```
The Second Extended Filesystem
==============================

ext2 was originally released in January 1993.  Written by R\'emy Card,
Theodore Ts'o and Stephen Tweedie, it was a major rewrite of the
Extended Filesystem.  It is currently still (April 2001) the predominant
filesystem in use by Linux.  There are also implementations available
for NetBSD, FreeBSD, the GNU HURD, Windows 95/98/NT, OS/2 and RISC OS.

Options
=======

Most defaults are determined by the filesystem superblock, and can be
set using tune2fs(8). Kernel-determined defaults are indicated by (*).

bsddf                   (*)     Makes `df' act like BSD.
minixdf                         Makes `df' act like Minix.

check=none, nocheck     (*)     Don't do extra checking of bitmaps on mount
                                (check=normal and check=strict options removed)

debug                           Extra debugging information is sent to the
                                kernel syslog.  Useful for developers.

errors=continue                 Keep going on a filesystem error.
errors=remount-ro               Remount the filesystem read-only on an error.
errors=panic                    Panic and halt the machine if an error occurs.

grpid, bsdgroups                Give objects the same group ID as their parent.
nogrpid, sysvgroups             New objects have the group ID of their creator.

nouid32                         Use 16-bit UIDs and GIDs.

oldalloc                        Enable the old block allocator. Orlov should
                                have better performance, we'd like to get some
                                feedback if it's the contrary for you.
orlov                   (*)     Use the Orlov block allocator.
                                (See http://lwn.net/Articles/14633/ and
                                http://lwn.net/Articles/14446/.)

resuid=n                        The user ID which may use the reserved blocks.
resgid=n                        The group ID which may use the reserved blocks.

sb=n                            Use alternate superblock at this location.

user_xattr                      Enable "user." POSIX Extended Attributes
                                (requires CONFIG_EXT2_FS_XATTR).
                                See also http://acl.bestbits.at
nouser_xattr                    Don't support "user." extended attributes.

acl                             Enable POSIX Access Control Lists support
                                (requires CONFIG_EXT2_FS_POSIX_ACL).
                                See also http://acl.bestbits.at
noacl                           Don't support POSIX ACLs.
```

```
nobh                             Do not attach buffer_heads to file pagecache.

xip                              Use execute in place (no caching) if possible

grpquota,noquota,quota,usrquota Quota options are silently ignored by ext2.
```

Specification

ext2 shares many properties with traditional Unix filesystems. It has the concepts of blocks, inodes and directories. It has space in the specification for Access Control Lists (ACLs), fragments, undeletion and compression though these are not yet implemented (some are available as separate patches). There is also a versioning mechanism to allow new features (such as journalling) to be added in a maximally compatible manner.

Blocks

The space in the device or file is split up into blocks. These are a fixed size, of 1024, 2048 or 4096 bytes (8192 bytes on Alpha systems), which is decided when the filesystem is created. Smaller blocks mean less wasted space per file, but require slightly more accounting overhead, and also impose other limits on the size of files and the filesystem.

Block Groups

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.

The Superblock

The superblock contains all the information about the configuration of the filing system. 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.

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 (see the Revisions section below) and which OS created it.

If the filesystem is revision 1 or higher, then there are 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.

Inodes

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 more 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).

The flags field contains some ext2-specific flags which aren't catered for by the standard chmod flags. These flags can be listed with lsattr and changed with the chattr command, and allow 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. Not all of these are supported yet.

Directories

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 (support for taking advantage of this feature does not yet exist in Glibc 2.2).

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

The current implementation of ext2 uses a singly-linked list to store the filenames in the directory; a pending enhancement uses hashing of the filenames to allow lookup without the need to scan the entire directory.

The current implementation never removes empty directory blocks once they have been allocated to hold more files.

Special files

Symbolic links are also filesystem objects with inodes. They deserve special mention because the data for them is stored within the inode itself if the symlink is less than 60 bytes long. 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.

Character and block special devices never have data blocks assigned to them. Instead, their device number is stored in the inode, again reusing the fields which would be used to point to the data blocks.

Reserved Space

In ext2, there is a mechanism for reserving a certain number of blocks for a particular user (normally the super-user). This is intended to allow for the system to continue functioning even if non-privileged users fill up all the space available to them (this is independent of filesystem quotas). It also keeps the filesystem from filling up entirely which helps combat fragmentation.

Filesystem check

At boot time, most systems run a consistency check (e2fsck) on their filesystems. The superblock of the ext2 filesystem contains several fields which indicate whether fsck should actually run (since checking the filesystem at boot can take a long time if it is large). fsck will run if the filesystem was not cleanly unmounted, if the maximum mount count has been exceeded or if the maximum time between checks has been exceeded.

Feature Compatibility

The compatibility feature mechanism used in ext2 is sophisticated. It safely allows features to be added to the filesystem, without unnecessarily sacrificing compatibility with older versions of the filesystem code. The feature compatibility mechanism is not supported by the original revision 0 (EXT2_GOOD_OLD_REV) of ext2, but was introduced in revision 1. There are three 32-bit fields, one for compatible features (COMPAT), one for read-only compatible (RO_COMPAT) features and one for incompatible (INCOMPAT) features.

These feature flags have specific meanings for the kernel as follows:

A COMPAT flag indicates that a feature is present in the filesystem, but the on-disk format is 100% compatible with older on-disk formats, so a kernel which didn't know anything about this feature could read/write the filesystem without any chance of corrupting the filesystem (or even making it inconsistent). This is essentially just a flag which says "this filesystem has a (hidden) feature" that the kernel or e2fsck may want to be aware of (more on e2fsck and feature flags later). The ext3 HAS_JOURNAL feature is a COMPAT flag because the ext3 journal is simply a regular file with data blocks in it so the kernel does not need to take any special notice of it if it doesn't understand ext3 journaling.

An RO_COMPAT flag indicates that the on-disk format is 100% compatible with older on-disk formats for reading (i.e. the feature does not change the visible on-disk format). However, an old kernel writing to such a filesystem would/could corrupt the filesystem, so this is prevented. The most common such feature, SPARSE_SUPER, is an RO_COMPAT feature because sparse groups allow file data blocks where superblock/group descriptor backups used to live, and ext2_free_blocks() refuses to free these blocks, which would leading to inconsistent bitmaps. An old kernel would also get an error if it tried to free a series of blocks which crossed a group boundary, but this is a legitimate layout in a SPARSE_SUPER filesystem.

An INCOMPAT flag indicates the on-disk format has changed in some way that makes it unreadable by older kernels, or would otherwise cause a problem if an old kernel tried to mount it. FILETYPE is an INCOMPAT flag because older kernels would think a filename was longer than 256 characters, which would lead to corrupt directory listings. The COMPRESSION flag is an obvious INCOMPAT flag - if the kernel doesn't understand compression, you would just get garbage back from read() instead of it automatically decompressing your data. The ext3 RECOVER flag is needed to prevent a kernel which does not understand the ext3 journal from mounting the filesystem without replaying the journal.

For e2fsck, it needs to be more strict with the handling of these flags than the kernel. If it doesn't understand ANY of the COMPAT, RO_COMPAT, or INCOMPAT flags it will refuse to check the filesystem, because it has no way of verifying whether a given feature is valid or not. Allowing e2fsck to succeed on a filesystem with an unknown feature is a false sense of security for the user. Refusing to check a filesystem with unknown features is a good incentive for the user to update to the latest e2fsck. This also means that anyone adding feature flags to ext2 also needs to update e2fsck to verify these features.

Metadata

It is frequently claimed that the ext2 implementation of writing asynchronous metadata is faster than the ffs synchronous metadata scheme but less reliable. Both methods are equally resolvable by their respective fsck programs.

If you're exceptionally paranoid, there are 3 ways of making metadata writes synchronous on ext2:

per-file if you have the program source: use the O_SYNC flag to open()
per-file if you don't have the source: use "chattr +S" on the file
per-filesystem: add the "sync" option to mount (or in /etc/fstab)

the first and last are not ext2 specific but do force the metadata to be written synchronously. See also Journaling below.

Limitations

There are various limits imposed by the on-disk layout of ext2. Other limits are imposed by the current implementation of the kernel code. Many of the limits are determined at the time the filesystem is first created, and depend upon the block size chosen. The ratio of inodes to data blocks is fixed at filesystem creation time, so the only way to increase the number of inodes is to increase the size of the filesystem. No tools currently exist which can change the ratio of inodes to blocks.

Most of these limits could be overcome with slight changes in the on-disk format and using a compatibility flag to signal the format change (at the expense of some compatibility).

Filesystem block size:	1kB	2kB	4kB	8kB
File size limit:	16GB	256GB	2048GB	2048GB
Filesystem size limit:	2047GB	8192GB	16384GB	32768GB

There is a 2.4 kernel limit of 2048GB for a single block device, so no filesystem larger than that can be created at this time. There is also an upper limit on the block size imposed by the page size of the kernel, so 8kB blocks are only allowed on Alpha systems (and other architectures which support larger pages).

There is an upper limit of 32000 subdirectories in a single directory.

There is a "soft" upper limit of about 10-15k files in a single directory with the current linear linked-list directory implementation. This limit stems from performance problems when creating and deleting (and also finding) files in such large directories. Using a hashed directory index (under development) allows 100k-1M+ files in a single directory without performance problems (although RAM size becomes an issue at this point).

The (meaningless) absolute upper limit of files in a single directory (imposed by the file size, the realistic limit is obviously much less) is over 130 trillion files. It would be higher except there are not enough 4-character names to make up unique directory entries, so they have to be 8 character filenames, even then we are fairly close to running out of unique filenames.

Journaling

A journaling extension to the ext2 code has been developed by Stephen Tweedie. It avoids the risks of metadata corruption and the need to wait for e2fsck to complete after a crash, without requiring a change to the on-disk ext2 layout. In a nutshell, the journal is a regular file which stores whole metadata (and optionally data) blocks that have been modified, prior to writing them into the filesystem. This means it is possible to add a journal to an existing ext2 filesystem without the need for data conversion.

When changes to the filesystem (e.g. a file is renamed) they are stored in a transaction in the journal and can either be complete or incomplete at the time of a crash. If a transaction is complete at the time of a crash (or in the normal case where the system does not crash), then any blocks in that transaction are guaranteed to represent a valid filesystem state, and are copied into the filesystem. If a transaction is incomplete at the time of the crash, then there is no guarantee of consistency for the blocks in that transaction so they are discarded (which means any filesystem changes they represent are also lost).
Check Documentation/filesystems/ext3.txt if you want to read more about ext3 and journaling.

References

```
The kernel source       file:/usr/src/linux/fs/ext2/
e2fsprogs (e2fsck)      http://e2fsprogs.sourceforge.net/
Design & Implementation http://e2fsprogs.sourceforge.net/ext2intro.html
Journaling (ext3)       ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/
Filesystem Resizing     http://ext2resize.sourceforge.net/
Compression (*)         http://e2compr.sourceforge.net/

Implementations for:
Windows 95/98/NT/2000   http://www.chrysocome.net/explore2fs
Windows 95 (*)          http://www.yipton.net/content.html#FSDEXT2
DOS client (*)          ftp://metalab.unc.edu/pub/Linux/system/filesystems/ext2/
OS/2 (+)                ftp://metalab.unc.edu/pub/Linux/system/filesystems/ext2/
RISC OS client
http://www.esw-heim.tu-clausthal.de/~marco/smorbrod/IscaFS/
```

(*) no longer actively developed/supported (as of Apr 2001)
(+) no longer actively developed/supported (as of Mar 2009)