection blocks), data bitmap blocks, and inode bitmap blocks. Other filesystems may use different metadata.

Several journaling filesystems, such as SGI's XFS and IBM's JFS, limit themselves to logging the operations affecting metadata. In fact, metadata's log records are sufficient to restore the consistency of the on-disk filesystem data structures. However, since operations on blocks of file data are not logged, nothing prevents a system failure from corrupting the contents of the files.

The Ext3 filesystem, however, can be configured to log the operations affecting both the filesystem metadata and the data blocks of the files. Because logging every kind of write operation leads to a significant performance penalty, Ext3 lets the system administrator decide what has to be logged; in particular, it offers three different journaling modes:

Journal

All filesystem data and metadata changes are logged into the journal. This mode minimizes the chance of losing the updates made to each file, but it requires many additional disk accesses. For example, when a new file is created, all its data blocks must be duplicated as log records. This is the safest and slowest Ext3 journaling mode.

Ordered

Only changes to filesystem metadata are logged into the journal. However, the Ext3 filesystem groups metadata and relative data blocks so that data blocks are written to disk *before* the metadata. This way, the chance to have data corruption inside the files is reduced; for instance, each write access that enlarges a file

is guaranteed to be fully protected by the journal. This is the default Ext3 journaling mode.

Writeback

Only changes to filesystem metadata are logged; this is the method found on the other journaling filesystems and is the fastest mode.

The journaling mode of the Ext3 filesystem is specified by an option of the *mount* system command. For instance, to mount an Ext3 filesystem stored in the */dev/sda2* partition on the */jdisk* mount point with the “writeback” mode, the system administrator can type the command:

```
# mount -t ext3 -o data=writeback /dev/sda2 /jdisk
```

The Journaling Block Device Layer

The Ext3 journal is usually stored in a hidden file named *.journal* located in the root directory of the filesystem.

The Ext3 filesystem does not handle the journal on its own; rather, it uses a general kernel layer named *Journaling Block Device*, or *JBD*. Right now, only Ext3 uses the JBD layer, but other filesystems might use it in the future.

The JBD layer is a rather complex piece of software. The Ext3 filesystem invokes the JBD routines to ensure that its subsequent operations don't corrupt the disk data structures in case of system failure. However, JBD typically uses the same disk to log the changes performed by the Ext3 filesystem, and it is therefore vulnerable to system failures as much as Ext3. In other words, JBD must also protect itself from system failures that could corrupt the journal.

Therefore, the interaction between Ext3 and JBD is essentially based on three fundamental units:

Log record

Describes a single update of a disk block of the journaling filesystem.

Atomic operation handle

Includes log records relative to a single high-level change of the filesystem; typically, each system call modifying the filesystem gives rise to a single atomic operation handle.

Transaction

Includes several atomic operation handles whose log records are marked valid for *e2fsck* at the same time.

Log records

A *log record* is essentially the description of a low-level operation that is going to be issued by the filesystem. In some journaling filesystems, the log record consists of exactly the span of bytes modified by the operation, together with the starting position

of the bytes inside the filesystem. The JBD layer, however, uses log records consisting of the whole buffer modified by the low-level operation. This approach may waste a lot of journal space (for instance, when the low-level operation just changes the value of a bit in a bitmap), but it is also much faster because the JBD layer can work directly with buffers and their buffer heads.

Log records are thus represented inside the journal as normal blocks of data (or metadata). Each such block, however, is associated with a small tag of type `journal_block_tag_t`, which stores the logical block number of the block inside the filesystem and a few status flags.

Later, whenever a buffer is being considered by the JBD, either because it belongs to a log record or because it is a data block that should be flushed to disk before the corresponding metadata block (in the “ordered” journaling mode), the kernel attaches a `journal_head` data structure to the buffer head. In this case, the `b_private` field of the buffer head stores the address of the `journal_head` data structure and the `BH_JBD` flag is set (see the section “Block Buffers and Buffer Heads” in Chapter 15).

Atomic operation handles

Every system call modifying the filesystem is usually split into a series of low-level operations that manipulate disk data structures.

For instance, suppose that Ext3 must satisfy a user request to append a block of data to a regular file. The filesystem layer must determine the last block of the file, locate a free block in the filesystem, update the data block bitmap inside the proper block group, store the logical number of the new block either in the file’s inode or in an indirect addressing block, write the contents of the new block, and finally, update several fields of the inode. As you see, the append operation translates into many lower-level operations on the data and metadata blocks of the filesystem.

Now, just imagine what could happen if a system failure occurred in the middle of an append operation, when some of the lower-level manipulations have already been executed while others have not. Of course, the scenario could be even worse, with high-level operations affecting two or more files (for example, moving a file from one directory to another).

To prevent data corruption, the Ext3 filesystem must ensure that each system call is handled in an atomic way. An *atomic operation handle* is a set of low-level operations on the disk data structures that correspond to a single high-level operation. When recovering from a system failure, the filesystem ensures that either the whole high-level operation is applied or none of its low-level operations is.

Each atomic operation handle is represented by a descriptor of type `handle_t`. To start an atomic operation, the Ext3 filesystem invokes the `journal_start()` JBD function, which allocates, if necessary, a new atomic operation handle and inserts it into the current transactions (see the next section). Because every low-level operation on the disk might suspend the process, the address of the active handle is stored in the

journal_info field of the process descriptor. To notify that an atomic operation is completed, the Ext3 filesystem invokes the journal_stop() function.

Transactions

For reasons of efficiency, the JBD layer manages the journal by grouping the log records that belong to several atomic operation handles into a single *transaction*. Furthermore, all log records relative to a handle must be included in the same transaction.

All log records of a transaction are stored in consecutive blocks of the journal. The JBD layer handles each transaction as a whole. For instance, it reclaims the blocks used by a transaction only after all data included in its log records is committed to the filesystem.

As soon as it is created, a transaction may accept log records of new handles. The transaction stops accepting new handles when either of the following occurs:

- A fixed amount of time has elapsed, typically 5 seconds.
- There are no free blocks in the journal left for a new handle.

A transaction is represented by a descriptor of type transaction_t. The most important field is t_state, which describes the current status of the transaction.

Essentially, a transaction can be:

Complete

All log records included in the transaction have been physically written onto the journal. When recovering from a system failure, *e2fsck* considers every complete transaction of the journal and writes the corresponding blocks into the filesystem. In this case, the t_state field stores the value T_FINISHED.

Incomplete

At least one log record included in the transaction has not yet been physically written to the journal, or new log records are still being added to the transaction. In case of system failure, the image of the transaction stored in the journal is likely not up-to-date. Therefore, when recovering from a system failure, *e2fsck* does not trust the incomplete transactions in the journal and skips them. In this case, the t_state field stores one of the following values:

T_RUNNING

Still accepting new atomic operation handles.

T_LOCKED

Not accepting new atomic operation handles, but some of them are still unfinished.

T_FLUSH

All atomic operation handles have finished, but some log records are still being written to the journal.

T_COMMIT

All log records of the atomic operation handles have been written to disk, but the transaction has yet to be marked as completed on the journal.

At any time the journal may include several transactions, but only one of them is in the T_RUNNING state—it is the *active transaction* that is accepting the new atomic operation handle requests issued by the Ext3 filesystem.

Several transactions in the journal might be incomplete, because the buffers containing the relative log records have not yet been written to the journal.

If a transaction is complete, all its log records have been written to the journal but some of the corresponding buffers have yet to be written onto the filesystem. A complete transaction is deleted from the journal when the JBD layer verifies that all buffers described by the log records have been successfully written onto the Ext3 filesystem.

How Journaling Works

Let's try to explain how journaling works with an example: the Ext3 filesystem layer receives a request to write some data blocks of a regular file.

As you might easily guess, we are not going to describe in detail every single operation of the Ext3 filesystem layer and of the JBD layer. There would be far too many issues to be covered! However, we describe the essential actions:

1. The service routine of the `write()` system call triggers the `write` method of the file object associated with the Ext3 regular file. For Ext3, this method is implemented by the `generic_file_write()` function, already described in the section “Writing to a File” in Chapter 16.
2. The `generic_file_write()` function invokes the `prepare_write` method of the `address_space` object several times, once for every page of data involved by the write operation. For Ext3, this method is implemented by the `ext3_prepare_write()` function.
3. The `ext3_prepare_write()` function starts a new atomic operation by invoking the `journal_start()` JBD function. The handle is added to the active transaction. Actually, the atomic operation handle is created only when executing the first invocation of the `journal_start()` function. Following invocations verify that the `journal_info` field of the process descriptor is already set and use the referenced handle.
4. The `ext3_prepare_write()` function invokes the `block_prepare_write()` function already described in Chapter 16, passing to it the address of the `ext3_get_block()` function. Remember that `block_prepare_write()` takes care of preparing the buffers and the buffer heads of the file's page.

5. When the kernel must determine the logical number of a block of the Ext3 filesystem, it executes the `ext3_get_block()` function. This function is actually similar to `ext2_get_block()`, which is described in the earlier section “Allocating a Data Block.” A crucial difference, however, is that the Ext3 filesystem invokes functions of the JBD layer to ensure that the low-level operations are logged:

- *Before* issuing a low-level write operation on a metadata block of the filesystem, the function invokes `journal_get_write_access()`. Basically, this latter function adds the metadata buffer to a list of the active transaction. However, it must also check whether the metadata is included in an older incomplete transaction of the journal; in this case, it duplicates the buffer to make sure that the older transactions are committed with the old content.
- *After* updating the buffer containing the metadata block, the Ext3 filesystem invokes `journal_dirty_metadata()` to move the metadata buffer to the proper dirty list of the active transaction and to log the operation in the journal.

Notice that metadata buffers handled by the JBD layer are not usually included in the dirty lists of buffers of the inode, so they are not written to disk by the normal disk cache flushing mechanisms described in Chapter 15.

6. If the Ext3 filesystem has been mounted in “journal” mode, the `ext3_prepare_write()` function also invokes `journal_get_write_access()` on every buffer touched by the write operation.

7. Control returns to the `generic_file_write()` function, which updates the page with the data stored in the User Mode address space and then invokes the `commit_write` method of the `address_space` object. For Ext3, the function that implements this method depends on how the Ext3 filesystem has been mounted:

- If the Ext3 filesystem has been mounted in “journal” mode, the `commit_write` method is implemented by the `ext3_journalled_commit_write()` function, which invokes `journal_dirty_metadata()` on every buffer of data (not metadata) in the page. This way, the buffer is included in the proper dirty list of the active transaction and not in the dirty list of the owner inode; moreover, the corresponding log records are written to the journal. Finally, `ext3_journalled_commit_write()` invokes `journal_stop()` to notify the JBD layer that the atomic operation handle is closed.
- If the Ext3 filesystem has been mounted in “ordered” mode, the `commit_write` method is implemented by the `ext3_ordered_commit_write()` function, which invokes the `journal_dirty_data()` function on every buffer of data in the page to insert the buffer in a proper list of the active transactions. The JBD layer ensures that all buffers in this list are written to disk before the metadata buffers of the transaction. No log record is written onto the journal. Next, `ext3_ordered_commit_write()` executes the normal `generic_commit_write()` function described in Chapter 15, which inserts the

data buffers in the list of the dirty buffers of the owner inode. Finally, `ext3_ordered_commit_write()` invokes `journal_stop()` to notify the JBD layer that the atomic operation handle is closed.

- If the Ext3 filesystem has been mounted in “writeback” mode, the `commit_write` method is implemented by the `ext3_writeback_commit_write()` function, which executes the normal `generic_commit_write()` function described in Chapter 15, which inserts the data buffers in the list of the dirty buffers of the owner inode. Then, `ext3_writeback_commit_write()` invokes `journal_stop()` to notify the JBD layer that the atomic operation handle is closed.
8. The service routine of the `write()` system call terminates here. However, the JBD layer has not finished its work. Eventually, our transaction becomes complete when all its log records have been physically written to the journal. Then `journal_commit_transaction()` is executed.
 9. If the Ext3 filesystem has been mounted in “ordered” mode, the `journal_commit_transaction()` function activates the I/O data transfers for all data buffers included in the list of the transaction and waits until all data transfers terminate.
 10. The `journal_commit_transaction()` function activates the I/O data transfers for all metadata buffers included in the transaction (and also for all data buffers, if Ext3 was mounted in “journal” mode).
 11. Periodically, the kernel activates a checkpoint activity for every complete transaction in the journal. The checkpoint basically involves verifying whether the I/O data transfers triggered by `journal_commit_transaction()` have successfully terminated. If so, the transaction can be deleted from the journal.

Of course, the log records in the journal never play an active role until a system failure occurs. Only during system reboot does the *e2fsck* utility program scan the journal stored in the filesystem and reschedule all write operations described by the log records of the complete transactions.