LogFS - finally a scalable flash file system

Jörn Engel
IBM Deutschland Entwicklung
<joern@wh.fh-wedel.de>

Robert Mertens
University of Osnabrück
<robert.mertens@uos.de>

Abstract

Currently, two different solutions for filesystems on flash storage exist: Flash Translation Layers in combination with traditional filesystems and log-structured flash file systems. This paper analyzes the weaknesses of both approaches and derives a number of requirements that help avoiding these weaknesses. Finally the basic concept of a tree structured flash file system fulfilling all of these requirements is presented.

1 Flash

1.1 Flash introduction

Many types of flash memory exist - NAND, NOR, AG-AND, ECC-NOR, ... All of them differ in details, but share the same principles. Flash differentiates between write and erase. Empty flash is completely filled with ones. A flash write will flip individual bits from 1 to 0. The only way to toggle bits back from 0 to 1 is by performing a flash erase.

Erases happen in coarse granularities of an erase block, which is usually a power of two from 32k to 256k. Writes can happen in smaller granularities of as little as one bit (NOR), 8-16 bytes (ECC-NOR), 256, 512 or 2048 bytes (NAND, AG-AND).

NAND manufacturers call the write granularity a ”page”, but most OS developers would confuse this with an MMU page, so the name should not be used. Filesystems also work with blocks, which are quite unlike flash erase blocks. For the purpose of this paper, the flash write granularity shall be a ”write block”, the erase granularity an ”erase block” and the filesystem granularity an ”fs block”.

1.2 Flash limitations

Hardware manufacturers aim to use flash as hard disk replacements. There are however a few relevant differences between the two that affect filesystem design:

1. Flash requires out of place updates of existing data. Before being able to write to a specific location, that erase block has to be erased. After being erased, all bits are set to 1. An unclean unmount at this time will cause data loss, as neither the old nor the new data can be retrieved.
2. Lifetime of flash erase blocks is limited by the number of erase cycles on them. Hardware manufacturers usually guarantee 100.000 erase cycles. This number is per individual erase block. Hard disks have no such limitation.
3. Erase blocks are significantly larger than hard disk sectors or filesystem blocks. Therefore, erase blocks must be shared by several filesystem blocks. During operation, erase blocks get partially obsoleted and require garbage collection to free space. For details, see 1.3.

When comparing these limitation with any filesystem designed for hard disk usage, the result is obvious. Trying to use conventional filesystems on raw flash will turn new and expensive devices into paperweights.

1.3 Garbage collection

Garbage collection for flash filesystems closely follows the "segment cleaning" methods described in [1]. Data is not deleted, but obsoleted. Obsolete data still occupies space, and cannot be deleted without also deleting valid data in the same erase block.

Therefore a garbage collector will clean an erase block by moving all valid data into a free block, obsoleting it in the old erase block. After this, all data in the old erase block is obsolete and the block can be deleted. The space previously consumed by obsoleted data is the net gain of this operation.

An important property during garbage collection is the ratio of obsolete and valid data. Garbage collecting blocks with 50% obsolete data requires processing of two blocks to gain one free. With only 10% obsolete data, ten blocks need to be processed for the same result. Obviously, the goal is to collect blocks containing as much obsolete data as possible.

2 Current flash users

2.1 Flash translation layers

One strategy to deal with the limitation described in 1.2 is to create a virtual block device for use by a regular filesystem. This is what the various flash translation layers (FTL) do. The three differences are dealt with inside the FTL - to the filesystem the virtual device supports writes of multiple sectors of 512 bytes to aligned offsets.

A write of block B to the virtual block device causes an FTL to do several things. First, the new content of block B is written to flash. Second, the flash area associated with the old content is obsoleted. Third, garbage collection may take place to free new blocks for subsequent writes to flash. Depending on the concrete FTL used, additional writes to flash may be necessary to store the mapping between block device and flash content, account for free blocks, etc.

But even in a perfect implementation, an FTL will still be inefficient. The block device abstraction has no distinction between free and used data - both are just the same. The distinction between free and used data on the block device can only be made by the user - in this case a filesystem. Independently of the content, the complete block device must be viewed as valid data.[1] Therefore, the flash medium contains only little obsolete data and garbage collection has a bad ratio of obsolete/valid data.

With complete knowledge of the filesystems internal structures, the FTL could detect deleted files and treat blocks previously occupied by these files as undefined and hence obsolete data. But filesystems never tell the FTL about obsolete blocks (i.e. due to file deletion), so the FTL could only guess by analyzing data written to the block device, for example by looking at the Ext2 free block bitmaps. But if such a guess ever goes wrong, data will be lost. Therefore, such techniques have only been implemented for FAT and even there range between ineffective and dangerous.

This GC-induced movement of obsolete data makes an FTL both slower than necessary and reduces the flash lifetime. People wanted something better.

[1]Actually, this is not quite true. Initially the whole content of the block device is undefined and can be treated as obsolete data. The state mentioned above becomes true once the complete block device contains defined data, i.e. has been written.

2.2 Flash filesystems

A flash filesystem differs from regular filesystems by operating directly on a flash device - MTD in Linux - instead of an intermediate block device. It is well aware of the difference between flash write and flash erase, knows the size of a erase block and can perform garbage collection.

All existing flash filesystems in the FOSS world, JFFS, JFFS2 and YAFFS, have a log structured design. All writes to flash happen in units of a node, which basically consists of some file data, the inode number, an offset within the inode, the data length and a version number[2]. See [3] for more details.

Updating an existing area in a file is done by simply writing out another node for this inode and offset, with a version number higher than all previous ones. So, as for all log structured filesystems, writes are extremely simple and fast.

Reading a file consists of finding all nodes for it, discarding those that have been replaced by newer nodes, and assembling everything. Since it is nowhere defined where the node for any particular file can be found, this effectively requires searching the complete medium for nodes - an expensive operation called a scan.

Performing a full scan on every file read would result in unacceptable benchmark numbers, so this operation is done just once during mount time. From the data retrieved, an in-memory tree is created, holding all necessary information to find all nodes for any particular file without doing the full scan.

But this mount time scan is still expensive. On the authors notebook, mounting an empty JFFS2 on a 1GiB USB stick takes around 15 minutes. That is a little slower than most users would expect a filesystem mount to happen. On top of that, the in-memory tree requires several hundred kiB or MiB, an amount that matters a lot for embedded systems.

Over the last few years, users have constantly complained about both mount time and memory usage. YAFFS has a slightly optimized de-

[2]Other fields also exists, but don't matter at this point

sign in this area, requiring less of each. But all existing flash filesystems are still O(n) - with different constant factors.

3 LogFS

3.1 Requirements

Based on above analysis, we can deduce several requirements that a scalable flash filesystem must fulfill:

1. Filesystem design is based on a tree, similar to Ext2.
2. Updates must be done out-of-place, unlike Ext2.
3. The root node must be locatable in O(1) time.
4. The filesystem must work on raw flash, without FTL.
5. Data corruption on crashes must be prevented.

The following is a design proposal that fulfills above requirements. It is by no means the only possible design[3], but likely the first. Where possible, it is based on Ext2 - the most commonly used filesystem for Linux.

3.2 Wandering trees

```
  A                                                     A'
 / \                                      C'           / \
B   C                         E'         / \          B   C'
   / \    becomes:  G'  =>   / \    =>  D   E'    =>     / \
  D   E                     F   G'         / \          D   E'
     / \                                  F   G'           / \
    F   G                                                 F   G'
```

Wandering trees allow tree updates with out-of-place writes. This means an update of node G is done in the following steps. First, a new node G' is written first. Then, the parent node E is updated by writing a new node E', pointing to G' and all previous children of E, is written. Continuing up the tree, all parents are updated in this manner until the root A is replaced with a new root A'.

[3]Artem Bityuckiy is working on a different design

3.3 Anchor nodes

Anchor nodes are the necessary amount of log-structured design to find a tree's root node in flash. A small area of at least two erase blocks are set apart for (version—offset) tuples. As usual, the highest version wins and all older ones are ignored. Therefore, the complete anchor area needs to be scanned once at mount time to find the newest anchor - which points to the current root node.

After an update to the wandering tree, all that is still required is to write a new anchor node with an incremented version number to finalize the data changes.

3.4 The hardlink problem

```
  A             A'
 / \           / \
B   C         B   C'
   / \    =>    /   \
  D   E        D     E'
   \ / \        \   / \
    F   G        F F'  G
```

The user view of a filesystem is often perceived as a tree, but under Unix this is only true for directories. Regular files can have several hard links, which effectively merges tree branches again. Since inodes only have information about the number of hard links to them, not their origins, it is impossible to follow all paths to a particular file without walking the complete directory tree.

Hard link handling is not a problem for hard disk based filesystems, since updates can happen in-place. With out-of-place updates, all references to an inode would need to be updated.
If we used a Unix filesystem structure directly as basis for our wandering trees, every update would automatically break hard links. While this can be a useful property as well [4], it breaks POSIX semantics. Therefore we need something more sophisticated.

3.5 Inode file

Unlike Ext2, LogFS does not have a specialized disk area to store inodes in. Instead, it uses a specialized file, the inode file. Apart from simplifying the code - file writes and inode writes are identical now -

this is the necessary layer of abstraction to solve the hardlink problem.

With an inode file, the tree consists of only two levels, the inode file being the root node and every inode being a leaf node. Directory entries simply contain the inode number, which is used as an offset within the inode file.

```
 Inode file
| | | | | | |
A B C D E F G
```

Hence, there is only one remaining reference to any inode - from the inode file. The data structure is actually a tree again and permits use of the wandering tree strategy. Plus, when walking up the tree, only the changed node and the inode file need to be updated.

3.6 Name choice

The name "LogFS" was picked as a development title only. It is a bad joke about JFFS2, the Journalling Flash Filesystem that is not journalling but log-structured. LogFS would be the first flash filesystem that is not log-structures, so its name would be just as wrong as JFFS2.
Proposals for a product name are welcome.

4 Conclusion

This paper has demonstrated the lack of a scalable flash filesystem and analyzed existing solutions, pointing out their individual shortcomings. A requirement list for a working solution is given and some of the central building blocks for a scalable flash filesystem are explained. While this is an important step, there is still work missing, esp. in the area of garbage collection and block handling - which will be presented in a future paper.

References

[1] The Design and Implementation of a Log-Structured File System http://citeseer.ist.psu.edu/rosenblum91design.html

[2] Understanding the Flash Translation Layer (FTL) Specification
http://www.intel.com/design/flcomp/applnots/29781602.pdf

[3] JFFS: The Journalling Flash File System
http://sources.redhat.com/jffs2/jffs2.pdf

[4] Cowlinks
http://wohnheim.fh-wedel.de/~joern/cowlink/