JFFS : The Journalling Flash File System

David Woodhouse, Red Hat Inc.

Presented by Viacheslav Fedorov

Outline

- Flash
  - NOR, NAND
- Using Flash
  - Emulating block-devices
    - Direct mapping
    - FTL
  - Efficient FS
    - JFFS
    - JFFS2
- Future development

Flash

- Solid state, non-volatile storage
  - Reliable, High speed, Relatively low cost
- Two types: NOR and NAND
  - Block structure („erase blocks“)
    - 128 KB block on NOR flash
    - 8 KB block on NAND flash
  - Erasing by blocks only!
  - Limited lifetime
    - Endurance ~ 10-100 k erases per block
    - „Wear levelling“ needed

Flash for File Storage

- Conventional approach

  Many „standard“ file systems exist

  Why not use one of them?

  - Simplest method – direct 1:1 mapping
    - Good for read-only operations
    - No wear levelling
    - Very unsafe (on power loss)
  - Flash Translation Layer – keep track of „sectors“
    - Suitable for a writable FS
    - Wear levelling, Reliable operation; **BUT**
    - One journalling FS on top of the other

Flash for File Storage

- Efficient approach
  - Design an FS specifically for flash
    - Built-in wear levelling
    - No extra translation layers
    - Reliable operation

Flash for File Storage

- Efficient approach
  - JFFS
    - Log-structured FS
    - Only one type of node in the log
    - Direct operation without Translation Layers

JFFS: Log-structured

- No file has a fixed location
- Nodes (packets of data) are stored sequentially in flash, as in a log
  - Wear Levelling!
- Each node:
  - Is associated with a file (filename, link to parent)
  - Has a unique Version number (among file's nodes)
  - Contains latest metadata (timestamp, permissions)
  - (Optionally) Contains some of the file's data and offset at which those data reside in the file

JFFS: Operation: Writing

User action	What is written
Write 200 bytes D at offset 0 in a file	Version: 1 Offset: 0 Length: 200 Data: DDDDD.....D
Write 200 bytes 'C' at offset 200 in file	Version: 2 Offset: 200 Length: 200 Data: CCCCC.....C
Write 100 bytes 'A' at offset 120 in file	Version: 3 Offset: 20 Length: 100 Data: AAA.....A

JFFS: Operation: Reading the FS

The log nodes are „played back“
in version order, to recreate a map of where each range of data is located on the flash

Node version 1
200 bytes at 0

0-200: node v1 address

Node version 2
200 bytes at 200

0-200: node v1 address
200-400: node v2 address

Node version 3
100 bytes at 120

0-120: node v1 address
120-220: node v3 address
220-400: node v2 address

JFFS: Obsoleted nodes

- A node is obsolete if some latter node(s) has new data for the same location in file

- Nodes can also be obsoleted when the user deletes a file
  - New node is written to the log, with a **`deleted`** flag set in the metadata
  - All following nodes are marked
  - After the last file handle is closed, all nodes from this file become obsolete

- Node data physically stays on flash (dirty space)

JFFS: Garbage collection

Nothing is erased, so sooner or later we will start to run out of space. We need to reclaim the „dirty space.“

JFFS: Garbage collection

Nothing is erased, so sooner or later we will start to run out of space. We need to reclaim the „dirty space.“

JFFS: Garbage collection

Nothing is erased, so sooner or later we will start to run out of space. We need to reclaim the „dirty space."

Now we can erase the whole block at the start

JFFS: Garbage collection

Nothing is erased, so sooner or later we will start to run out of space. We need to reclaim the „dirty space.“

Now we can erase the whole block at the start

JFFS: Garbage collection

Nothing is erased, so sooner or later we will start to run out of space. We need to reclaim the „dirty space."

Now we can erase the whole block at the start

JFFS: Implementation flaws

- Garbage collection
  - Rewriting even the „clean“ nodes

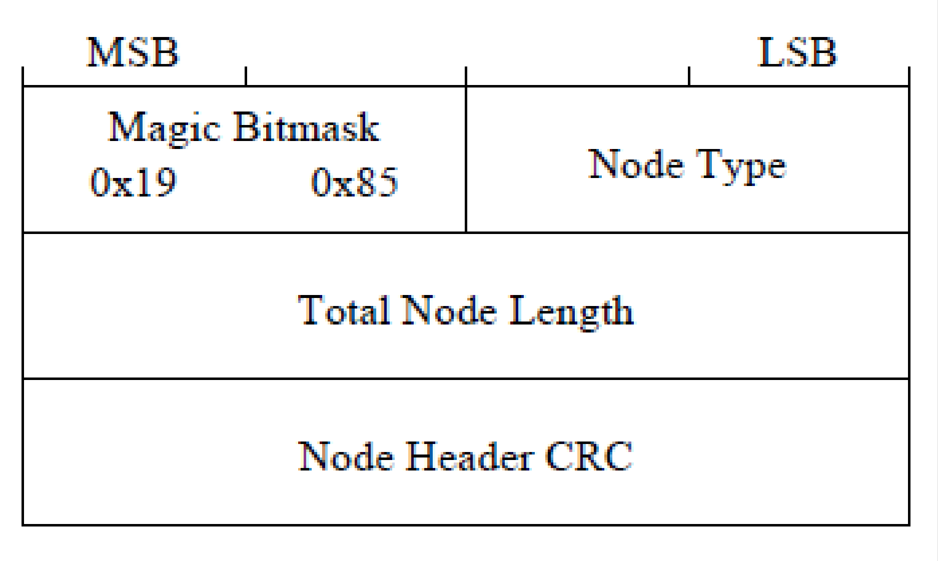
  - Perfect „wear levelling“;
    however, too frequent erasures
- Compression not supported
  - Too bad, was a very important feature at that time
- Filenames and metadata stored in each node
  - Waste of space

JFFS2

- Compression
- New node types
  - Inode, DirEnt, CleanMarker
- Non-sequential structure
  - Erase blocks treated individually
- Better memory economy

JFFS2: Node format

- Different node types for the log entries
  - JFFS used one type of node
- Common node layout
  - Backwards compatibility



JFFS2: Compatibility

- JFFS2_FEATURE_INCOMPAT
  - Refuse to mount the FS
- JFFS2_FEATURE_ROCOMPAT
  - Read-only FS
- JFFS2_FEATURE_RWCOMPAT_DELETE
  - Delete on Garbage Collection
- JFFS2_FEATURE_RWCOMPAT_COPY
  - Copy on GC

JFFS2: Node types

- Inode data node (file data)
  - Similar to JFFS node
    - Size
    - Metadata
    - Offset + data (optionally)
  - No filename or links to parent
  - Data may be compressed
    - „None“
    - „Zero“
    - Zlib compressed
  - No more than 1 page of data

JFFS2: Node types

- Directory Entry node
  - Link to Parent (directory inode number)
  - Link to File itself (inode number)
    - Inode num = 0 to unlink the file
  - Name
  - Version

Renaming done in two stages

- Write new DirEnt with the new name
- Write DirEnt with original name and inode num=0

JFFS2: Node types

- Clean Block marker node
  - Written to the cleanly erased block
  - Used to deal with partially erased blocks
    (Not fully erased due to power loss during an erase cycle)

JFFS2: Log structure

- Erase blocks are treated individually
- Several lists to store references to them:
  - clean_list – blocks containing only valid nodes
  - dirty_list – blocks containing some dirty nodes
  - free_list – erased blocks ready to be written to
    - Contain one node – Clean Block Marker

JFFS2: Operation: Writing

- Similar to JFFS
  - Write nodes sequentially until a block is filled
- Take a new block from the free_list and continue
- When free_list's size reaches a threshold
  - Garbage collection to reclaim blocks

JFFS2: Garbage Collection

- Pick a block from dirty_list, write out all its clean nodes, and erase the block
  - 99 times in 100
- 1 time in 100, pick a block from the clean_list to ensure wear is levelled

JFFS2: Mounting

- Physical scan, data structures allocation, node information caching
- Pass 1: data maps built for each file,
  nlink calculated for each file
- Pass 2: files with nlnks=0 are deleted
- Pass 3: temporarily cached information freed

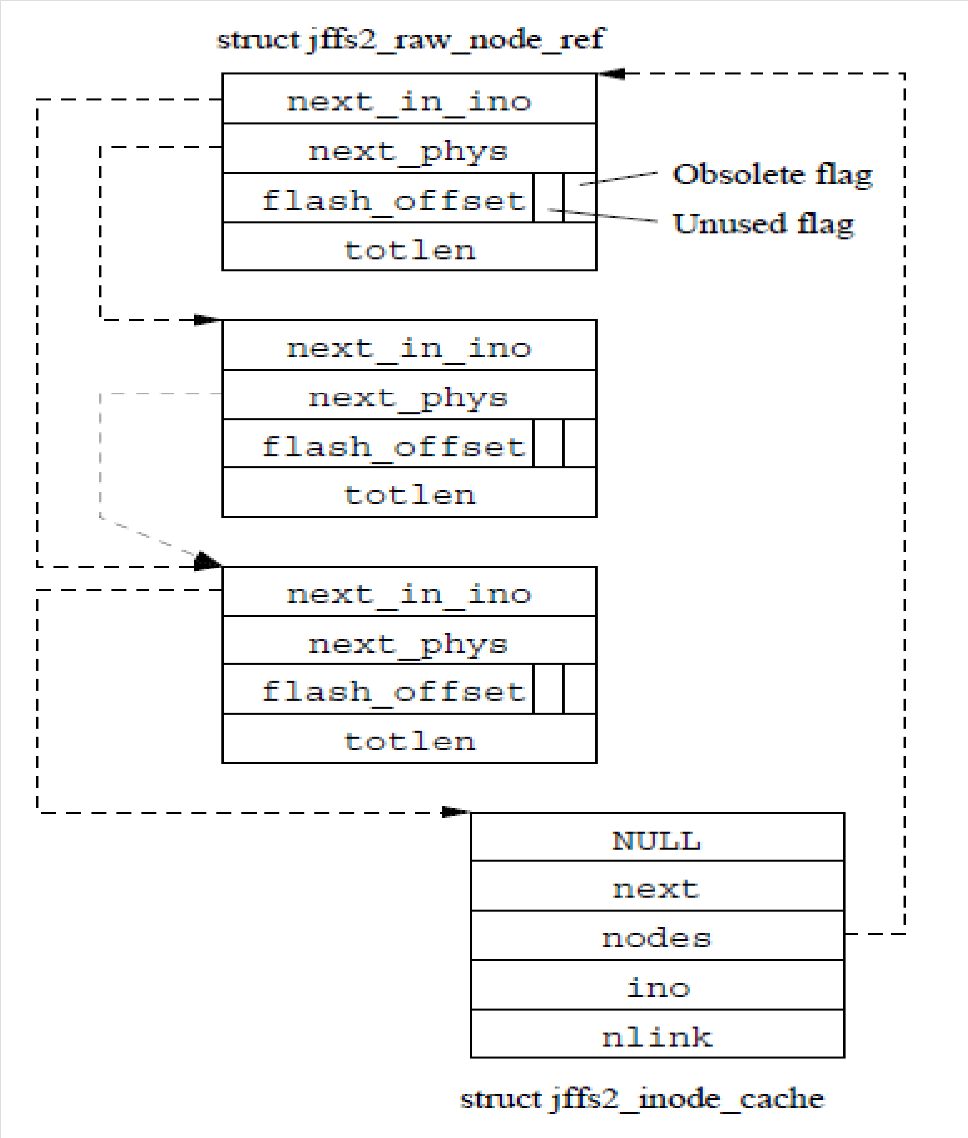
JFFS2: Operation: Data structures

What is kept in memory:

- For each inode
  - inode_cache

- For each node
  - raw_node_ref

Full map of data regions is built only on file access



Future development

- Improved fault tolerance
  - Error correction
  - Lists of bad blocks
- Lower Garbage Collection overhead
  - Currently minimum 5 free blocks
  - Possible to reduce to 1-2 blocks
- Database support
  - Exposing transactions to userspace

Thank you!