SQUASHFS 4.0 FILESYSTEM

Squashfs is a compressed read-only filesystem for Linux.
It uses zlib compression to compress files, inodes and directories.
Inodes in the system are very small and all blocks are packed to minimise
data overhead. Block sizes greater than 4K are supported up to a maximum
of 1Mbytes (default block size 128K).

Squashfs is intended for general read-only filesystem use, for archival
use (i.e. in cases where a .tar.gz file may be used), and in constrained
block device/memory systems (e.g. embedded systems) where low overhead is
needed.

Mailing list: squashfs-devel@lists.sourceforge.net
Web site: www.squashfs.org

1. FILESYSTEM FEATURES

Squashfs filesystem features versus Cramfs:

	Squashfs	Cramfs
Max filesystem size:	2^64	256 MiB
Max file size:	~ 2 TiB	16 MiB
Max files:	unlimited	unlimited
Max directories:	unlimited	unlimited
Max entries per directory:	unlimited	unlimited
Max block size:	1 MiB	4 KiB
Metadata compression:	yes	no
Directory indexes:	yes	no
Sparse file support:	yes	no
Tail-end packing (fragments):	yes	no
Exportable (NFS etc.):	yes	no
Hard link support:	yes	no
"." and ".." in readdir:	yes	no
Real inode numbers:	yes	no
32-bit uids/gids:	yes	no
File creation time:	yes	no
Xattr support:	yes	no
ACL support:	no	no

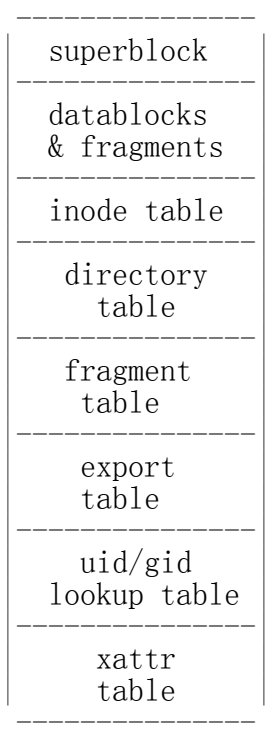
Squashfs compresses data, inodes and directories. In addition, inode and
directory data are highly compacted, and packed on byte boundaries. Each
compressed inode is on average 8 bytes in length (the exact length varies on
file type, i.e. regular file, directory, symbolic link, and block/char device
inodes have different sizes).

2. USING SQUASHFS

As squashfs is a read-only filesystem, the mksquashfs program must be used to
create populated squashfs filesystems. This and other squashfs utilities
can be obtained from http://www.squashfs.org. Usage instructions can be
obtained from this site also.

3. SQUASHFS FILESYSTEM DESIGN

A squashfs filesystem consists of a maximum of eight parts, packed together on a byte
alignment:



Compressed data blocks are written to the filesystem as files are read from the source directory, and checked for duplicates. Once all file data has been written the completed inode, directory, fragment, export and uid/gid lookup tables are written.

3.1 Inodes

Metadata (inodes and directories) are compressed in 8Kbyte blocks. Each compressed block is prefixed by a two byte length, the top bit is set if the block is uncompressed. A block will be uncompressed if the -noI option is set, or if the compressed block was larger than the uncompressed block.

Inodes are packed into the metadata blocks, and are not aligned to block boundaries, therefore inodes overlap compressed blocks. Inodes are identified by a 48-bit number which encodes the location of the compressed metadata block containing the inode, and the byte offset into that block where the inode is placed (<block, offset>).

To maximise compression there are different inodes for each file type (regular file, directory, device, etc.), the inode contents and length varying with the type.

To further maximise compression, two types of regular file inode and
directory inode are defined: inodes optimised for frequently occurring
regular files and directories, and extended types where extra
information has to be stored.

3.2 Directories

Like inodes, directories are packed into compressed metadata blocks, stored
in a directory table. Directories are accessed using the start address of
the metablock containing the directory and the offset into the
decompressed block (<block, offset>).

Directories are organised in a slightly complex way, and are not simply
a list of file names. The organisation takes advantage of the
fact that (in most cases) the inodes of the files will be in the same
compressed metadata block, and therefore, can share the start block.
Directories are therefore organised in a two level list, a directory
header containing the shared start block value, and a sequence of directory
entries, each of which share the shared start block. A new directory header
is written once/if the inode start block changes. The directory
header/directory entry list is repeated as many times as necessary.

Directories are sorted, and can contain a directory index to speed up
file lookup. Directory indexes store one entry per metablock, each entry
storing the index/filename mapping to the first directory header
in each metadata block. Directories are sorted in alphabetical order,
and at lookup the index is scanned linearly looking for the first filename
alphabetically larger than the filename being looked up. At this point the
location of the metadata block the filename is in has been found.
The general idea of the index is ensure only one metadata block needs to be
decompressed to do a lookup irrespective of the length of the directory.
This scheme has the advantage that it doesn't require extra memory overhead
and doesn't require much extra storage on disk.

3.3 File data

Regular files consist of a sequence of contiguous compressed blocks, and/or a
compressed fragment block (tail-end packed block).   The compressed size
of each datablock is stored in a block list contained within the
file inode.

To speed up access to datablocks when reading 'large' files (256 Mbytes or
larger), the code implements an index cache that caches the mapping from
block index to datablock location on disk.

The index cache allows Squashfs to handle large files (up to 1.75 TiB) while
retaining a simple and space-efficient block list on disk. The cache
is split into slots, caching up to eight 224 GiB files (128 KiB blocks).
Larger files use multiple slots, with 1.75 TiB files using all 8 slots.
The index cache is designed to be memory efficient, and by default uses
16 KiB.

3.4 Fragment lookup table

Regular files can contain a fragment index which is mapped to a fragment location on disk and compressed size using a fragment lookup table. This fragment lookup table is itself stored compressed into metadata blocks. A second index table is used to locate these. This second index table for speed of access (and because it is small) is read at mount time and cached in memory.

3.5 Uid/gid lookup table

For space efficiency regular files store uid and gid indexes, which are converted to 32-bit uids/gids using an id look up table. This table is stored compressed into metadata blocks. A second index table is used to locate these. This second index table for speed of access (and because it is small) is read at mount time and cached in memory.

3.6 Export table

To enable Squashfs filesystems to be exportable (via NFS etc.) filesystems can optionally (disabled with the -no-exports Mksquashfs option) contain an inode number to inode disk location lookup table. This is required to enable Squashfs to map inode numbers passed in filehandles to the inode location on disk, which is necessary when the export code reinstantiates expired/flushed inodes.

This table is stored compressed into metadata blocks. A second index table is used to locate these. This second index table for speed of access (and because it is small) is read at mount time and cached in memory.

3.7 Xattr table

The xattr table contains extended attributes for each inode. The xattrs for each inode are stored in a list, each list entry containing a type, name and value field. The type field encodes the xattr prefix ("user.", "trusted." etc) and it also encodes how the name/value fields should be interpreted. Currently the type indicates whether the value is stored inline (in which case the value field contains the xattr value), or if it is stored out of line (in which case the value field stores a reference to where the actual value is stored). This allows large values to be stored out of line improving scanning and lookup performance and it also allows values to be de-duplicated, the value being stored once, and all other occurences holding an out of line reference to that value.

The xattr lists are packed into compressed 8K metadata blocks.
To reduce overhead in inodes, rather than storing the on-disk location of the xattr list inside each inode, a 32-bit xattr id is stored. This xattr id is mapped into the location of the xattr list using a second xattr id lookup table.

4. TODOS AND OUTSTANDING ISSUES

4.1 Todo list

Implement ACL support.

4.2 Squashfs internal cache

Blocks in Squashfs are compressed. To avoid repeatedly decompressing recently accessed data Squashfs uses two small metadata and fragment caches.

The cache is not used for file datablocks, these are decompressed and cached in the page-cache in the normal way. The cache is used to temporarily cache fragment and metadata blocks which have been read as a result of a metadata (i.e. inode or directory) or fragment access. Because metadata and fragments are packed together into blocks (to gain greater compression) the read of a particular piece of metadata or fragment will retrieve other metadata/fragments which have been packed with it, these because of locality-of-reference may be read in the near future. Temporarily caching them ensures they are available for near future access without requiring an additional read and decompress.

In the future this internal cache may be replaced with an implementation which uses the kernel page cache. Because the page cache operates on page sized units this may introduce additional complexity in terms of locking and associated race conditions.