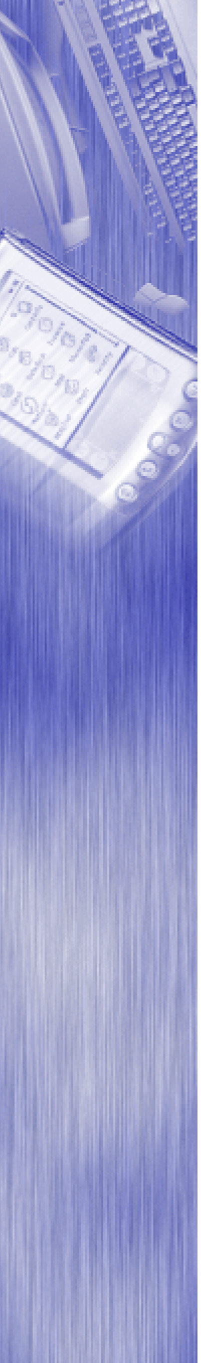


File Systems for Flash Memories

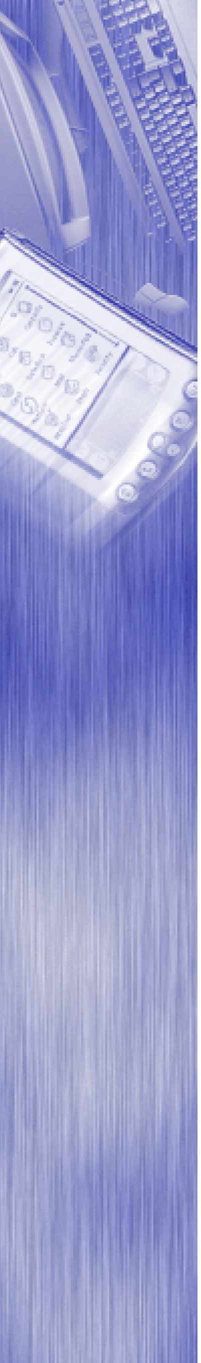
한국과학기술원 전산학과

김 진 수

(jinsoo@cs.kaist.ac.kr)

Outline

- Introduction to Flash Memories
- File Systems for Flash Memories
- JFFS / JFFS2
- LFFS



Introduction to Flash Memories

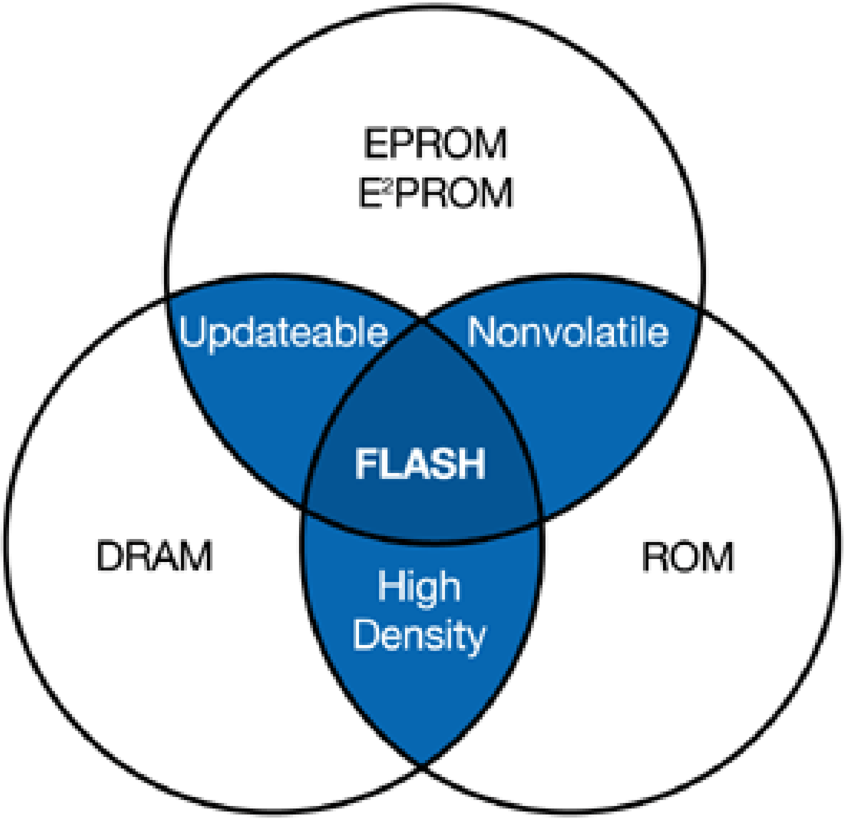
Memory Types

FLASH

- High-density
- Low-cost
- High-speed
- Low-power
- High reliability

DRAM

- High-density
- Low-cost
- High-speed
- High-power



EPROM

- Non-volatile
- High-density
- Ultraviolet light for erasure

EEPROM

- Non-volatile
- Lower reliability
- Higher cost
- Lowest density
- Electrically byte-erasable

ROM

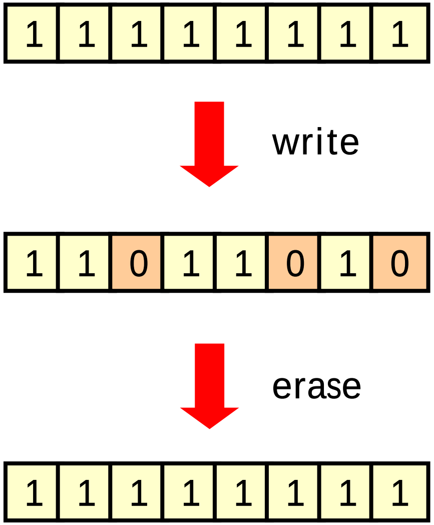
- High-density
- Reliable
- Low-cost
- Suitable for high production with stable code

Source: Intel Corporation.

Flash Memory Characteristics

Operations

- Read
- Write or Program – change state from 1 to 0
- Erase – change state from 0 to 1



1 1 1 1 1 1 1 1

1 1 0 1 1 0 1 0

1 1 1 1 1 1 1 1

Unit

- Page (sector) – management or program unit
- Block – erase unit

NOR vs. NAND Flash (1)

NOR Flash

- Random, direct access interface
- Fast random reads
- Slow erase and write
- Mainly for code storage
- Intel (28%), Spansion (25%), STMicro (13%), Samsung (7%), Toshiba (5%), ...

Source: iSuppli Corp. (Q2/2005)

NAND Flash

- I/O mapped access
- Smaller cell size
- Lower cost
- Smaller size erase blocks
- Better performance for erase and write
- Mainly for data storage
- Samsung (55%) , Toshiba (23%), Hynix (10%), Renesas (6%), STMicro (2%), Infineon (2%), Micron (2%)

NOR vs. NAND Flash (2)

Mass Storage-NAND

Memory Cards
(mobile computers)

Solid-State Disk
(rugged & reliable storage)

Digital Camera
(still & moving pictures)

Voice/Audio Recorder
(near CD quality)

- Low Cost and High Density
- Good P/E Cycling Endurance

Code Memory-NOR

BIOS/Networking
(PC/router/hub)

Telecommunications
(switcher)

Cellular Phone
(code & data)

POS / PDA / PCA
(code & data)

- Fast Random Access
- XIP

Source: Samsung Electronics

NOR vs. NAND Flash (3)

- **Access times comparison**

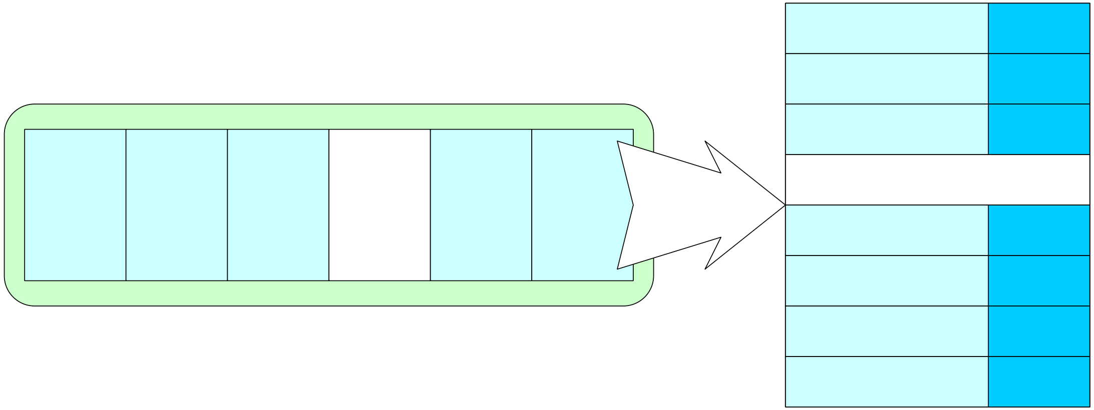
Media	Read	Write	Erase
DRAM	60ns (2B) 2.56us (512B)	60ns (2B) 2.56us (512B)	-
NOR Flash	150ns (2B) 14.4us (512B)	211ns (2B) 3.53ms (512B)	1.2s (128KB)
NAND Flash	10.2us (2B) 35.9us (512B)	201us (2B) 226us (512B)	2ms (16KB)
Disk	12.4ms (512B) (average)	12.4ms (512B) (average)	-

Flash: Beauty and the Beast

- **Flash memory is a beauty.**
  - Small, light-weight, robust, low-cost, low-power non-volatile device
- **Flash memory is a beast.**
  - Much slower program/erase operations
  - No in-place-update
  - Erase unit > write unit
  - Limited lifetime (100K~1M program/erase cycles)
  - Bad blocks (for NAND), ...
- **Software support for flash memory is very important for performance & reliability.**

NAND Flash Memory

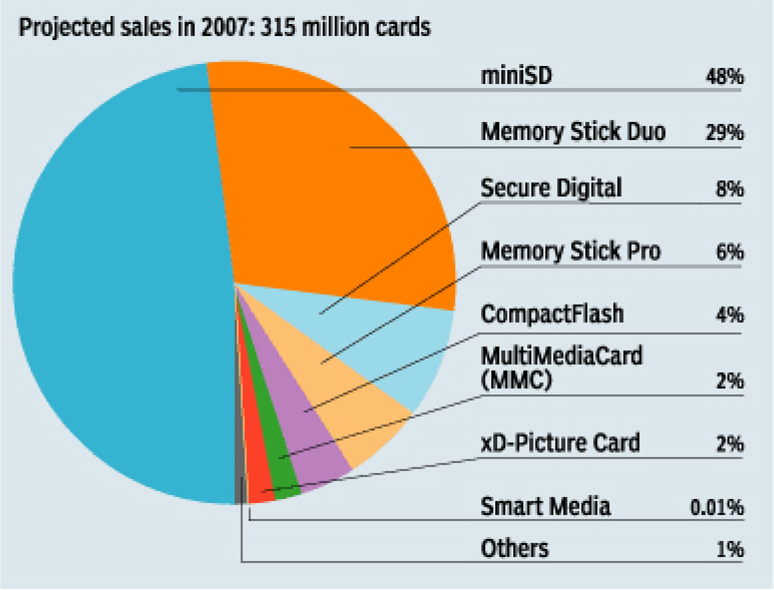
- **NAND Flash memory structure**



- Small Block NAND: (512+16)B/page, 32pages/block
- Large Block NAND: (2K+64)B/page, 64pages/block
- Limited NOP (Number of Programming): Usually 4

NAND Flash-based Storage (1)

- **Flash cards**
  - CompactFlash, MMC, SD/miniSD, Memory Stick, xD, ...



Source: IDC (from http://www.bitmicro.com)

NAND Flash-based Storage (2)

- **Flash SSDs (Solid State Disks)**
  - M-Systems FFD (Fast Flash Disk) 2.5”
    - – Solid-state flash disk in a 2.5” disk
    - – Up to 90GB
    - – ATA-6: interface speed of 100MB/s
    - – 40MB/s sustained read/write rates
    - – Released: March 10, 2004
    - – ~$40,000 for 90GB
  - BiTMICRO E-Disk
    - – Battery-backed DRAM + NAND Flash
  - Samsung Flash SSDs

NAND Flash-based Storage (3)

OKI Wireless Base Station

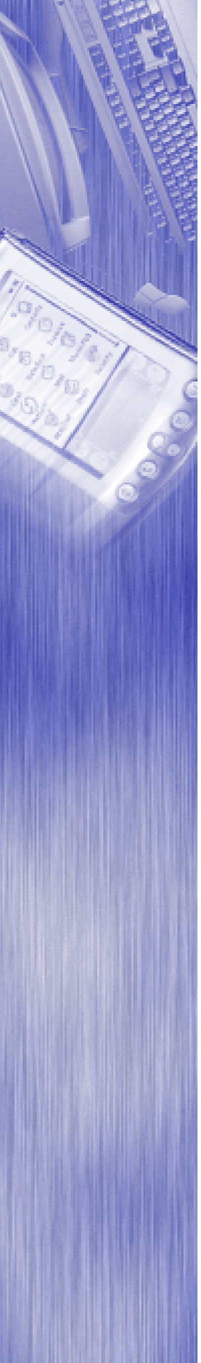
F-18 Hornet

Nortel Networks Optical Switches

Eurocopter AS 532U2 Cougar

NAND Flash-based Storage (4)

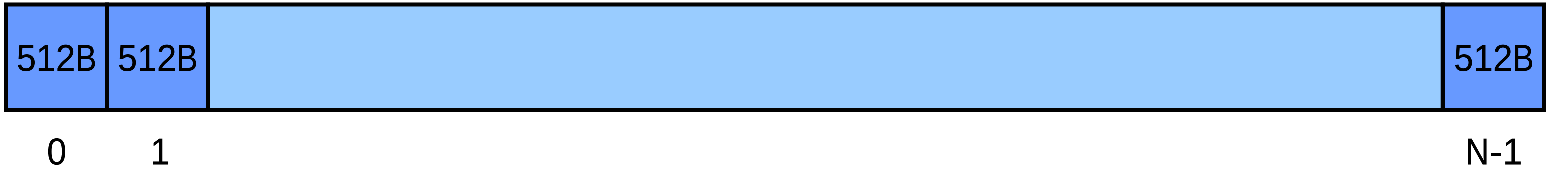
- **Flash-embedded devices**
  - Handheld phones
  - MP3 players
  - PMPs
  - PDAs
  - Digital TVs
  - Set-top boxes
  - Car navigation & entertainment systems
  - ...



File Systems for Flash Memories

Storage: A Logical View

- **Abstraction given by block device drivers:**



- **Operations**
  - Identify(): returns N
  - Read(start sector #, # of sectors)
  - Write(start sector #, # of sectors)

Source: Sang Lyul Min (Seoul National Univ.)

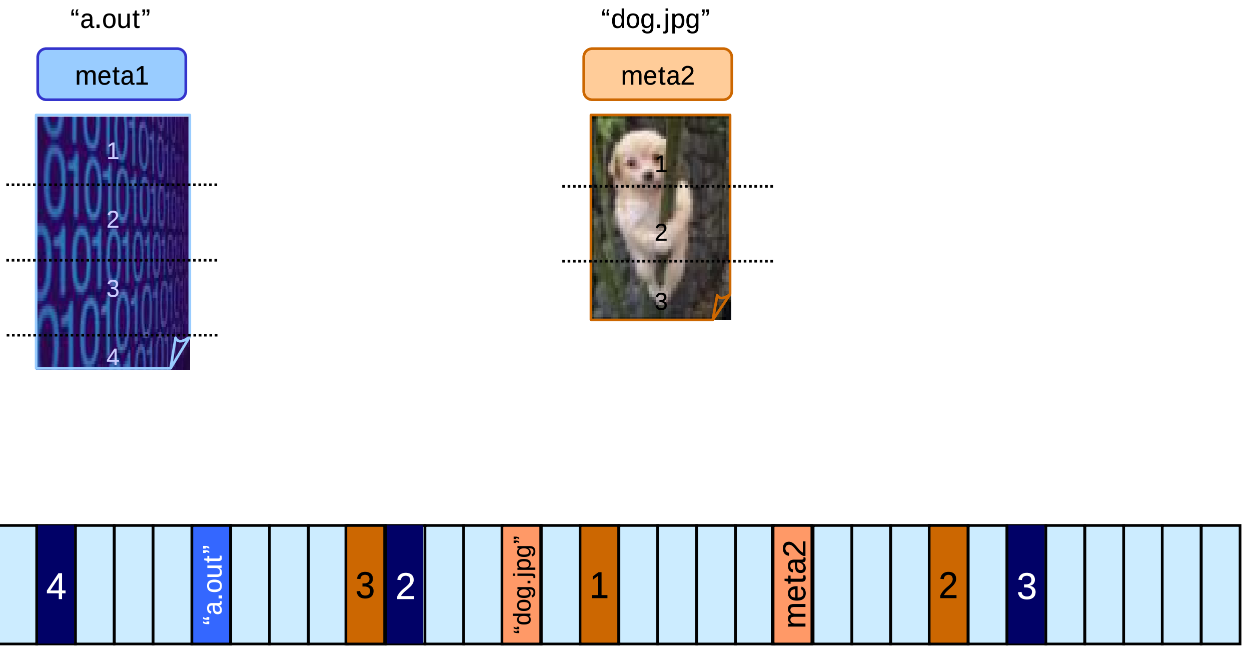
File System Basics (1)

- **For each file, we have**
  - File contents (data)
    - Nobody cares what they are.
  - File attributes (metadata)
    - File size
    - Owner, access control lists
    - Creation time, last access time, last modification time, ...
  - File name

- **File access begins with...**
  - File name
    - open ("/etc/passwd", O_RDONLY);

File System Basics (2)

- **File system: A mapping problem**
  - <filename, data, metadata> → <a set of blocks>



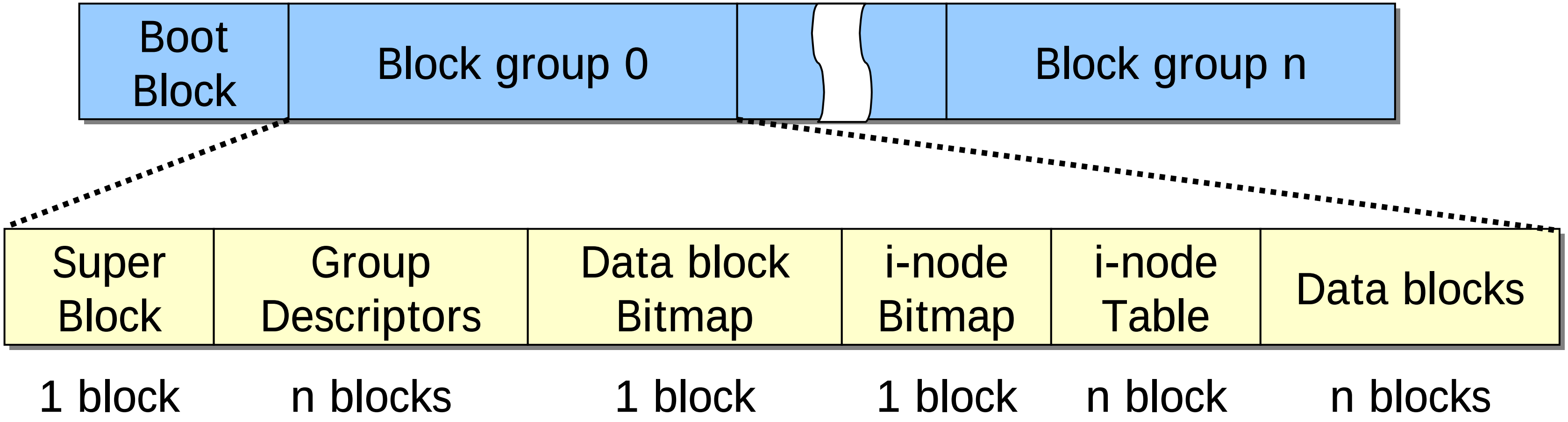
File System Basics (3)

- **Goals**
  - Performance + Reliability

- **Design Issues**
  - What information should be kept in metadata?
  - How to locate metadata?
    - – Mapping from pathname to metadata
  - How to locate data blocks?
  - How to manage metadata and data blocks?
    - – Allocation, reclamation, free space management, etc.
  - How to recover the file system after a crash?
  - ...

File System Example

- **Ext2 file system**
  - A disk-based file system for Linux
    - Similar to UNIX Fast File System (FFS)
    - Evolved to Ext3 File system (with journaling)
  - Directory: pathname → metadata (i-node)
  - Direct/indirect block pointers: i-node → data blocks



Flash File Systems

Disks vs. NAND Flash

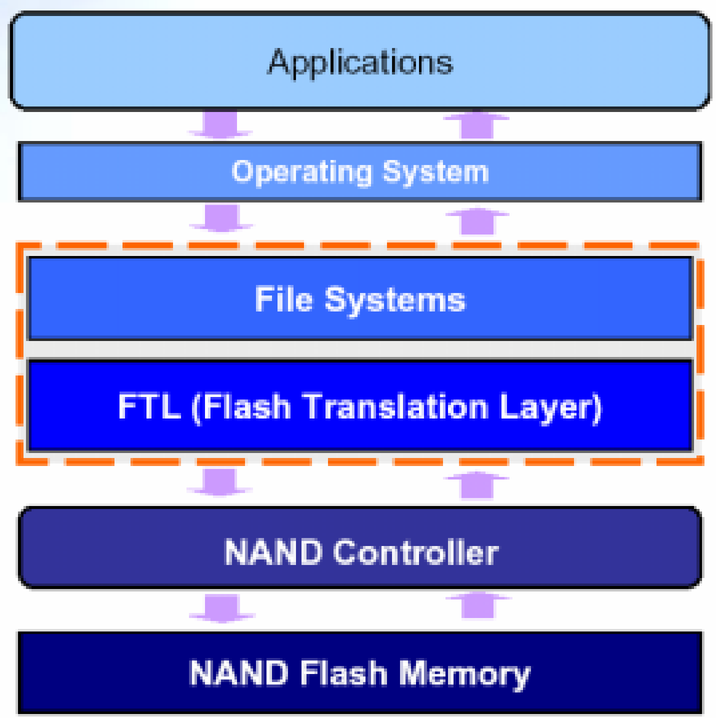
- No seek time
- Asymmetric read/write cost
- No in-place-update
- Wear-leveling

Approaches to flash file systems

- Layered approach
  - – Block device emulation using FTL (Flash Translation Layer)
- Native (or cross-layer) approach

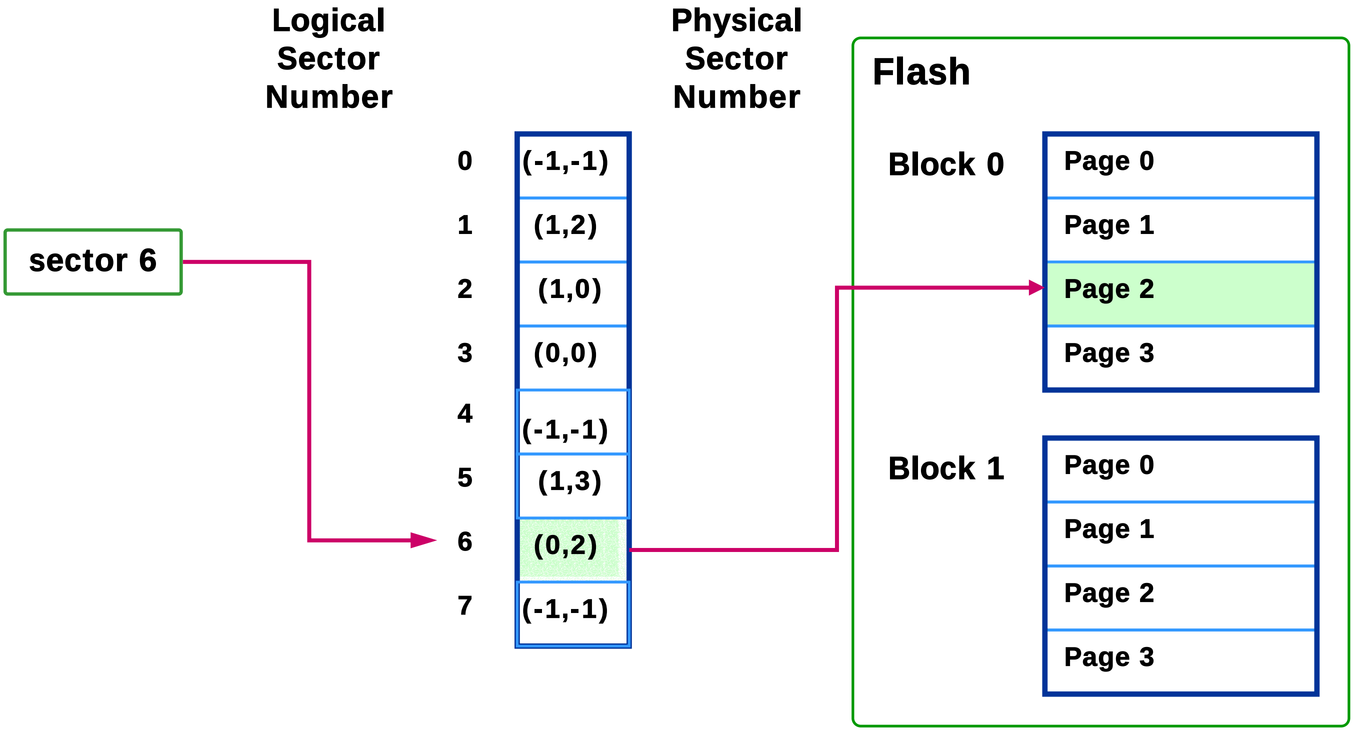
Layered Approach (1)

- **Flash Translation Layer (FTL)**
  - A software layer to make NAND flash fully emulate magnetic disks.
  - Sector mapping
  - Garbage collection
  - Power-off recovery
  - Bad block management
  - Wear-leveling
  - Error correction code (ECC)
  - Power management



Layered Approach (2)

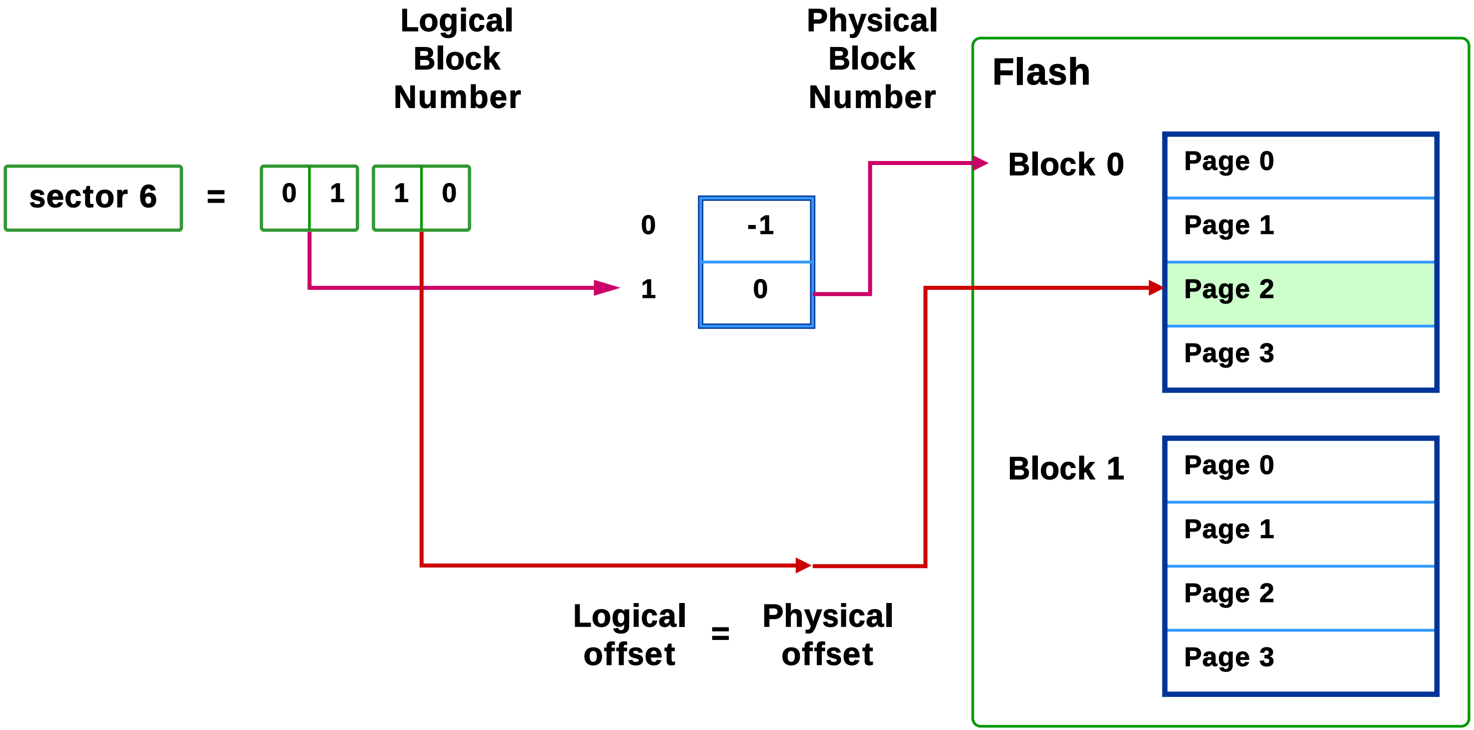
- **Page mapping in FTL**



Logical Sector Number	Physical Sector Number
0	(-1,-1)
1	(1,2)
2	(1,0)
3	(0,0)
4	(-1,-1)
5	(1,3)
6	(0,2)
7	(-1,-1)

Layered Approach (3)

- **Block mapping in FTL**



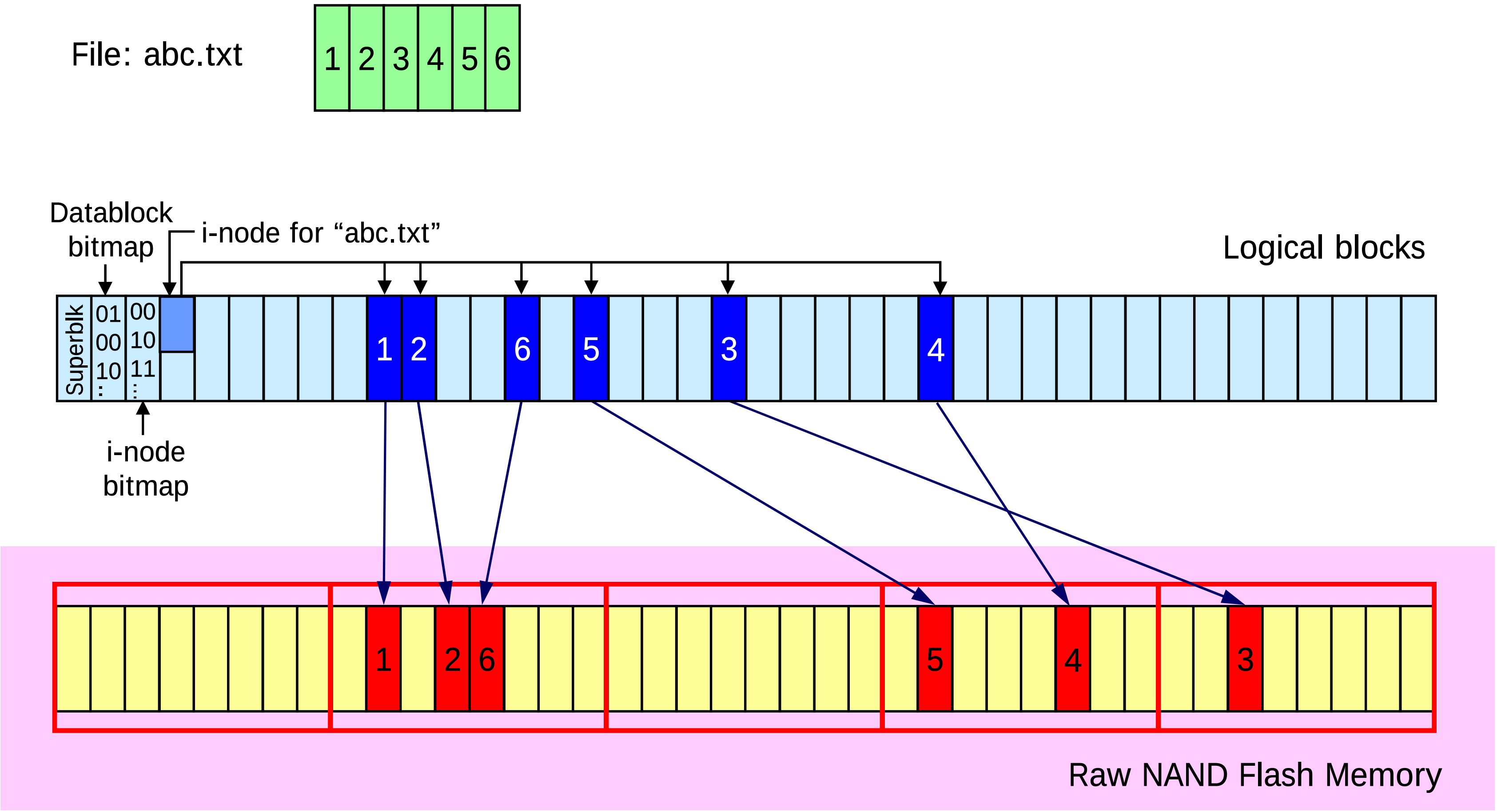
Layered Approach (4)

- **Benefits**
  - Easy to deploy.
    - No modification is required for upper layers.
    - Legacy file systems or swap space can be built.
  - Flash cards or flash SSDs already come with FTL.

- **Limitations**
  - Most FTLs are patented.
  - FTL can not make use of kernel-level information.
  - Kernel is not aware of the presence of flash memory.

Layered Approach (5)

- **What happens on file deletion?**



Native Approach

- **Cross-layer optimization**
  - Kernel manages raw flash memory directly.
  - More opportunities to optimize the performance.
  - Kernel is involved in some FTL functionalities.
    - Sector mapping, garbage collection, wear-leveling, power-off recovery, etc.
  - Example:
    - Flash-aware file systems: JFFS/JFFS2, YAFFS
  - Limitations
    - Need to change the host operating system
    - Only applicable Flash-embedded devices

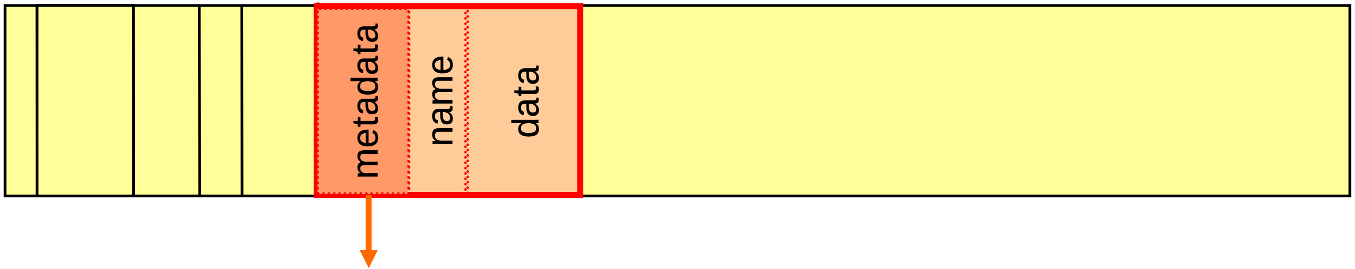
JFFS/JFFS2

JFFS (1)

- **JFFS (Journaling Flash File Systems)**
  - Developed by Axis Communications, Sweden in 1999.
  - Released under GNU GPL
  - Designed for small NOR flashes
  - A log-structured file system
    - Any file system modification is appended to the log.
    - The log is the only data structure on the flash media.
      Log = <metadata, (name), (data)>
    - A file is obsoleted by a later log in whole or in part.
    - Obsoleted logs are reclaimed via garbage collection.
  - Rely on special in-core data structures for filename→metadata, metadata→data mappings.

JFFS (2)

- **JFFS architecture**



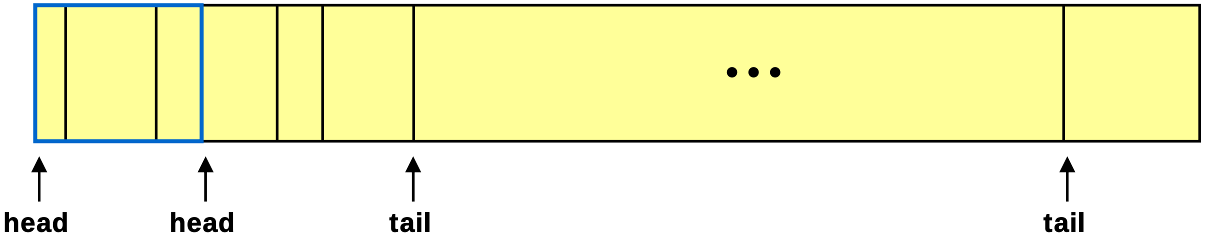
jffs_raw_inode

magic	: magic number
ino	: inode number
pino	: parent inode number
version	: version number
mode	: file's type or mode
uid **gid**	: file's owner and group
atime	: last access time
mtime	: last modification time
ctime	: creation time
offset	: where to begin to write
dsize	: size of the node's data
rsize	: how much are going to be replaced?
nsize **nlink** **flags**	: name length, number of links, flags for rename/deleted/accurate
dchksum	: checksum for the data
nchksum **chksum**	: checksums for the name and the raw inode

JFFS (3)

Garbage collection

- The free space is eventually exhausted. Now what?
- Erase the oldest block in the log.



- Live nodes should be moved.
- Perfectly wear-leveled.

JFFS2 (1)

JFFS limitations

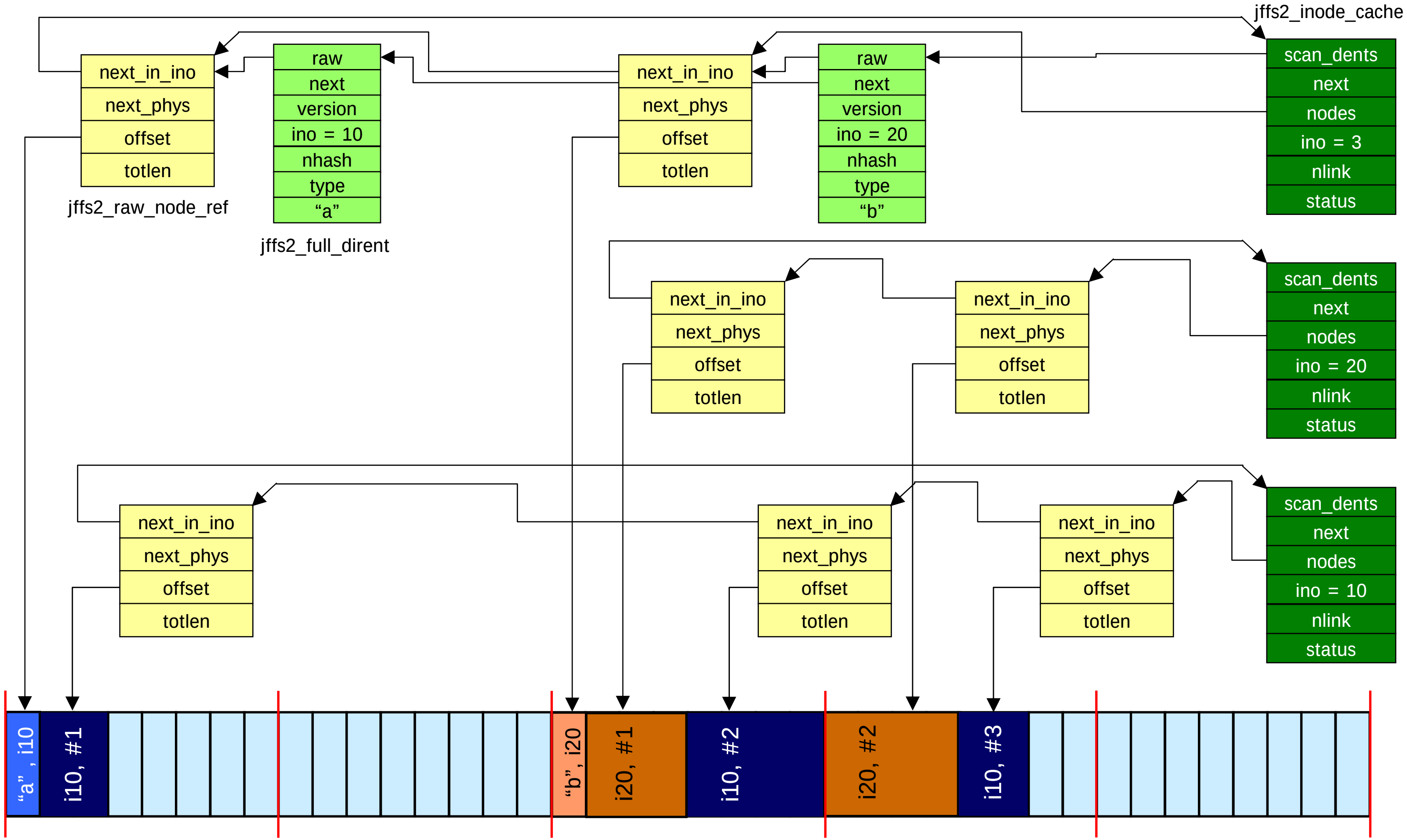
- Poor garbage collection performance
  - A block is garbage collected even if it contains only clean nodes.
  - In many cases, there are static data. (libraries, program executables, etc.)
- No compression support
  - Flash memories are expensive.
- No support for hard links
  - File name and parent i-node are stored in each i-node.
- No support for NAND flashes

JFFS2 (2)

- **Node types**
  - JFFS2_NODETYPE_INODE
    - – Similar to jffs_raw_inode
    - – No filename, no parent i-node number
    - – Compression support
  - JFFS2_NODETYPE_DIRENT
    - – Represent a directory entry, or a link
    - – File name, i-node, parent i-node (directory's i-node), etc.
    - – File name with i-node = 0: deleted file
  - JFFS2_NODETYPE_CLEANMARKER
    - – To deal with the problem of partially-erased blocks due to the power failure during erase operation

JFFS2 (3)

- JFFS2 architecture



JFFS2 (4)

- **What happens on mount:**
  - Physically scan the whole flash media
    - – Check CRC
    - – Build in-core data structures
      - » jffs2_raw_node_ref, jffs2_inode_cache, jffs2_full_dirent, etc.
  - Scan the directory tree, calculating nlink for each inode
  - Scan for inodes with nlink == 0 and remove them
  - Free temporary data structures
    - – e.g., jffs2_full_dirent

JFFS2 (5)

- **Block lists**
  - free_list: empty blocks
  - clean_list: blocks full of valid nodes
  - dirty_list: blocks containing at least one obsoleted node

- **Garbage collection**
  - Invoked if the size of free_list is less than the threshold.
  - Which blocks?
    - 99% from dirty_list (jiffies % 100 != 0)
    - 1% from clean_list (for wear-leveling)
  - Small nodes can be merged by GC.

JFFS2 (6)

JFFS2 limitations

- Large memory consumption
  - In-core data structures
    - » jffs2_raw_node_ref (16bytes/node), jffs2_inode_cache
- Slow mount time
  - 4 sec for 4MB!
- Runtime overheads (space & time)
  - Build child directory entries from flash on directory access
  - Build node fragments on file access
  - All the inode’s nodes should be examined (with CRC checked)
- Do not utilize NAND OOB area

JFFS2 (7)

JFFS2 memory consumption example

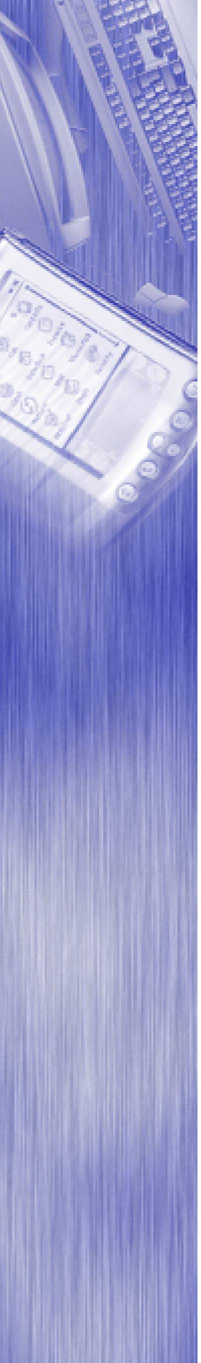
- JFFS2 with 64MB NAND flash

– Typical Linux root FS: (719 directories, 2995 regular files)	2.2MB
– 64MB file with 512bytes/node:	6.7MB
– 64MB file with 10bytes/node:	47.6MB

- JFFS2 with 1GB NAND flash (estimated)

– Typical Linux root FS:	34.7MB
– 64MB file with 512bytes/node:	104.2MB
– 64MB file with 10bytes/node:	743.6MB

(Source: JFFS3 Design Issues, June 4, 2005)



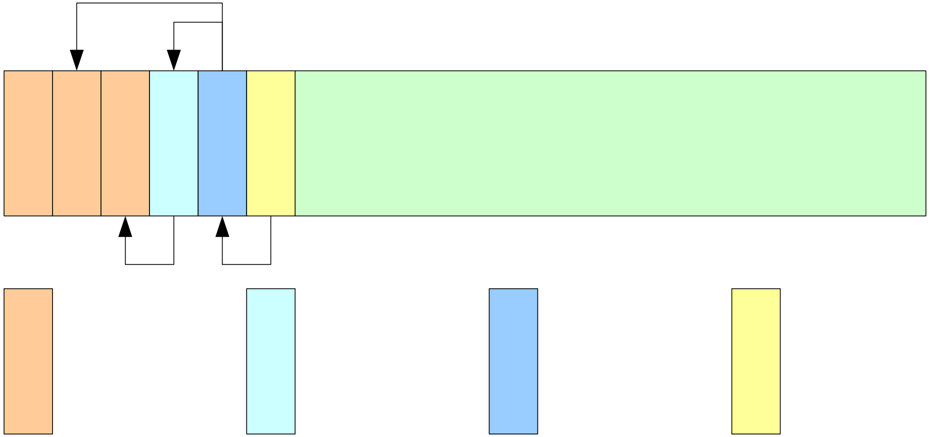
LFFS

Design Objectives

- **LFFS (Log-structured Flash File System)**
  - A file system for large block NAND flash memories running over Linux MTD
  - Scalable file system
    - Supporting up to several GB
  - Fast mount
  - Small memory footprint
  - Comparable performance to JFFS2

LFFS Approach (1)

- **Back to the original LFS design**



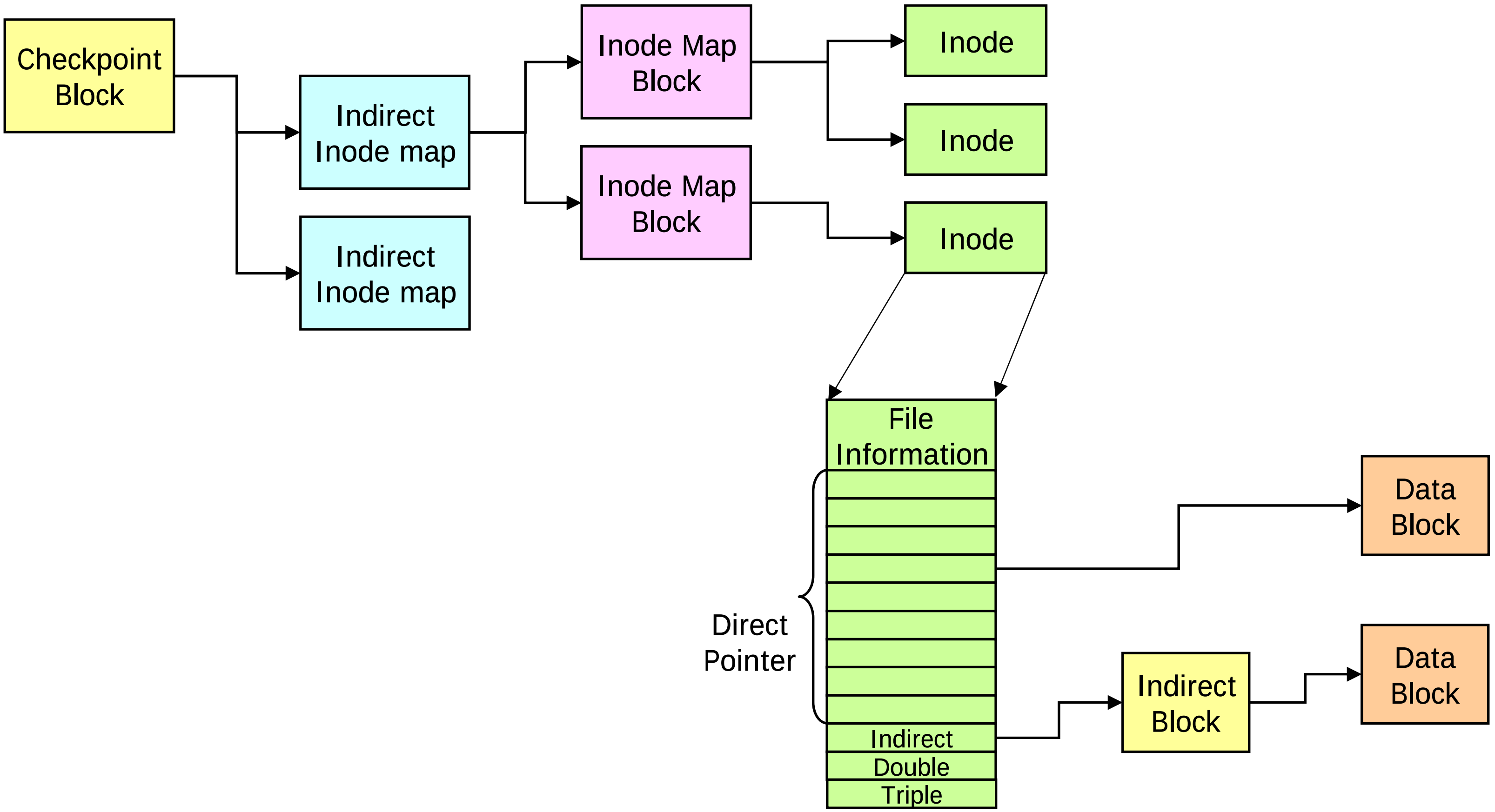
- VFS-compliant metadata structure and caching
- Fast mount and recovery using checkpoints

LFFS Approach (2)

- **LFFS differences**
  - Use multiple checkpoint blocks
    - – LFS: small (two) fixed checkpoint slots
    - – Avoid wear-out of checkpoint region
  - Introduce Indirect inode map block
    - – Points to the locations of inode map blocks
    - – Reduces the size of checkpoint data
  - Make use of OOB area in NAND flash
    - – Segment summary info.

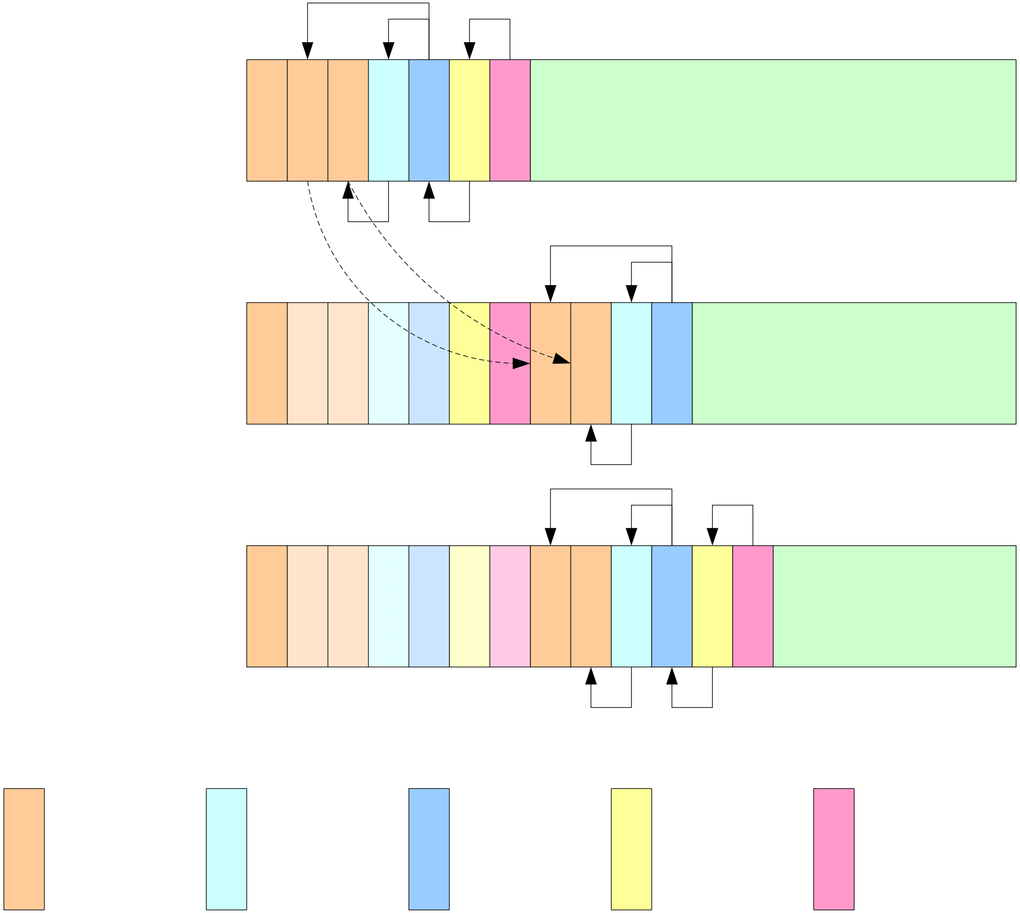
LFFS (1)

- **LFFS data structures**



LFFS (2)

- **LFFS architecture**



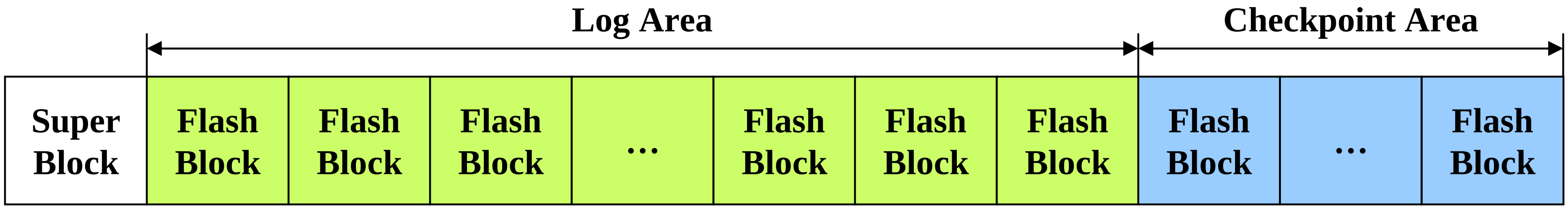
LFFS (3)

- **Blocks in LFFS**
  - Inode block
  - Indirect block
  - Data (directory) block
  - Inode map
    - Points to inode positions within flash memory
  - Indirect inode map
    - Points to inode map blocks (fully cached, 128KB)
  - Checkpoint block
  - OOB data
    - Bad block indicator, next log block, ECC
    - Inode number and offset for recovery

LFFS (4)

Multiple checkpoint blocks

- Checkpoint area
  - Recovery data and file system metadata
  - 256KB or 512KB



Log Area

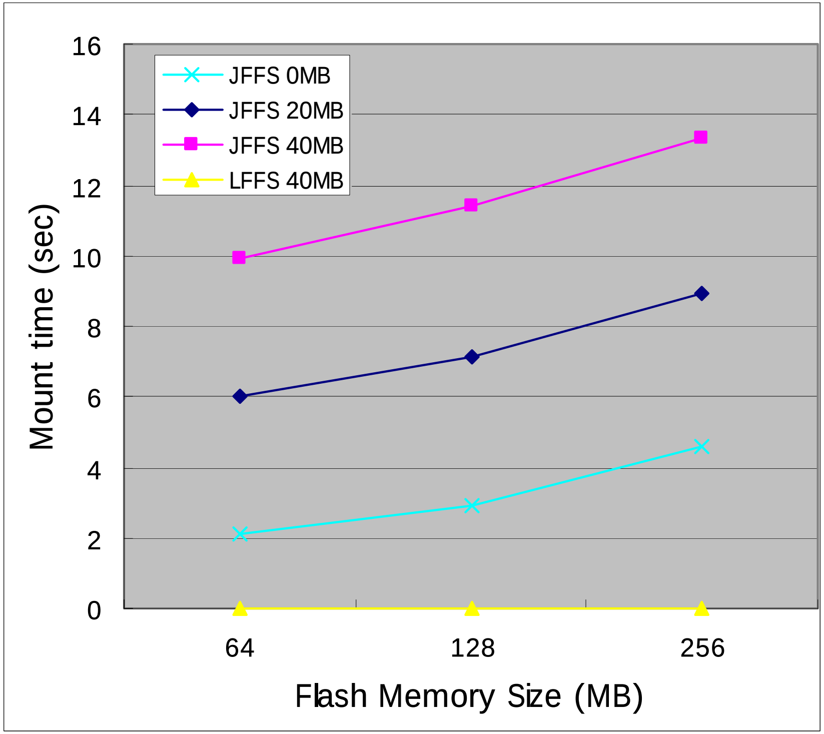
Checkpoint Area

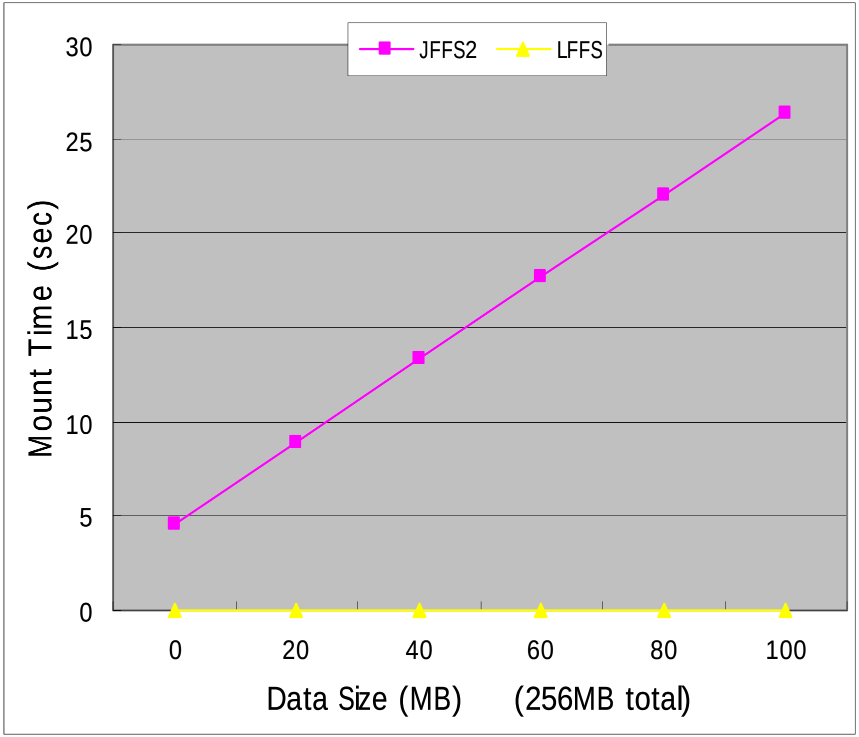
Super Block	Flash Block	Flash Block	Flash Block	…	Flash Block	Flash Block	Flash Block	Flash Block	…	Flash Block

- Total lifetime (checkpointing at every 15sec)
  - 15sec * (512KB/1KB * 100,000) ≈ 24 years

LFFS Performance (1)

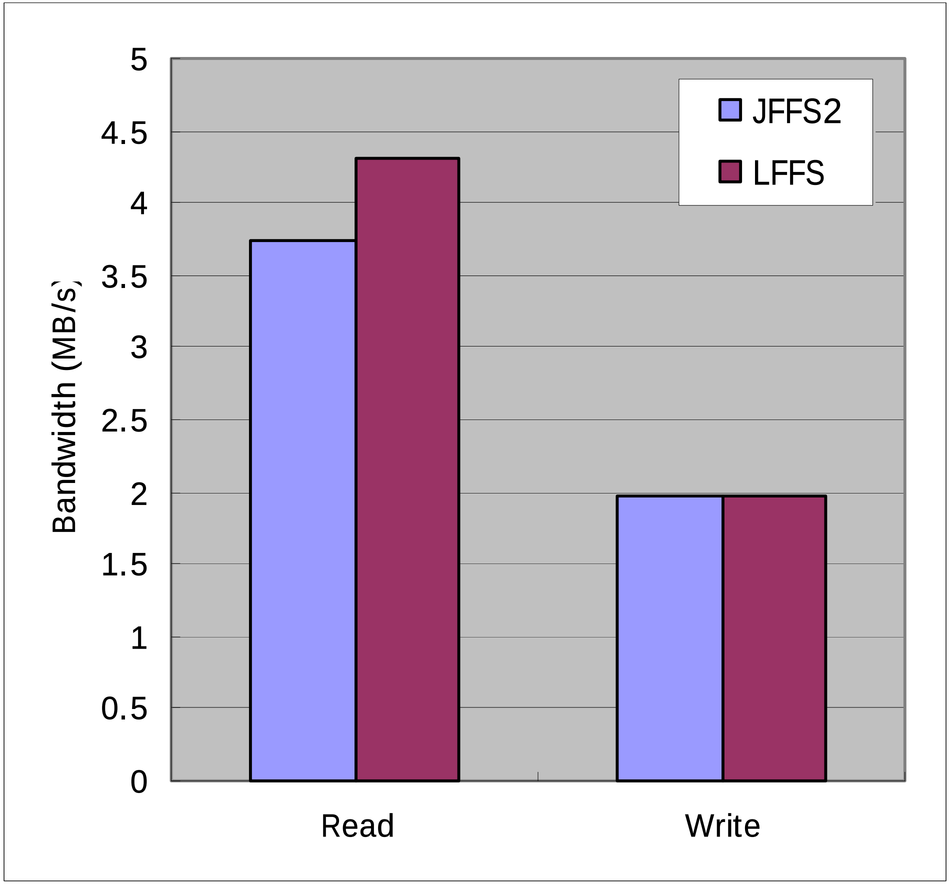
- **Mount time**





LFFS Performance (2)

- File read and write performance
  - Raw performance with nandsim
    - Read: 4.51MB/s
    - Write: 2.02MB/s
  - LFFS performance
    - Read: 4.31MB/s
    - Write: 1.98MB/s



Conclusion

- **Flash-aware file system has many opportunities.**

- **JFFS2 has drawbacks.**
  - Large memory footprints
  - Slow mount time

- **No clear winner, yet.**

- **Is an LFS-style file system the answer?**