The Ext2 File System

Table of Contents

Introduction

The Second Extended Filesystem (Ext2) was introduced in 1994, to substitute the extedended File System (ext FS), which offered unsatisfactory performance. Ext2 is quite efficient and robust and has become the most widely used Linux filesystem. The following features contribute to the efficiency of Ext2:

- When creating an Ext2 filesystem, the optimal block size (from 1,024 to 4,096 bytes) can be chosen, 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. Larger block sizes are instead preferable when the average file length is expected to be greater than a few thousand bytes. 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 Ext2 file system divides the logical partition that it occupies into Block 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 are supported. If the pathname of the symbolic link is less than 60 bytes long, it is stored in the inode and can thus be translated without reading a data block.

The ext2 File System also includes other features that make it both robust and flexible:

- A careful implementation of the file-updating strategy 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 incremented 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 mountings (a counter is incremented 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). The second option can be useful for log files.
- Compatibility with both the Unix System V Release 4 and the BSD semantics of the Group ID for a new file. In SVR4, the new file assumes the Group ID of the process that creates it; in BSD, the new file inherits the Group ID of the directory containing it. Ext2 includes a mount option that specifies which semantic is used.

The Ext2 filesystem is a mature, stable program, and it has not evolved significantly in recent years. Several additional features, however, 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.

Access Control Lists (ACL)

Instead of classifying the users of a file under three classes—owner, group, and others—this list is associated with each file to specify the access rights for any specific users or combinations of users.

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.

The following sections describe in more detail the Ext2 filesystem.

The Ext2 inode

In the Ext2 file system, the inode is the basic building block; every file and directory in the file system is described by one and only one inode. The Ext2 inodes for each Block Group are kept in the inode table together with a bitmap that allows the system to keep track of allocated and unallocated inodes. Figure 1 shows the format of an Ext2 inode, amongst other information, it contains the following fields:

mode

This holds two pieces of information; what does this inode describe and the permissions that users have to it. For Ext2, an inode can describe one of file, directory, symbolic link, block device, character device or FIFO.

Owner Information

The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size

The size of the file in bytes.

Timestamps

The time that the inode was created and the last time that it was modified.

Datablocks

Pointers to the blocks that contain the data that this inode is describing. The first twelve are pointers to the physical blocks containing the data described by this inode and the last three pointers contain more and more levels of indirection. For example, the double indirect blocks pointer points at a block of pointers to blocks of pointers to data blocks. This means that files less than or equal to twelve data blocks in length are more quickly accessed than larger files.

You should note that Ext2 inodes can describe special device files. These are not real files but handles that programs can use to access devices. All of the device files in /dev are there to allow programs to access Linux's devices. For example the mount program takes the device file that it wishes to mount as an argument.

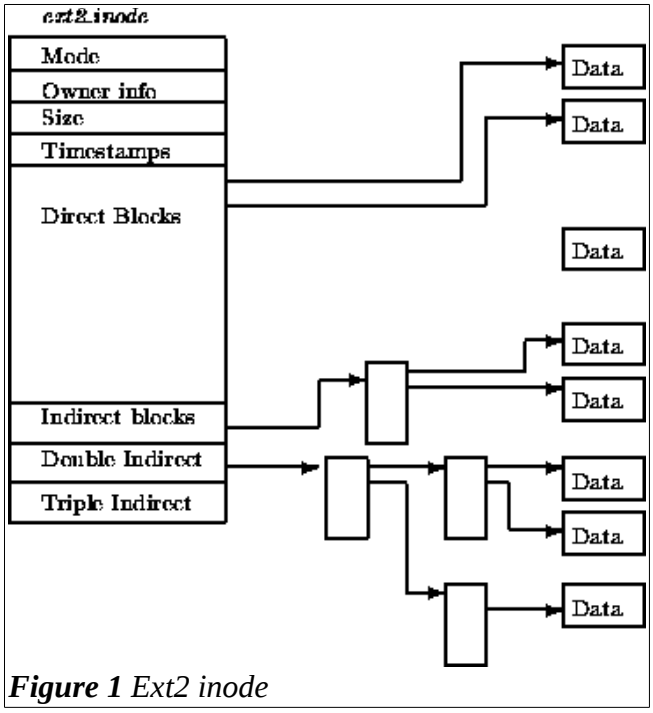

Figure 1 *Ext2 inode*

Disk data Structure

In any Ext2 partition, the first block is reserved for the partition boot sector. The rest of the Ext2 partition is split into *block groups* (see Figure 2). All block groups in the filesystem have the same size and are stored sequentially. This allows the kernel to easily derive the location of a block group in a disk from its integer index. Moreover, some data structures must fit in exactly one block, while others may require more than one block

Block group splitting reduce file fragmentation, since the kernel tries to keep the data blocks belonging to a file in the same block group, if possible. Every block group contains the following pieces of information:

- A copy of the filesystem's superblock
- A copy of the block group descriptors
- A data block bitmap which is used to identify the free blocks inside the group
- An inode bitmap, which is used to identify the free inodes inside the group
- inode table: it consists of a series of consecutive blocks, each of which contains a predefined

number of inodes. 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. Note that 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

- data blocks, containing files. Any block which does not contain any meaningful information, it is said to be *free*.

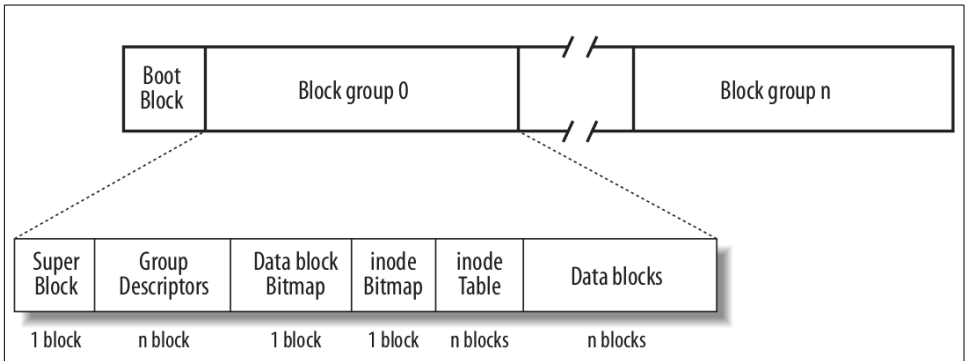

***Figure 2** Ext2 partition layout*

Note that although 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 becomes 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.

As concerns the block group number in a partition, it depends both on the partition size and the block size. This constraint depends on the fact that the block bitmap must be stored in a single block. Therefore, in each block group, there can be at most 8×b blocks, where b is the block size in bytes. Thus, the total number of block groups is roughly s/(8×b), where s is the partition size in blocks. For example, if we consider an 8 GB Ext2 partition with a 4-KB block size, each 4-KB block bitmap describes 32K data blocks—that is, 128 MB. Therefore, at most 64 block groups are needed. Clearly, the smaller the block size, the larger the number of block groups.

The superblock

The Superblock contains a description of the basic size and shape of this file system. Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption. Amongst other information it holds the:

Magic Number

This allows the mounting software to check that this is indeed the Superblock for an Ext2 file system. For the current version of Ext2 this is 0xEF53.

Revision Level

The major and minor revision levels allow the mounting code to determine whether or not this file system supports features that are only available in particular revisions of the file system. There are also feature compatibility fields which help the mounting code to determine which new features can safely be used on this file system.

Mount Count and Maximum Mount Count

Together these allow the system to determine if the file system should be fully checked. The mount count is incremented each time the file system is mounted and when it equals the maximum mount count the warning message ``maximal mount count reached, running e2fsck is recommended'' is displayed.

Block Group Number

The Block Group number that holds this copy of the Superblock,

Block Size

The size of the block for this file system in bytes, for example 1024 bytes,

Blocks per Group

The number of blocks in a group. Like the block size this is fixed when the file system is created,

Free Blocks

The number of free blocks in the file system,

Free inodes

The number of free inodes in the file system,

First inode

This is the inode number of the first inode in the file system. The first inode in an Ext2 root file system would be the directory entry for the '/' directory.

The Group Descriptor

Each Block Group has a data structure describing it. Like the Superblock, all the group descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption. Each Group Descriptor contains the following information:

Blocks Bitmap

The block number of the block bitmap for this Block Group. This is used during block allocation and deallocation.

inode Bitmap

The block number of the inode allocation bitmap for this Block Group. This is used during inode allocation and deallocation.

inode Table

The block number of the starting block for the inode table for this Block Group. Each inode is represented by the Ext2 inode data structure described below.

Free blocks count, Free Inodes count, Used directory count

The name of this fields is self-explanatory

Note that the group descriptors are placed one after another and together they make the group descriptor table. Each Blocks Group contains the entire table of group descriptors after its copy of the Superblock. Only the first copy (in Block Group 0) is actually used by the Ext2 file system. The other copies are there, like the copies of the Superblock, in case the main copy is corrupted.

Directories

Ext2 implements directories as a special kind of file, which contain file names together with the corresponding inode numbers. A directory file is a list of directory variable length entries, each one containing the following information:

inode number

The inode number for this directory entry. This entry is 4 byte long.

Entry length

This entry is 2 bytes long and contains the length of this directory entry in bytes. Note that this 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. In the example shown in Figure ??, the fifth entry of Figure ?? (describing the file named “oldfile”) was deleted because the field of usr is set to 12+16 (the lengths of the usr and oldfile entries).

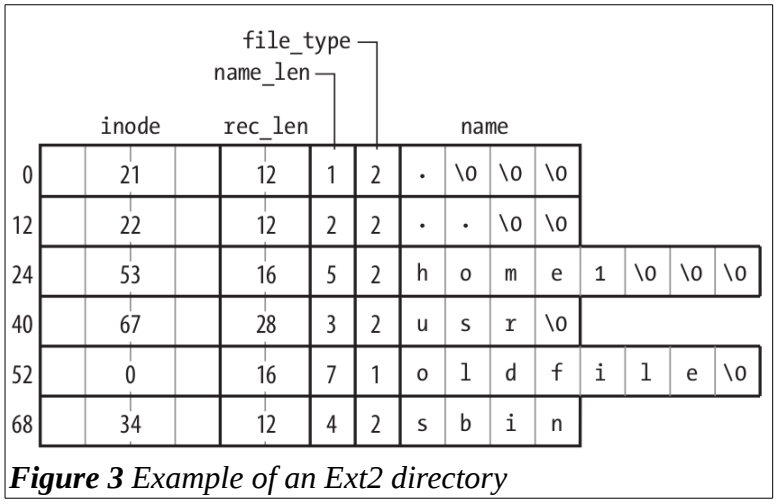

Figure 3 *Example of an Ext2 directory*

Name length

The length of the file name (1 byte)

File type

The type of file (1 byte). Eigth type of files are possible: unknown , regular file, directory , character device , block device , named pipe , socket , symbolic link.

The name of this directory entry

This 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.

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, slow, 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 free inodes count field in the superblock and of the free inodes count field in the proper group descriptor must be decremented.
- 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 free blocks count field in the superblock and of the 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 last write time field of the Ext2 superblock.

Since all Ext2 disk data structures are stored in blocks of the Ext2 partition, the kernel uses the buffer cache and the page cache to keep them up to date. Table 2 specifies, for each type of data related to Ext2 filesystems, the way they are cached in memory. As one may expect, very frequently updated data is always cached; that is, the data is permanently stored in memory and included in the buffer cache or in the page cache until the corresponding Ext2 partition is unmounted. Thus it is

never necessary to read the this data from disk (periodically, however, the data must be written back to disk). As concerns the *dynamic* mode, instead, 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. Finally, the never-cached data, is not kept in any cache since it does not represent meaningful information. In between these extremes lies the.
It is worth noting 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.

Type	Caching mode
Superblock	Always cached
Group descriptor	Always cached
Block bitmap	Dynamic
Inode bitmap	Dynamic
Inode	Dynamic
Data block	Dynamic
Free inode	Never
Free block	Never

Table 1 *Caching mode of the Ext2 data structures*

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 earlier . 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 us consider, for example, how an Ext2 1.44 MB floppy disk is initialized by mke2fs with the default options. Once mounted, it appears as a volume consisting of 1,412 blocks; each one is 1,024 bytes in length. The floppy disk layout after the Ext2 file system creation is shown in Table 2.

Block	Content
0	Boot block
1	Superblock
2	Block containing a single block group descriptor
3	Data block bitmap
4	inode bitmap
5-27	inode table: inodes up to 10: reserved (inode 2 is the root); inode 11: lost+found; inodes 12–184: free
28	Root directory (includes . , .. , and lost+found)
29	lost+found directory (includes . and ..)
30-40	Reserved blocks preallocated for lost+found directory
41-1439	Free blocks

Table 2 *Ext2 block allocation for a floppy disk (default options)*

Finding a File in an Ext2 File System

A Linux filename has the same format as all Unix filenames have. It is a series of directory names seperated by forward slashes (“/”) and ending in the file's name. One example filename would be “/home/francesco/myfile.txt” where /home and /francesco are directory names and the file's name is myfile.txt. Note that the in Linux the filename can be of any length and contain any printable character. To find the inode representing this file within an Ext2 file system the system must parse the filename a directory at a time until we get to the file itself.
The first inode that we need is the inode number for the root of the file system, which is available in the file system's superblock. To read an Ext2 inode we must look for it in the inode table of the appropriate Block Group. If, for example, the root inode number is 42 then we need the 42nd inode from the inode table of Block Group 0. The root inode describes a directory and it's data blocks contain Ext2 directory entries. Home is just one of the many directory entries and this directory entry gives us the number of the inode describing the /home directory. We have to read this directory (by first reading its inode and then reading the directory entries from the data blocks described by its inode) to find the francesco entry which gives us the number of the inode describing the /home/francesco directory. Afterwards, we read the directory entries contained in the blocks pointed indexed by the inode describing the /home/francesco directory to find the inode number of the myfile.txt file. Finally, from this inode we get the data blocks containing the information in the file.

File size changing

One common problem with a file system is its tendency to fragment. The blocks that hold the file's data get spread all over the file system and this makes sequentially accessing the data blocks of a file more and more inefficient the further apart the data blocks are. The Ext2 file system tries to overcome this by allocating the new blocks for a file physically close to its current data blocks or at least in the same Block Group as its current data blocks. Only when this fails does it allocate data blocks in another Block Group.
Whenever a process attempts to write data into a file the Linux file system checks to see if the data has gone off the end of the file's last allocated block. If it has, then it must allocate a new data block for this file. Until the allocation is complete, the process cannot run, it must wait for the file system to allocate a new data block and write the rest of the data to it before it can continue. The first thing that the Ext2 block allocation routines do is to lock the Ext2 Superblock for this file system.

Allocating and deallocating changes fields within the superblock and the Linux file system cannot allow more than one process to do this at the same time. If another process needs to allocate more data blocks then it will have to wait until this process has finished. Processes waiting for the superblock are suspended, unable to run, until control of the superblock is relinquished by its current user. Access to the superblock is granted on a first come, first served basis and once a process has control of the superblock then it keeps control until it has finished. Having locked the superblock, the process checks that there are enough free blocks left in this file system. If there are not enough free blocks then this attempt to allocate more will fail and the process will relinquish control of this file system's superblock. If there are enough free blocks in the file system, the process tries to allocate one.
If the Ext2 file system has been built to preallocate data blocks then we may be able to take one of those. The preallocated blocks do not actually exist, they are just reserved within the allocated block bitmap. The VFS inode representing the file that we are trying to allocate a new data block for has two Ext2 specific fields, prealloc_block and prealloc_count which are the block number of the first preallocated data block and how many of them there are respectively. If there were no preallocated blocks or block preallocation is not enabled, the Ext2 file system must allocate a new block. The Ext2 file system first looks to see if the data block after the last data block in the file is free. Logically, this is the most efficient block to allocate as it makes sequential accesses much quicker. If this block is not free, then the search widens and it looks for a data block within 64 blocks of the ideal block. This block, although not ideal is at least fairly close and within the same Block Group of the other data blocks belonging to this file.
If even that block is not free, the process starts looking in all of the other Block Groups in turn until it finds some free blocks. The block allocation code looks for a cluster of eight free data blocks somewhere in one of the Block Groups. If it cannot find eight together, it will settle for less. If block preallocation is wanted and enabled it will update prealloc_block and prealloc_count accordingly.
Wherever it found the free block, the block allocation code updates the Block Group's block bitmap and allocates a data buffer in the buffer cache. That data buffer is uniquely identified by the file system's supporting device identifier and the block number of the allocated block. The data in the buffer is zero'd and the buffer is marked as ``dirty'' to show that it's contents have not been written to the physical disk. Finally, the superblock itself is marked as ``dirty'' to show that it has been changed and it is unlocked. If there were any processes waiting for the superblock, the first one in the queue is allowed to run again and will gain exclusive control of the superblock for its file operations. The process's data is written to the new data block and, if that data block is filled, the entire process is repeated and another data block allocated.

The Ext3 Filesystem

The Ext3 file system is an enhanced filesystem that has evolved from Ext2. Developers had two main objectives, while designing the Ext3 file system: to be a journaling filesystem; 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. In practice, cleanly unmounted Ext3 filesystems can be remounted as an Ext2 filesystems; 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 also apply to Ext3.

Journaling file systems

Updates to filesystem blocks might be kept in dynamic memory for long period of time before being flushed to disk. Events 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 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 the filesystem was not properly unmounted, the e2fsck starts checking all disk data structures of the filesystem. The time spent depends on the number of files and directories to be examined and mainly on the disk size. With filesystems reaching hundreds of gigabytes, a single consistency check may take hours, a downtime which may result unacceptable for many systems. 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.

It is worth noting that journaling 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
```