Linux Kernel Programming

Flash Memory and Embedded Flash Management in Linux

Pierre Olivier

Systems Software Research Group @ Virginia Tech

April 6, 2017

Outline

Outline

Flash memory: general presentation

Flash usage in computer systems

- Some benefits: storage density (small size), shock resistance, low power consumption
  - **Flash is the main secondary storage media in embedded systems**

Flash memory: general presentation

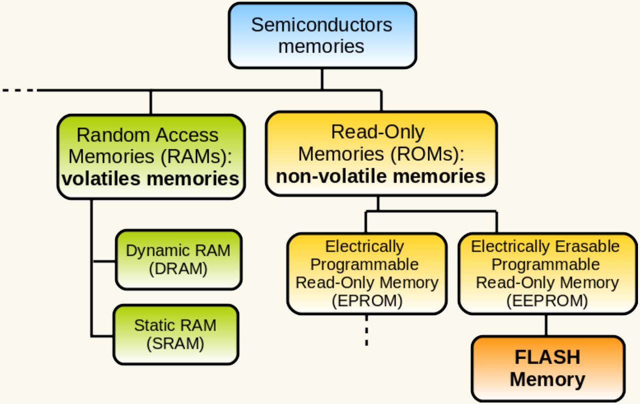
Flash usage in computer systems (2)

- **Solid-state drives**: “disks” based on flash memory
  - High performance compared to hard disk drives
- SSD are now widely used in:
  - Datacenters: HPC, big data processing, etc.
  - Laptops, regular desktop PCs

Flash memory: general presentation

Flash & semiconductor memories

- Flash is a **non-volatile** memory
- Flash is a sub-type of **EEPROM** memory



Flash memory: general presentation

Flash memory types: NOR flash

- Flash memory types are named from the logic gate used for their design
- **NOR flash**
  - High cost/bit, low density
    - **low capacity**
  - Random access (i.e. byte granularity)
  - Fast reads, slow writes
  - XIP: *eXecute In Place*
    - Code in NOR flash can be directly executed without going through the RAM
  - Used in motherboards BIOS, device firmwares, etc.
  - NOR is not the topic of these slides

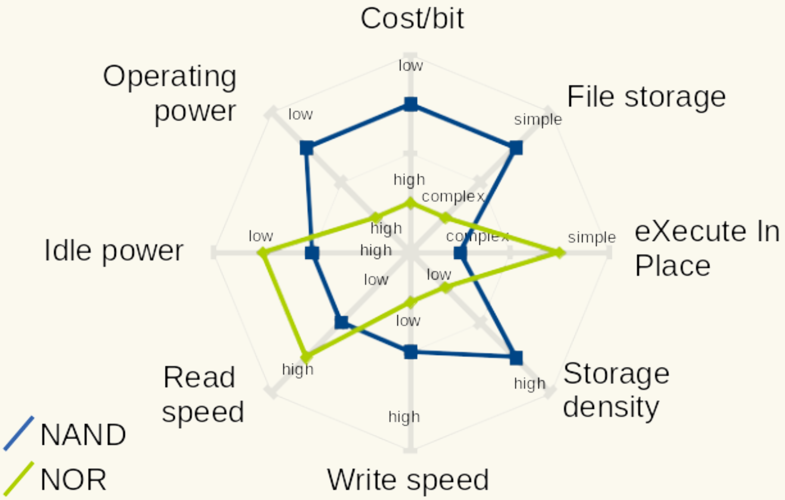
Flash memory: general presentation

Flash memory types: NAND flash

- **NAND flash**
  - Low cost per bit, high storage density
    - **Used for secondary storage**
  - Block level access
    - Chunks of bytes, no random access
  - Good and balanced read/write performance
  - **In this lecture we focus on NAND flash**
- Other types less popular, mostly used in embedded systems [1]

Flash memory: general presentation

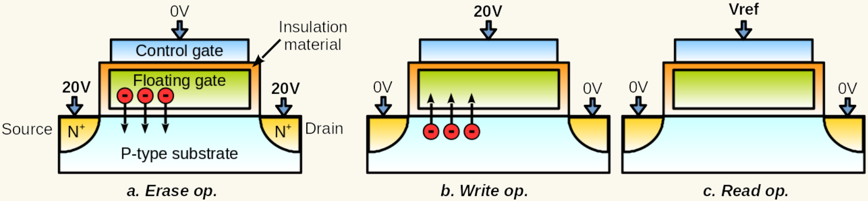
Flash memory types: NAND vs NOR



Flash memory: general presentation

Technology and micro-architecture: floating gate transistor

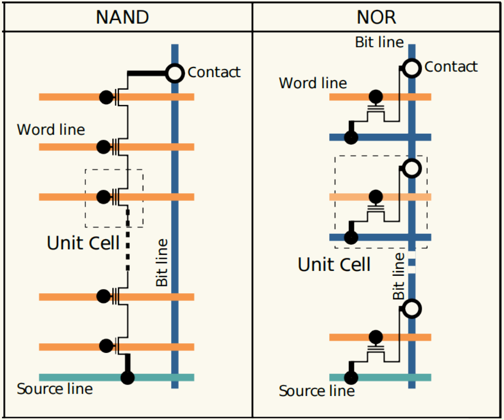
- **Floating gate transistor** gives to flash its non-volatile property



Flash memory: general presentation

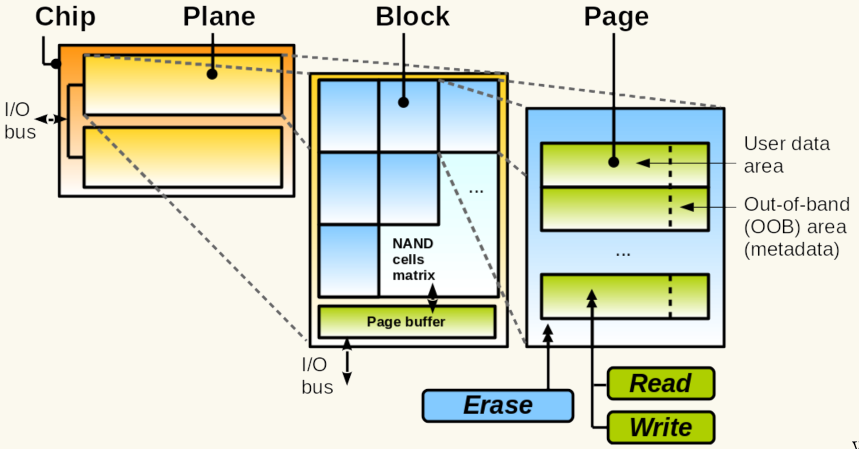
Technology and micro-architecture: NAND micro-architecture

- Transistors are assembled in serial in NAND flash: it is accessed by blocks
- One NAND flash cell can store:
  - 1 bit: *Single-Level Cell* (SLC)
  - 2 bits: *Multi-Level Cell* (MLC)
  - 3 bits: *Triple-Level Cell* (TLC)



Flash memory: general presentation

Technology and micro-architecture: NAND micro-architecture (2)



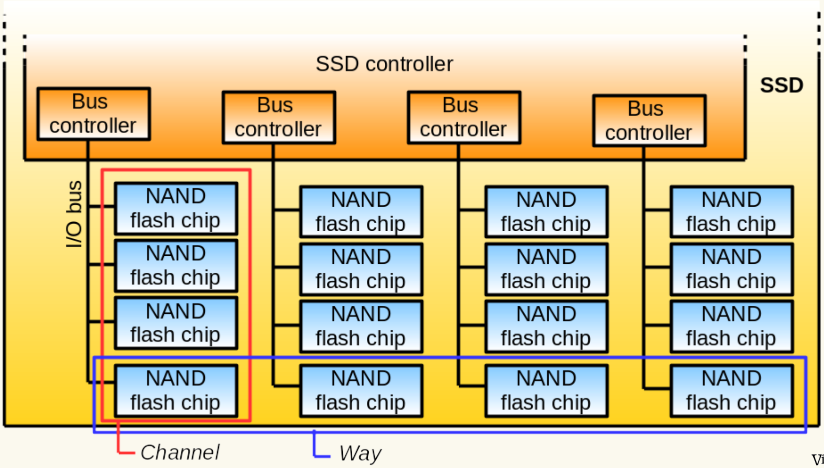
Flash memory: general presentation

Technology and micro-architecture: NAND micro-architectural characteristics

Architectural characteristic	Values among chip models
Flash page size	from 512 (+16 OOB) to 8192 (+128 OOB), power of two
Number of pages per block	32 or 64 or 128
Number of blocks per plane	1024 or 2048 or 4096
Number of planes per chip	1 or 2 or 4
I/O bus width	8 or 16 bits

Flash memory: general presentation

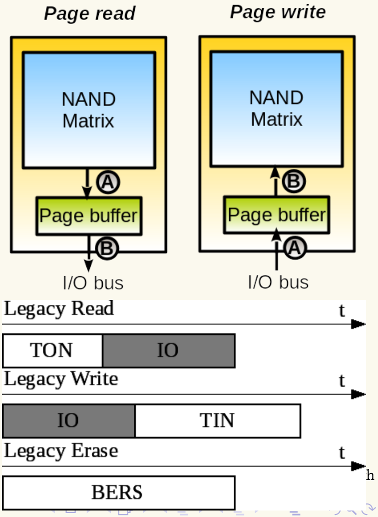
Technology and micro-architecture: SSD internals



Flash memory: general presentation

Operations

- A NAND flash chip supports 3 main operations called **legacy** operations
  - **Page read**: ∼30 μs + IO for SLC, ∼30 to 100 μs + IO for MLC
  - **Page write**: IO + ∼200 μs (SLC), IO + ∼300 to 2000 μs (MLC)
  - **Block erase**: 500 to 2000 μs (SLC), ∼3000 μs (MLC)
- Legacy vs *advanced* operations:
  - Cache mode, copy-back, multi-plane/chip/channels



Outline

Flash constraints and limitation

Presentation

- **Specific constraints in flash memory operation**

1. **Erase-before-write rule**
2. **Flash wear**
3. **Reliability**
4. Plus special constraints on advanced operations

Flash constraints and limitation

Presentation

- **Specific constraints in flash memory operation**

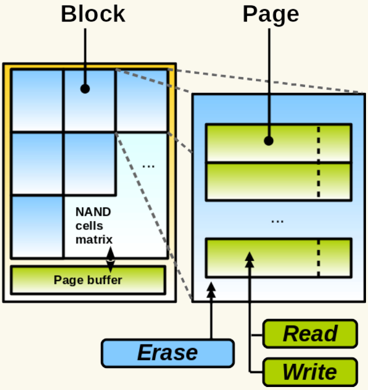
1. **Erase-before-write rule**
2. **Flash wear**
3. **Reliability**
4. Plus special constraints on advanced operations

Leads to the presence of specific flash constraints management systems in flash-based storage subsystems

Flash constraints and limitation

The erase-before-write rule

- Erase-before-write rule:
  - **It is not possible to perform a write operation in a page that already contains data**
  - The page first needs to be erased → issues arise:
    - Target of the erase operation is an entire block!
    - Erase operation takes time!
    - Write/erase operations are **asymmetrical**



Flash: no in-place data updates (overwrites)

Flash constraints and limitation

Flash wear

- **Flash wears with usage: a block can only sustain a limited number of erase operations**
  - Once a given threshold is reached the block cannot contain data anymore: it is called a **bad block**
    - Bad block appear during flash lifetime
    - Some bad blocks are also present when the chip comes out of the factory
- Threshold:
  - SLC: 100 000 erase operations
  - MLC: 10 000 erase operations
  - TLC: 5 000 erase operations
- SLC vs MLC/TLC: trade-off performance/endurance/capacity

Flash constraints and limitation

Reliability

- Due to the technology and the high voltages applied to the memory cells during operation:
  - *Bitflips* occur on data read/written on flash, as well as adjacent on-flash data
  - Error rate is higher in MLC/TLC than in SLC
  - To reduce the frequency of errors, **pages should (SLC) or must (MLC/TLC) be written sequentially within the containing block**
- Retention is generally 5 to 10 years
  - Can go down to 1 year for chips with high number of write/erase cycles

Outline

Constraints management

Presentation

Constraints summary

- **Erase-before-write**
- **Wear**
- Reliability

- Constraints are dealt with in flash-based storage subsystems with so-called **flash management systems**
- These system abstract the constraints from the upper layers (user/programmer)
  - Allow using flash in computer systems

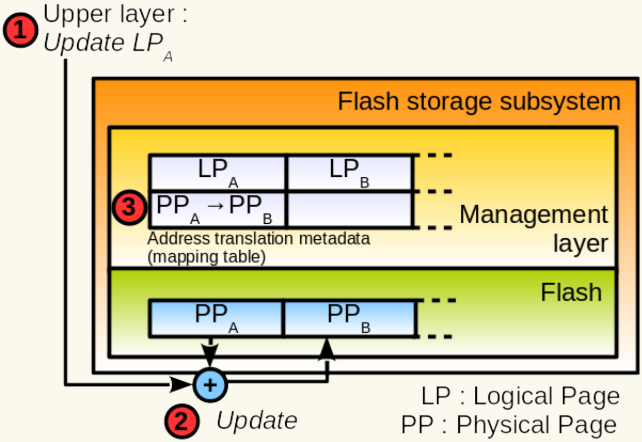
Constraints management

Managing the erase-before-write rule

- **Erase-before-write**: no in-place updates
- The programmer still expects the capability to perform overwrites on stored data independently of the storage media
- **Solution: out-of-place data updates through *logical to physical mapping***
  - Logical addresses are viewed by the programmer
  - Physical addresses are actual flash pages/blocks

Constraints management

Managing the erase-before-write rule

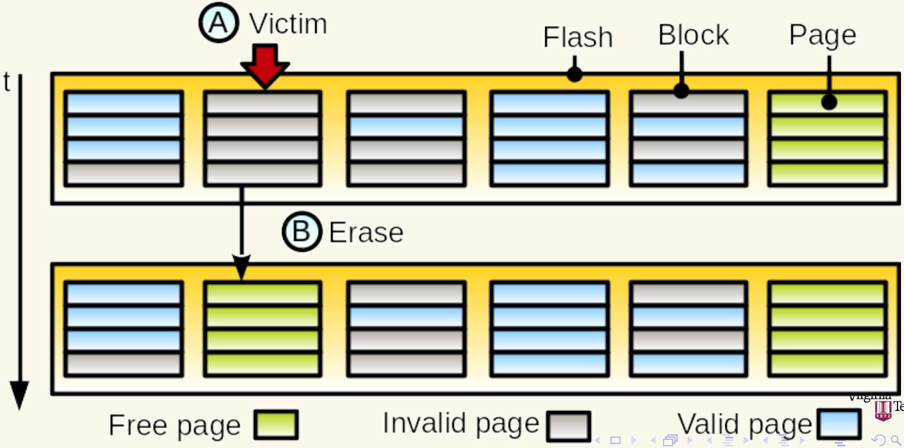


- States for a page: free, used (valid), **used (invalid)**

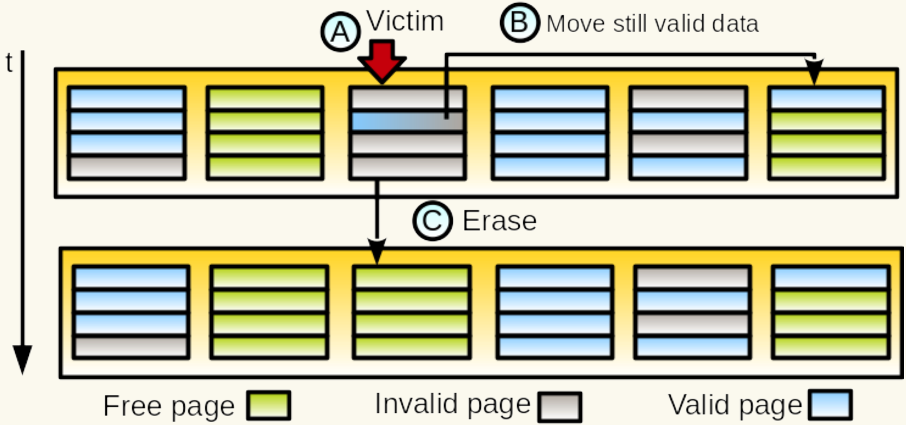
Constraints management

Managing the erase-before-write rule

- Flash management systems implement a **garbage collector** to recycle invalid pages into free space



Constraints management
Managing the erase-before-write rule (2)



Constraints management

Wear leveling (3)

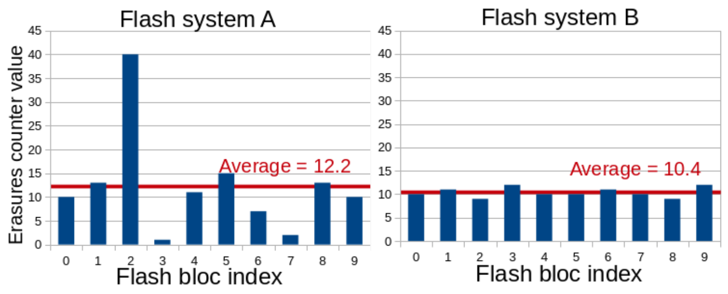
- **Victim block selection policy:**
  - *Greedy*: choose the bloc containing the largest amount of invalid pages
    - Good performance as this minimizes the amount of still valid data recopy
    - Does not take the flash wear into account
  - *Cost/benefit*: computes a score for each block:

$$\frac{\text{Estimation of the current wear for the concerned block}}{\text{Number of invalid pages in the concerned block}}$$

    - Wear can be estimated through erase counters, usage frequency, time since the last use

Constraints management

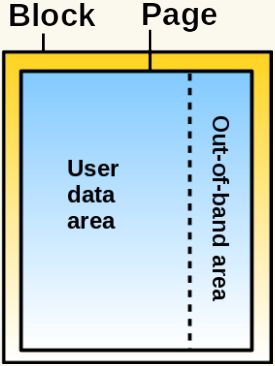
Wear leveling: example



Metric	**System A**	**System B**
Total number of erase operations	122	104
Average value of per-block erase counters	12.2	10.4
Standard deviation of the erase counter distribution	10.8	1.07
Erase counter difference between the most and the less erased block	39	3

Constraints management
Bad blocks management

- Bitflips are handled with **error correcting code** (ECC)
  - Detect and correct bitflips
- Page data hash stored in OOB area
- ECC types used:
  - SLC: **Hamming** - 2 detection/1 correction per page, simple implementation
  - MLC/TLC: **Reed-Solomon**, **Bose-Chaudhuri-Hocquenghem** - detect and correct several errors per page, implementation more complex



- ECC implemented in software (OS/driver) or in hardware (dedicated circuit in the flash device controller)

Constraints management

Conclusion

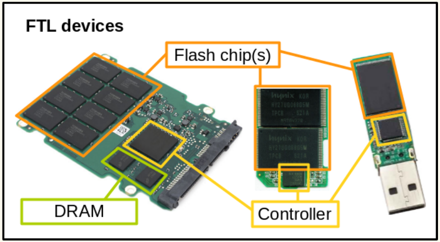
- **Erase-before-write rule**
  - Solved by performing **out-of-place updates**
    - Implies logical to physical address translation
    - Implies the introduction of the **invalid state** for a page, and the implementation of a **garbage collector**
- **Flash wear**
  - Solved with **wear-leveling** policies
- **Reliability** (bitflips):
  - Solved through the use of error-correcting codes

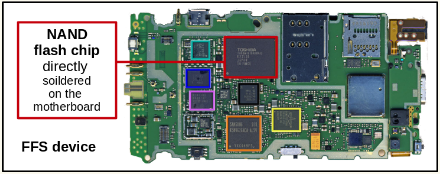
Outline

Embedded flash management with Linux

Flash Translation Layer vs Flash File Systems

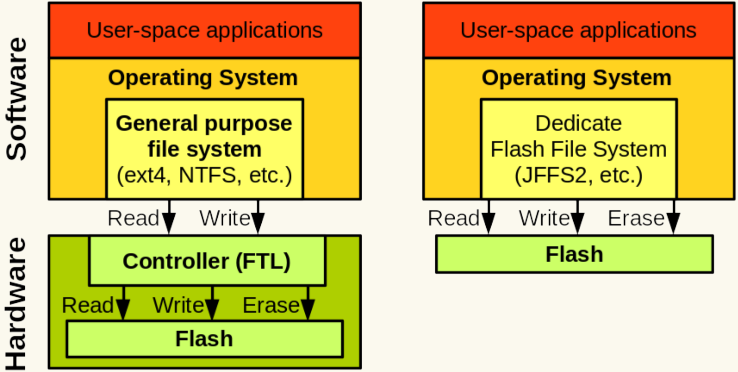
- Two main classes of flash management systems:
  1. Flash Translation Layer
     - SSDs, SD/MMC cards, USB flash drives
     - Hardware-based solution implemented in the device controller
  2. **Flash File Systems**
     - Mostly used in embedded systems: smartphones, tablets, etc
     - Software solution





Embedded flash management with Linux

Flash Translation Layer vs Flash File Systems (2)



Embedded flash management with Linux

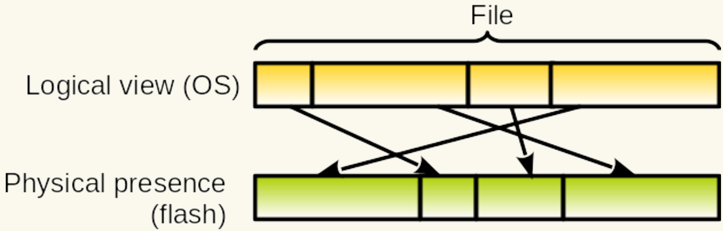
Flash File Systems: Presentation

- Pure software-based solution
  - FFS implemented as a filesystem in the OS code
- FFS used to manage *raw/bare* flash chip
  - Directly soldered on the motherboard
  - Mostly present in **embedded systems**
- Roles:
  - **Manage flash constraints**
  - **Manage embedded constraints**
  - **Manage regular filesystem operations**
- **Linux supports the most popular FFS**:
  - JFFS2, UBIFS, YAFFS2

Embedded flash management with Linux

Flash File Systems: Roles

- **Flash constraints management:**
  - Address translation
  - Wear leveling
  - Error correcting code processing implemented in the Linux NAND driver
- Address translation:



Embedded flash management with Linux

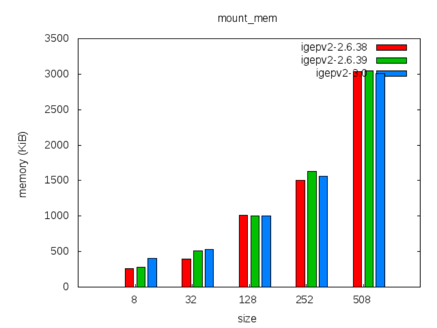
Flash File Systems: Roles (2)

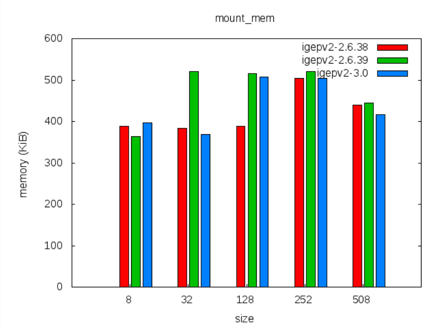
- **Embedded constraints**:
  - Limited resources (CPU power, RAM capacity)
  - Unclean unmount tolerance (ex: power cut):
    - Journaling, log-based structures, atomic operations
- FFS scalability with the managed flash space size:
  - Crucial metrics:
    - Mount time
    - RAM footprint
  - Linear vs logarithmic evolution

Embedded flash management with Linux

Flash File Systems: scalability

- RAM footprint of JFFS2 and UBIFS according to the managed flash size





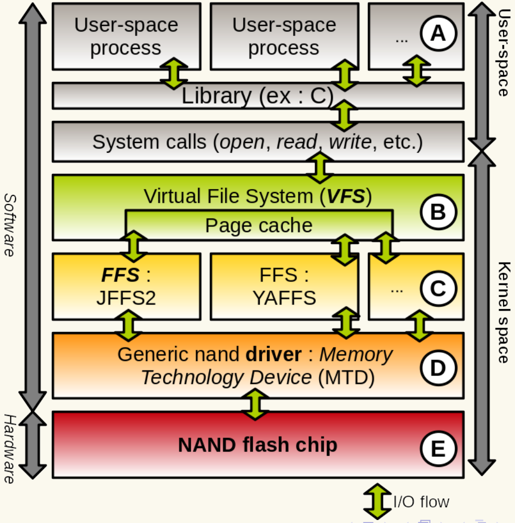
Source:

`http://elinux.org/Flash_Filesystem_Benchmarks_Kernel_Evolution`

Embedded flash management with Linux

FFS integration in Linux

1. VFS
2. FFS
3. Driver
4. Flash



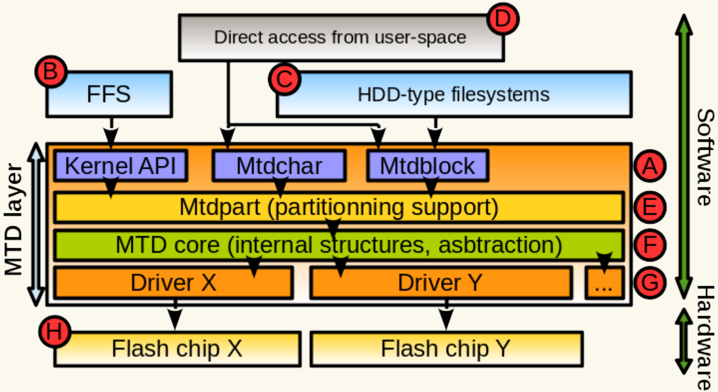
Embedded flash management with Linux

FFS integration in Linux: The Virtual File System

- **The Virtual File System (VFS)**
  - Abstraction layer for all filesystems supported by Linux
- Maintains a cache of file data, the **page cache**
  - All file accesses are buffered as long as there is some free RAM
- Some mechanisms associated with the page cache:
  - **Read-ahead**: data pre-fetching during read operations
  - Page cache **write-back**: buffer writes in RAM, and postpone them to absorb temporal locality
- Both of these topics (VFS & page cache) will have a dedicated lecture session

Embedded flash management with Linux

FFS integration in Linux: Memory Technology Device, the embedded flash driver



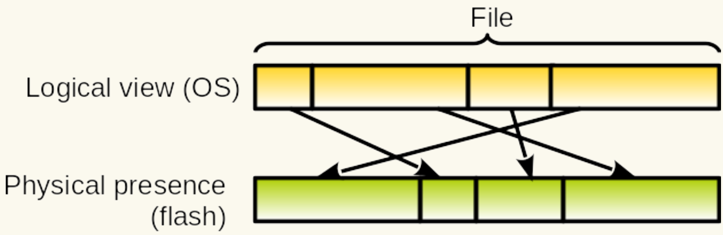
Embedded flash management with Linux

FFS integration in Linux: FFS implementation: JFFS2

- **JFFS2** integrated in the kernel mainline in 2001 (Linux 2.4.10)
  - Relatively mature and stable
- Each modification to the filesystem is packed in a **node** written (synchronously) on flash
- Nodes are indexed with a **table**
  - The entire managed flash partition is scanned at mount time to rebuild the table $\rightarrow$ 15 min. to mount a 1GB partition
  - **Linear scalability**
- Garbage collection uses lists of blocks with different states
  - Victims are blocks with large amount of invalid data
  - Wear leveling: 1 time upon 100, victim is a fully valid block
  - GC launched when free space is low, as well as in the background through a kernel thread

Embedded flash management with Linux

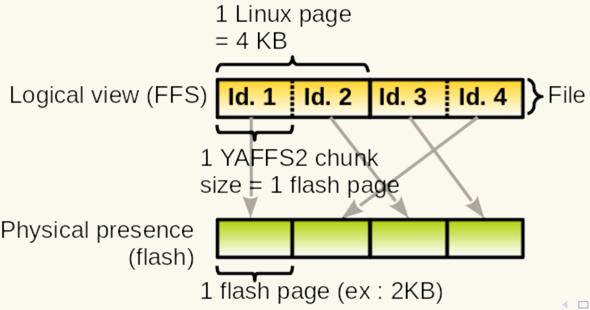
FFS integration in Linux: FFS implementation: JFFS2 (2)



Embedded flash management with Linux

FFS integration in Linux: FFS implementation: YAFFS2

- **YAFFS2** dates from 2002
- Extensively used in Android up to 2011/2012
- File data is divided into *chunks*
  - Fixed size: the size of one underlying flash page
- **Table** is used for chunk indexation → **linear scalability**

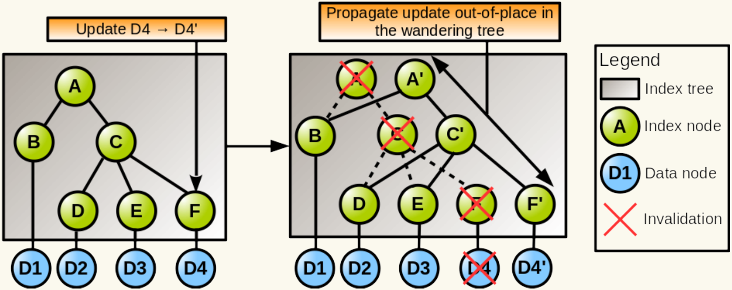


- YAFFS is not directly present in the kernel mainline
- Can be easily integrated through a patch

Embedded flash management with Linux

FFS integration in Linux: FFS implementation: UBIFS

- **UBIFS** integrated in Linux mainline in 2008 (2.6.27)
- Nodes indexation is done through a **tree** that is stored in flash
  - Only a subset of the tree is cached in RAM, brought on-demand
  - **logarithmic scalability**
  - Because there is no in-place updates on physical flash, the tree moves: **wandering tree**



Embedded flash management with Linux

JFFS2 vs YAFFS2 vs UBIFS

Feature	JFFS2	YAFFS2	UBIFS
Supported flash memory type	NOR, NAND	NAND	NOR, NAND
Virtual device type used	MTD	MTD	UBI (on MTD)
File indexing structure	Table	Table	(Wandering) tree
Compression algorithms supported	LZO, Zlib, Rtime	None	LZO, Zlib
Mount time scalability	Linear	Linear	Linear (UBI)
Memory footprint scalability	Linear	Linear	Logarithmic
Integration in Linux mainline	Yes	No (patch)	Yes

Outline

Conclusion

- **NAND flash memory**
  - Since a long time the main storage media in embedded systems
  - Widely present in servers, HPC, but also desktop/laptop computers (SSDs)
- **Specific constraints**:
  1. Erase-before-write
  2. Flash wear
  3. Reliability
  - Leads to **specific and complex management systems**
    - Flash Translation Layer
    - Flash File Systems

Bibliography I

[1] HIDAKA, H.
Evolution of embedded flash memory technology for mcu.
In *2011 IEEE International Conference on IC Design Technology* (May 2011), pp. 1–4.