YAFFS
A NAND flash filesystem

Wookey
wookey@wookware.org

Aleph One Ltd

Balloonboard.org

Toby Churchill Ltd

Embedded Linux Conference - Europe
Linz

Project Genesis

- TCL needed a reliable FS for NAND
- Considered Smartmedia compatibility
- 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

Access mode	Linear random access	Page access
Replaces	ROM	Mass Storage
Cost	Expensive	Cheap
Device Density	Low (64MB)	High (1GB)
Erase block size	8k to 128K typical	32x2K pages
Endurance	100k to 1M erasures	10k to 100k erasures
Erase time	1second	2ms
Programming	Byte by Byte, no limit on writes	Page programming, must be erased before re-writing
Data sense	Program byte to change 1s to 0s. Erase block to change 0s to 1s	Program page to change 1s to 0s. Erase to change 0s to 1s
Write Ordering	Random access programming	Pages must be written sequentially within block
Bad blocks	None when delivered, but will wear out so filesystems should be fault tolerant	Bad blocks expected when delivered. More will appear with use. Thus fault tolerance is a necessity.

NAND reliabilty

NAND is unreliable - bad blocks, data errors

Affected by temp, storage time, manufacturing, voltage

- Program/erase failure
- YAFFS internal cache
    - Detected in hardware. YAFFS copies data and retires block
- Charge Leakage - bitrot over time
    - ECC
- 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: (1 million SLC, 100,000 MLC)
    - 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 programand read voltages

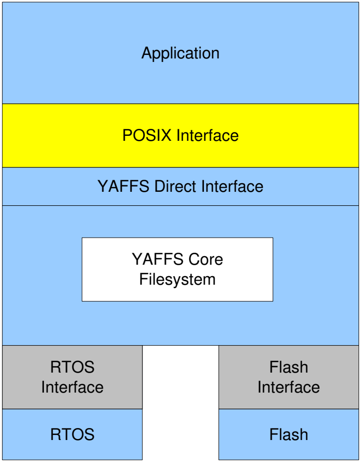
Mechanisms to deal with NAND problems

	Chip Fault	Degre-dation	Prog/Erase failure	Leakage	Write Disturb	Read Disturb
NAND self-check	Yes		Yes			
Block Retirement	Yes	Yes	Yes			
Wear Levelling		Yes				
Write Verification					Yes	
Read counting /re-write						Future
Infrequent Read Checking				Future	Future	Future
ECC		Yes		Yes	Yes	Yes

Design approach

- OS 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

YAFFS Architecture



Terminology

- Flash-defined
  - Page - 2k flash page (512 byte YAFFS1)
  - Block - Erasable set of pages (typically 32)
- YAFFS-defined
  - Chunk - YAFFS tracking unit.
    usually==page. Can be bigger

Process

- Each file has an id - equivalent to inode. id 0 indicates 'deleted'
- File data stored in chunks, same size as flash pages (2K/512 bytes)
- Chunks numbered 1,2,3,4 etc - 0 is header.
- Each flash page is marked with file id and chunk number
- These tags are stored in the OOB - 64bits: including file id, chunk number, write serial number, tag ECC and bytes-in-page-used
- On overwriting the relevant chunks are replaced by writing new pages with new data but same tags - the old page is marked 'discarded'
- File headers (mode, uid, length etc) get a page of their own (chunk 0)
- Pages also have a 2-bit serial number - incremented on write
- Allows crash-recovery when two pages have same tags (because old page has not yet been marked 'discarded').
- Discarded blocks are garbage-collected.

Log-structured Filesystem (1)

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

Flash Blocks

Block	Chunk	ObjId	ChunkId	DelFlag	Comment
0	0	500	0	Live	Object header for this file (length 0)

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

Flash Blocks

Block	Chunk	ObjId	ChunkId	DelFlag	Comment
0	0	500	0	Live	Object header for this file (length 0)
0	1	500	1	Live	First chunk of data
0	2	500	2	Live	Second chunk of data
0	3	500	3	Live	Third chunk of data

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.

Flash Blocks

Block	Chunk	ObjId	ChunkId	DelFlag	Comment
0	0	500	0	Del	Obsoleted object header (length 0)
0	1	500	1	Live	First chunk of data
0	2	500	2	Live	Second chunk of data
0	3	500	3	Live	Third chunk of data
1	0	500	0	Live	New object header (length n)

Log-structured Filesystem (3)

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.

Flash Blocks

Block	Chunk	ObjId	ChunkId	DelFlag	Comment
0	0	500	0	Del	Obsoleted object header (length 0)
0	1	500	1	Del	Obsoleted first chunk of data
0	2	500	2	Live	Second chunk of data
0	3	500	3	Live	Third chunk of data
1	0	500	0	Del	Obsoleted object header
1	1	500	1	Live	New first chunk of file
1	2	500	0	Live	New object header

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.

Flash Blocks

Block	Chunk	ObjId	ChunkId	DelFlag	Comment
0	0	500	0	Del	Obsoleted object header (length 0)
0	1	500	1	Del	Obsoleted first chunk of data
0	2	500	2	Del	Second chunk of data
0	3	500	3	Del	Third chunk of data
1	0	500	0	Del	Obsoleted object header
1	1	500	1	Del	Deleted first chunk of file
1	2	500	0	Del	Obsoleted object header
1	3	500	0	Live	New object header (length 0)

Note all the pages in block 0 are now marked as deleted. So we can now erase block 0 and re-use the space.

Log-structured Filesystem (6)

We will now rename the file.
To do this we write a new object header for the file

Flash Blocks

Block	Chunk	ObjId	ChunkId	Del	Comment
0	0				Erased
0	1				Erased
0	2				Erased
0	3				Erased
1	0	500	0	Del	Obsoleted object header
1	1	500	1	Del	Deleted first chunk of file
1	2	500	0	Del	Obsoleted object header
1	3	500	0	Del	Obsoleted object header
2	0	500	0	Live	New object header showing new name

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
  - 4GB max filesystem size (MTD 32-bit limit)
  - (16GB tested - limited by RAM footprint (4TB flash needs 1GB RAM))

Devices, hardlinks, softlinks, pipes supported

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

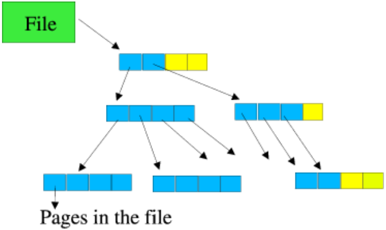
- zero re-writes means can’t use ‘discarded’ flag
- Genuinely log-structured
- Instead track block allocation order (with sequence number)
- Delete by making chunks available for GC and move file to special ‘unlinked’ directory until all chunks in it are ‘stale’
- GC gets more complex to keep ‘sense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

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
  - MTD-determined layout (on linux)
  - MTD 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/dir/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
  - Level 0-Tnodes: 256K
  - Level 1-Tnodes: 32K
  - Level 2-Tnodes: 4K
  - Objects: 120K
  - 412K 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
- 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
  - `while(program_is_being_stupid)`
    `write(f,buf,1);`
- 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
- 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

- Linux
- Wince (3 and 6)
- NetBSD
- pSOS
- ThreadX
- DSP_BIOS
- Bootloaders

YAFFS in use

- Formatting is simpy 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, 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
- Case sensitive
- No UID/GIDS
- Flat 32-bit time
- Thread safe - one mutex
- Multiple devices - eg /ram /boot /flash

Licensing

- GPL - Good Thing (TM), patents
- 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'