The Journalling Flash File System

http://sources.redhat.com/jffs2/

David Woodhouse
dwmw2@cambridge.redhat.com

The Grand Plan

- What is Flash?
- How is it used?
  - Flash Translation Layer (FTL)
  - NFTL
- Better ways of using it
  - JFFS
  - JFFS2
- The Future

Flash memory technology - NOR flash

- Low power, high density non-volatile storage
- Linearly accessible memory
- Individually clearable bits
- Bits reset only in *"erase blocks"* of typically 128KiB
- Limited lifetime - typ. 100,000 erase cycles

Flash memory technology - NAND flash

- Cheaper, higher tolerances than NOR flash
- Smaller erase blocks *(typ. 8 KiB)*
- Subdivided into 512 byte *"pages"*
- Not linearly accessible
- Uniform interface — 8-bit data/address bus + 3 control lines
- "Out-Of-Band" data storage - 16 bytes in 512 for metadata/ECC

So what do we do with it?

Traditional answer (FTL and NFTL):

- Emulate a standard block device
- Use a normal file system on top of that

So what do we do with it?

Traditional answer (FTL and NFTL):

- Emulate a standard block device
- Use a normal file system on top of that

This sucks. Obviously you need a journalling file system on your emulated block device, which is itself a kind of journalling pseudo-filesystem. Two layers of journalling on top of each other aren't the best way to ensure efficient operation.

#include "CompactFlash_is_not_flash.h"

Can we do better?

Yes!

We want a journalling file system designed specifically for use on flash devices, with built-in wear levelling.

This lends itself to a purely log-structured file system writing log nodes directly to the flash. The log-structured nature of such a file system will provide automatic wear levelling.

And lo... our prayers were answered

In 1999, Axis Communications AB released exactly the file system that we had been talking about.

- Log structured file system
- Direct operation on flash devices
- GPL'd code for Linux 2.0.

And lo... our prayers were answered

In 1999, Axis Communications AB released exactly the file system that we had been talking about.

- Log structured file system
- Direct operation on flash devices
- GPL'd code for Linux 2.0.

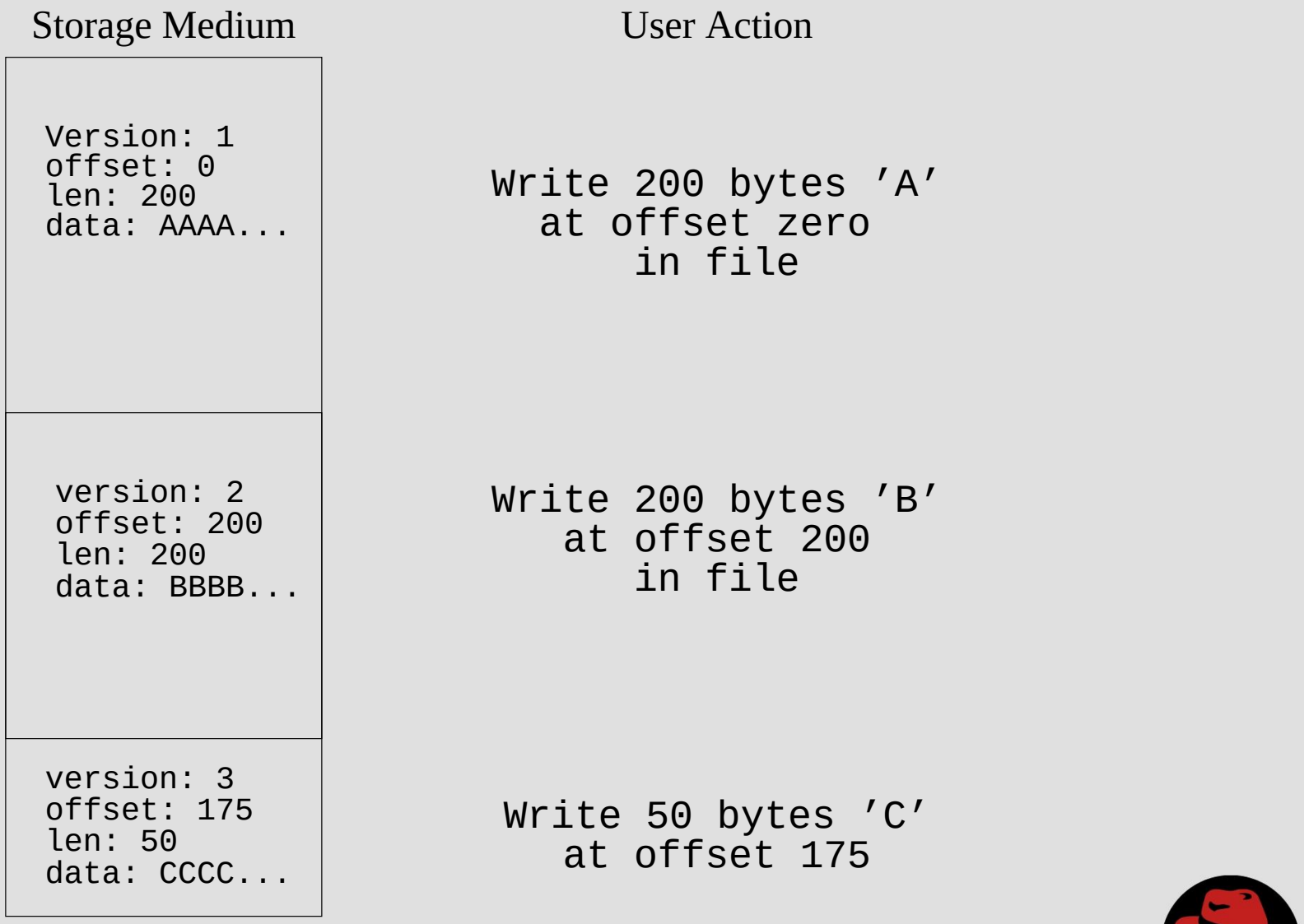
Ported to 2.4 and the generic Memory Technology Device system by a developer in Sweden, and subsequently backported to 2.2 by Red Hat for a customer to use in a web pad device.

What does "Log structured" mean?

- Data stored on medium in no particular location

- Packets, or "nodes" of data written sequentially to a log which records all changes, containing:
  - Identification of file to which the node belongs
  - A "version" field, indicating the chronological sequence of the nodes belonging to this file
  - Current inode metadata (uid, gid, etc.)
  - Optionally: Some data, and the offset within the file at which the data should appear

What does “Log structured” mean?



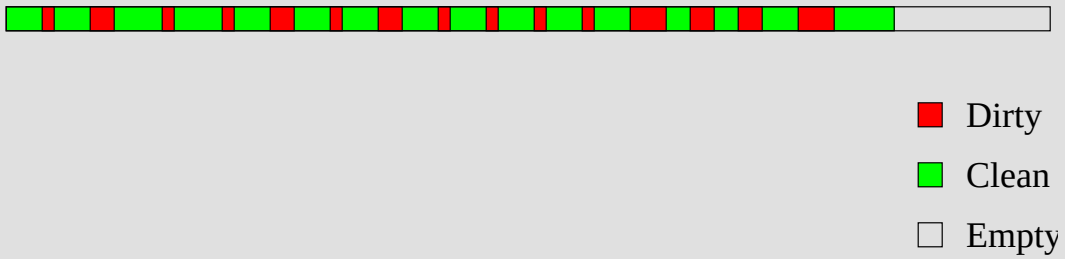
Playing back the log

To read the file system, the log nodes are played back in version order, to recreate a map of where each range of data is located on the physical medium.

Node playback	List State
Node version 1: 200 bytes @ 0	0–200: v1
Node version 2: 200 bytes @ 200	0–200: v1 200–400: v2
Node version 3: 50 bytes @ 175	0–175: v1 175–225: v3 225–400: v2

Dirty space

Some nodes are completely obsoleted by later writes to the same location in the file. They create “dirty space” within the file system.



Garbage Collection

So far so good. But soon the log reaches the end of the medium. At this point we need to start to reclaim some of the dirty space.

Garbage Collection

So far so good. But soon the log reaches the end of the medium. At this point we need to start to reclaim some of the dirty space.

So we copy the still-valid data from the beginning of the log to the remaining space at the end...

Garbage Collection

So far so good. But soon the log reaches the end of the medium. At this point we need to start to reclaim some of the dirty space.

So we copy the still-valid data from the beginning of the log to the remaining space at the end...

Garbage Collection

So far so good. But soon the log reaches the end of the medium. At this point we need to start to reclaim some of the dirty space.

So we copy the still-valid data from the beginning of the log to the remaining space at the end...

...until we can erase a block at the start.

Limitations of the original JFFS

- Poor garbage collection performance on full file systems

- No compression
- File names and parent inode stored in each node along with other metadata
  - Wasting space
  - Preventing POSIX hard links

Enter JFFS2

JFFS2 started off as a project to add compression to JFFS, but because of the other problems with JFFS, it seemed like the right time to do a complete rewrite to address them all at once.

- Non-sequential log structure
- Compression
- Different node types on medium
- Improved memory usage

Log structure

Erase blocks are treated individually and references to each are stored on one of many lists in the JFFS2 data structures.

- `clean_list` — Erase blocks with only valid nodes
- `dirty_list` — Erase blocks with one or more obsoleted nodes
- `free_list` — Empty erase blocks waiting to be filled
- ...and others...

Garbage Collection

- 99 times in 100, pick a block from the `dirty_list` to be garbage collected, for optimal performance
- The remaining 1 in 100 times, pick a clean block, to ensure that data are moved around the medium and wear levelling is achieved

Compression

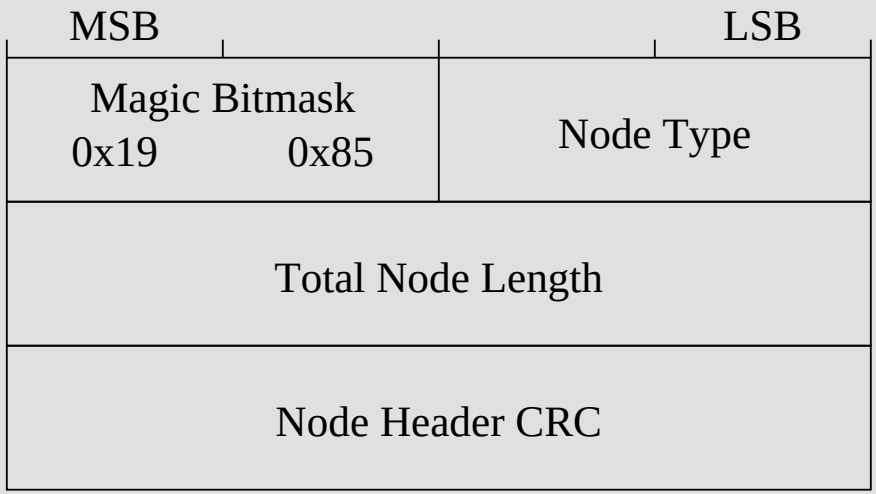
Although ostensibly the purpose of the exercise, compression was the easy part. Some useful and quick compression algorithms were implemented, followed by the import of yet another copy of zlib.c into the kernel tree.

In order to facilitate quick decompression, data are compressed in chunks no larger than the hardware page size.

Node types - common node header

JFFS2 introduces different node types for the entries in the log, where JFFS only used one type of structure in the log.

The nodes share a common layout, allowing JFFS2 implementations which don't understand a new node type to deal with it appropriately.



Compatibility types

The Node Type field in the header has a unique identification number for the node type, and the two most significant bits are used to indicate the expected behaviour if the node is not supported.

- JFFS2_FEATURE_INCOMPAT
- JFFS2_FEATURE_ROCOMPAT
- JFFS2_FEATURE_RWCOMPAT_DELETE
- JFFS2_FEATURE_RWCOMPAT_COPY

Directory entry nodes

- Parent (directory) inode number
- Name
- Inode number
- Version

Inode number zero used to signify unlinking

Inode data nodes

Very similar to JFFS v1 nodes, except without the parent and filename fields:

- User ID, Group ID, Permissions, etc.
- Current inode size
- Optional data, not crossing page boundary, possibly compressed

Clean block marker nodes

Introduced to deal with the problem of partially-erased blocks.

Losing power during an erase cycle can result in a block which appears to be erased, but which contains a few bits which are in fact returning random data.

Writing a marker to the beginning of the block after successful completion of an erase cycle allows JFFS2 to be certain the block is in a usable state.

Memory Usage

Polite behaviour under system memory pressure through normal actions of VM — `prune_icache`

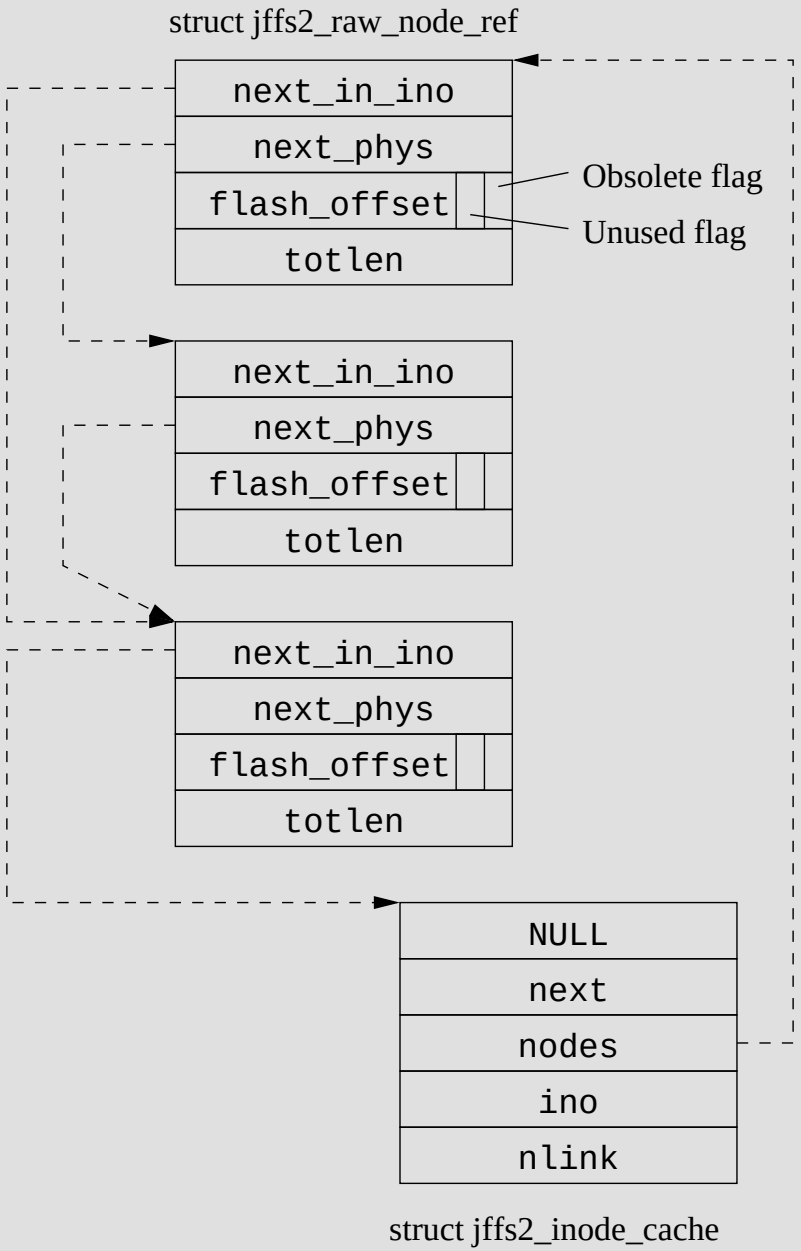
- Store in-core at all times only the bare minimum amount of data required to find inodes
- Build full map of data regions for an inode only on `read_inode()` being called
- Free all extra data on `clear_inode()`

Mounting a JFFS2 filesystem

Four-pass process

- Physical scan, allocating data structures and caching node information.
- Pass 1: Build data maps and calculate nlink for each inode, adding `jffs2_inode_cache` entries to hash table.
- Pass 2: Delete inodes with nlink == 0
- Pass 3: Free temporary cached information

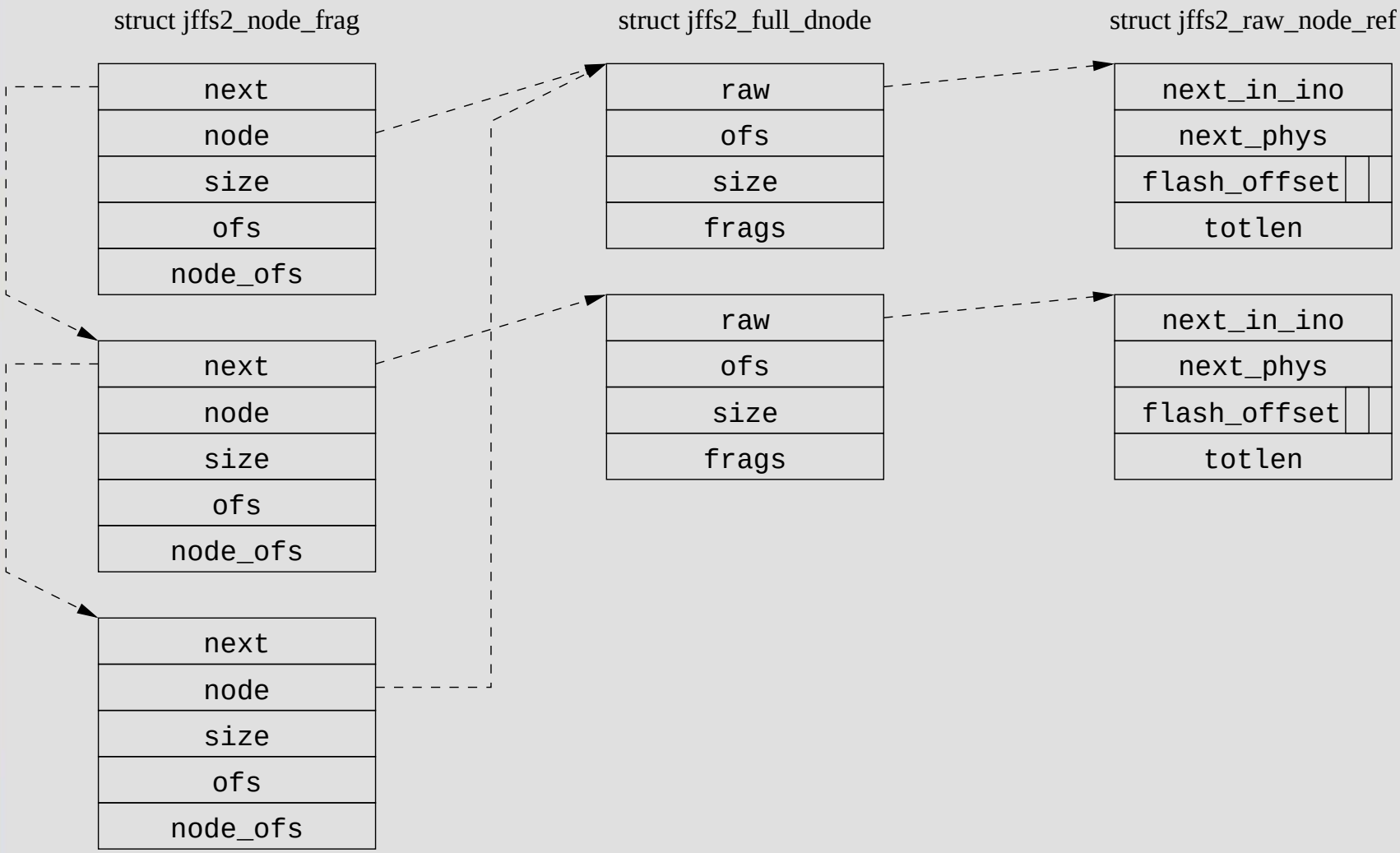
Data structures - raw node tracking



Read inode

On `jffs2_read_inode()` calls, we look up the `jffs2_inode_cache` in the hash table, and read each physical node belonging to the inode in question, building up a *fraglist* representing the whole range of data in the file.

Data structures - node fragments



File read

- Look up file range in fraglist.
- For each frag in range:
  - Call `jffs2_read_dnode()` to read the range indicated by the node fragment.

File read

- Look up file range in fraglist.
- For each frag in range:
  - Call `jffs2_read_dnode()` to read the range indicated by the node fragment.

This means that where two ranges of bytes in a given node are visible, we read and decompress the whole node twice. We could probably optimise this to do only one read/decompress cycle.

Flash space allocation

Allocate flash space with `jffs2_reserve_space()` function:

- Caller specifies the *minimum* acceptable allocation.
- Garbage collection is triggered if necessary to make space.
- Returns the *maximum* amount of space which is currently available (or `-ENOSPC`).
- Successful allocations lock the `alloc_sem` semaphore, used to ensure sequential writes.

File write

- Allocate space with `jffs2_reserve_space()` as shown.
- Compress as much data as we can into the available space.
- Write node.
- Adjust inode fragment list accordingly.
- Call `jffs2_complete_reservation()` to release `alloc_sem`

Garbage Collection - core operation

For each `jffs2_raw_node_ref` in the block to be erased:

- If it's already obsolete, skip it.
- Follow the `next_in_ino` chain to find the inode number.
- Call `iget` for the inode in question to ensure the fraglist etc. is built.
- Obsolete the node we're looking at by writing the same data out again.

Garbage Collection - continued

Each type of node requires different stuff to be written out to obsolete it:

- Normal directory entries - just write the same out again.
- "Deletion" directory entries - see *TODO*.
- Data nodes with data - write new data node with the *current* data for the same range of the file.
- Data nodes without data. Erm, yes...

Fun stuff - truncation and holes

Potential problems with data "showing through" the holes left by file truncation and subsequent expansion.

JFFS1 didn't suffer this problem, because of linear garbage collection.

Initially attempted to solve it by writing zeroes to the space between the new and old end-of-file on truncation. Garbage collection was too hard.

Instead we writing zero data in the gaps whenever we *expand* a file, to ensure that old data remain dead.

Future improvements — Checkpointing

Scanning the entire flash during mount is *slow*. The suggested solution is to store sufficient information in *checkpoint* nodes to avoid the need to read the whole flash at startup.

- What to store?
  - Per-inode nlink, list of raw node offsets/lengths.
  - Per-eraseblock info
- When to store it?
  - Periodic opportunistic checkpoints
  - Checkpoint on clean unmount only

Future improvements — NAND support

Interesting problems for NAND:

- Easy bit — Moving the `CLEANMARKER` node.
- Harder bit — Garbage collection fixes.
- Mindbogglingly painful bit — Write batching, and the associated problems:
  - Early block erasure
  - `fsync()`, `sys_sync()`
  - Write errors on delayed writes

Future improvements — Other

- Expose transactions to userspace
- Reduce garbage collection overhead
- Improve fault tolerance
- Not eXecute-In-Place
- Extra per-inode flags
  - Compression control
  - Preloading

The Journalling Flash File System

http://sources.redhat.com/jffs2/

David Woodhouse
dwmw2@cambridge.redhat.com