The LogFS Flash Filesystem

Specification

Superblocks

Two superblocks exist at the beginning and end of the filesystem.
Each superblock is 256 Bytes large, with another 3840 Bytes reserved
for future purposes, making a total of 4096 Bytes.

Superblock locations may differ for MTD and block devices. On MTD the
first non-bad block contains a superblock in the first 4096 Bytes and
the last non-bad block contains a superblock in the last 4096 Bytes.
On block devices, the first 4096 Bytes of the device contain the first
superblock and the last aligned 4096 Byte-block contains the second
superblock.

For the most part, the superblocks can be considered read-only. They
are written only to correct errors detected within the superblocks,
move the journal and change the filesystem parameters through tunefs.
As a result, the superblock does not contain any fields that require
constant updates, like the amount of free space, etc.

Segments

The space in the device is split up into equal-sized segments.
Segments are the primary write unit of LogFS. Within each segments,
writes happen from front (low addresses) to back (high addresses. If
only a partial segment has been written, the segment number, the
current position within and optionally a write buffer are stored in
the journal.

Segments are erased as a whole. Therefore Garbage Collection may be
required to completely free a segment before doing so.

Journal

The journal contains all global information about the filesystem that
is subject to frequent change. At mount time, it has to be scanned
for the most recent commit entry, which contains a list of pointers to
all currently valid entries.

Object Store

All space except for the superblocks and journal is part of the object
store. Each segment contains a segment header and a number of
objects, each consisting of the object header and the payload.
Objects are either inodes, directory entries (dentries), file data
blocks or indirect blocks.

Levels

Garbage collection (GC) may fail if all data is written
indiscriminately. One requirement of GC is that data is separated
roughly according to the distance between the tree root and the data.
Effectively that means all file data is on level 0, indirect blocks
are on levels 1, 2, 3 4 or 5 for 1x, 2x, 3x, 4x or 5x indirect blocks,
respectively. Inode file data is on level 6 for the inodes and 7-11
for indirect blocks.

Each segment contains objects of a single level only. As a result,
each level requires its own separate segment to be open for writing.

Inode File

All inodes are stored in a special file, the inode file. Single
exception is the inode file's inode (master inode) which for obvious
reasons is stored in the journal instead. Instead of data blocks, the
leaf nodes of the inode files are inodes.

Aliases

Writes in LogFS are done by means of a wandering tree. A naïve
implementation would require that for each write or a block, all
parent blocks are written as well, since the block pointers have
changed. Such an implementation would not be very efficient.

In LogFS, the block pointer changes are cached in the journal by means
of alias entries. Each alias consists of its logical address - inode
number, block index, level and child number (index into block) - and
the changed data. Any 8-byte word can be changes in this manner.

Currently aliases are used for block pointers, file size, file used
bytes and the height of an inodes indirect tree.

Segment Aliases

Related to regular aliases, these are used to handle bad blocks.
Initially, bad blocks are handled by moving the affected segment
content to a spare segment and noting this move in the journal with a
segment alias, a simple (to, from) tupel. GC will later empty this
segment and the alias can be removed again. This is used on MTD only.

Vim

By cleverly predicting the life time of data, it is possible to
separate long-living data from short-living data and thereby reduce
the GC overhead later. Each type of distinc life expectency (vim) can
have a separate segment open for writing. Each (level, vim) tupel can
be open just once. If an open segment with unknown vim is encountered

at mount time, it is closed and ignored henceforth.

Indirect Tree

Inodes in LogFS are similar to FFS-style filesystems with direct and indirect block pointers. One difference is that LogFS uses a single indirect pointer that can be either a 1x, 2x, etc. indirect pointer. A height field in the inode defines the height of the indirect tree and thereby the indirection of the pointer.

Another difference is the addressing of indirect blocks. In LogFS, the first 16 pointers in the first indirect block are left empty, corresponding to the 16 direct pointers in the inode. In ext2 (maybe others as well) the first pointer in the first indirect block corresponds to logical block 12, skipping the 12 direct pointers. So where ext2 is using arithmetic to better utilize space, LogFS keeps arithmetic simple and uses compression to save space.

Compression

Both file data and metadata can be compressed. Compression for file data can be enabled with chattr +c and disabled with chattr -c. Doing so has no effect on existing data, but new data will be stored accordingly. New inodes will inherit the compression flag of the parent directory.

Metadata is always compressed. However, the space accounting ignores this and charges for the uncompressed size. Failing to do so could result in GC failures when, after moving some data, indirect blocks compress worse than previously. Even on a 100% full medium, GC may not consume any extra space, so the compression gains are lost space to the user.

However, they are not lost space to the filesystem internals. By cheating the user for those bytes, the filesystem gained some slack space and GC will run less often and faster.

Garbage Collection and Wear Leveling

Garbage collection is invoked whenever the number of free segments falls below a threshold. The best (known) candidate is picked based on the least amount of valid data contained in the segment. All remaining valid data is copied elsewhere, thereby invalidating it.

The GC code also checks for aliases and writes then back if their number gets too large.

Wear leveling is done by occasionally picking a suboptimal segment for garbage collection. If a stale segments erase count is significantly lower than the active segments' erase counts, it will be picked. Wear leveling is rate limited, so it will never monopolize the device for more than one segment worth at a time.

Values for "occasionally", "significantly lower" are compile time constants.

Hashed directories

To satisfy efficient lookup(), directory entries are hashed and located based on the hash. In order to both support large directories and not be overly inefficient for small directories, several hash tables of increasing size are used. For each table, the hash value modulo the table size gives the table index.

Tables sizes are chosen to limit the number of indirect blocks with a fully populated table to 0, 1, 2 or 3 respectively. So the first table contains 16 entries, the second 512-16, etc.

The last table is special in several ways. First its size depends on the effective 32bit limit on telldir/seekdir cookies. Since logfs uses the upper half of the address space for indirect blocks, the size is limited to 2^31. Secondly the table contains hash buckets with 16 entries each.

Using single-entry buckets would result in birthday "attacks". At just 2^16 used entries, hash collisions would be likely (P >= 0.5). My math skills are insufficient to do the combinatorics for the 17x collisions necessary to overflow a bucket, but testing showed that in 10,000 runs the lowest directory fill before a bucket overflow was 188,057,130 entries with an average of 315,149,915 entries. So for directory sizes of up to a million, bucket overflows should be virtually impossible under normal circumstances.

With carefully chosen filenames, it is obviously possible to cause an overflow with just 21 entries (4 higher tables + 16 entries + 1). So there may be a security concern if a malicious user has write access to a directory.

Open For Discussion
===================

Device Address Space

A device address space is used for caching. Both block devices and MTD provide functions to either read a single page or write a segment. Partial segments may be written for data integrity, but where possible complete segments are written for performance on simple block device flash media.

Meta Inodes

Inodes are stored in the inode file, which is just a regular file for most purposes. At umount time, however, the inode file needs to remain open until all dirty inodes are written. So generic_shutdown_super() may not close this inode, but shouldn't complain about remaining inodes due to the inode file either. Same

goes for mapping inode of the device address space.

Currently logfs uses a hack that essentially copies part of fs/inode.c
code over. A general solution would be preferred.

Indirect block mapping

With compression, the block device (or mapping inode) cannot be used
to cache indirect blocks. Some other place is required. Currently
logfs uses the top half of each inode's address space. The low 8TB
(on 32bit) are filled with file data, the high 8TB are used for
indirect blocks.

One problem is that 16TB files created on 64bit systems actually have
data in the top 8TB. But files >16TB would cause problems anyway, so
only the limit has changed.