YAFFS
A NAND flash filesystem

Wookey
wookey@wookware.org

Aleph One Ltd
Balloonboard.org
Toby Churchill Ltd

Embedded Linux Conference - Europe
Linz
1. Project Genesis
2. Flash hardware
3. YAFFS fundamentals
4. Filesystem Details
5. Embedded Use
Project Genesis

- TCL needed a reliable FS for NAND
- Charles Manning is the man
- Considered Smartmedia compatible scheme (FAT+FTL)
- Considered JFFS2
  - Better than FTL
  - High RAM use
  - Slow boot times
History

- Decided to create ‘YAFFS’ - Dec 2001
- Working on NAND emulator - March 2002
- Working on real NAND (Linux) - May 2002
- WinCE version - Aug 2002
- ucLinux use - Sept 2002
- Linux rootfs - Nov 2002
- pSOS version - Feb 2003
- Shipping commercially - Early 2003
- Linux 2.6 supported - Aug 2004
- YAFFS2 - Dec 2004
- Checkpointing - May 2006
## Flash primer - NOR vs NAND

<table>
<thead>
<tr>
<th></th>
<th>NOR flash</th>
<th>NAND flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access mode:</td>
<td>Linear random access</td>
<td>Page access</td>
</tr>
<tr>
<td>Replaces:</td>
<td>ROM</td>
<td>Mass Storage</td>
</tr>
<tr>
<td>Cost:</td>
<td>Expensive</td>
<td>Cheap</td>
</tr>
<tr>
<td>Device Density:</td>
<td>Low (64MB)</td>
<td>High (1GB)</td>
</tr>
<tr>
<td>Erase block size:</td>
<td>8k to 128K typical</td>
<td>32x512b / 64x2K pages</td>
</tr>
<tr>
<td>Endurance:</td>
<td>100k to 1M erasures</td>
<td>10k to 100k erasures</td>
</tr>
<tr>
<td>Erase time:</td>
<td>1second</td>
<td>2ms</td>
</tr>
<tr>
<td>Programming:</td>
<td>Byte by Byte, no limit on writes</td>
<td>Page programming, must be erased before re-writing</td>
</tr>
<tr>
<td>Data sense:</td>
<td>Program byte to change 1s to 0s.</td>
<td>Program page to change 1s to 0s.</td>
</tr>
<tr>
<td></td>
<td>Erase block to change 0s to 1s.</td>
<td>Erase to change 0s to 1s</td>
</tr>
<tr>
<td>Write Ordering:</td>
<td>Random access programming</td>
<td>Pages must be written sequentially within block</td>
</tr>
<tr>
<td>Bad blocks:</td>
<td>None when delivered, but will wear out so filesystems</td>
<td>Bad blocks expected when delivered. More will appear with use.</td>
</tr>
<tr>
<td></td>
<td>should be fault tolerant</td>
<td>Thus fault tolerance is a necessity.</td>
</tr>
<tr>
<td>OOB data:</td>
<td>No</td>
<td>Yes (16 bytes)</td>
</tr>
</tbody>
</table>
NAND reliability

NAND is unreliable - bad blocks, data errors
Affected by temp, storage time, manufacturing, voltage

- Program/erase failure
  - Detected in hardware. YAFFS copies data and retires block
- Charge Leakage - bitrot over time
  - ECC - Error Correction Codes
- Write disturb: (extra bits set to 0 in page/block)
  - YAFFS2 minimises write disturb (sequential block writes, no re-writing)
- Read disturb, other pages in block energised.
  - minor effect - needs 10*endurance reads to give errors
  - ECC (not sufficient)
  - count page reads, rewriting block at threshold
  - Read other pages periodically (e.g. every 256 reads)

MLC makes all this worse - multiple program and read voltages
# Mechanisms to deal with NAND problems

<table>
<thead>
<tr>
<th>Feature</th>
<th>Chip Fault</th>
<th>Degredation</th>
<th>Prog/Erase failure</th>
<th>Leakage</th>
<th>Write Disturb</th>
<th>Read Disturb</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND self-check</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block Retirement</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wear Levelling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Verification</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Read counting/re-write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Future</td>
</tr>
<tr>
<td>Infrequent Read Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Future</td>
<td>Future</td>
</tr>
<tr>
<td>ECC</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Design approach

- OS and compiler neutral
- Portable - OS interface, guts, hardware interface, app interface
- Log-structured - Tags break down dependence on physical location
- Configurable - chunk size, file limit, OOB layout, features
- Single threaded (don’t need separate GC thread like NOR)
- Follow hardware characteristics (OOB, no re-writes)
- Developed on NAND emulator in userspace
- Abstract types allow Unicode or ASCII operation
**Terminology**

- **Flash-defined**
  - Page - 2k flash page (512 byte YAFFS1)
  - Block - Erasable set of pages (typically 64 on 2K NAND)

- **YAFFS-defined**
  - Chunk - YAFFS tracking unit. usually == page.
    Can be bigger, e.g. 2x2K NAND in parallel as 4K chunks)
Each file has an id - equivalent to inode. id 0 indicates ’invalid’
File data stored in chunks, same size as flash pages (2K/512 bytes)
Chunks numbered 1,2,3,4 etc - 0 is header.
Header gives type (device/file/dir) and hold mode/uid/length etc
Each flash page is marked with file id and chunk number
These tags are stored in the OOB - file id, chunk number, write serial/sequence number, tag ECC and bytes-in-page-used
When changing a file the relevant chunks are replaced by writing new pages with new data but same tags - the old page is marked ’discarded’
Pages have a serial number incremented on write (2-bit serial in YAFFS1, sequence in YAFFS2). Allows crash-recovery when two pages have same tags.
Discarded blocks are garbage-collected.
Deleted items placed in ‘deleted’ dir (YAFFS2)
Log-structured Filesystem (1)

Imagine flash chip with 4 pages per block. First we’ll create a file.

<table>
<thead>
<tr>
<th>Block</th>
<th>Chunk</th>
<th>ObjId</th>
<th>ChunkId</th>
<th>DelFlag</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Live</td>
<td>Object header for this file (length 0)</td>
</tr>
</tbody>
</table>

Next we write a few chunks worth of data to the file.

<table>
<thead>
<tr>
<th>Block</th>
<th>Chunk</th>
<th>ObjId</th>
<th>ChunkId</th>
<th>DelFlag</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Live</td>
<td>Object header for this file (length 0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>Live</td>
<td>First chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>500</td>
<td>2</td>
<td>Live</td>
<td>Second chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>500</td>
<td>3</td>
<td>Live</td>
<td>Third chunk of data</td>
</tr>
</tbody>
</table>
Log-structured Filesystem (2)

Next we close the file. This writes a new object header for the file. Notice how the previous object header is deleted.

<table>
<thead>
<tr>
<th>Block</th>
<th>Chunk</th>
<th>ObjId</th>
<th>ChunkId</th>
<th>DelFlag</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header (length 0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>Live</td>
<td>First chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>500</td>
<td>2</td>
<td>Live</td>
<td>Second chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>500</td>
<td>3</td>
<td>Live</td>
<td>Third chunk of data</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Live</td>
<td>New object header (length n)</td>
</tr>
</tbody>
</table>
Let’s now open the file for read/write, overwrite part of the first chunk in the file and close the file. The replaced data and object header chunks become deleted.

<table>
<thead>
<tr>
<th>Block</th>
<th>Chunk</th>
<th>ObjId</th>
<th>ChunkId</th>
<th>DelFlag</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header (length 0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>Del</td>
<td>Obsoleted first chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>500</td>
<td>2</td>
<td>Live</td>
<td>Second chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>500</td>
<td>3</td>
<td>Live</td>
<td>Third chunk of data</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>Live</td>
<td>New first chunk of file</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>500</td>
<td>0</td>
<td>Live</td>
<td>New object header</td>
</tr>
</tbody>
</table>
Log-structured Filesystem (5)

Now let’s resize the file to zero by opening the file with `O_TRUNC` and closing the file. This writes a new object header with length 0 and marks the data chunks deleted.

<table>
<thead>
<tr>
<th>Block</th>
<th>Chunk</th>
<th>ObjId</th>
<th>ChunkId</th>
<th>DelFlag</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header (length 0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>Del</td>
<td>Obsoleted first chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>500</td>
<td>2</td>
<td>Del</td>
<td>Second chunk of data</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>500</td>
<td>3</td>
<td>Del</td>
<td>Third chunk of data</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>Del</td>
<td>Deleted first chunk of file</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>500</td>
<td>0</td>
<td>Live</td>
<td>New object header (length 0)</td>
</tr>
</tbody>
</table>

Note all the pages in block 0 are now marked as deleted. So we can now erase block 0 and re-use the space.
We will now rename the file.
To do this we write a new object header for the file

<table>
<thead>
<tr>
<th>Block</th>
<th>Chunk</th>
<th>ObjId</th>
<th>ChunkId</th>
<th>Del</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Erased</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Erased</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>Erased</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Erased</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>500</td>
<td>1</td>
<td>Del</td>
<td>Deleted first chunk of file</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>500</td>
<td>0</td>
<td>Del</td>
<td>Obsoleted object header</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>Live</td>
<td>New object header showing new name</td>
</tr>
</tbody>
</table>
**YAFFS2**

- Specced Dec 2002, working Dec 2004
- Designed for new hardware:
  - >=1k page size
  - no re-writing
  - simultaneous page programming
  - 16-bit bus on some parts
- Main difference is ‘discarded’ status tracking
- ECC done by driver (MTD in Linux case)
- Extended Tags (Extra metadata to improve performance)
- RAM footprint 25-50% less
- faster (write 1-3x, read 1-2x, delete 4-34x, GC 2-7x)
**YAFFS2 - Discarded status mechanism**

- Zero re-writes means can’t use ‘discarded’ flag
- Genuinely log-structured
- Instead track block allocation order (with sequence number)
- Delete by making chunks available for GC and move file to special ‘unlinked’ directory until all chunks in it are ‘stale’
- GC gets more complex to keep ‘sense of history’
- Scanning runs backwards - reads sequence numbers chronologically
Filesystem Limits

- **YAFFS1**
  - $2^{18}$ files (>260,000)
  - $2^{20}$ max file size (512MB)
  - 1GB max filesystem size

- **YAFFS2 - All tweakable**
  - 2GB max file size
  - filesystem max size set by RAM footprint (4TB flash needs 1GB RAM)
  - 4GB max filesystem size set by MTD (32-bit)
  - 8GB shipping, 16GB tested
OOB data

- **YAFFS1:**
  - Derived from Smartmedia, (e.g byte 5 is bad block marker)
  - 16 bytes: 7 tags, 2 status, 6 ECC
  - YAFFS/Smartmedia or JFFS2 format ECC

- **YAFFS2:**
  - 64 bytes available in 2k page
  - MTD-determined layout (on linux)
  - MTD or hardware does ECC - 38 bytes free on 2.6.21
  - Tags normally 28 bytes (16 data, 12ecc)
  - Sometimes doesn’t fit (eg oneNAND - 20 free)
RAM Data Structures

- Not fundamental - needed for speed
- Yaffs_Object - per file/directory/link/device
- T-node tree covering all allocated chunks
  - As the file grows in size, the levels increase.
  - The T-nodes are 32 bytes. (16bytes on 2k arrays <=128MB)
  - Level 0 is 16 2-byte entries giving an index to chunkId.
  - Higher level T-nodes are 8 4-byte pointers to other tnodes
  - Allocated in blocks of 100 (reduced overhead & fragmentation)
RAM usage

- Level0-Tnodes:
  - Chunksize
  - RAM use/MB NAND
  - 256MB NAND
    - 512b: 4K, 1MB
    - 2k: 1K, 256K
    - 4k: 0.5K, 128K

Can change chunk size, and/or parallel chips.

- Higher-level Tnodes: 0-Tnodes/8, etc

- Objects: 24bytes (+17 with short name caching) per file

- For 256MB 2K chunk NAND with 3000 files/dirs/devices
  - 128k chunks needs 18bits per Tnode cell, so
    - Level 0-Tnodes: 288K
    - Level 1-Tnodes: 36K
    - Level 2-Tnodes: 5K
    - Objects: 120K
    - 449K total
Partitioning

- Internal - give start and end block
- MTD partitioning (partition appears as device)
Checkpointing

- RAM structures saved on flash at unmount (10 blocks)
- Structures re-read, avoiding boot scan
- sub-second boots on multi-GB systems
- Invalidated by any write
- Lazy Loading also reduces mount time.
Garbage Collection and Threads

- Single threaded - Gross locking, matches NAND
- 3 blocks reserved for GC (384K)
- If no deleted blocks, GC dirtiest
- Soft Background deletion:
  - Delete/Resize large files can take up to 0.5s
  - Incorporated with GC
  - Spread over several writes
- GC is determinsitic - does one block for each write (default)
- Worst case - nearly full disk, blocks have n-1 chunks valid
- Can give GC own thread, so operates in ‘dead time’
Caching

- Linux VFS has cache, WinCE and RTOS don’t
- YAFFS internal cache
  - 15x speed-up for short writes on WinCE
  - Allows non-aligned writes
- Choose generic read/write (VFS) or direct read/write (MTD)
  - Generic is cached *(usually* reads much faster 10x, writes 5% slower)
  - Direct is more robust on power fail
ECC

- Needs Error Correction Codes for reliable use
- ECC on Tags and data
- 22bits per 256 bytes, 1-bit correction, 2-bit detection
- CPU/RAM intensive
- Lots of options:
  - Hardware or software
  - YAFFS or MTD
  - New MTD, old MTD or YAFFS/Smartmedia positioning
- Make sure bootloader, OS and FS generation all match!
- Can be disabled - not recommended!
OS portability

- **Native**
  - Linux
  - WinCE
  - NetBSD

- **Yaffs Direct Interface**
  - pSOS
  - ThreadX
  - DSP_BIOS
  - Others

- **Bootloaders - simple read-only YAFFS**
YAFFS in use

- Formatting is simply blanking
- `mount -t yaffs /dev/mtd0 /
- Creating a filesystem image needs to generate OOB data
  - YAFFS1: mkyaffsimage tool - generates images
  - YAFFS2: mkyaffs2image - often customised
  - Use nandutils if possible
YAFFS Direct Interface

- YDI replaces Linux VFS/WinCE FSD layer
- open, close, stat, read, write, rename, mount etc
- Caching of unaligned accesses
- Port needs 5 OS functions, 6 NAND functions:
  - Lock and Unlock (mutex)
  - current time (for time stamping)
  - Set Error (to return errors)
  - Init to initialise RTOS context
  - NAND access (read, write, markbad, queryblock, initnand, erase).
Embedded system use - YAFFS Direct Interface (2)

- No CSD - all filenames in full
- Configurable case sensitivity
- No UID/GIDS
- Flat 32-bit/64-bit time
- Thread safe - one mutex
- Multiple devices - eg /ram /boot /flash
Licensing

- GPL - Good Thing (TM)
- Bootloader/headers LGPL to allow incorporation
- YAFFS in proprietary OSes (pSOS, ThreadX, VxWorks)
  - Wider use
  - Aleph One Licence - MySQL/sleepycat-style: ‘If you don’t want to play then you can pay’
Future Developments

- In-band tags (Done, being tested now)
- Counter Read-/write- disturb
- Continuous block summaries for checkpointing
- BCH codes instead of ECC
- RAM reduction - mixed chunk sizes
- Small files in headers - less wasted space
- Mainlining
The end

http://yaffs.net

Questions?