The Second Extended File System

Internal Layout

Dave Poirier

<instinc@gmail.com>

Table of Contents

List of Tables

List of Figures

About this book

The latest version of this document may be downloaded from http://www.freesoftware.fsf.org/ext2-doc/

This book is intended as an introduction and guide to the Second Extended File System, also known as Ext2. The reader should have a good understanding of the purpose of a file system as well as the associated vocabulary (file, directory, partition, etc).

Implementing file system drivers is already a daunting task, unfortunately except for tidbits of information here and there most of the documentation for the Second Extended Filesystem is in source files.

Hopefully this document will fix this problem, may it be of help to as many of you as possible.

Unless otherwise stated, all values are stored in little endian byte order.

Chapter 1. Historical Background

Written by Remy Card, Theodore Ts'o and Stephen Tweedie as a major rewrite of the Extended Filesystem, it was first released to the public on January 1993 as part of the Linux kernel. One of its greatest achievement is the ability to extend the file system functionalities while maintaining the internal structures. This allowed an easier development of the Third Extended Filesystem (ext3) and the Fourth Extended Filesystem (ext4).

There are implementations available in most operating system including but not limited to NetBSD, FreeBSD, the GNU HURD, Microsoft Windows, IBM OS/2 and RISC OS.

Although newer file systems have been designed, such as Ext3 and Ext4, the Second Extended Filesystem is still prefered on flash drives as it requires fewer write operations (since it has no journal). The structures of Ext3 and Ext4 are based on Ext2 and add some additional options such as journaling, journal checksums, extents, online defragmentation, delayed allocations and larger directories to name but a few.

Chapter 2. Definitions

The Second Extended Filesystem uses blocks as the basic unit of storage, inodes as the mean of keeping track of files and system objects, block groups to logically split the disk into more manageable sections, directories to provide a hierarchical organization of files, block and inode bitmaps to keep track of allocated blocks and inodes, and superblocks to define the parameters of the file system and its overall state.

Ext2 shares many properties with traditional Unix filesystems. It has space in the specification for Access Control Lists (ACLs), fragments, undeletion and compression. There is also a versioning mechanism to allow new features (such as journalling) to be added in a maximally compatible manner; such as in Ext3 and Ext4.

2.1. Blocks

A partition, disk, file or block device formated with a Second Extended Filesystem is divided into small groups of sectors called "blocks". These blocks are then grouped into larger units called block groups.

The size of the blocks are usually determined when formatting the disk and will have an impact on performance, maximum file size, and maximum file system size. Block sizes commonly implemented include 1KiB, 2KiB, 4KiB and 8KiB although provisions in the superblock allow for block sizes as big as 1024 * (2^31)-1
(see s_log_block_size).

Depending on the implementation, some architectures may impose limits on which block sizes are supported. For example, a Linux 2.6 implementation on DEC Alpha uses blocks of 8KiB but the same implementation on a Intel 386 processor will support a maximum block size of 4KiB.

Table 2-1. Impact of Block Sizes

Upper Limits	1KiB	2KiB	4KiB	8KiB
file system blocks	2,147,483,647	2,147,483,647	2,147,483,647	2,147,483,647
blocks per block group	8,192	16,384	32,768	65,536
inodes per block group	8,192	16,384	32,768	65,536
bytes per block group	8,388,608 (8MiB)	33,554,432 (32MiB)	134,217,728 (128MiB)	536,870,912 (512MiB)
file system size (real)	4,398,046,509,056 (4TiB)	8,796,093,018,112 (8TiB)	17,592,186,036,224 (16TiB)	35,184,372,080,640 (32TiB)
file system size (Linux)	2,199,023,254,528 (2TiB) [a]	8,796,093,018,112 (8TiB)	17,592,186,036,224 (16TiB)	35,184,372,080,640 (32TiB)
blocks per file	16,843,020	134,217,728	1,074,791,436	8,594,130,956
file size (real)	17,247,252,480 (16GiB)	274,877,906,944 (256GiB)	2,199,023,255,552 (2TiB)	2,199,023,255,552 (2TiB)
file size (Linux 2.6.28)	17,247,252,480 (16GiB)	274,877,906,944 (256GiB)	2,199,023,255,552 (2TiB)	2,199,023,255,552 (2TiB)

Notes:
a. This limit comes from the maximum size of a block device in Linux 2.4; it is unclear whether a Linux 2.6 kernel using a 1KiB block size could properly format and mount a Ext2 partition larger than 2TiB.

Note: the 2TiB file size is limited by the i_blocks value in the inode which indicates the number of 512-bytes sector rather than the actual number of ext2 blocks allocated.

2.2. Block Groups

This definition comes from the Linux Kernel Documentation.

Blocks are clustered into block groups in order to reduce fragmentation and minimise the amount of head seeking when reading a large amount of consecutive data. Information about each block group is kept in a descriptor table stored in the block(s) immediately after the superblock. Two blocks near the start of each group are reserved for the block usage bitmap and the inode usage bitmap which show which blocks and inodes are in use. Since each bitmap is limited to a single block, this means that the maximum size of a block group is 8 times the size of a block.

The block(s) following the bitmaps in each block group are designated as the inode table for that block group and the remainder are the data blocks. The block allocation algorithm attempts to allocate data blocks in the same block group as the inode which contains them.

2.3. Directories

This definition comes from the Linux Kernel Documentation with some minor alterations.

A directory is a filesystem object and has an inode just like a file. It is a specially formatted file containing records which associate each name with an inode number. Later revisions of the filesystem also encode the type of the object (file, directory, symlink, device, fifo, socket) to avoid the need to check the inode itself for this information

The inode allocation code should try to assign inodes which are in the same block group as the directory in which they are first created.

The original Ext2 revision used singly-linked list to store the filenames in the directory; newer revisions are able to use hashes and binary trees.

Also note that as directory grows additional blocks are assigned to store the additional file records. When filenames are removed, some implementations do not free these additional blocks.

2.4. Inodes

This definition comes from the Linux Kernel Documentation with some minor alterations.

The inode (index node) is a fundamental concept in the ext2 filesystem. Each object in the filesystem is represented by an inode. The inode structure contains pointers to the filesystem blocks which contain the data held in the object and all of the metadata about an object except its name. The metadata about an object includes the permissions, owner, group, flags, size, number of blocks used, access time, change time, modification time, deletion time, number of links, fragments, version (for NFS) and extended attributes (EAs) and/or Access Control Lists (ACLs).

There are some reserved fields which are currently unused in the inode structure and several which are overloaded. One field is reserved for the directory ACL if the inode is a directory and alternately for the top 32 bits of the file size if the inode is a regular file (allowing file sizes larger than 2GB). The translator field is unused under Linux, but is used by the HURD to reference the inode of a program which will be used to interpret this object. Most of the remaining reserved fields have been used up for both Linux and the HURD for larger owner and group fields, The HURD also has a larger mode field so it uses another of the remaining fields to store the extra bits.

There are pointers to the first 12 blocks which contain the file's data in the inode. There is a pointer to an indirect block (which contains pointers to the next set of blocks), a pointer to a doubly-indirect block (which contains pointers to indirect blocks) and a pointer to a trebly-indirect block (which contains pointers to doubly-indirect blocks).

Some filesystem specific behaviour flags are also stored and allow for specific filesystem behaviour on a per-file basis. There are flags for secure deletion, undeletable, compression, synchronous updates, immutability, append-only, dumpable, no-atime, indexed directories, and data-journaling.

Many of the filesystem specific behaviour flags, like journaling, have been implemented in newer filesystems like Ext3 and Ext4, while some other are still under development.

All the inodes are stored in inode tables, with one inode table per block group.

2.5. Superblocks

This definition comes from the Linux Kernel Documentation with some minor alterations.

The superblock contains all the information about the configuration of the filesystem. The information in the superblock contains fields such as the total number of inodes and blocks in the filesystem and how many are free, how many inodes and blocks are in each block group, when the filesystem was mounted (and if it was cleanly unmounted), when it was modified, what version of the filesystem it is and which OS created it.

The primary copy of the superblock is stored at an offset of 1024 bytes from the start of the device, and it is essential to mounting the filesystem. Since it is so important, backup copies of the superblock are stored in block groups throughout the filesystem.

The first version of ext2 (revision 0) stores a copy at the start of every block group, along with backups of the group descriptor block(s). Because this can consume a considerable amount of space for large filesystems, later revisions can optionally reduce the number of backup copies by only putting backups in specific groups (this is the sparse superblock feature). The groups chosen are 0, 1 and powers of 3, 5 and 7.

Revision 1 and higher of the filesystem also store extra fields, such as a volume name, a unique identification number, the inode size, and space for optional filesystem features to store configuration info.

All fields in the superblock (as in all other ext2 structures) are stored on the disc in little endian format, so a filesystem is portable between machines without having to know what machine it was created on.

2.6. Symbolic Links

This definition comes from Wikipedia.org with some minor alterations.

A symbolic link (also symlink or soft link) is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path and that affects pathname resolution.

Symbolic links operate transparently for most operations: programs which read or write to files named by a symbolic link will behave as if operating directly on the target file. However, programs that need to handle symbolic links specially (e.g., backup utilities) may identify and manipulate them directly.

A symbolic link merely contains a text string that is interpreted and followed by the operating system as a path to another file or directory. It is a file on its own and can exist independently of its target. The symbolic links do not affect an inode link count. If a symbolic link is deleted, its target remains unaffected. If the target is moved, renamed or deleted, any symbolic link that used to point to it continues to exist but now points to a non-existing file. Symbolic links pointing to non-existing files are sometimes called "orphaned" or "dangling".

Symbolic links are also filesystem objects with inodes. For all symlink shorter than 60 bytes long, the data is stored within the inode itself; it uses the fields which would normally be used to store the pointers to data blocks. This is a worthwhile optimisation as it we avoid allocating a full block for the symlink, and most symlinks are less than 60 characters long.

Symbolic links can also point to files or directories of other partitions and file systems.

Chapter 3. Disk Organization

An Ext2 file systems starts with a superblock located at byte offset 1024 from the start of the volume. This is block 1 for a 1KiB block formatted volume or within block 0 for larger block sizes. Note that the size of the superblock is constant regardless of the block size.

On the next block(s) following the superblock, is the Block Group Descriptor Table; which provides an overview of how the volume is split into block groups and where to find the inode bitmap, the block bitmap, and the inode table for each block group.

In revision 0 of Ext2, each block group consists of a copy superblock, a copy of the block group descriptor table, a block bitmap, an inode bitmap, an inode table, and data blocks.

With the introduction of revision 1 and the sparse superblock feature in Ext2, only specific block groups contain copies of the superblock and block group descriptor table. All block groups still contain the block bitmap, inode bitmap, inode table, and data blocks. The shadow copies of the superblock can be located in block groups 0, 1 and powers of 3, 5 and 7.

The block bitmap and inode bitmap are limited to 1 block each per block group, so the total blocks per block group is therefore limited. (More information in the Block Size Impact table).

Each data block may also be further divided into "fragments". As of Linux 2.6.28, support for fragment was still not implemented in the kernel; it is therefore suggested to ensure the fragment size is equal to the block size so as to maintain compatibility.

Table 3-1. Sample Floppy Disk Layout, 1KiB blocks

Block Offset	Length	Description
byte 0	512 bytes	boot record (if present)
byte 512	512 bytes	additional boot record data (if present)
-- block group 0, blocks 1 to 1439 --		
byte 1024	1024 bytes	superblock
block 2	1 block	block group descriptor table
block 3	1 block	block bitmap
block 4	1 block	inode bitmap
block 5	23 blocks	inode table
block 28	1412 blocks	data blocks

For the curious, block 0 always points to the first sector of the disk or partition and will always contain the boot record if one is present.

The superblock is always located at byte offset 1024 from the start of the disk or partition. In a 1KiB block-size formatted file system, this is block 1, but it will always be block 0 (at 1024 bytes within block 0) in larger block size file systems.

And here's the organisation of a 20MB ext2 file system, using 1KiB blocks:

Table 3-2. Sample 20mb Partition Layout

Block Offset	Length	Description
byte 0	512 bytes	boot record (if present)
byte 512	512 bytes	additional boot record data (if present)
-- block group 0, blocks 1 to 8192 --		
byte 1024	1024 bytes	superblock
block 2	1 block	block group descriptor table
block 3	1 block	block bitmap
block 4	1 block	inode bitmap
block 5	214 blocks	inode table
block 219	7974 blocks	data blocks
-- block group 1, blocks 8193 to 16384 --		
block 8193	1 block	superblock backup
block 8194	1 block	block group descriptor table backup
block 8195	1 block	block bitmap
block 8196	1 block	inode bitmap
block 8197	214 blocks	inode table
block 8408	7974 blocks	data blocks
-- block group 2, blocks 16385 to 24576 --		
block 16385	1 block	block bitmap
block 16386	1 block	inode bitmap
block 16387	214 blocks	inode table
block 16601	3879 blocks	data blocks

The layout on disk is very predictable as long as you know a few basic information; block size, blocks per group, inodes per group. This information is all located in, or can be computed from, the `superblock` structure.

Nevertheless, unless the image was crafted with controlled parameters, the position of the various structures on disk (except the superblock) should never be assumed. Always load the superblock first.

Notice how block 0 is not part of the block group 0 in 1KiB block size file systems. The reason for this is block group 0 always starts with the block containing the superblock. Hence, on 1KiB block systems, block group 0 starts at block 1, but on larger block sizes it starts on block 0. For more information, see the s_first_data_block superblock entry.

3.1. Superblock

The superblock is always located at byte offset 1024 from the beginning of the file, block device or partition formatted with Ext2 and later variants (Ext3, Ext4).

Its structure is mostly constant from Ext2 to Ext3 and Ext4 with only some minor changes.

Table 3-3. Superblock Structure

Offset (bytes)	Size (bytes)	Description
0	4	s_inodes_count

Offset (bytes)	Size (bytes)	Description
4	4	s_blocks_count
8	4	s_r_blocks_count
12	4	s_free_blocks_count
16	4	s_free_inodes_count
20	4	s_first_data_block
24	4	s_log_block_size
28	4	s_log_frag_size
32	4	s_blocks_per_group
36	4	s_frags_per_group
40	4	s_inodes_per_group
44	4	s_mtime
48	4	s_wtime
52	2	s_mnt_count
54	2	s_max_mnt_count
56	2	s_magic
58	2	s_state
60	2	s_errors
62	2	s_minor_rev_level
64	4	s_lastcheck
68	4	s_checkinterval
72	4	s_creator_os
76	4	s_rev_level
80	2	s_def_resuid
82	2	s_def_resgid
-- EXT2_DYNAMIC_REV Specific --		
84	4	s_first_ino
88	2	s_inode_size
90	2	s_block_group_nr
92	4	s_feature_compat
96	4	s_feature_incompat
100	4	s_feature_ro_compat
104	16	s_uuid
120	16	s_volume_name
136	64	s_last_mounted
200	4	s_algo_bitmap
-- Performance Hints --		
204	1	s_prealloc_blocks
205	1	s_prealloc_dir_blocks
206	2	(alignment)
-- Journaling Support --		
208	16	s_journal_uuid
224	4	s_journal_inum
228	4	s_journal_dev
232	4	s_last_orphan
-- Directory Indexing Support --		
236	4 x 4	s_hash_seed
252	1	s_def_hash_version
253	3	padding - reserved for future expansion
-- Other options --		
256	4	s_default_mount_options
260	4	s_first_meta_bg
264	760	Unused - reserved for future revisions

3.1.1. s_inodes_count

32bit value indicating the total number of inodes, both used and free, in the file system. This value must be lower or equal to (s_inodes_per_group * number of block groups). It must be equal to the sum of the inodes defined in each block group.

3.1.2. s_blocks_count

32bit value indicating the total number of blocks in the system including all used, free and reserved. This value must be lower or equal to (s_blocks_per_group * number of block groups). It must be equal to the sum of the blocks defined in each block group.

3.1.3. s_r_blocks_count

32bit value indicating the total number of blocks reserved for the usage of the super user. This is most useful if for some reason a user, maliciously or not, fill the file system to capacity; the super user will have this specified amount of free blocks at his disposal so he can edit and save configuration files.

3.1.4. s_free_blocks_count

32bit value indicating the total number of free blocks, including the number of reserved blocks (see s_r_blocks_count). This is a sum of all free blocks of all the block groups.

3.1.5. s_free_inodes_count

32bit value indicating the total number of free inodes. This is a sum of all free inodes of all the block groups.

3.1.6. s_first_data_block

32bit value identifying the first data block, in other word the id of the block containing the `superblock` structure.

Note that this value is always 0 for file systems with a block size larger than 1KB, and always 1 for file systems with a block size of 1KB. The `superblock` is *always* starting at the 1024th byte of the disk, which normally happens to be the first byte of the 3rd sector.

3.1.7. s_log_block_size

The block size is computed using this 32bit value as the number of bits to shift left the value 1024. This value may only be positive.

```
block size = 1024 << s_log_block_size;
```

Common block sizes include 1KiB, 2KiB, 4KiB and 8Kib. For information about the impact of selecting a block size, see Impact of Block Sizes.

> In Linux, at least up to 2.6.28, the block size must be at least as large as the sector size of the block device, and cannot be larger than the supported memory page of the architecture.

3.1.8. s_log_frag_size

The fragment size is computed using this 32bit value as the number of bits to shift left the value 1024. Note that a negative value would shift the bit right rather than left.

```
if( positive )
  fragmnet size = 1024 << s_log_frag_size;
else
  framgnet size = 1024 >> -s_log_frag_size;
```

> As of Linux 2.6.28 no support exists for an Ext2 partition with fragment size smaller than the block size, as this feature seems to not be available.

3.1.9. s_blocks_per_group

32bit value indicating the total number of blocks per group. This value in combination with s_first_data_block can be used to determine the block groups boundaries.

3.1.10. s_frags_per_group

32bit value indicating the total number of fragments per group. It is also used to determine the size of the `block bitmap` of each block group.

3.1.11. s_inodes_per_group

32bit value indicating the total number of inodes per group. This is also used to determine the size of the `inode bitmap` of each block group. Note that you cannot have more than (block size in bytes * 8) inodes per group as the inode bitmap must fit within a single block. This value must be a perfect multiple of the number of inodes that can fit in a block ((1024<<s_log_block_size)/s_inode_size).

3.1.12. s_mtime

Unix time, as defined by POSIX, of the last time the file system was mounted.

3.1.13. s_wtime

Unix time, as defined by POSIX, of the last write access to the file system.

3.1.14. s_mnt_count

32bit value indicating how many time the file system was mounted since the last time it was fully verified.

3.1.15. s_max_mnt_count

32bit value indicating the maximum number of times that the file system may be mounted before a full check is performed.

3.1.16. s_magic

16bit value identifying the file system as Ext2. The value is currently fixed to `EXT2_SUPER_MAGIC` of value 0xEF53.

3.1.17. s_state

16bit value indicating the file system state. When the file system is mounted, this state is set to `EXT2_ERROR_FS`. After the file system was cleanly unmounted, this value is set to `EXT2_VALID_FS`.

When mounting the file system, if a valid of `EXT2_ERROR_FS` is encountered it means the file system was not cleanly unmounted and most likely contain errors that will need to be fixed. Typically under Linux this means running fsck.

Table 3-4. Defined s_state Values

Constant Name	Value	Description
EXT2_VALID_FS	1	Unmounted cleanly

Constant Name	Value	Description
EXT2_ERROR_FS	2	Errors detected

3.1.18. s_errors

16bit value indicating what the file system driver should do when an error is detected. The following values have been defined:

Table 3-5. Defined s_errors Values

Constant Name	Value	Description
EXT2_ERRORS_CONTINUE	1	continue as if nothing happened
EXT2_ERRORS_RO	2	remount read-only
EXT2_ERRORS_PANIC	3	cause a kernel panic

3.1.19. s_minor_rev_level

16bit value identifying the minor revision level within its revision level.

3.1.20. s_lastcheck

Unix time, as defined by POSIX, of the last file system check.

3.1.21. s_checkinterval

Maximum Unix time interval, as defined by POSIX, allowed between file system checks.

3.1.22. s_creator_os

32bit identifier of the os that created the file system. Defined values are:

Table 3-6. Defined s_creator_os Values

Constant Name	Value	Description
EXT2_OS_LINUX	0	Linux
EXT2_OS_HURD	1	GNU HURD
EXT2_OS_MASIX	2	MASIX
EXT2_OS_FREEBSD	3	FreeBSD
EXT2_OS_LITES	4	Lites

3.1.23. s_rev_level

32bit revision level value.

Table 3-7. Defined s_rev_level Values

Constant Name	Value	Description
EXT2_GOOD_OLD_REV	0	Revision 0
EXT2_DYNAMIC_REV	1	Revision 1 with variable inode sizes, extended attributes, etc.

3.1.24. s_def_resuid

16bit value used as the default user id for reserved blocks.

> In Linux this defaults to `EXT2_DEF_RESUID` of 0.

3.1.25. s_def_resgid

16bit value used as the default group id for reserved blocks.

> In Linux this defaults to `EXT2_DEF_RESGID` of 0.

3.1.26. s_first_ino

32bit value used as index to the first inode useable for standard files. In revision 0, the first non-reserved inode is fixed to 11 (`EXT2_GOOD_OLD_FIRST_INO`). In revision 1 and later this value may be set to any value.

3.1.27. s_inode_size

16bit value indicating the size of the inode structure. In revision 0, this value is always 128 (`EXT2_GOOD_OLD_INODE_SIZE`). In revision 1 and later, this value must be a perfect power of 2 and must be smaller or equal to the block size (1<<s_log_block_size).

3.1.28. s_block_group_nr

16bit value used to indicate the block group number hosting this superblock structure. This can be used to rebuild the file system from any superblock backup.

3.1.29. s_feature_compat

32bit bitmask of compatible features. The file system implementation is free to support them or not without risk of damaging the meta-data.

Table 3-8. Defined s_feature_compat Values

Constant Name	Value	Description
EXT2_FEATURE_COMPAT_DIR_PREALLOC	0x0001	Block pre-allocation for new directories
EXT2_FEATURE_COMPAT_IMAGIC_INODES	0x0002	
EXT3_FEATURE_COMPAT_HAS_JOURNAL	0x0004	An Ext3 journal exists
EXT2_FEATURE_COMPAT_EXT_ATTR	0x0008	Extended inode attributes are present
EXT2_FEATURE_COMPAT_RESIZE_INO	0x0010	Non-standard inode size used
EXT2_FEATURE_COMPAT_DIR_INDEX	0x0020	Directory indexing (HTree)

3.1.30. s_feature_incompat

32bit bitmask of incompatible features. The file system implementation should refuse to mount the file system if any of the indicated feature is unsupported.

An implementation not supporting these features would be unable to properly use the file system. For example, if compression is being used and an executable file would be unusable after being read from the disk if the system does not know how to uncompress it.

Table 3-9. Defined s_feature_incompat Values

Constant Name	Value	Description
EXT2_FEATURE_INCOMPAT_COMPRESSION	0x0001	Disk/File compression is used
EXT2_FEATURE_INCOMPAT_FILETYPE	0x0002	
EXT3_FEATURE_INCOMPAT_RECOVER	0x0004	
EXT3_FEATURE_INCOMPAT_JOURNAL_DEV	0x0008	
EXT2_FEATURE_INCOMPAT_META_BG	0x0010	

3.1.31. s_feature_ro_compat

32bit bitmask of "read-only" features. The file system implementation should mount as read-only if any of the indicated feature is unsupported.

Table 3-10. Defined s_feature_ro_compat Values

Constant Name	Value	Description
EXT2_FEATURE_RO_COMPAT_SPARSE_SUPER	0x0001	Sparse Superblock
EXT2_FEATURE_RO_COMPAT_LARGE_FILE	0x0002	Large file support, 64-bit file size
EXT2_FEATURE_RO_COMPAT_BTREE_DIR	0x0004	Binary tree sorted directory files

3.1.32. s_uuid

128bit value used as the volume id. This should, as much as possible, be unique for each file system formatted.

3.1.33. s_volume_name

16 bytes volume name, mostly unusued. A valid volume name would consist of only ISO-Latin-1 characters and be 0 terminated.

3.1.34. s_last_mounted

64 bytes directory path where the file system was last mounted. While not normally used, it could serve for auto-finding the mountpoint when not indicated on the command line. Again the path should be zero terminated for compatibility reasons. Valid path is constructed from ISO-Latin-1 characters.

3.1.35. s_algo_bitmap

32bit value used by compression algorithms to determine the compression method(s) used.

> Compression is supported in Linux 2.4 and 2.6 via the e2compr patch. For more information, visit http://e2compr.sourceforge.net/

Table 3-11. Defined s_algo_bitmap Values

Constant Name	Bit Number	Description
EXT2_LZV1_ALG	0	Binary value of 0x00000001
EXT2_LZRW3A_ALG	1	Binary value of 0x00000002
EXT2_GZIP_ALG	2	Binary value of 0x00000004
EXT2_BZIP2_ALG	3	Binary value of 0x00000008
EXT2_LZO_ALG	4	Binary value of 0x00000010

3.1.36. s_prealloc_blocks

8-bit value representing the number of blocks the implementation should attempt to pre-allocate when creating a new regular file.

Linux 2.6.28 will only perform pre-allocation using Ext4 although no problem is expected if any version of Linux encounters a file with more blocks present than required.

3.1.37. s_prealloc_dir_blocks

8-bit value representing the number of blocks the implementation should attempt to pre-allocate when creating a new directory.

Linux 2.6.28 will only perform pre-allocation using Ext4 and only if the `EXT4_FEATURE_COMPAT_DIR_PREALLOC` flag is present. Since Linux does not de-allocate blocks from directories after they were allocated, it should be safe to perform pre-allocation and maintain compatibility with Linux.

3.1.38. s_journal_uuid

16-byte value containing the uuid of the journal superblock. See Ext3 Journaling for more information.

3.1.39. s_journal_inum

32-bit inode number of the journal file. See Ext3 Journaling for more information.

3.1.40. s_journal_dev

32-bit device number of the journal file. See Ext3 Journaling for more information.

3.1.41. s_last_orphan

32-bit inode number, pointing to the first inode in the list of inodes to delete. See Ext3 Journaling for more information.

3.1.42. s_hash_seed

An array of 4 32bit values containing the seeds used for the hash algorithm for directory indexing.

3.1.43. s_def_hash_version

An 8bit value containing the default hash version used for directory indexing.

3.1.44. s_default_mount_options

A 32bit value containing the default mount options for this file system. TODO: Add more information here!

3.1.45. s_first_meta_bg

A 32bit value indicating the block group ID of the first meta block group. TODO: Research if this is an Ext3-only extension.

3.2. Block Group Descriptor Table

The block group descriptor table is an array of block group descriptor, used to define parameters of all the block groups. It provides the location of the inode bitmap and inode table, block bitmap, number of free blocks and inodes, and some other useful information.

The block group descriptor table starts on the first block following the superblock. This would be the third block on a 1KiB block file system, or the second block for 2KiB and larger block file systems. Shadow copies of the block group descriptor table are also stored with every copy of the superblock.

Depending on how many block groups are defined, this table can require multiple blocks of storage. Always refer to the superblock in case of doubt.

The layout of a block group descriptor is as follows:

Table 3-12. Block Group Descriptor Structure

Offset (bytes)	Size (bytes)	Description
0	4	bg_block_bitmap
4	4	bg_inode_bitmap
8	4	bg_inode_table
12	2	bg_free_blocks_count
14	2	bg_free_inodes_count
16	2	bg_used_dirs_count
18	2	bg_pad
20	12	bg_reserved

For each block group in the file system, such a `group_desc` is created. Each represent a single block group within the file system and the information within any one of them is pertinent only to the group it is describing. Every block group descriptor table contains all the information about all the block groups.

NOTE: All indicated "block id" are absolute.

3.2.1. bg_block_bitmap

32bit block id of the first block of the "block bitmap" for the group represented.

The actual block bitmap is located within its own allocated blocks starting at the block ID specified by this value.

3.2.2. bg_inode_bitmap

32bit block id of the first block of the "inode bitmap" for the group represented.

3.2.3. bg_inode_table

32bit block id of the first block of the "inode table" for the group represented.

3.2.4. bg_free_blocks_count

16bit value indicating the total number of free blocks for the represented group.

3.2.5. bg_free_inodes_count

16bit value indicating the total number of free inodes for the represented group.

3.2.6. bg_used_dirs_count

16bit value indicating the number of inodes allocated to directories for the represented group.

3.2.7. bg_pad

16bit value used for padding the structure on a 32bit boundary.

3.2.8. bg_reserved

12 bytes of reserved space for future revisions.

3.3. Block Bitmap

On small file systems, the "Block Bitmap" is normally located at the first block, or second block if a superblock backup is present, of each block group. Its official location can be determined by reading the "bg_block_bitmap" in its associated group descriptor.

Each bit represent the current state of a block within that block group, where 1 means "used" and 0 "free/available". The first block of this block group is represented by bit 0 of byte 0, the second by bit 1 of byte 0. The 8th block is represented by bit 7 (most significant bit) of byte 0 while the 9th block is represented by bit 0 (least significant bit) of byte 1.

3.4. Inode Bitmap

The "Inode Bitmap" works in a similar way as the "Block Bitmap", difference being in each bit representing an inode in the "Inode Table" rather than a block.

There is one inode bitmap per group and its location may be determined by reading the "bg_inode_bitmap" in its associated group descriptor.

When the inode table is created, all the reserved inodes are marked as used. In revision 0 this is the first 11 inodes.

3.5. Inode Table

The inode table is used to keep track of every directory, regular file, symbolic link, or special file; their location, size, type and access rights are all stored in inodes. There is no filename stored in the inode itself, names are contained in directory files only.

There is one inode table per block group and it can be located by reading the bg_inode_table in its associated group descriptor. There are s_inodes_per_group inodes per table.

Each inode contain the information about a single physical file on the system. A file can be a directory, a socket, a buffer, character or block device, symbolic link or a regular file. So an inode can be seen as a block of information related to an entity, describing its location on disk, its size and its owner. An inode looks like this:

Table 3-13. Inode Structure

Offset (bytes)	Size (bytes)	Description
0	2	i_mode
2	2	i_uid

Offset (bytes)	Size (bytes)	Description
4	4	i_size
8	4	i_atime
12	4	i_ctime
16	4	i_mtime
20	4	i_dtime
24	2	i_gid
26	2	i_links_count
28	4	i_blocks
32	4	i_flags
36	4	i_osd1
40	15 x 4	i_block
100	4	i_generation
104	4	i_file_acl
108	4	i_dir_acl
112	4	i_faddr
116	12	i_osd2

The first few entries of the inode tables are reserved. In revision 0 there are 11 entries reserved while in revision 1 (EXT2_DYNAMIC_REV) and later the number of reserved inodes entries is specified in the s_first_ino of the superblock structure. Here's a listing of the known reserved inode entries:

Table 3-14. Defined Reserved Inodes

Constant Name	Value	Description
EXT2_BAD_INO	1	bad blocks inode
EXT2_ROOT_INO	2	root directory inode
EXT2_ACL_IDX_INO	3	ACL index inode (deprecated?)
EXT2_ACL_DATA_INO	4	ACL data inode (deprecated?)
EXT2_BOOT_LOADER_INO	5	boot loader inode
EXT2_UNDEL_DIR_INO	6	undelete directory inode

3.5.1. i_mode

16bit value used to indicate the format of the described file and the access rights. Here are the possible values, which can be combined in various ways:

Table 3-15. Defined i_mode Values

Constant	Value	Description
-- file format --		
EXT2_S_IFSOCK	0xC000	socket
EXT2_S_IFLNK	0xA000	symbolic link
EXT2_S_IFREG	0x8000	regular file
EXT2_S_IFBLK	0x6000	block device
EXT2_S_IFDIR	0x4000	directory
EXT2_S_IFCHR	0x2000	character device
EXT2_S_IFIFO	0x1000	fifo
-- process execution user/group override --		
EXT2_S_ISUID	0x0800	Set process User ID
EXT2_S_ISGID	0x0400	Set process Group ID
EXT2_S_ISVTX	0x0200	sticky bit
-- access rights --		
EXT2_S_IRUSR	0x0100	user read
EXT2_S_IWUSR	0x0080	user write
EXT2_S_IXUSR	0x0040	user execute
EXT2_S_IRGRP	0x0020	group read
EXT2_S_IWGRP	0x0010	group write
EXT2_S_IXGRP	0x0008	group execute
EXT2_S_IROTH	0x0004	others read
EXT2_S_IWOTH	0x0002	others write
EXT2_S_IXOTH	0x0001	others execute

3.5.2. i_uid

16bit user id associated with the file.

3.5.3. i_size

In revision 0, (signed) 32bit value indicating the size of the file in bytes. In revision 1 and later revisions, and only for regular files, this represents the lower 32-bit of the file size; the upper 32-bit is located in the i_dir_acl.

3.5.4. i_atime

32bit value representing the number of seconds since january 1st 1970 of the last time this inode was accessed.

3.5.5. i_ctime

32bit value representing the number of seconds since january 1st 1970, of when the inode was created.

3.5.6. i_mtime

32bit value representing the number of seconds since january 1st 1970, of the last time this inode was modified.

3.5.7. i_dtime

32bit value representing the number of seconds since january 1st 1970, of when the inode was deleted.

3.5.8. i_gid

16bit value of the POSIX group having access to this file.

3.5.9. i_links_count

16bit value indicating how many times this particular inode is linked (referred to). Most files will have a link count of 1. Files with hard links pointing to them will have an additional count for each hard link.

Symbolic links do not affect the link count of an inode. When the link count reaches 0 the inode and all its associated blocks are freed.

3.5.10. i_blocks

32-bit value representing the total number of 512-bytes blocks reserved to contain the data of this inode, regardless if these blocks are used or not. The block numbers of these reserved blocks are contained in the i_block array.

Since this value represents 512-byte blocks and not file system blocks, this value should not be directly used as an index to the i_block array. Rather, the maximum index of the i_block array should be computed from i_blocks / ((1024<<s_log_block_size)/512), or once simplified, i_blocks/(2<<s_log_block_size).

3.5.11. i_flags

32bit value indicating how the ext2 implementation should behave when accessing the data for this inode.

Table 3-16. Defined i_flags Values

Constant Name	Value	Description
EXT2_SECRM_FL	0x00000001	secure deletion
EXT2_UNRM_FL	0x00000002	record for undelete
EXT2_COMPR_FL	0x00000004	compressed file
EXT2_SYNC_FL	0x00000008	synchronous updates
EXT2_IMMUTABLE_FL	0x00000010	immutable file
EXT2_APPEND_FL	0x00000020	append only
EXT2_NODUMP_FL	0x00000040	do not dump/delete file
EXT2_NOATIME_FL	0x00000080	do not update .i_atime
-- Reserved for compression usage --		
EXT2_DIRTY_FL	0x00000100	Dirty (modified)
EXT2_COMPRBLK_FL	0x00000200	compressed blocks
EXT2_NOCOMPR_FL	0x00000400	access raw compressed data
EXT2_ECOMPR_FL	0x00000800	compression error
-- End of compression flags --		
EXT2_BTREE_FL	0x00001000	b-tree format directory
EXT2_INDEX_FL	0x00001000	hash indexed directory
EXT2_IMAGIC_FL	0x00002000	AFS directory
EXT3_JOURNAL_DATA_FL	0x00004000	journal file data
EXT2_RESERVED_FL	0x80000000	reserved for ext2 library

3.5.12. i_osd1

32bit OS dependant value.

3.5.12.1. Hurd

32bit value labeled as "translator".

3.5.12.2. Linux

32bit value currently reserved.

3.5.12.3. Masix

32bit value currently reserved.

3.5.13. i_block

15 x 32bit block numbers pointing to the blocks containing the data for this inode. The first 12 blocks are direct blocks. The 13th entry in this array is the block number of the first indirect block; which is a block containing an array of block ID containing the data. Therefore, the 13th block of the file will be the first block ID contained in the indirect block. With a 1KiB block size, blocks 13 to 268 of the file data are contained in this indirect block.

The 14th entry in this array is the block number of the first doubly-indirect block; which is a block containing an array of indirect block IDs, with each of those indirect blocks containing an array of blocks containing the data. In a 1KiB block size, there would be 256 indirect blocks per doubly-indirect block, with 256 direct blocks per indirect block for a total of 65536 blocks per doubly-indirect block.

The 15th entry in this array is the block number of the triply-indirect block; which is a block containing an array of doubly-indrect block IDs, with each of those doubly-indrect block containing an array of indrect block, and each of those indirect block containing an array of direct block. In a 1KiB file system, this would be a total of 16777216 blocks per triply-indirect block.

A value of 0 in this array effectively terminates it with no further block being defined. All the remaining entries of the array should still be set to 0.

3.5.14. i_generation

32bit value used to indicate the file version (used by NFS).

3.5.15. i_file_acl

32bit value indicating the block number containing the extended attributes. In revision 0 this value is always 0.

> Patches and implementation status of ACL under Linux can generally be found at http://acl.bestbits.at/

3.5.16. i_dir_acl

In revision 0 this 32bit value is always 0. In revision 1, for regular files this 32bit value contains the high 32 bits of the 64bit file size.

> Linux sets this value to 0 if the file is not a regular file (i.e. block devices, directories, etc). In theory, this value could be set to point to a block containing extended attributes of the directory or special file.

3.5.17. i_faddr

32bit value indicating the location of the file fragment.

> In Linux and GNU HURD, since fragments are unsupported this value is always 0. In Ext4 this value is now marked as obsolete.
>
> In theory, this should contain the block number which hosts the actual fragment. The fragment number and its size would be contained in the i_osd2 structure.

3.5.18. Inode i_osd2 Structure

96bit OS dependant structure.

3.5.18.1. Hurd

Table 3-17. Inode i_osd2 Structure: Hurd

Offset (bytes)	Size (bytes)	Description
0	1	h_i_frag
1	1	h_i_fsize
2	2	h_i_mode_high
4	2	h_i_uid_high
6	2	h_i_gid_high
8	4	h_i_author

3.5.18.1.1. h_i_frag

8bit fragment number. Always 0 GNU HURD since fragments are not supported. Obsolete with Ext4.

3.5.18.1.2. h_i_fsize

8bit fragment size. Always 0 in GNU HURD since fragments are not supported. Obsolete with Ext4.

3.5.18.1.3. h_i_mode_high

High 16bit of the 32bit mode.

3.5.18.1.4. h_i_uid_high

High 16bit of user id.

3.5.18.1.5. h_i_gid_high

High 16bit of group id.

3.5.18.1.6. h_i_author

32bit user id of the assigned file author. If this value is set to -1, the POSIX user id will be used.

3.5.18.2. Linux

Table 3-18. Inode i_osd2 Structure: Linux

Offset (bytes)	Size (bytes)	Description
0	1	l_i_frag
1	1	l_i_fsize
2	2	reserved
4	2	l_i_uid_high
6	2	l_i_gid_high
8	4	reserved

3.5.18.2.1. l_i_frag

8bit fragment number.

> Always 0 in Linux since fragments are not supported.
>
> A new implementation of Ext2 should completely disregard this field if the i_faddr value is 0; in Ext4 this field is combined with l_i_fsize to become the high 16bit of the 48bit blocks count for the inode data.

3.5.18.2.2. l_i_fsize

8bit fragment size.

> Always 0 in Linux since fragments are not supported.
>
> A new implementation of Ext2 should completely disregard this field if the i_faddr value is 0; in Ext4 this field is combined with l_i_frag to become the high 16bit of the 48bit blocks count for the inode data.

3.5.18.2.3. l_i_uid_high

High 16bit of user id.

3.5.18.2.4. l_i_gid_high

High 16bit of group id.

3.5.18.3. Masix

Table 3-19. Inode i_osd2 Structure: Masix

Offset (bytes)	Size (bytes)	Description
0	1	m_i_frag
1	1	m_i_fsize
2	10	reserved

3.5.18.3.1. m_i_frag

8bit fragment number. Always 0 in Masix as framgents are not supported. Obsolete with Ext4.

3.5.18.3.2. m_i_fsize

8bit fragment size. Always 0 in Masix as fragments are not supported. Obsolete with Ext4.

3.6. Locating an Inode

Inodes are all numerically ordered. The "inode number" is an index in the inode table to an inode structure. The size of the inode table is fixed at format time; it is built to hold a maximum number of entries. Due to the large amount of entries created, the table is quite big and thus, it is split equally among all the block groups (see Chapter 3 for more information).

The s_inodes_per_group field in the superblock structure tells us how many inodes are defined per group. Knowing that inode 1 is the first inode defined in the inode table, one can use the following formulaes:

```
block group = (inode - 1) / s_inodes_per_group
```

Once the block is identified, the local inode index for the local inode table can be identified using:

```
local inode index = (inode - 1) % s_inodes_per_group
```

Here are a couple of sample values that could be used to test your implementation:

Table 3-20. Sample Inode Computations

Inode Number	Block Group Number	Local Inode Index
s_inodes_per_group = 1712		
1	0	0
963	0	962
1712	0	1711
1713	1	0
3424	1	1711
3425	2	0

As many of you are most likely already familiar with, an index of 0 means the first entry. The reason behind using 0 rather than 1 is that it can more easily be multiplied by the structure size to find the final byte offset of its location in memory or on disk.

Chapter 4. Directory Structure

Directories are used to hierarchically organize files. Each directory can contain other directories, regular files and special files.

Directories are stored as data block and referenced by an inode. They can be identified by the file type `EXT2_S_IFDIR` stored in the i_mode field of the inode structure.

The second entry of the Inode table contains the inode pointing to the data of the root directory; as defined by the `EXT2_ROOT_INO` constant.

In revision 0 directories could only be stored in a linked list. Revision 1 and later introduced indexed directories. The indexed directory is backward compatible with the linked list directory; this is achieved by inserting empty directory entry records to skip over the hash indexes.

4.1. Linked List Directory

A directory file is a linked list of directory entry structures. Each structure contains the name of the entry, the inode associated with the data of this entry, and the distance within the directory file to the next entry.

In revision 0, the type of the entry (file, directory, special file, etc) has to be looked up in the inode of the file. In revision 0.5 and later, the file type is also contained in the directory entrystructure.

Table 4-1. Linked Directory Entry Structure

Offset (bytes)	Size (bytes)	Description
0	4	inode
4	2	rec_len
6	1	name_len[a]
7	1	file_type[b]
8	0-255	name

Notes:
a. Revision 0 of Ext2 used a 16bit `name_len`; since most implementations restricted filenames to a maximum of 255 characters this value was truncated to 8bit with the upper 8bit recycled as file_type.
b. Not available in revision 0; this field was part of the 16bit name_len field.

4.1.1. inode

32bit inode number of the file entry. A value of 0 indicate that the entry is not used.

4.1.2. rec_len

16bit unsigned displacement to the next directory entry from the start of the current directory entry. This field must have a value at least equal to the length of the current record.

The directory entries must be aligned on 4 bytes boundaries and there cannot be any directory entry spanning multiple data blocks. If an entry cannot completely fit in one block, it must be pushed to the next data block and the rec_len of the previous entry properly adjusted.

> Since this value cannot be negative, when a file is removed the previous record within the block has to be modified to point to the next valid record within the block or to the end of the block when no other directory entry is present.
>
> If the first entry within the block is removed, a blank record will be created and point to the next directory entry or to the end of the block.

4.1.3. name_len

8bit unsigned value indicating how many bytes of character data are contained in the name.

> This value must never be larger than rec_len - 8. If the directory entry name is updated and cannot fit in the existing directory entry, the entry may have to be relocated in a new directory entry of sufficient size and possibly stored in a new data block.

4.1.4. file_type

8bit unsigned value used to indicate file type.

> In revision 0, this field was the upper 8-bit of the then 16-bit name_len. Since all implementations still limited the file names to 255 characters this 8-bit value was always 0.
>
> This value must match the inode type defined in the related inode entry.

Table 4-2. Defined Inode File Type Values

Constant Name	Value	Description
EXT2_FT_UNKNOWN	0	Unknown File Type
EXT2_FT_REG_FILE	1	Regular File
EXT2_FT_DIR	2	Directory File
EXT2_FT_CHRDEV	3	Character Device
EXT2_FT_BLKDEV	4	Block Device
EXT2_FT_FIFO	5	Buffer File
EXT2_FT_SOCK	6	Socket File
EXT2_FT_SYMLINK	7	Symbolic Link

4.1.5. name

Name of the entry. The ISO-Latin-1 character set is expected in most system. The name must be no longer than 255 bytes after encoding.

4.1.6. Sample Directory

Here's a sample of the home directory of one user on my system:

```
$ ls -1a ~
.
..
.bash_profile
.bashrc
mbox
public_html
tmp
```

For which the following data representation can be found on the storage device:

Table 4-3. Sample Linked Directory Data Layout, 4KiB blocks

Offset (bytes)	Size (bytes)	Description
Directory Entry 0		
0	4	inode number: 783362
4	2	record length: 12
6	1	name length: 1
7	1	file type: EXT2_FT_DIR=2
8	1	name: .
9	3	padding
Directory Entry 1		
12	4	inode number: 1109761
16	2	record length: 12
18	1	name length: 2
19	1	file type: EXT2_FT_DIR=2
20	2	name: ..
22	2	padding
Directory Entry 2		
24	4	inode number: 783364
28	2	record length: 24
30	1	name length: 13
31	1	file type: EXT2_FT_REG_FILE
32	13	name: .bash_profile
45	3	padding
Directory Entry 3		
48	4	inode number: 783363
52	2	record length: 16
54	1	name length: 7
55	1	file type: EXT2_FT_REG_FILE
56	7	name: .bashrc
63	1	padding
Directory Entry 4		
64	4	inode number: 783377
68	2	record length: 12
70	1	name length: 4
71	1	file type: EXT2_FT_REG_FILE
72	4	name: mbox
Directory Entry 5		
76	4	inode number: 783545
80	2	record length: 20
82	1	name length: 11
83	1	file type: EXT2_FT_DIR=2
84	11	name: public_html
95	1	padding
Directory Entry 6		
96	4	inode number: 669354
100	2	record length: 12
102	1	name length: 3
103	1	file type: EXT2_FT_DIR=2
104	3	name: tmp
107	1	padding
Directory Entry 7		
108	4	inode number: 0
112	2	record length: 3988
114	1	name length: 0
115	1	file type: EXT2_FT_UNKNOWN
116	0	name:
116	3980	padding

4.2. Indexed Directory Format

Using the standard linked list directory format can become very slow once the number of files starts growing. To improve performances in such a system, a hashed index is used, which allow to quickly locate the particular file searched.

Bit EXT2_INDEX_FL in the i_flags of the directory inode is set if the indexed directory format is used.

In order to maintain backward compatibility with older implementations, the indexed directory also maintains a linked directory format side-by-side. In case there's any discrepency between the indexed and linked directories, the linked directory is preferred.

This backward compatibility is achieved by placing a fake directory entries at the beginning of block 0 of the indexed directory data blocks. These fake entries are part of the dx_root structure and host the linked directory information for the "." and ".." folder entries.

Immediately following the Section 4.2.1 structure is an array of Section 4.2.2 up to the end of the data block or until all files have been indexed.

When the number of files to be indexed exceeds the number of Section 4.2.2 that can fit in a block (Section 4.2.2.3), a level of indirect indexes is created. An indirect index is another data block allocated to the directory inode that contains directory entries.

4.2.1. Indexed Directory Root

Table 4-4. Indexed Directory Root Structure

Offset (bytes)	Size (bytes)	Description
-- Linked Directory Entry: . --		
0	4	inode: this directory
4	2	rec_len: 12
6	1	name_len: 1
7	1	file_type: `EXT2_FT_DIR`=2
8	1	name: .
9	3	padding
-- Linked Directory Entry: .. --		
12	4	inode: parent directory
16	2	rec_len: (blocksize - this entry's length(12))
18	1	name_len: 2
19	1	file_type: `EXT2_FT_DIR`=2
20	2	name: ..
22	2	padding
-- Indexed Directory Root Information Structure --		
24	4	reserved, zero
28	1	hash_version
29	1	info_length
30	1	indirect_levels
31	1	reserved - unused flags

4.2.1.1. hash_version

8bit value representing the hash version used in this indexed directory.

Table 4-5. Defined Indexed Directory Hash Versions

Constant Name	Value	Description
DX_HASH_LEGACY	0	TODO: link to section
DX_HASH_HALF_MD4	1	TODO: link to section
DX_HASH_TEA	2	TODO: link to section

4.2.1.2. info_length

8bit length of the indexed directory information structure (dx_root); currently equal to 8.

4.2.1.3. indirect_levels

8bit value indicating how many indirect levels of indexing are present in this hash.

> In Linux, as of 2.6.28, the maximum indirect levels value supported is 1.

4.2.2. Indexed Directory Entry

The indexed directory entries are used to quickly lookup the inode number associated with the hash of a filename. These entries are located immediately following the fake linked directory entry of the directory data blocks, or immediately following the Section 4.2.1.

The first indexed directory entry, rather than containing an actual hash and block number, contains the maximum number of indexed directory entries that can fit in the block and the actual number of indexed directory entries stored in the block. The format of this special entry is detailed in Table 4-7.

The other directory entries are sorted by hash value starting from the smallest to the largest numerical value.

Table 4-6. Indexed Directory Entry Structure (dx_entry)

Offset (bytes)	Size (bytes)	Description
0	4	hash
4	4	block

Table 4-7. Indexed Directory Entry Count and Limit Structure

Offset (bytes)	Size (bytes)	Description
0	2	limit
2	2	count

4.2.2.1. hash

32bit hash of the filename represented by this entry.

4.2.2.2. block

32bit block index of the directory inode data block containing the (linked) directory entry for the filename.

4.2.2.3. limit

16bit value representing the total number of indexed directory entries that fit within the block, after removing the other structures, but including the count/limit entry.

4.2.2.4. count

16bit value representing the total number of indexed directory entries present in the block. TODO: Research if this value includes the count/limit entry.

4.2.3. Lookup Algorithm

Lookup is straightforword:

```
- Compute a hash of the name
- Read the index root
- Use binary search (linear in the current code) to find the
  first index or leaf block that could contain the target hash
  (in tree order)
- Repeat the above until the lowest tree level is reached
- Read the leaf directory entry block and do a normal Ext2
  directory block search in it.
- If the name is found, return its directory entry and buffer
- Otherwise, if the collision bit of the next directory entry is
  set, continue searching in the successor block
```

Normally, two logical blocks of the file will need to be accessed, and one or two metadata index blocks. The effect of the metadata index blocks can largely be ignored in terms of disk access time since these blocks are unlikely to be evicted from cache. There is some small CPU cost that can be addressed by moving the whole directory into the page cache.

4.2.4. Insert Algorithm

Insertion of new entries into the directory is considerably more complex than lookup, due to the need to split leaf blocks when they become full, and to satisfy the conditions that allow hash key collisions to be handled reliably and efficiently. I'll just summarize here:

```
- Probe the index as for lookup
- If the target leaf block is full, split it and note the block
  that will receive the new entry
- Insert the new entry in the leaf block using the normal Ext2
  directory entry insertion code.
```

The details of splitting and hash collision handling are somewhat messy, but I will be happy to dwell on them at length if anyone is interested.

4.2.5. Splitting

In brief, when a leaf node fills up and we want to put a new entry into it the leaf has to be split, and its share of the hash space has to be partitioned. The most straightforward way to do this is to sort the entrys by hash value and split somewhere in the middle of the sorted list. This operation is log(number_of_entries_in_leaf) and is not a great cost so long as an efficient sorter is used. I used Combsort for this, although Quicksort would have been just as good in this case since average case performance is more important than worst case.

An alternative approach would be just to guess a median value for the hash key, and the partition could be done in linear time, but the resulting poorer partitioning of hash key space outweighs the small advantage of the linear partition algorithm. In any event, the number of entries needing sorting is bounded by the number that fit in a leaf.

4.2.6. Key Collisions

Some complexity is introduced by the need to handle sequences of hash key collisions. It is desireable to avoid splitting such sequences between blocks, so the split point of a block is adjusted with this in mind. But the possibility still remains that if the block fills up with identically-hashed entries, the sequence may still have to be split. This situation is flagged by placing a 1 in the low bit of the index entry that points at the sucessor block, which is naturally interpreted by the index probe as an intermediate value without any special coding. Thus, handling the collision problem imposes no real processing overhead, just come extra code and a slight reduction in the hash key space. The hash key space remains sufficient for any conceivable number of directory entries, up into the billions.

4.2.7. Hash Function

The exact properties of the hash function critically affect the performance of this indexing strategy, as I learned by trying a number of poor hash functions, at times intentionally. A poor hash function will result in many collisions or poor partitioning of the hash space. To illustrate why the latter is a problem, consider what happens when a block is split such that it covers just a few distinct hash values. The probability of later index entries hashing into the same, small hash space is very small. In practice, once a block is split, if its hash space is too small it tends to stay half full forever, an effect I observed in practice.

After some experimentation I came up with a hash function that gives reasonably good dispersal of hash keys across the entire 31 bit key space. This improved the average fullness of leaf blocks considerably, getting much closer to the theoretical average of 3/4 full.

But the current hash function is just a place holder, waiting for an better version based on some solid theory. I currently favor the idea of using crc32 as the default hash function, but I welcome suggestions.

Inevitably, no matter how good a hash function I come up with, somebody will come up with a better one later. For this reason the design allows for additional hash functiones to be added, with backward compatibility. This is accomplished simply, by including a hash function number in the index root. If a new, improved hash function is added, all the previous versions remain available, and previously created indexes remain readable.

Of course, the best strategy is to have a good hash function right from the beginning. The initial, quick hack has produced results that certainly have not been disappointing.

4.2.8. Performance

OK, if you have read this far then this is no doubt the part you've been waiting for. In short, the performance improvement over normal Ext2 has been stunning. With very small directories performance is similar to standard Ext2, but as directory size increases standard Ext2 quickly blows up quadratically, while htree-enhanced Ext2 continues to scale linearly.

Uli Luckas ran benchmarks for file creation in various sizes of directories ranging from 10,000 to 90,000 files. The results are pleasing: total file creation time stays very close to linear, versus quadratic increase with normal Ext2.

Time to create:

Figure 4-1. Performance of Indexed Directories

```
               Indexed        Normal
               =======        ======
10000 Files:   0m1.350s       0m23.670s
20000 Files:   0m2.720s       1m20.470s
30000 Files:   0m4.330s       3m9.320s
40000 Files:   0m5.890s       5m48.750s
50000 Files:   0m7.040s       9m31.270s
60000 Files:   0m8.610s       13m52.250s
70000 Files:   0m9.980s       19m24.070s
80000 Files:   0m12.060s      25m36.730s
90000 Files:   0m13.400s      33m18.550s
```

A graph is posted at: http://www.innominate.org/~phillips/htree/performance.png

All of these tests are CPU-bound, which may come as a surprise. The directories fit easily in cache, and the limiting factor in the case of standard Ext2 is the looking up of directory blocks in buffer cache, and the low level scan of directory entries. In the case of htree indexing there are a number of costs to be considered, all of them pretty well bounded. Notwithstanding, there are a few obvious optimizations to be done:

```
- Use binary search instead of linear search in the interior index
  nodes.

- If there is only one leaf block in a directory, bypass the index
  probe, go straight to the block.

- Map the directory into the page cache instead of the buffer cache.
```

Each of these optimizations will produce a noticeable improvement in performance, but naturally it will never be anything like the big jump going from N**2 to Log512(N), ~= N. In time the optimizations will be applied and we can expect to see another doubling or so in performance.

There will be a very slight performance hit when the directory gets big enough to need a second level. Because of caching this will be very small. Traversing the directories metadata index blocks will be a bigger cost, and once again, this cost can be reduced by moving the directory blocks into the page cache.

Typically, we will traverse 3 blocks to read or write a directory entry, and that number increases to 4-5 with really huge directories. But this is really nothing compared to normal Ext2, which traverses several hundred blocks in the same situation.

Chapter 5. File Attributes

Most of the file (also directory, symlink, device...) attributes are located in the inode associated with the file. Some other attributes are only available as extended attributes.

5.1. Standard Attributes

5.1.1. SUID, SGID and -rwxrwxrwx

There isn't much to say about those, they are located with the SGID and SUID bits in ext2_inode.i_mode.

5.1.2. File Size

The size of a file can be determined by looking at the ext2_inode.i_size field.

5.1.3. Owner and Group

Under most implementations, the owner and group are 16bit values, but on some recent Linux and Hurd implementations the owner and group id are 32bit. When 16bit values are used, only the "low" part should be used as valid, while when using 32bit value, both the "low" and "high" part should be used, the high part being shifted left 16 places then added to the low part.

The low part of owner and group are located in ext2_inode.i_uid and ext2_inode.i_gid respectively.

The high part of owner and group are located in ext2_inode.osd2.hurd.h_i_uid_high and ext2_inode.osd2.hurd.h_i_gid_high, respectively, for Hurd and located inext2_inode.osd2.linux.l_i_uid_high and ext2_inode.osd2.linux.l_i_gid_high, respectively, for Linux.

5.2. Extended Attributes

Extended attributes are name:value pairs associated permanently with files and directories, similar to the environment strings associated with a process. An attribute may be defined or undefined. If it is defined, its value may be empty or non-empty.

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system. They are often used to provide additional functionality to a filesystem - for example, additional security features such as Access Control Lists (ACLs) may be implemented using extended attributes.

Extended attributes are accessed as atomic objects. Reading retrieves the whole value of an attribute and stores it in a buffer. Writing replaces any previous value with the new value.

Extended attributes are stored on disk blocks allocated outside of any inode. The i_file_acl field (for regular files) or the i_dir_acl field (for directories) fields contain the block number of the allocated data block used to store the extended attributes.

> Inodes which have all identical extended attributes may share the same extended attribute block.

The attribute values are on the same block as their attribute entry descriptions, aligned to the end of the attribute block. This allows for additional attributes to be added more easily. The size of entry headers varies with the length of the attribute name.

5.2.1. Extended Attribute Block Layout

The block header is followed by multiple entry descriptors. These entry descriptors are variable in size, and aligned to `EXT2_XATTR_PAD` (4) byte boundaries. The entry descriptors are sorted by attribute name, so that two extended attribute blocks can be compared efficiently.

Attribute values are aligned to the end of the block, stored in no specific order. They are also padded to `EXT2_XATTR_PAD` (4) byte boundaries. No additional gaps are left between them.

Table 5-1. Extended Attribute Block Layout

Attribute Block Header		
Attribute Entry 1	|	
Attribute Entry 2	|	growing downwards
Attribute Entry 3	V	
4 null bytes		
unused space...		
Attribute Value 1	^	
Attribute Value 3	|	growing upwards
Attribute Value 2	|	

5.2.2. Extended Attribute Block Header

Table 5-2. ext2_xattr_header structure

Offset (bytes)	Size (bytse)	Description
0	4	h_magic
4	4	h_refcount
8	4	h_blocks
12	4	h_hash
16	16	reserved

5.2.2.1. h_magic

32bit magic number of identification, `EXT2_XATTR_MAGIC` = 0xEA020000.

5.2.2.2. h_refcount

32bit value used as reference count. This value is incremented everytime a link is created to this attribute block and decremented when a link is destroyed. Whenever this value reaches 0 the attribute block can be freed.

5.2.2.3. h_blocks

32bit value indicating how many blocks are currently used by the extended attributes.

> In Linux a value of h_blocks higher than 1 is considered invalid. This effectively restrict the amount of extended attributes to what can be fit in a single block.
>
> There does not seem to be any support for extended attributes in Ext2 under GNU HURD.

5.2.2.4. h_hash

32bit hash value of all attribute entry header hashes.

Procedure to compute Extended Attribute Header Hash

1. Initialize the 32bit hash to 0
2. Check if there are any extended attribute entry to process, if not we are done.
3. Do a cyclic bit shift of 16 bits to the left of the 32bits hash value, effectively swapping the upper and lower 16bits of the hash
4. Perform a bitwise OR between the extended attribute entry hash and the header hash being computed.
5. Go back to step 2>.

5.2.3. Attribute Entry Header

Figure 5-1. ext2_xattr_header structure

```
offset  size    description
------- ------- -----------
      0       1 e_name_len
      1       1 e_name_index
```

```
 2       2 e_value_offs
 4       4 e_value_block
 8       4 e_value_size
12       4 e_hash
16     ... e_name
```

The total size of an attribute entry is always rounded to the next 4-bytes boundary.

5.2.3.1. e_name_len

8bit unsigned value indicating the length of the name.

5.2.3.2. e_name_index

8bit unsigned value used as attribute name index.

5.2.3.3. e_value_offs

16bit unsigned offset to the value within the value block.

5.2.3.4. e_value_block

32bit id of the block holding the value.

5.2.3.5. e_value_size

32bit unsigned value indicating the size of the attribute value.

5.2.3.6. e_hash

32bit hash of attribute name and value.

5.2.3.7. e_name

Attribute name.

5.3. Behaviour Control Flags

The i_flags value in the inode structure allows to specify how the file system should behave in regard to the file. The following bits are currently defined:

Table 5-3. Behaviour Control Flags

EXT2_SECRM_FL	0x00000001	secure deletion
EXT2_UNRM_FL	0x00000002	record for undelete
EXT2_COMPR_FL	0x00000004	compressed file
EXT2_SYNC_FL	0x00000008	synchronous updates
EXT2_IMMUTABLE_FL	0x00000010	immutable file
EXT2_APPEND_FL	0x00000020	append only
EXT2_NODUMP_FL	0x00000040	do not dump/delete file
EXT2_NOATIME_FL	0x00000080	do not update .i_atime
EXT2_DIRTY_FL	0x00000100	dirty (file is in use?)
EXT2_COMPRBLK_FL	0x00000200	compressed blocks
EXT2_NOCOMPR_FL	0x00000400	access raw compressed data
EXT2_ECOMPR_FL	0x00000800	compression error
EXT2_BTREE_FL	0x00001000	b-tree format directory
EXT2_INDEX_FL	0x00001000	Hash indexed directory
EXT2_IMAGIC_FL	0x00002000	?
EXT3_JOURNAL_DATA_FL	0x00004000	journal file data
EXT2_RESERVED_FL	0x80000000	reserved for ext2 implementation

5.3.1. EXT2_SECRM_FL - Secure Deletion

Enabling this bit will cause random data to be written over the flie's content several time before the blocks are unlinked. Note that this is highly implementation dependant and as such, it should not be assumed to be 100% secure. Make sure to study the implementation notes before relying on this option.

5.3.2. EXT2_UNRM_FL - Record for Undelete

When supported by the implementation, setting this bit will cause the deleted data to be moved to a temporary location, where the user can restore the original file without any risk of data lost. This is most useful when using ext2 on a desktop or workstation.

5.3.3. EXT2_COMPR_FL - Compressed File

The file's content is compressed. There is no note about the particular algorithm used other than maybe the s_algo_bitmap field of the superblock structure.

5.3.4. EXT2_SYNC_FL - Synchronous Updates

The file's content in memory will be constantly synchronized with the content on disk. This is mostly used for very sensitive boot files or encryption keys that you do not want to lose in case of a crash.

5.3.5. EXT2_IMMUTABLE_FL - Immutable File

The blocks associated with the file will not be exchanged. If for any reason a file system defragmentation is launched, such files will not be moved. Mostly used for stage2 and stage1.5 boot loaders.

5.3.6. EXT2_APPEND_FL - Append Only

Writing can only be used to append content at the end of the file and not modify the current content. Example of such use could be mailboxes, where anybody could send a message to a user but not modify any already present.

5.3.7. EXT2_NODUMP_FL - Do No Dump/Delete

Setting this bit will protect the file from deletion. As long as this bit is set, even if the i_links_count is 0, the file will not be removed.

5.3.8. EXT2_NOATIME_FL - Do Not Update .i_atime

The i_atime field of the inode structure will not be modified when the file is accessed if this bit is set. The only good use I can think of that are related to security.

5.3.9. EXT2_DIRTY_FL - Dirty

I do not have information at this moment about the use of this bit.

5.3.10. EXT2_COMPRBLK_FL - Compressed Blocks

This flag is set if one or more blocks are compressed. You can have more information about compression on ext2 at http://www.netspace.net.au/~reiter/e2compr/ Note that the project has not been updated since 1999.

5.3.11. EXT2_NOCOMPR_FL - Access Raw Compressed Data

When this flag is set, the file system implementation will not uncompress the data before fowarding it to the application but will rather give it as is.

5.3.12. EXT2_ECOMPR_FL - Compression Error

This flag is set if an error was detected when trying to uncompress the file.

5.3.13. EXT2_BTREE_FL - B-Tree Format Directory

5.3.14. EXT2_INDEX_FL - Hash Indexed Directory

When this bit is set, the format of the directory file is hash indexed. This is covered in details in Section 4.2.

5.3.15. EXT2_IMAGIC_FL -

5.3.16. EXT2_JOURNAL_DATA_FL - Journal File Data

5.3.17. EXT2_RESERVED_FL - Reserved

Appendix A. Credits

I would like to personally thank everybody who contributed to this document, you are numerous and in many cases I haven't kept track of all of you. Be sure that if you are not in this list, it's a mistake and do not hesitate to contact me, it will be a pleasure to add your name to the list.

```
Peter Rottengatter (Peter.Rottengatter@bakerhughes.com)
  Corrections to Section 3.1.11
  Corrections to Table 3-1 and Table 3-2
  Corrections to Section 3.2

Ryan Cuthbertson (ryan.cuthbertson@adelaide.edu.au)
  Corrections to Section 3.5.10
  Corrections to Chapter 3

Andreas Gruenbacher (a.gruenbacher@bestbits.at)
  Section 5.2

Daniel Phillips (phillips@innominate.de)
  Section 4.2.3
```

Section 4.2.4
Section 4.2.5
Section 4.2.6
Section 4.2.7
Section 4.2.8

Jeremy Stanley of Access Data Inc.
Pointed out the inversed values for EXT2_S_IFSOCK and EXT2_S_IFLNK