Flash-aware File System

Flash-aware Computing

Instructor:

Prof. Sungjin Lee (sungjin.lee@dgist.ac.kr)

Today

- **File System Basics**
- Traditional Flash File Systems
- SSD-Friendly Flash File Systems
- Reference

What is a File System?

- **Provides a virtualized logical view of information stored on various storage media, such as disks, tapes, and flash-based SSDs**

- **Two key abstractions have developed over time in the virtualization of storage**
  - ***File***: A linear array of bytes, each of which you can read or write
    - Its contents are defined by a creator (e.g., text and binary)
    - It is often referred to as its *inode* number
  - ***Directory***: A special file that is a collection of files and other directories
    - Its contents are quite specific – it contains a list of (user-readable name, inode #) pairs (e.g., (“foo”, 10))
    - It has a hierarchical organization (e.g., tree, acyclic-graph, and graph)
    - It is also identified by an inode number

Operations on Files and Directories

POSIX Operations on Files

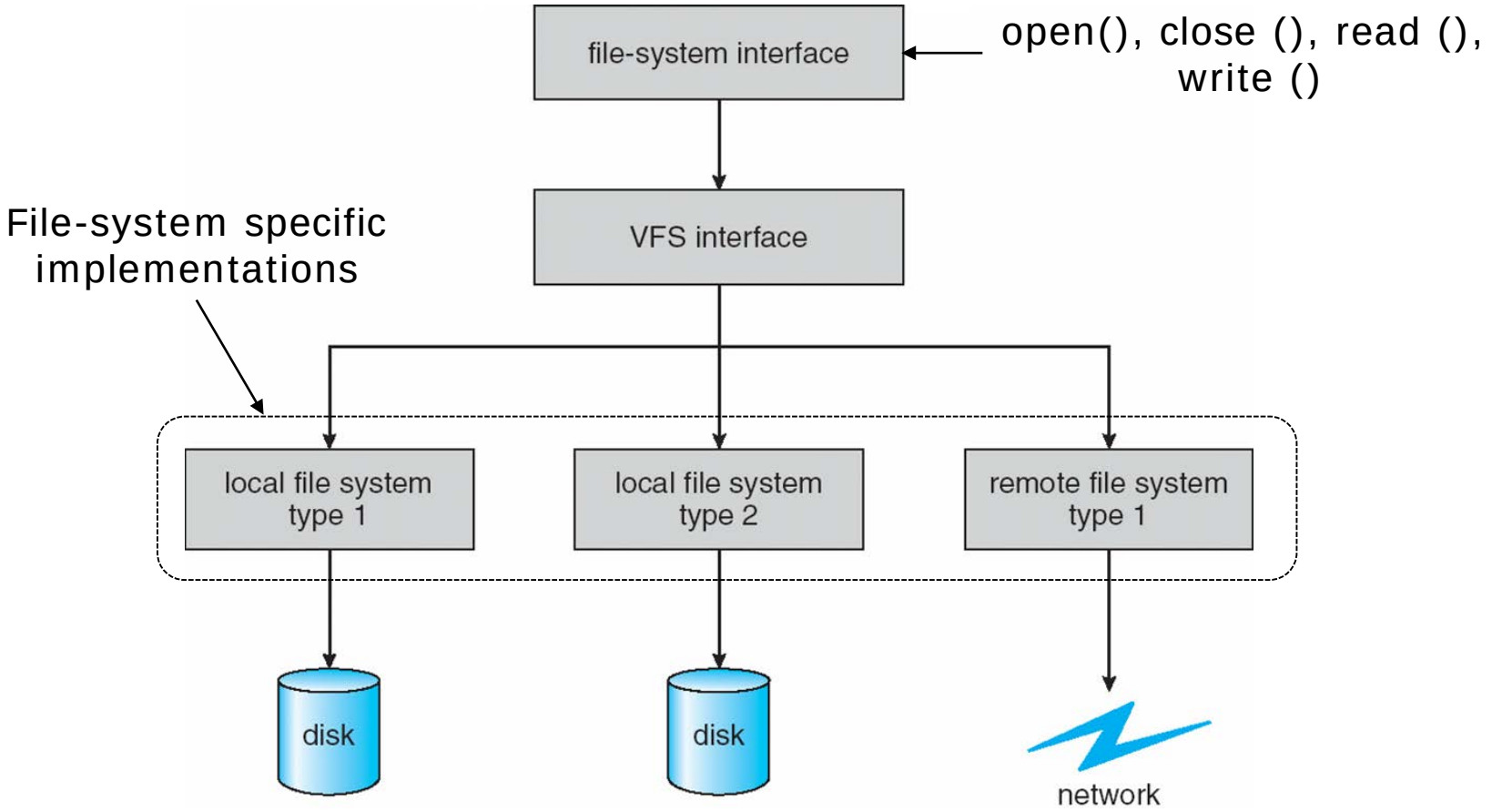
POSIX APIs	Description
creat ()	Create a file
open ()	Create/open a file
write ()	Write bytes to a file
read ()	Read bytes from a file
lseek ()	Move byte position inside a file
unlink ()	Remove a file
truncate ()	Resize a file
close ()	Close a file
	…

POSIX Operations on Directories

POSIX APIs	Description
opendir ()	Open a directory for reading
closedir ()	Close a directory
readdir ()	Read one directory entry
rewinddir ()	Rewind a directory so it can be reread
mkdir ()	Create a new directory
rmdir ()	Remove a directory
	…

Virtual File System

- **The POSIX API is to the VFS interface, rather than any specific type of file system**

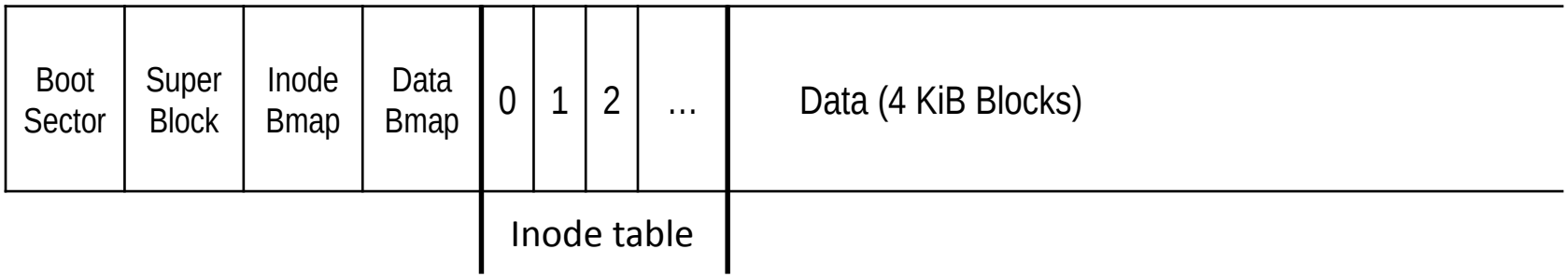


File System Implementation

- UNIX File System
- Journaling File System
- Log-structured or Copy-on Write File Systems

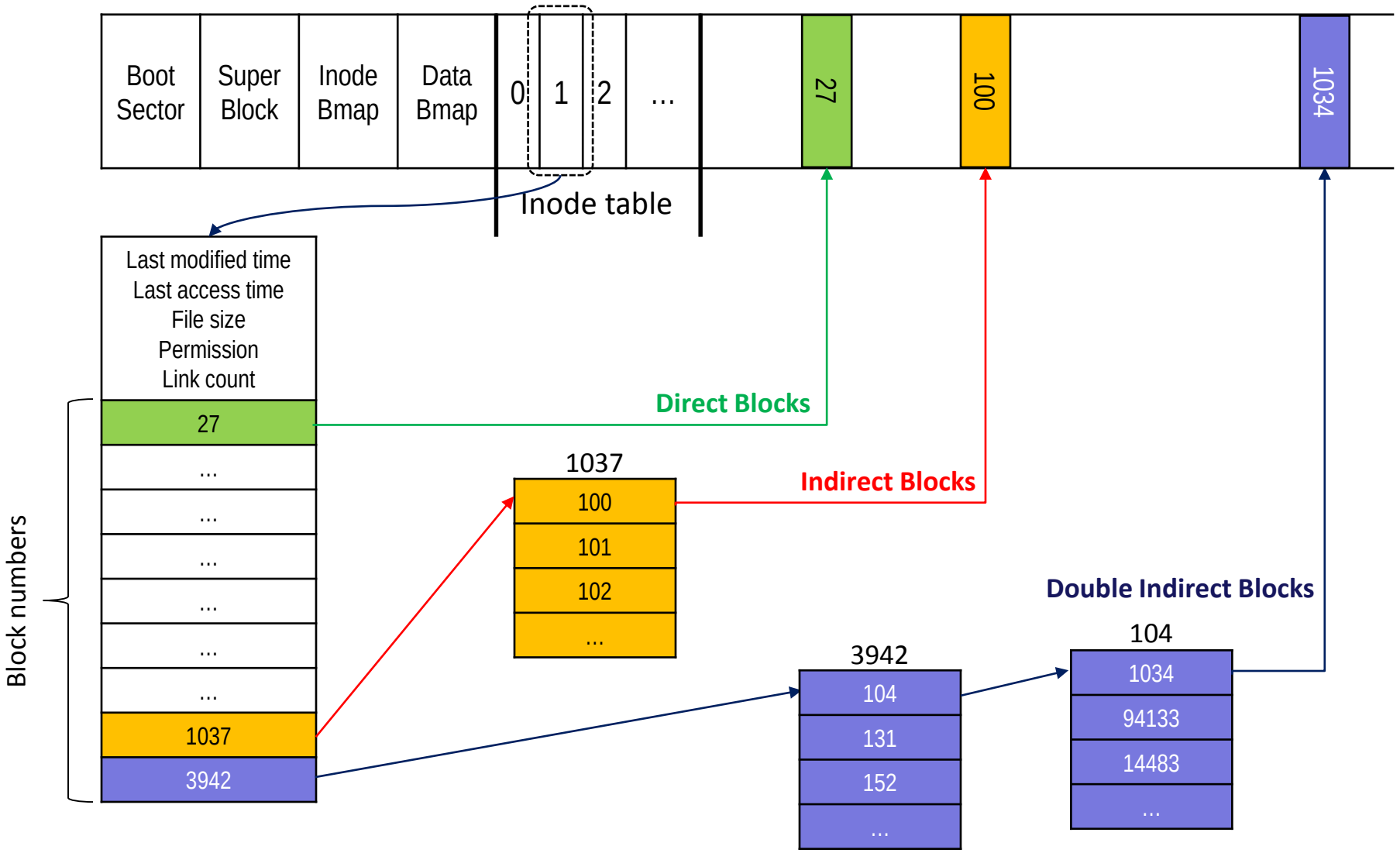
UNIX File System

- A traditional file system first developed for UNIX systems



- **Boot sector**: Information to be loaded into RAM to boot up the OS
- **Superblock**: File system's metadata (e.g., file system type, size, …)
- **Inode & data Bmaps**: Keep the status of blocks belonging to an inode table and data blocks
- **Inode table**: Keep file's metadata (e.g., size, permission, …) and data block pointers
- **Data blocks**: Keep users' file data

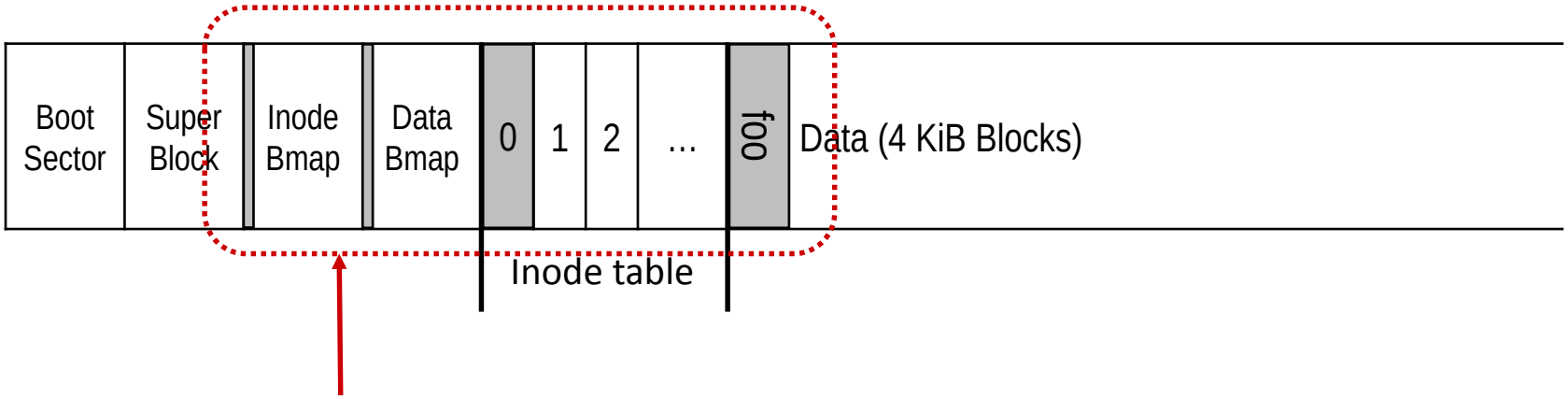
Inode & Block Pointers



Consistent Update Problem

- What happens if sudden power loss occurs while writing data to a file

write (0, “foo”, strlen (“foo”));

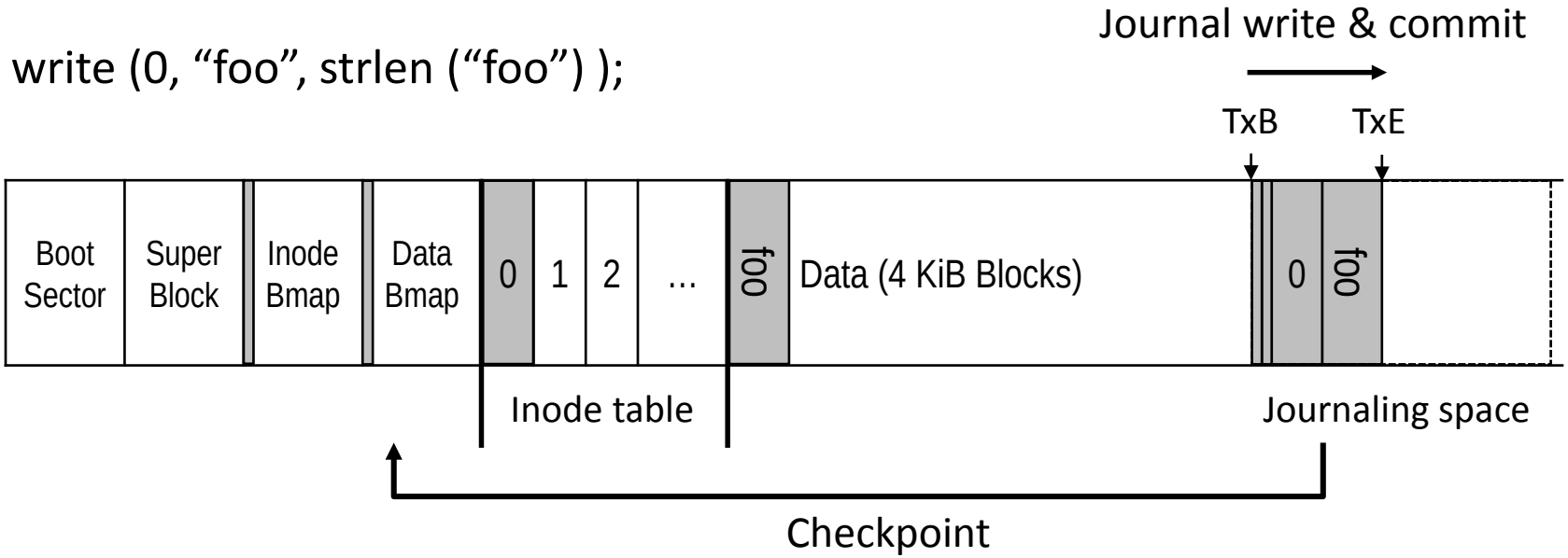


The file system will be inconsistent!!!

→ **Consistent update problem**

Journaling File System

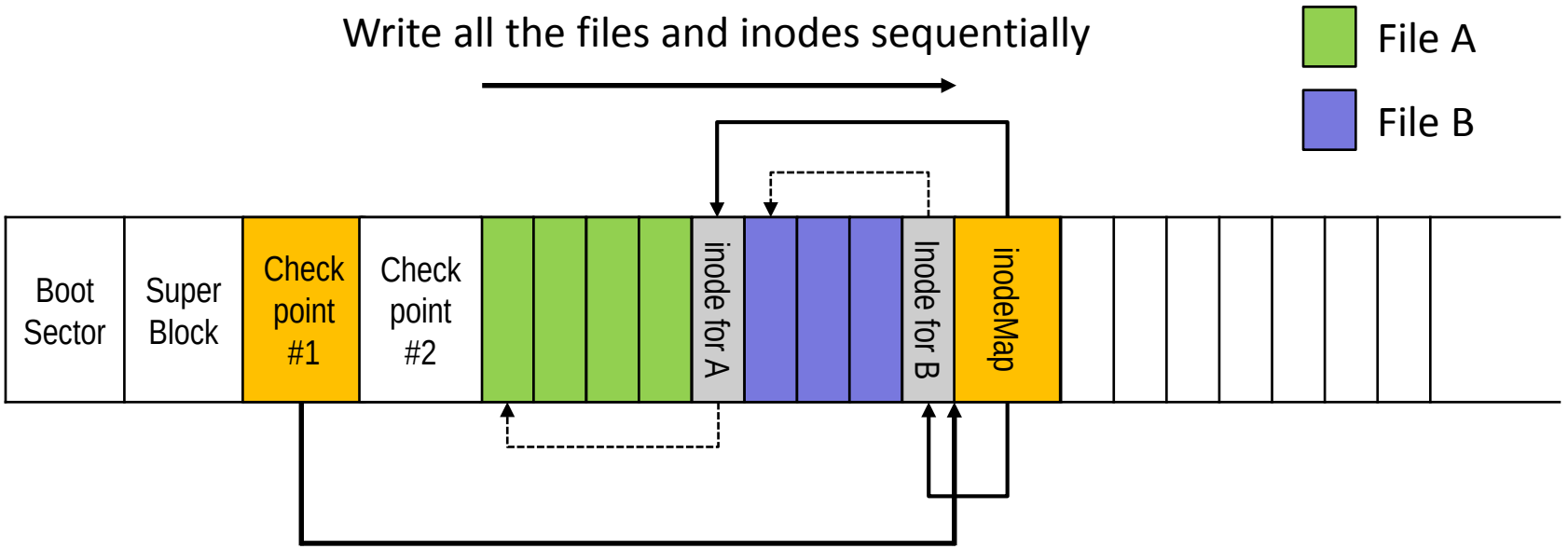
- Journaling file systems address the consistent update problem by adopting an idea of *write-ahead logging (or journaling)* from database systems
- Ext3, Ext4, ReiserFS, XFS, and NTFS are based on journaling



- Double writes could degrade overall write performance!

Log-structured File System

- Log-structured file systems (LFS) treat a storage space as a huge log, appending all files and directories sequentially
- The state-of-the-art file systems are based on LFS or CoW
  - e.g., Sprite LFS, F2FS, NetApp's WAFL, Btrfs, ZFS, …

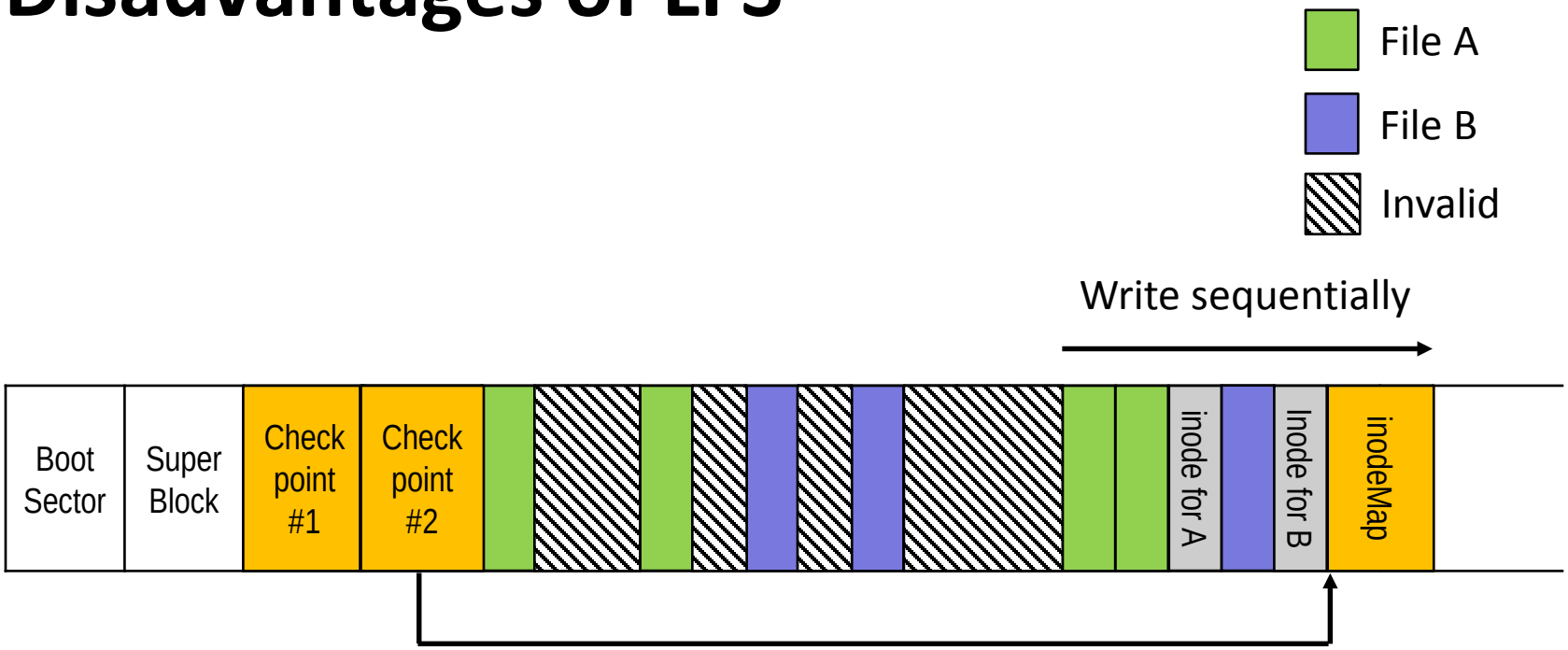


Log-structured File System (Cont.)

- **Advantages**
  - **(+)** No consistent update problem
  - **(+)** No double writes – an LFS itself is a log!
  - **(+)** Provide excellent write performance – disks are optimized for sequential I/O operations
  - **(+)** Reduce the movements of disk headers further (e.g., inode update and file updates)

- **Disadvantages**
  - **(–)** Expensive garbage collection cost
  - **(–)** Slow read performance

Disadvantages of LFS



- **Expensive garbage collection cost**: invalid blocks must be reclaimed for future writes; otherwise, free disk space will be exhausted
- **Slow read performance**: involve more head movements for future reads (e.g., when reading the file A)

Write Cost

- **Write cost with GC is modeled as follows**
  - Note: a segment (seg) is a unit of space allocation and GC

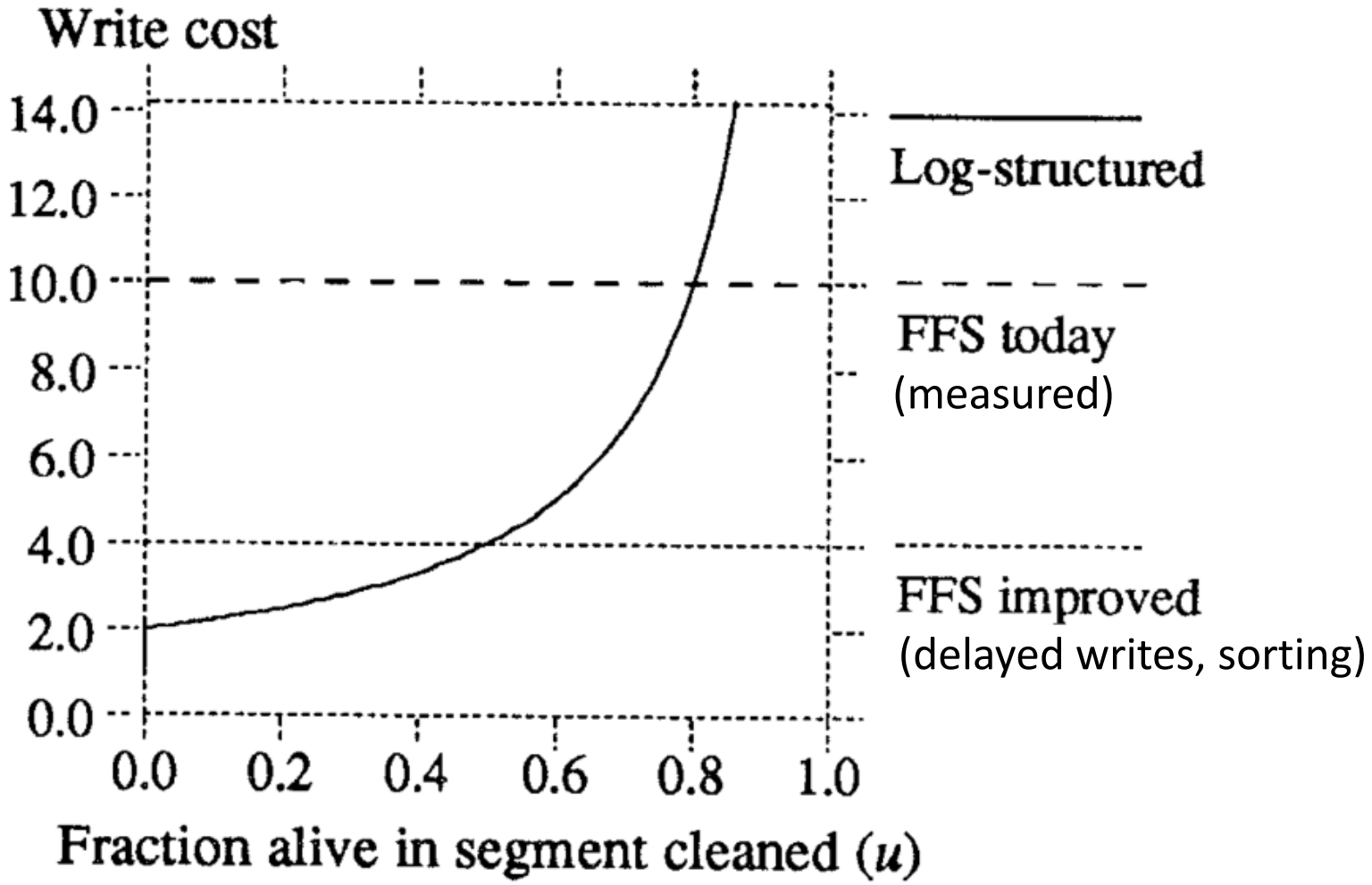
$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u}$$

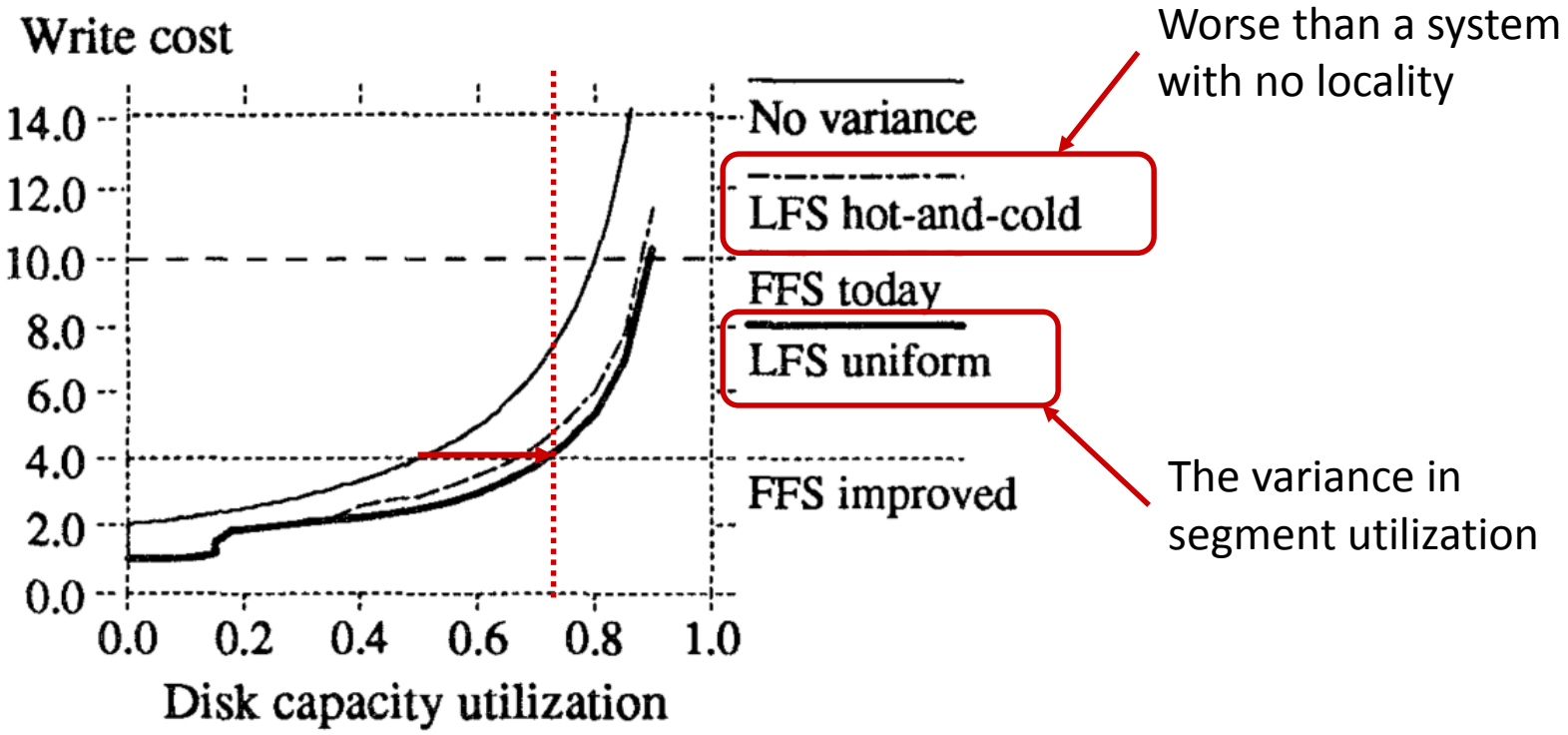
  - N is the number of segments
  - μ is the utilization of the segments ($0 \leq \mu < 1$)
  - If segments have no live data ($\mu = 0$), write cost becomes 1.0

Write Cost Comparison



Greedy Policy

- The cleaner chooses the *least-utilized segments* and *sorts the live data by age* before writing it out again
- Workloads: 4 KB files with two overwrite patterns
  - (1) **Uniform**: No locality – equal likelihood of being overwritten
  - (2) **Hot-and-cold**: Locality – 10:90



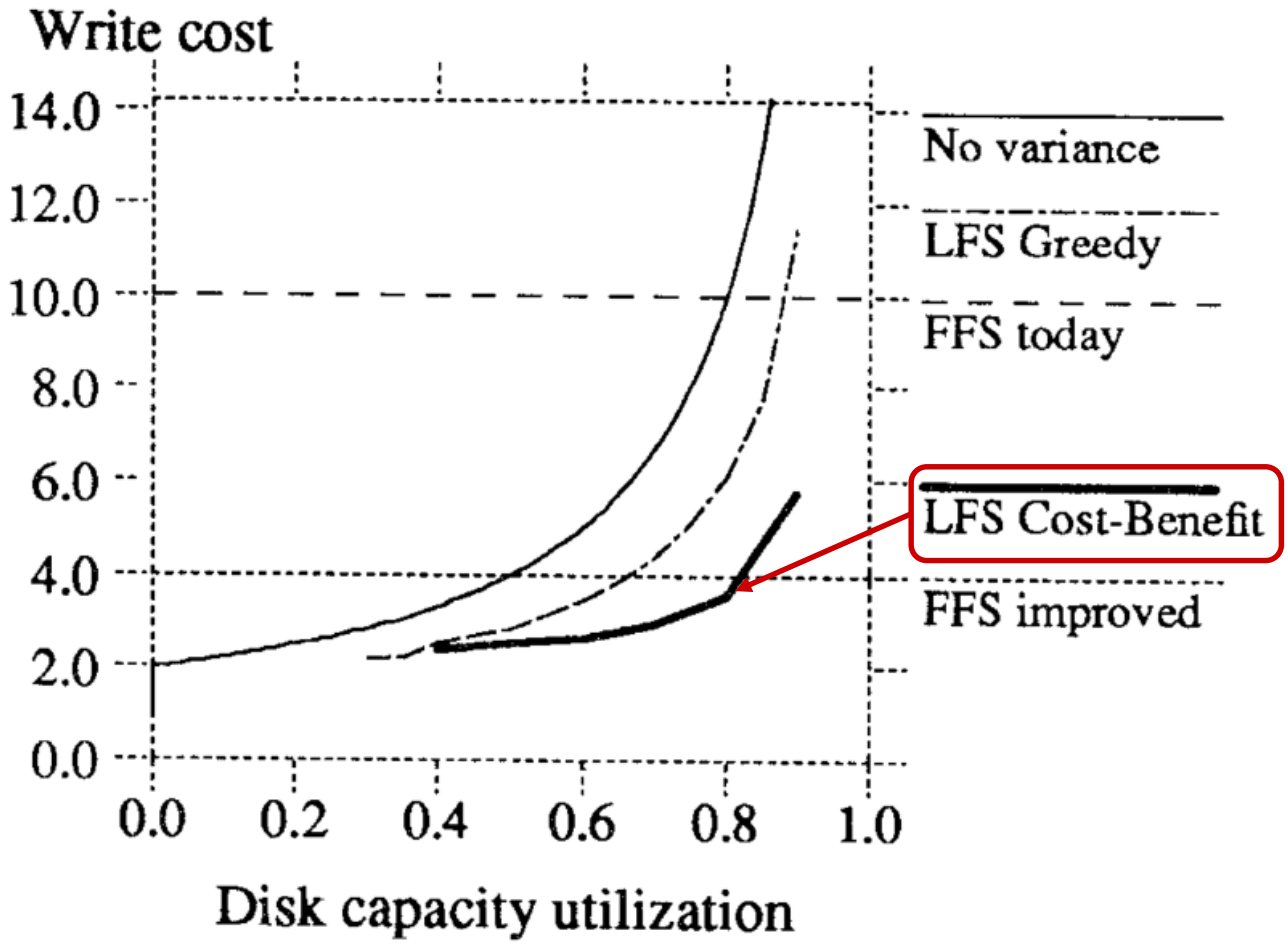
Cost-Benefit Policy

- Hot segments are frequently selected as victims even though their utilizations would drop further
  - It is necessary to delay cleaning and let more of the blocks die
  - On the other hand, free space in cold segments are valuable

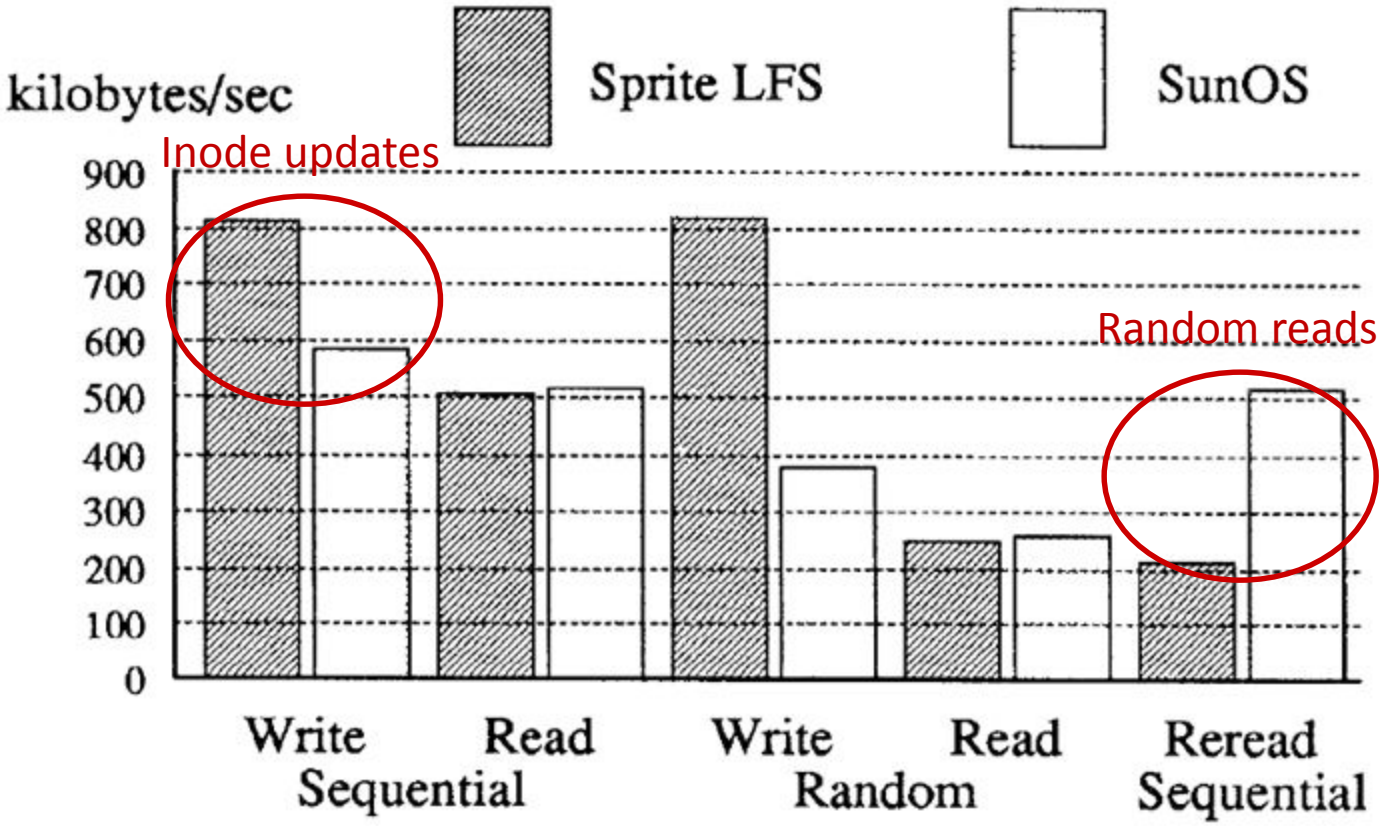
- Cost-benefit policy:

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated*age of data}}{\text{cost}} = \frac{(1-u)*\text{age}}{1+u}.$$

Cost-Benefit Policy (Cont.)



LFS Performance

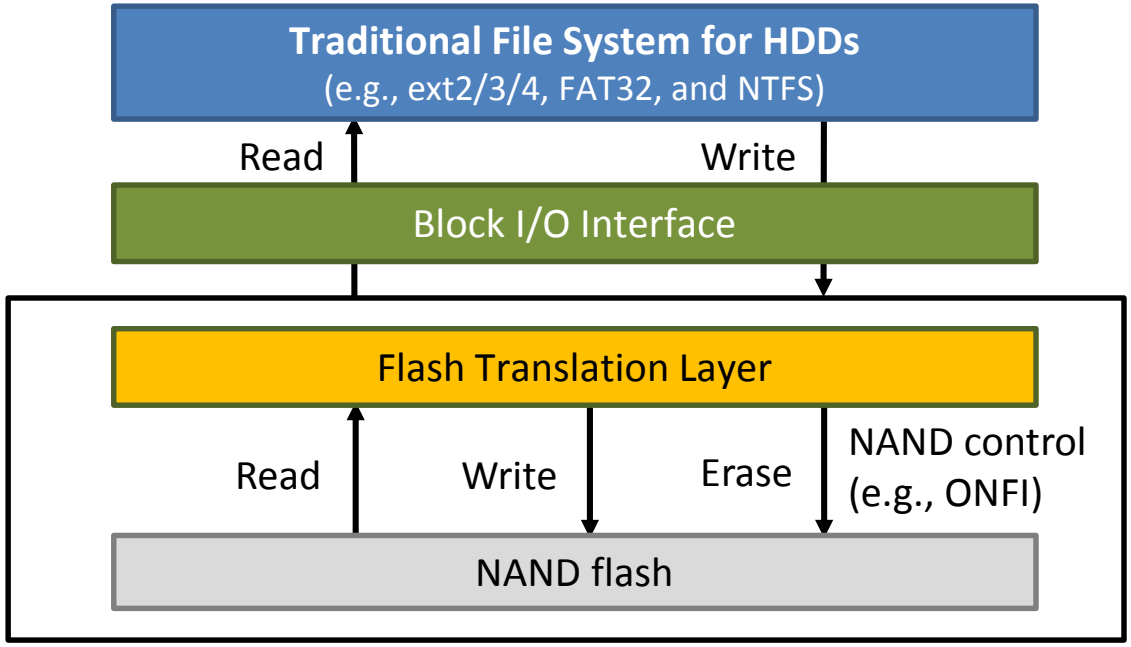


Today

- File System Basics
- **Traditional Flash File Systems**
  - JFFS2: Journaling Flash File System
  - YAFFS2: Yet Another Flash File System
  - UBIFS: Unsorted Bock Image File System
- SSD-Friendly File Systems
- Reference

Traditional File Systems for Flash

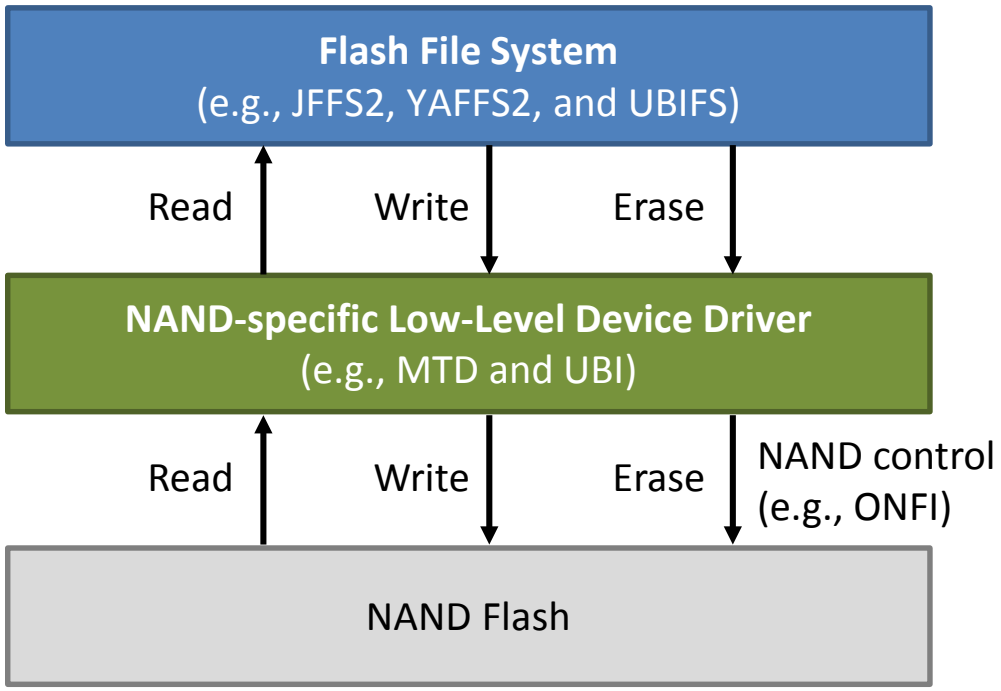
- **Originally designed for block devices like HDDs**
  - e.g., ext2/3/4, FAT32, and NTFS
- **But, NAND flash memory is not a block device**
  - The FTL provides block-device views outside, hiding the unique properties of NAND flash memory



Flash-based SSDs

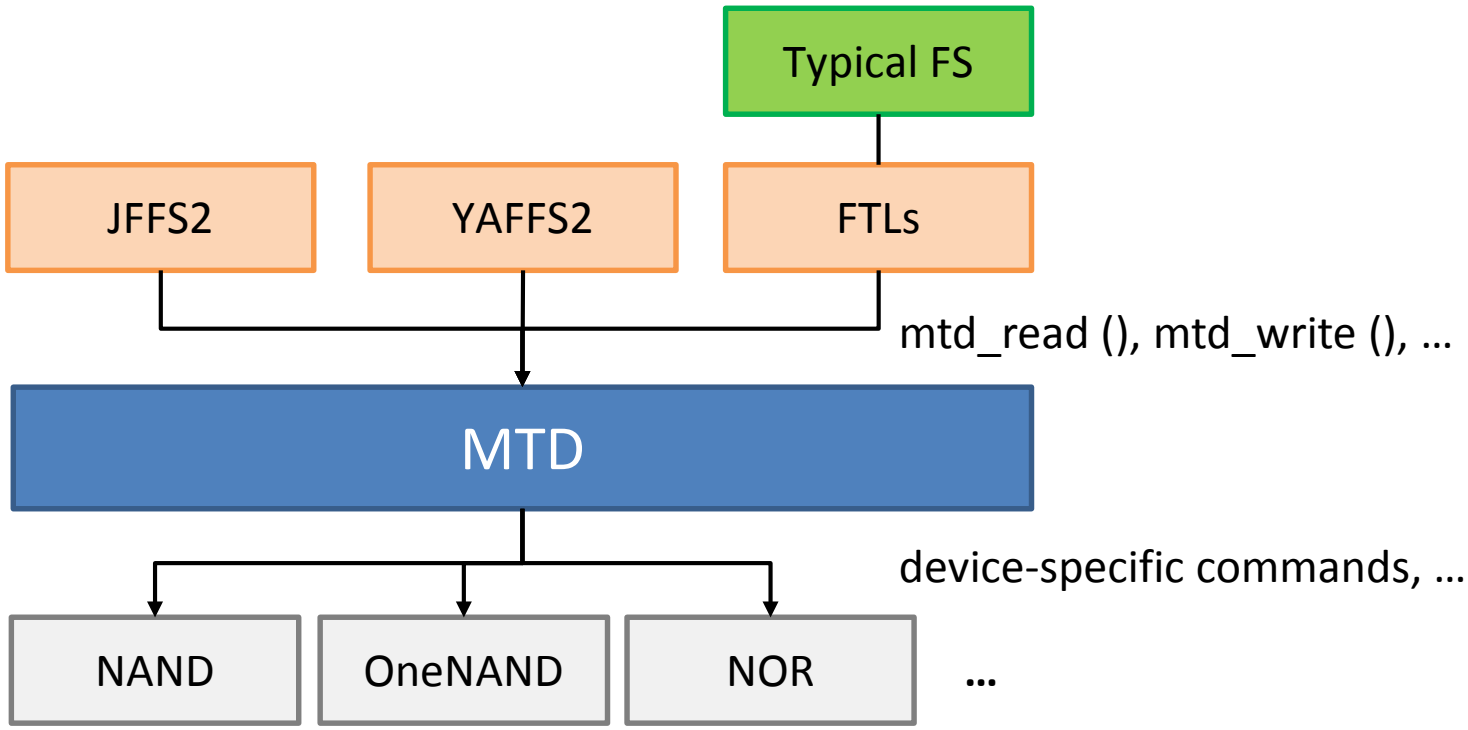
Flash File Systems

- **Directly manage raw NAND flash memory**
  - Internally performing address mapping, garbage collection, and wear-leveling by itself
- **Representative flash file systems**
  - JFFS2, YAFFS2, and UBIFS



Memory Technology Device (MTD)

- **MTD is the lowest level for accessing flash chips**
  - Offer the same APIs for different flash types and technologies
    - e.g., NAND, OneNAND, and NOR
- **JFFS2 and YAFFS2 run on top of MTD**



Traditional File Systems vs. Flash File Systems

	File System + FTL	Flash File System
Method	- Access a flash device via FTL	- Access a flash device directly
Pros	- High interoperability - No difficulties in managing recent NAND flash with new constraints	- High-level optimization with system-level information - Flash-aware storage management
Cons	- Lack of system-level information - Flash-unaware storage management	- Low interoperability - Must be redesigned to handle new NAND constraints

Flash file systems now become obsolete because of difficulties for the adoption to new types of NAND devices

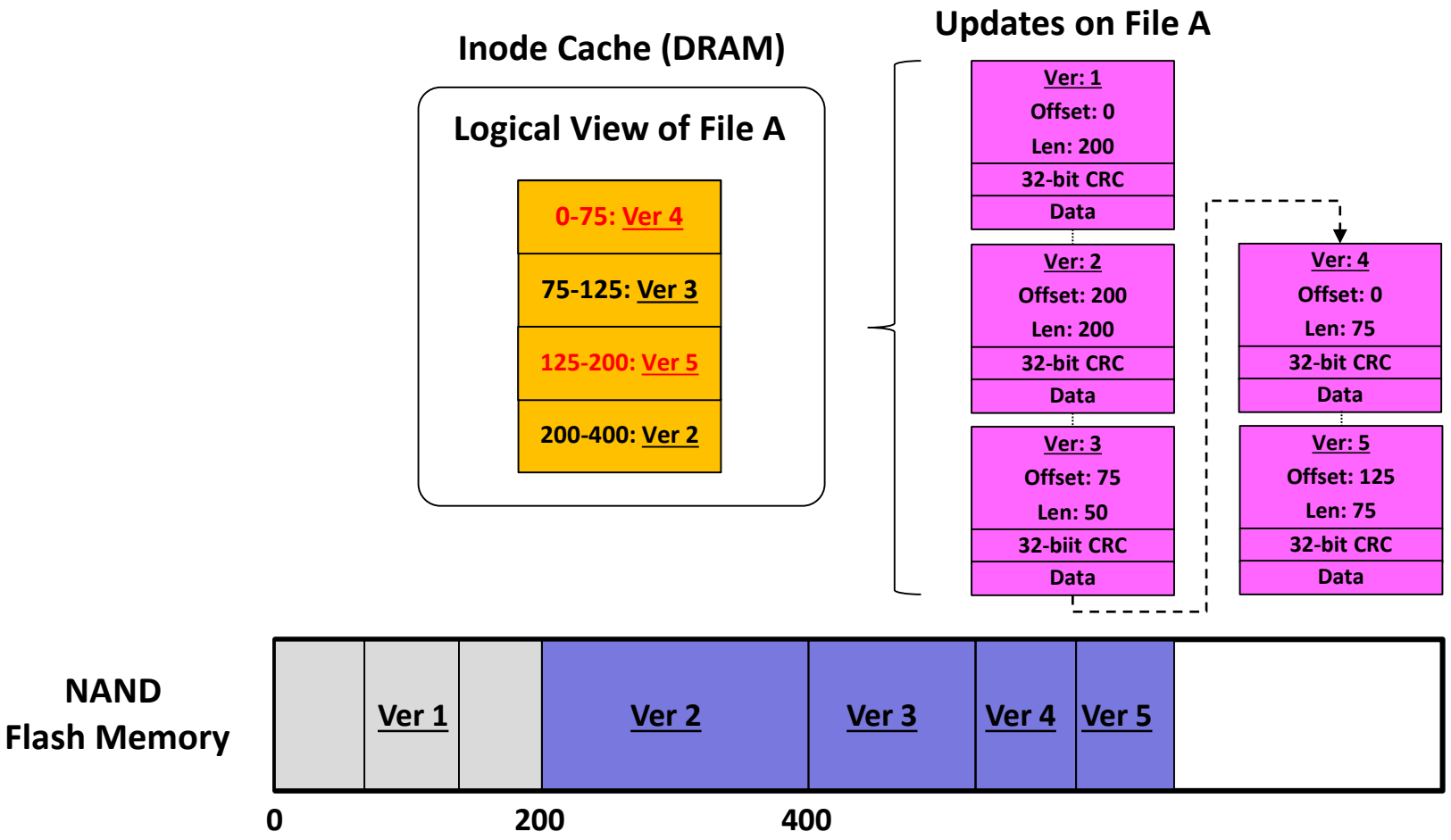
JFFS2: Journaling Flash File System

- **A log-structured file system (LFS) for use with NAND flash**
  - Unlike LFS, however, it *does not allow* any in-place updates!!!

- **Main features of JFFS2**
  - File data and metadata stored as nodes in NAND flash memory
  - Keep an inode cache holding the information of nodes in DRAM
  - A greedy garbage collection algorithm
    - Select cheapest blocks as a victim for garbage collection
  - A simple wear-leveling algorithm combined with GC
    - Consider the wearing rate of flash blocks when choosing a victim block for GC
  - Optional data compression

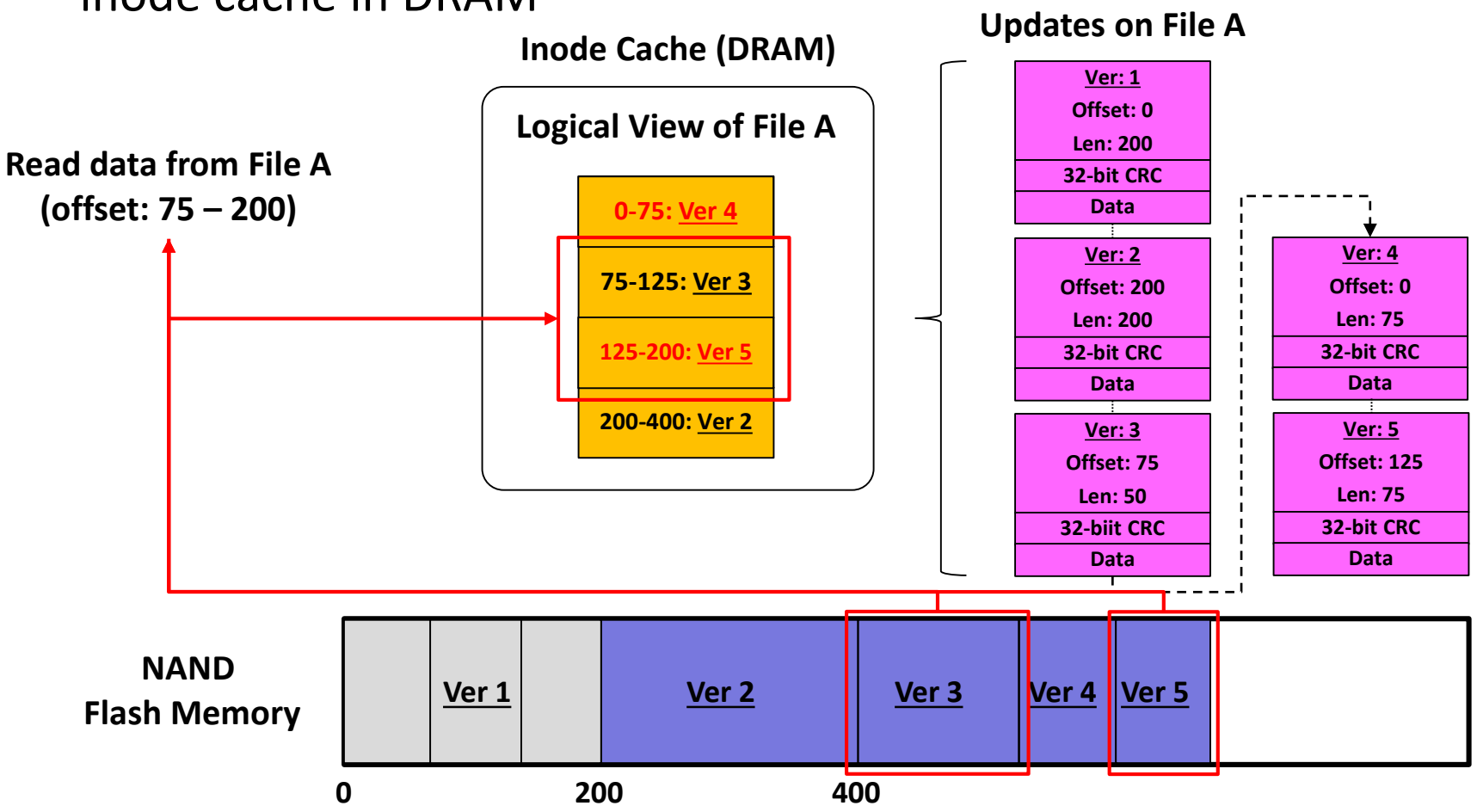
JFFS2: Write Operation

- All data are written sequentially to a log which records all changes



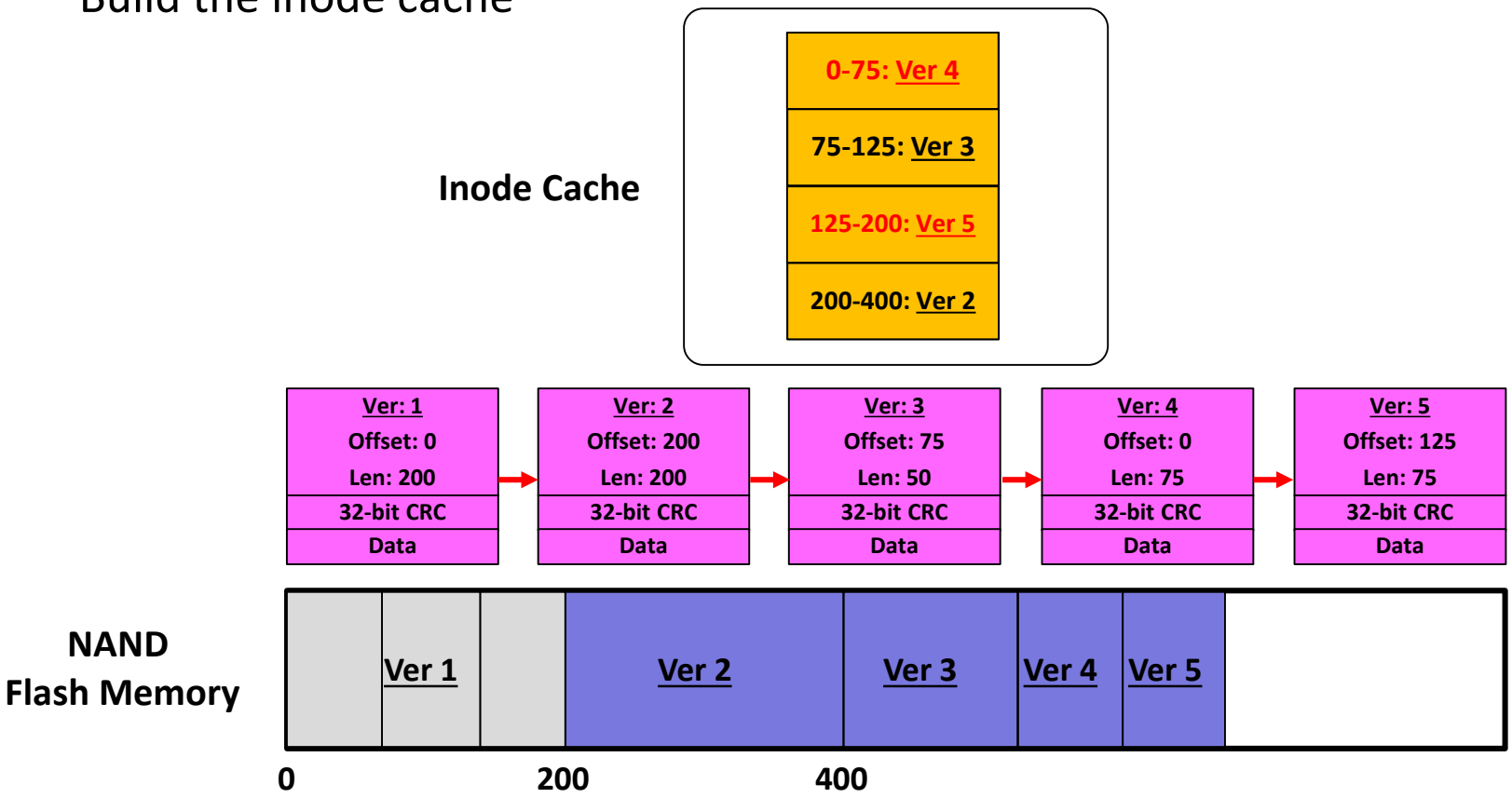
JFFS2: Read Operation

- The latest data can be read from NAND flash by referring to the inode cache in DRAM



JFFS2: Mount

- **Scan the flash memory medium after rebooting**
  - Check the CRC for written data and mark the obsolete data
  - Build the inode cache



JFFS2: Problems

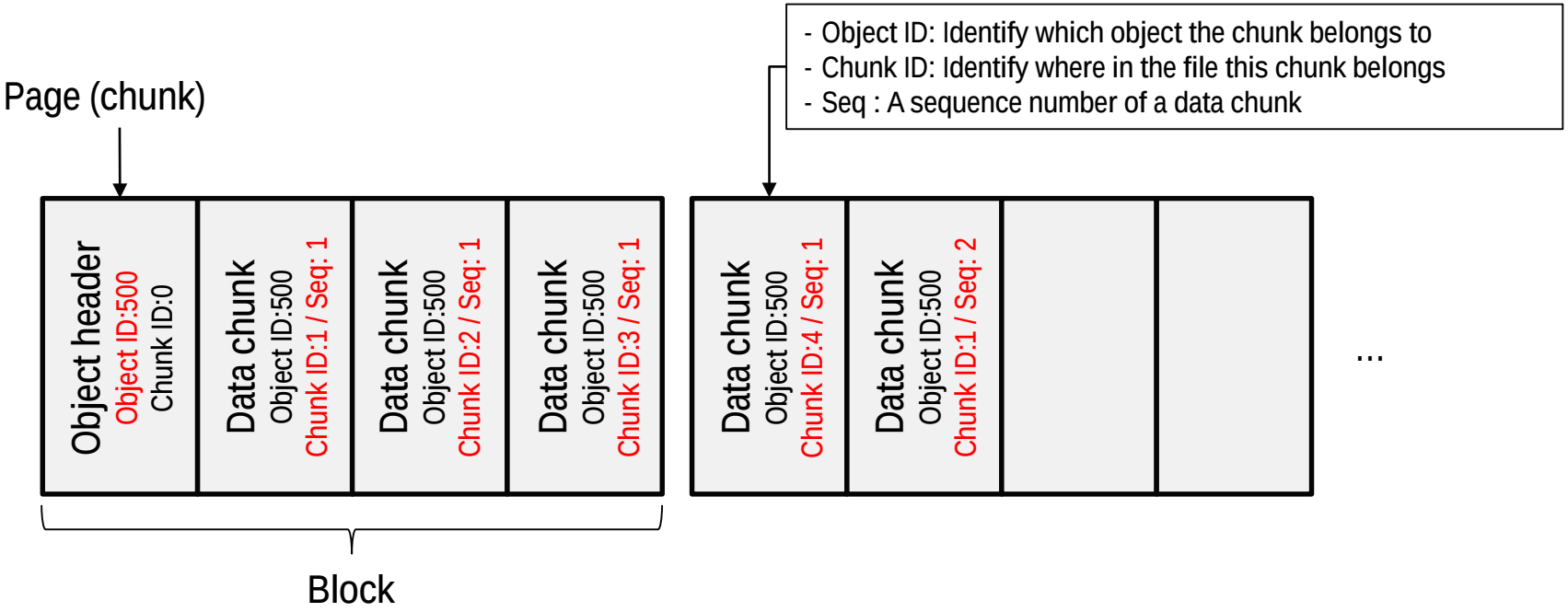
- **Slow mount time**
  - All nodes must be scanned at mount time
  - Mount time increases in proportion to flash size and file system contents
- **High memory consumption**
  - All node information must be maintained in DRAM
  - Memory consumption linearly depends on file system contents
- **Low scalability**
  - Infeasible for a large-scale flash device
  - Mount time and memory consumption increase according to a flash size

YAFFS2: Yet Another Flash File System

- **Another log-structured file system for flash memory**
  - Store data to flash memory like a log with a unique sequence number like JFFS2
  - Reads and writes are performed similar to JFFS2
- **Mitigate the problems raised by JFFS2**
  - Relatively frugal with memory resource
  - Checkpoints for a fast file system mount
  - Dynamic wear-leveling
- **Support multiple platforms**
  - Linux, WinCE, RTOSs, etc

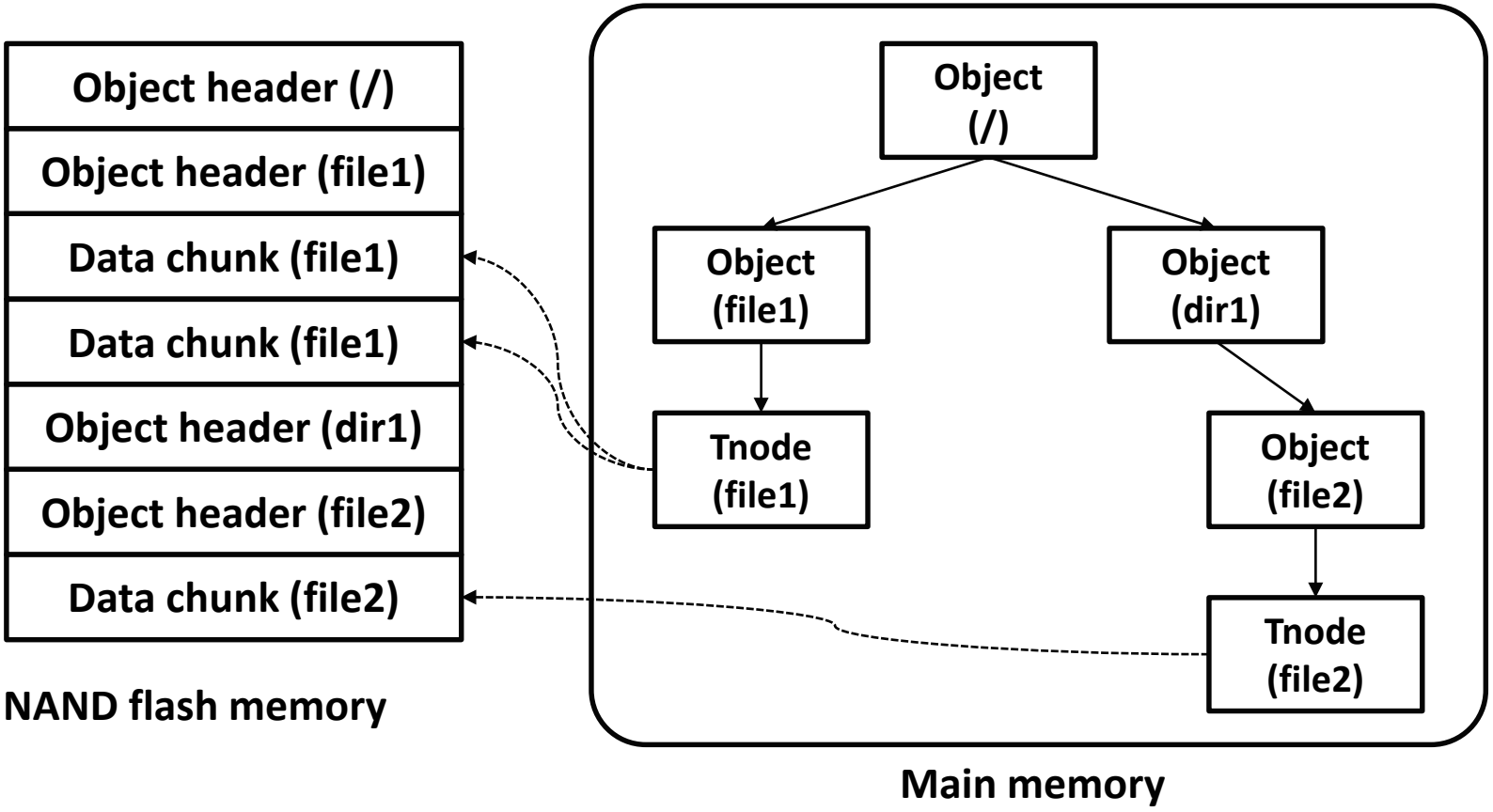
YAFFS2: Physical Layout

- **The entries in the log are all one chunk (one page) in size and can hold one of two types of chunk**
  - Data chunk: A chunk holding regular data file contents
  - Object header: A descriptor for an object, such as a directory, a regular file, etc; similar to `struct stat` but include dentry



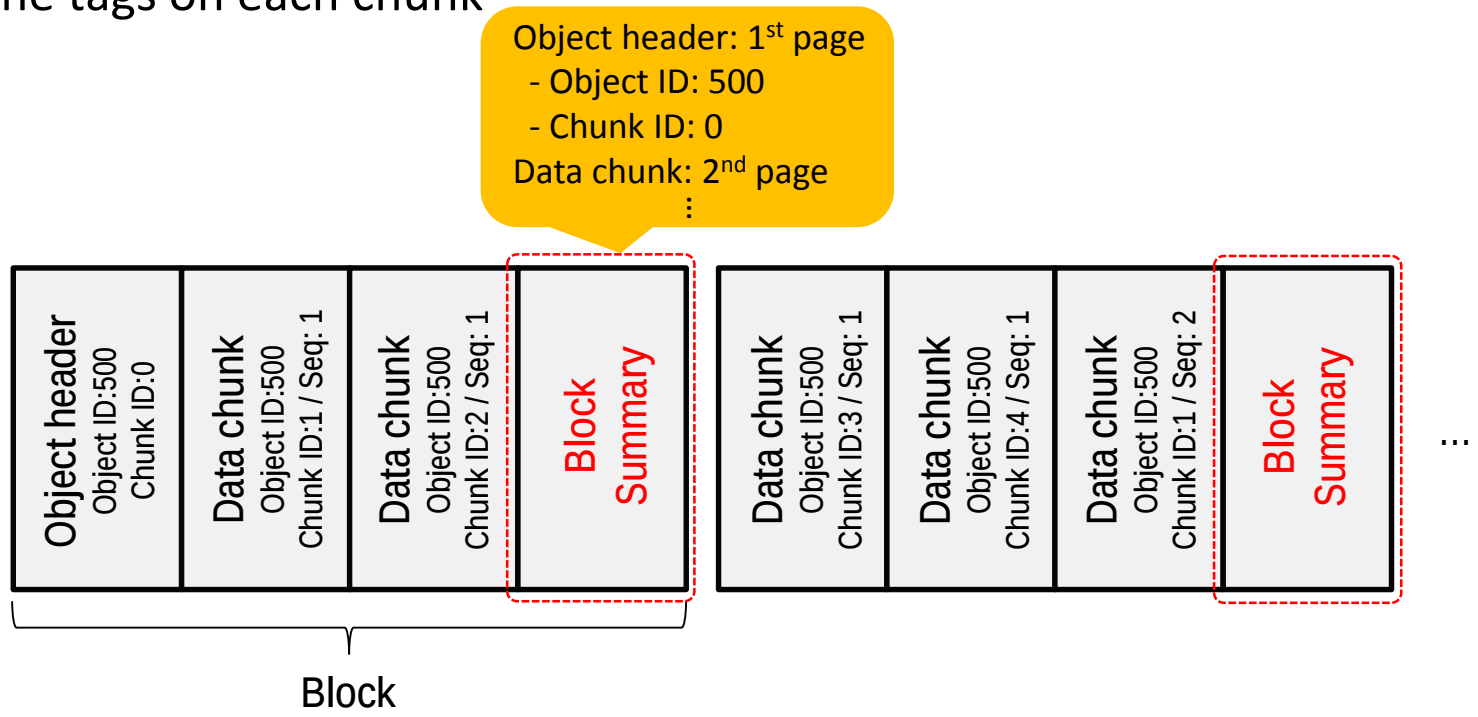
YAFFS2: File System Layout

- Maintain the information about objects and chunks in DRAM



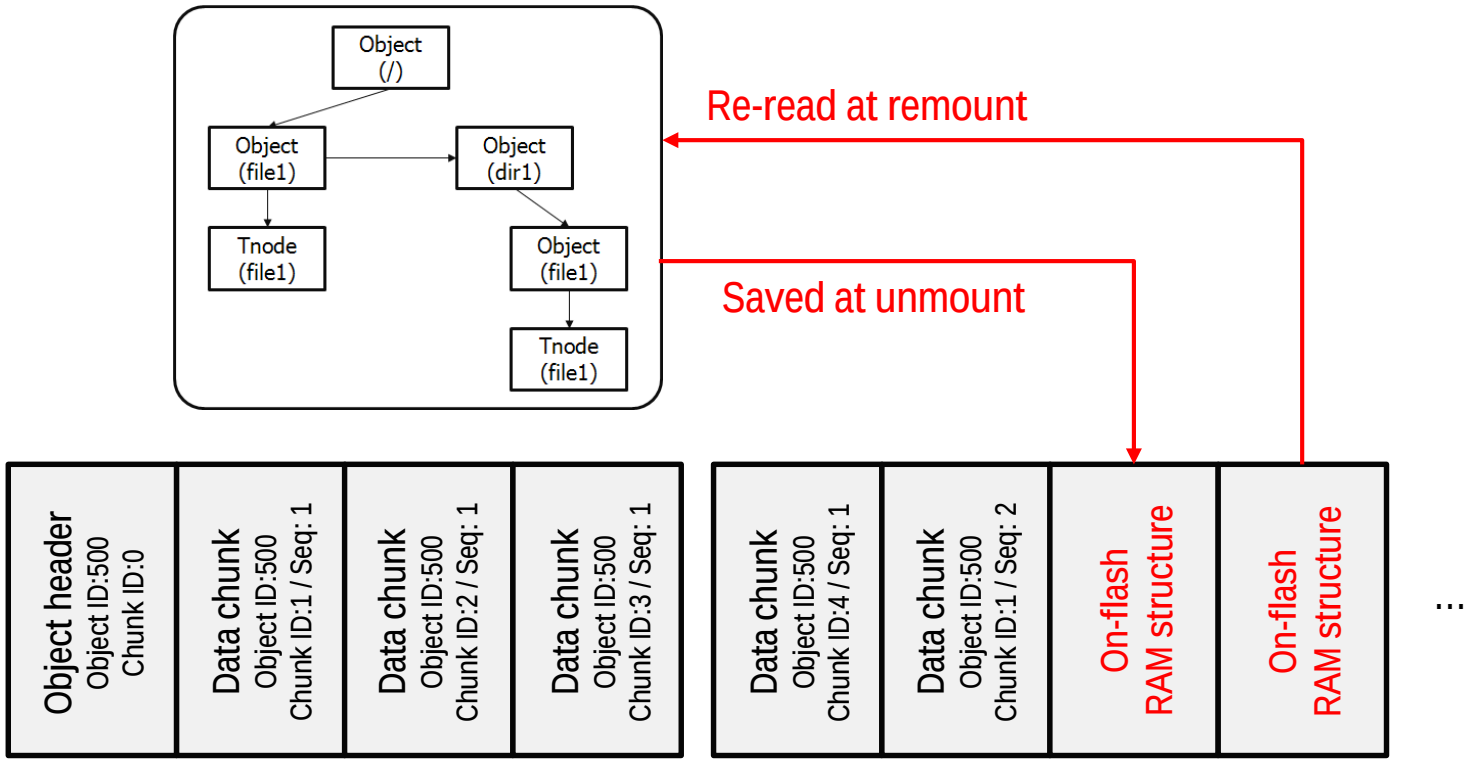
YAFFS2: Block Summary

- A block summary (including object id and chunk id) for chunks in the block are written to the last chunk
- This allows all the tags for chunks in that block to be read in one hit, avoiding full disk scan
- If a block summary is not available for a block, then a full scan is used to get the tags on each chunk



YAFFS2: Checkpoints

- DRAM structures are saved on flash at unmount
- Structures re-read, avoiding boot scan
- Lazy loading also reduces mount time



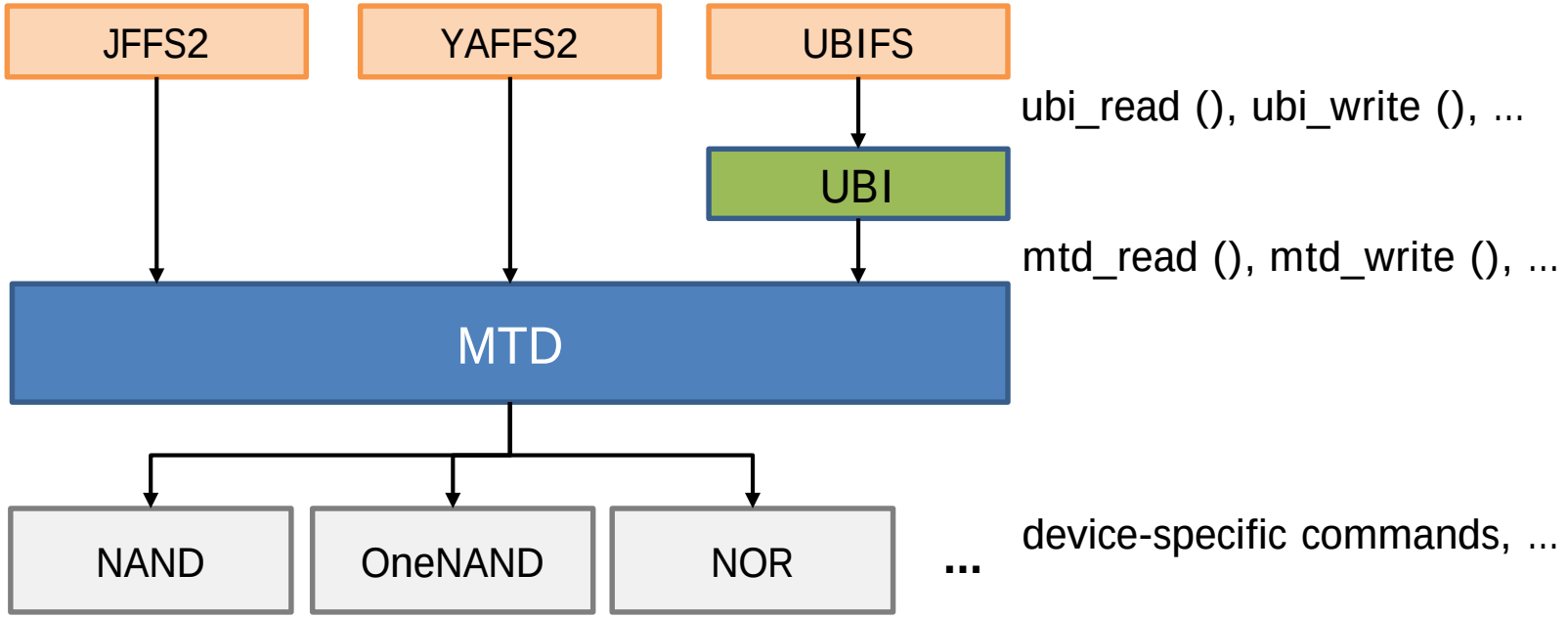
UBIFS: Unsorted Bock Image File System

- **A new flash file system developed by Nokia**
  - Considered as the next generation of JFFS2

- **New features of UBIFS**
  - Scalability
    - Scale well with respect to flash size
    - Memory size and mount time do not depend on flash size
  - Fast mount
    - Do not have to scan the whole media when mounting
  - Write-back support
    - Dramatically improve the throughput of the file system in many workloads
  - …

UBI: Unsorted Block Image

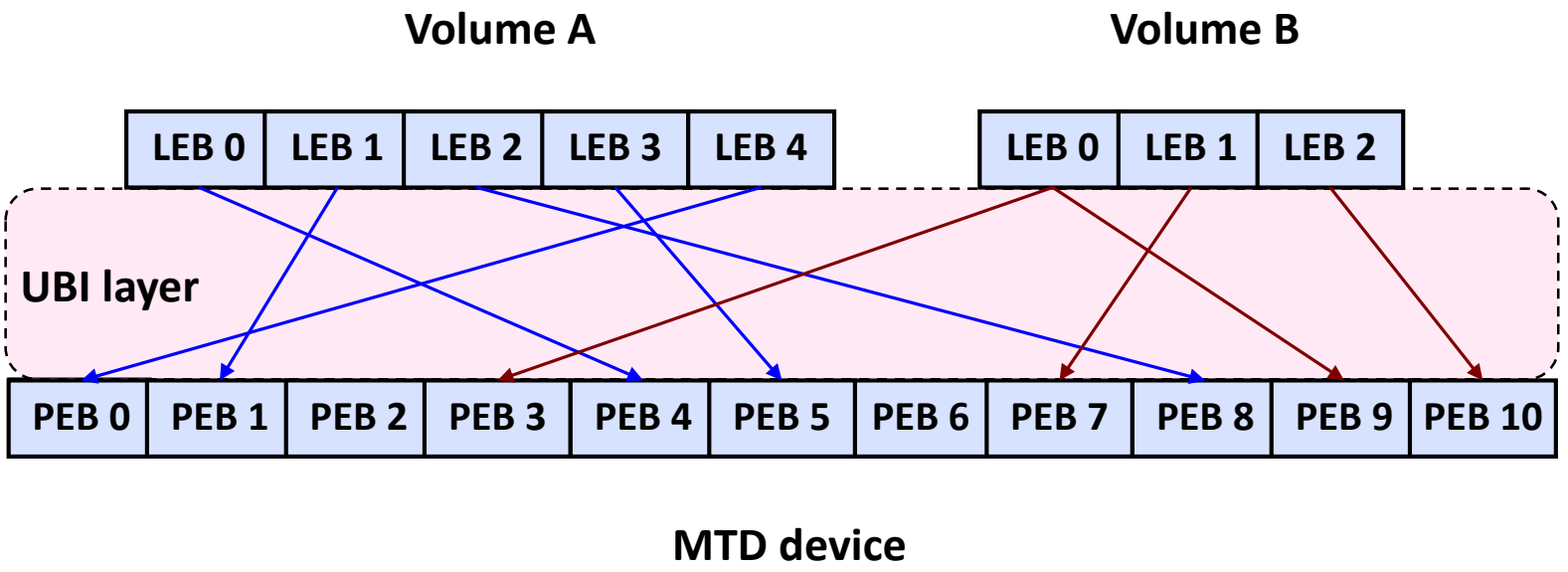
- **UBIFS runs on top of UBI volume**
  - UBI supports multiple volumes, bad block management, wear-leveling, and bit-flips error management
  - The upper level software can be simpler with UBI



How UBI works

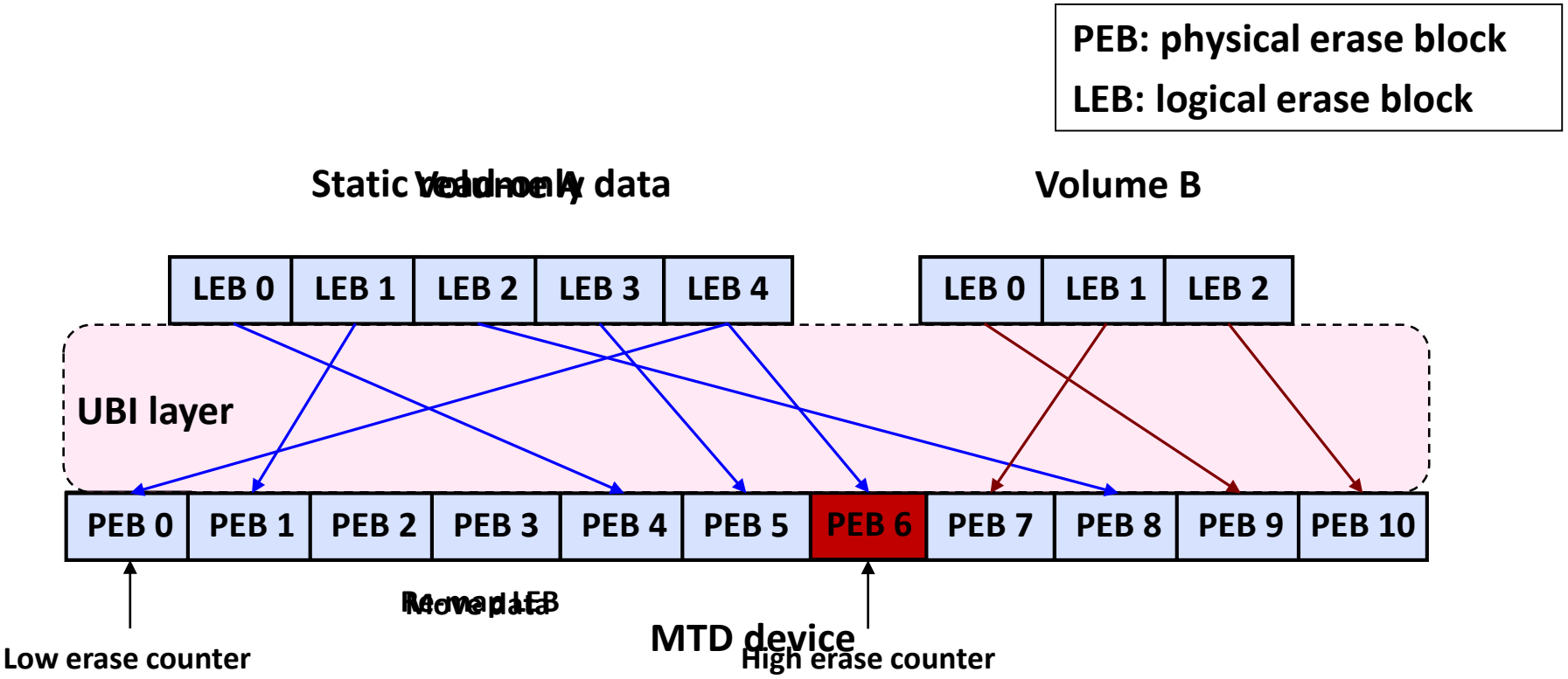
- **Logical erase blocks (LEBs) are mapped to physical erase blocks (PEBs)**
  - Any LEB can be mapped to any PEB

PEB: physical erase block
LEB: logical erase block



How UBI works

- UBI has its own wear-leveling algorithm that moves the data kept in highly erased blocks to lower one

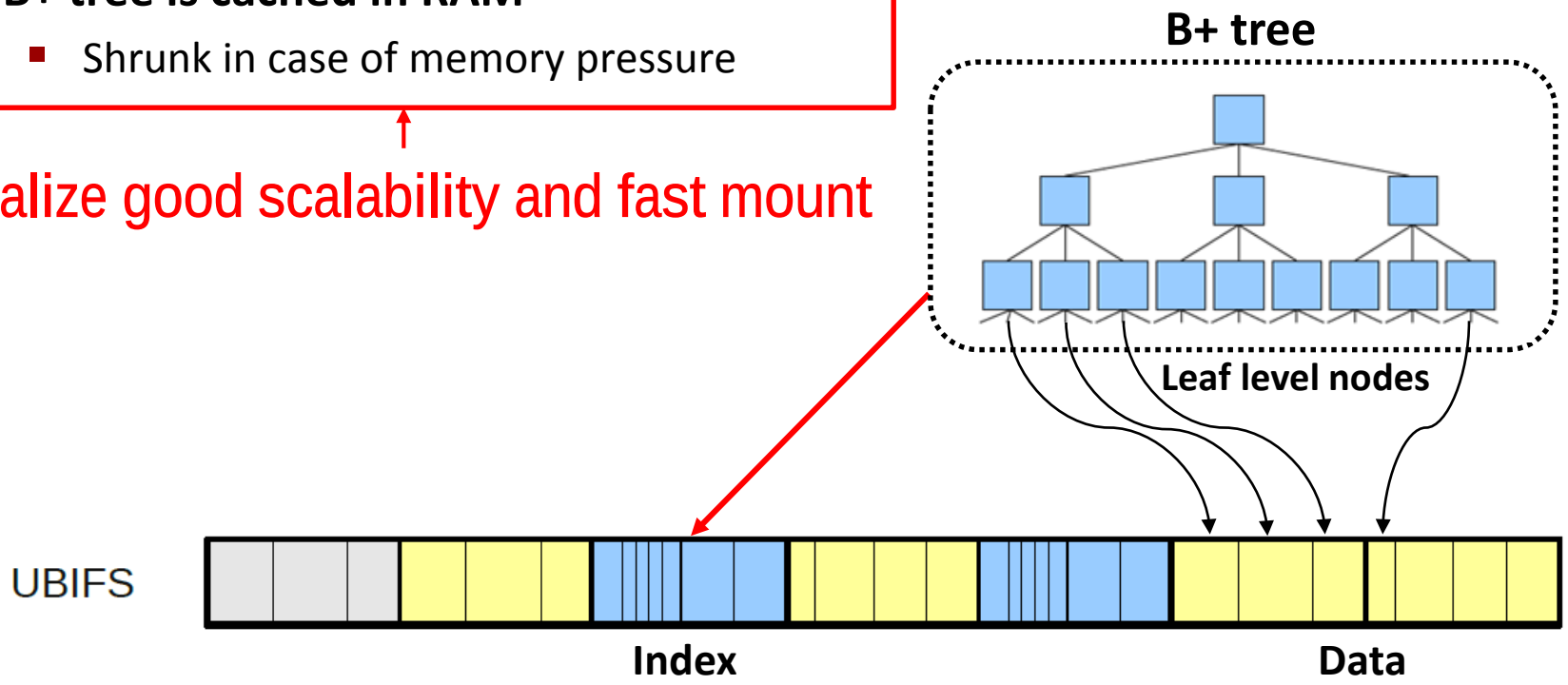


UBIFS: Indexing with B+ Tree

- **UBIFS index is a B+ tree and is stored on NAND flash**
  - c.f., JFFS2 does not store the index on flash
- **Leaf level contains data**

- **Full scanning is not needed**
- **B+ tree is cached in RAM**
  - Shrunk in case of memory pressure

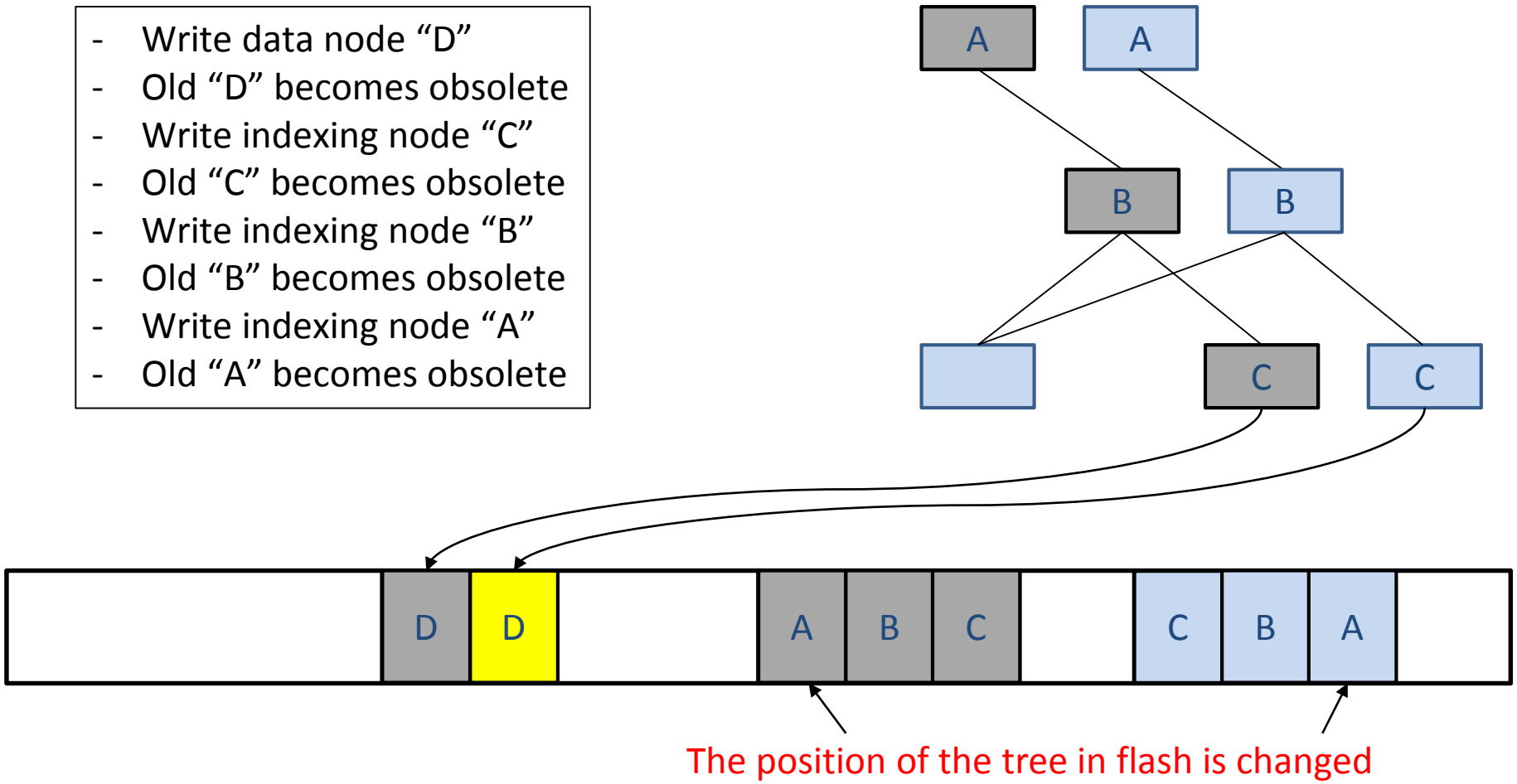
Realize good scalability and fast mount



UBIFS: Wandering Tree

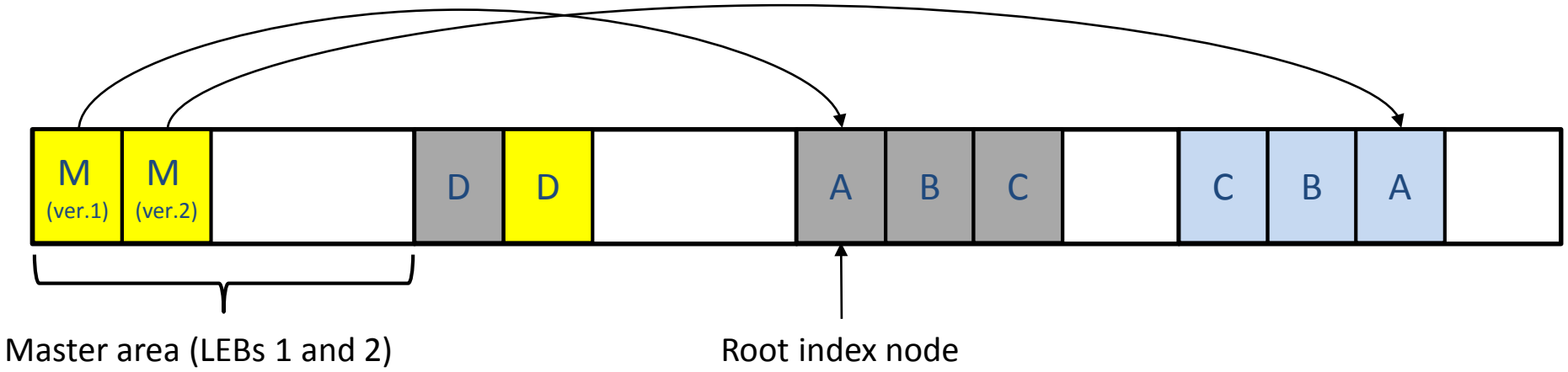
- **How to find the root of the tree?**

- Write data node “D”
- Old “D” becomes obsolete
- Write indexing node “C”
- Old “C” becomes obsolete
- Write indexing node “B”
- Old “B” becomes obsolete
- Write indexing node “A”
- Old “A” becomes obsolete

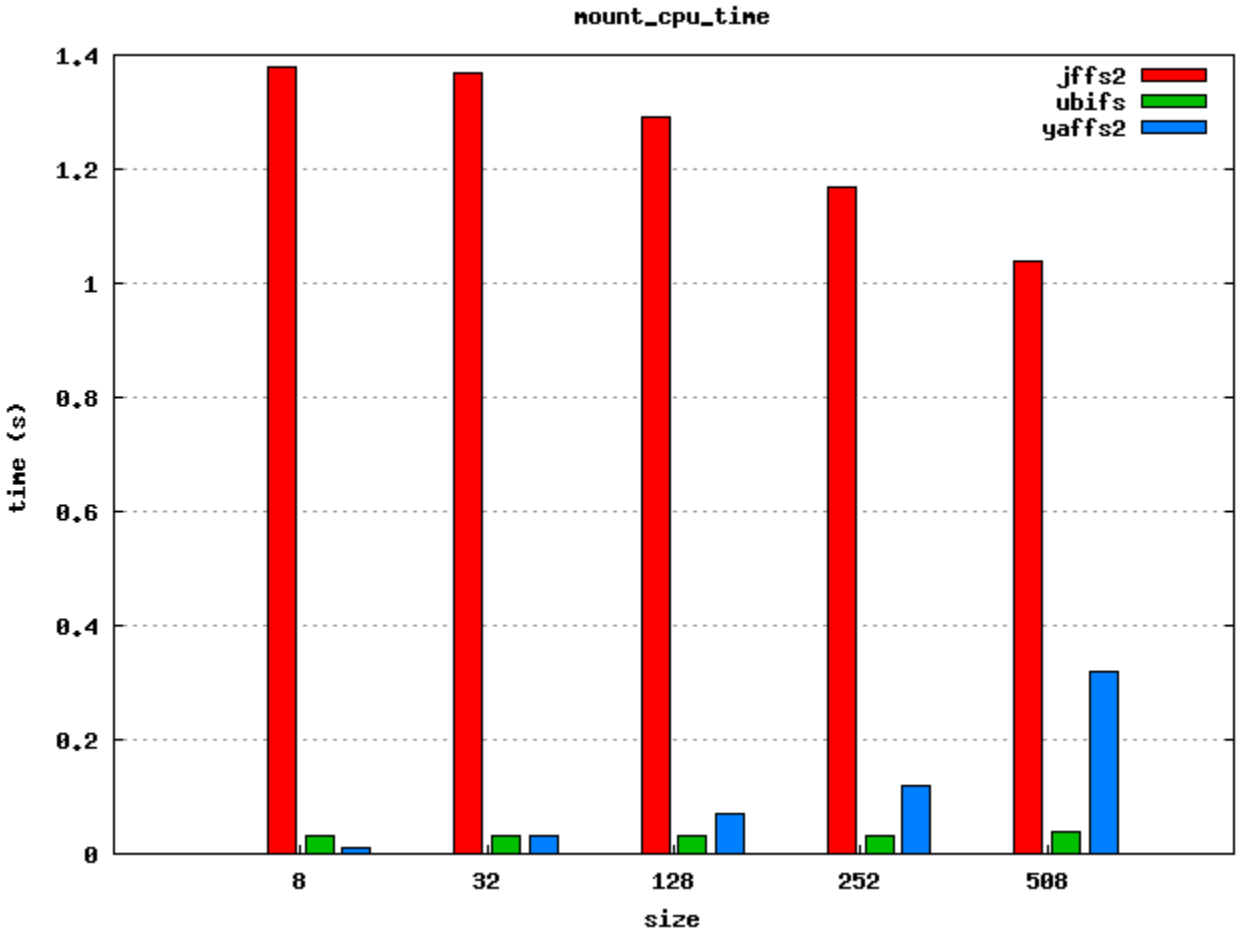


UBIFS: Master Area

- LEBs 1/2 are reserved for *a master area* pointing to *a root index node*
  - The master area can be quickly found on mount since its location is fixed (LEBs 1 and 2)
  - Using the root index node, B+ tree can be quickly constructed
- A mater area could have multiple nodes
  - A valid master node is found by scanning master area



Mount Time Comparison



Summary

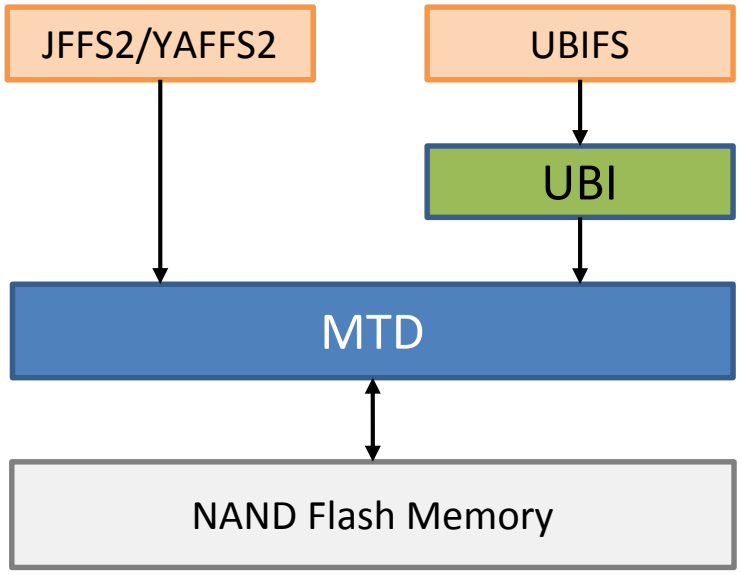
- **JFFS2: Journaling Flash File System version 2**
  - Commonly used for low volume flash devices
  - Compression is supported
  - Long mount time & High memory consumption
- **YAFFS2: Yet Another Flash File System version 2**
  - Fast mount time with check-pointing
- **UBIFS: Unsorted Block Image File System**
  - Fast mount time and low memory consumption by adopting B+ tree indexing

Today

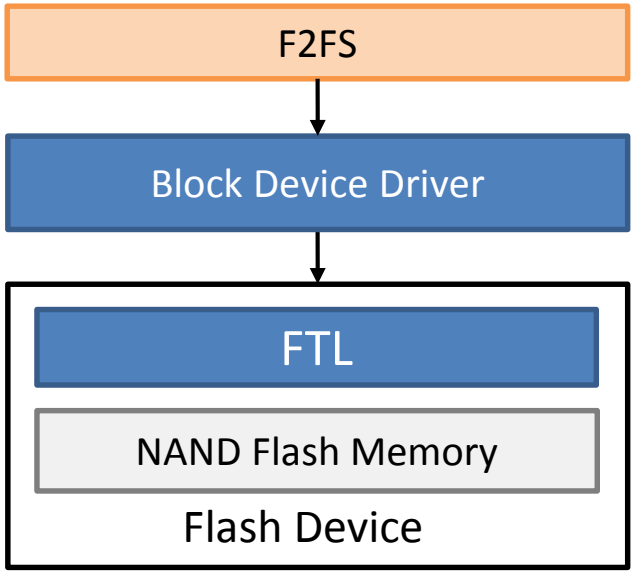
- File System Basics
- Traditional Flash File Systems
- **SSD-Friendly Flash File Systems**
  - F2FS: Flash-friendly File System
- Reference

F2FS: Flash-friendly File System

- **Log-structured file system for FTL devices**
  - Unlike other flash file systems, it runs atop FTL-based flash storage and is optimized for it
  - Exploit system-level information for better performance and reliability (e.g., better hot-cold separation, background GC, …)



Traditional flash file system



F2FS

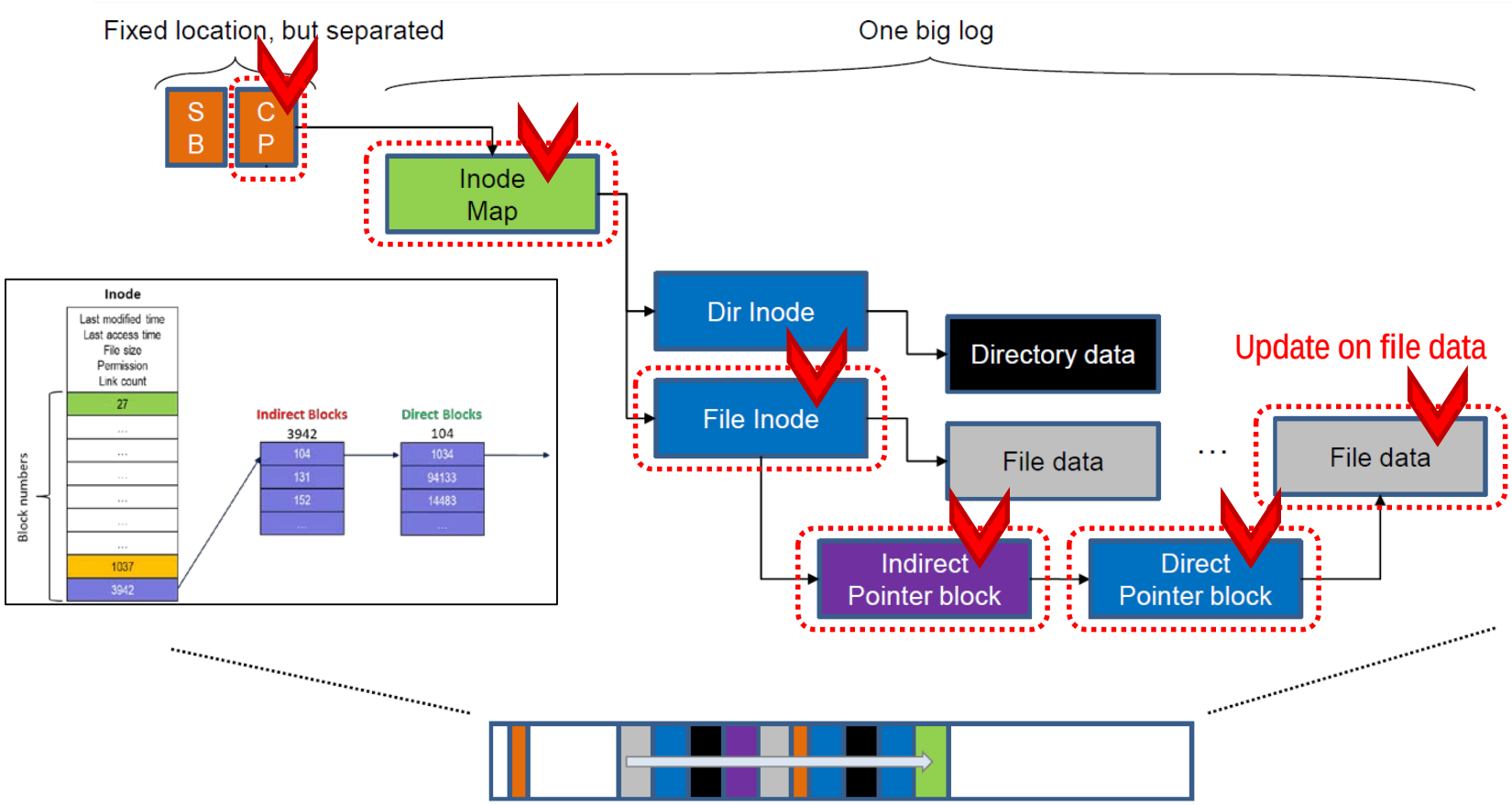
Design Concept of F2FS

- **Designed to take advantage of both of two approaches**
  - Exploit high-level system information
  - Flash-aware storage management
  - Handle new NAND flash without redesign

	File System + FTL	Flash File System
Method	- Access a flash device via FTL	- Access a flash device directly
Pros	- High interoperability - No difficulties in managing recent NAND flash with new constraints	- High-level optimization with system-level information - Flash-aware storage management
Cons	- Lack of system-level information - Flash-unaware storage management	- Low interoperability - Must be redesigned to handle new constraints

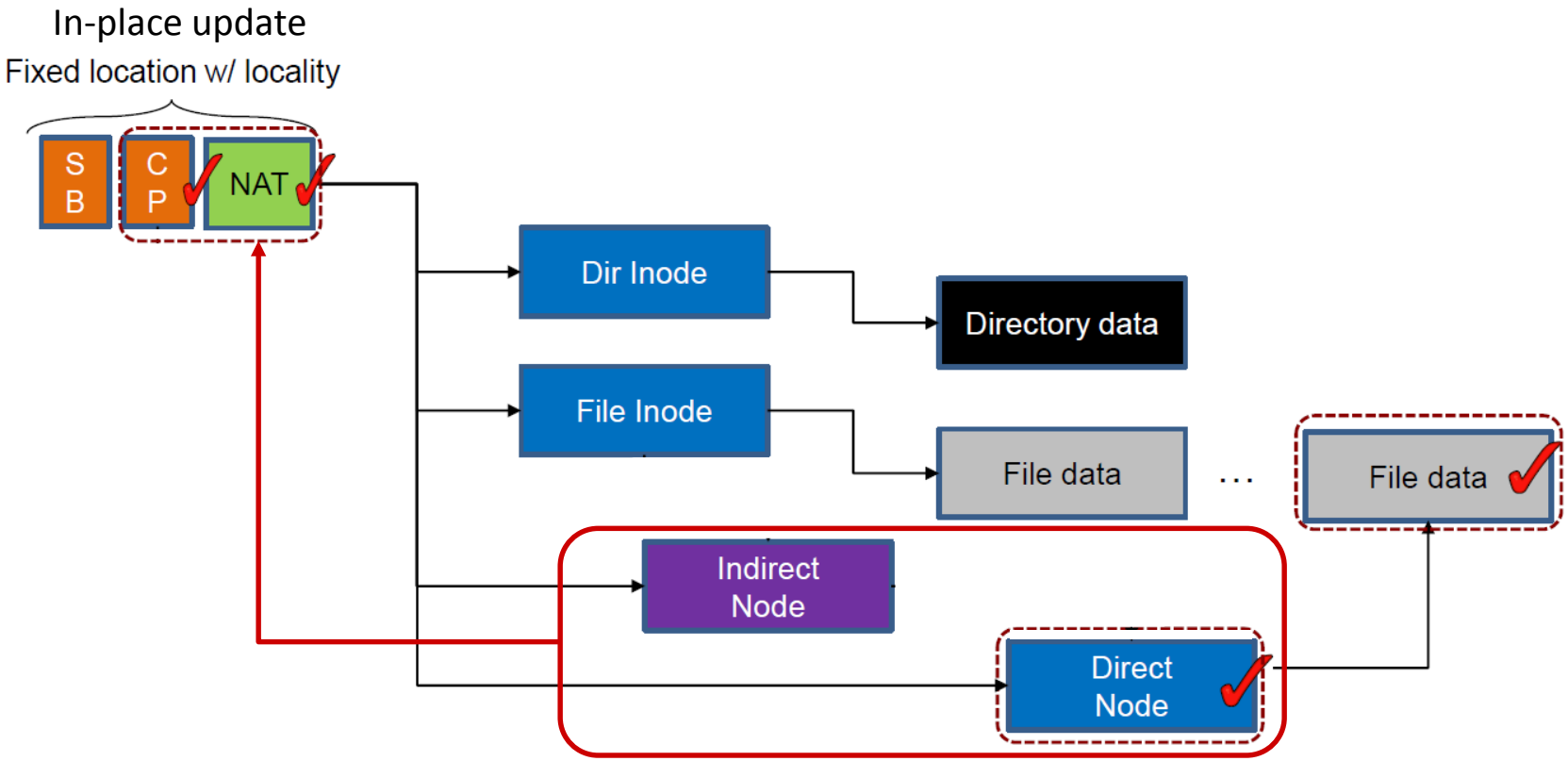
Index Structure of LFS

- **LFS and UBIFS suffer from the wandering tree problem**
  - Update on a file causes several extra writes



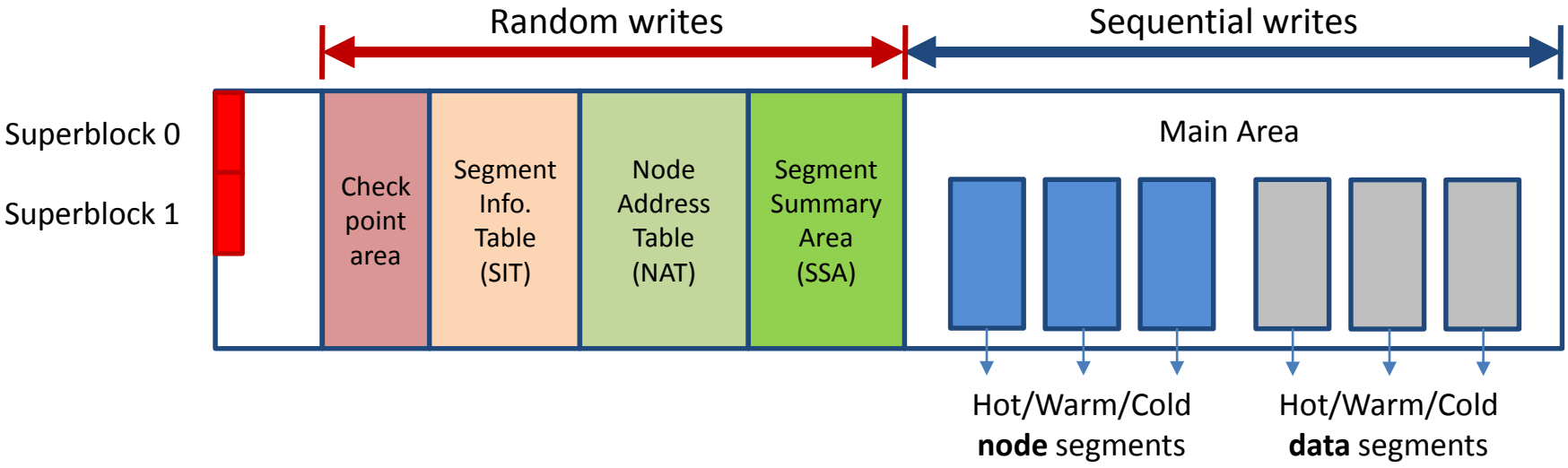
Index Structure of F2FS

- Introduce Node Address Table (NAT) containing the locations of all the node blocks, including indirect and direct nodes
- NAT allows us to eliminate the wandering tree problem



Logical Storage Layout

- **File-system's metadata is located together for locality**
  - Use an "in-place update" strategy for metadata
- **Files are written sequentially for performance**
  - Use an "out-of-place-update" strategy to exploit high throughput of multiple NAND chips
  - Six active logs for static hot and cold data separation



Cleaning Process

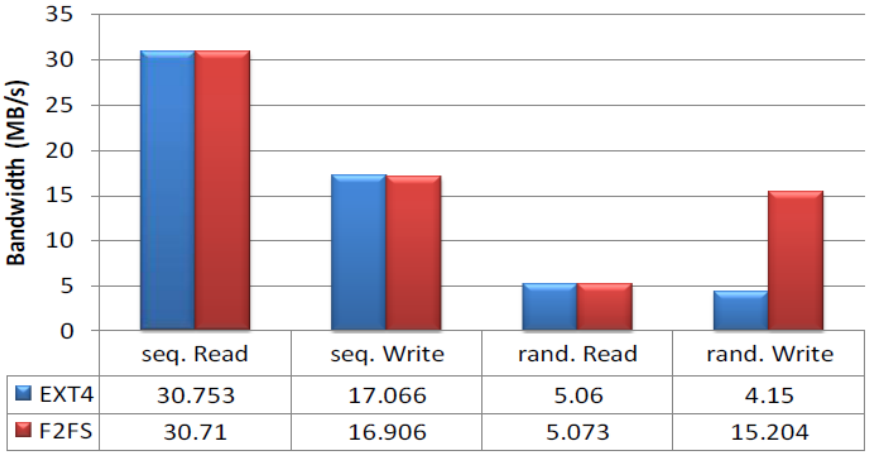
- **Background cleaning process**
  - A kernel thread doing the cleaning job periodically at idle time

- **Victim selection policies**
  - Greedy algorithm for foreground cleaning job
    - Reduce the amount of data moved for cleaning
  - Cost-benefit algorithm for background cleaning job
    - Reclaim obsolete space in a file system
      - Improve the lifetime of a storage device
      - Improve the overall I/O performance

F2FS Performance

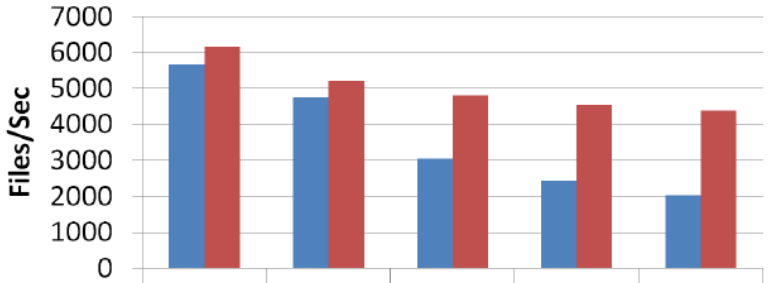
[System Specification]

CPU	ARM Coretex-A9 1.2GHz
DRAM	1GB
Storage	Samsung eMMC 64GB
Kernel	***Linux 3.3***
Partition Size	***12 GB***



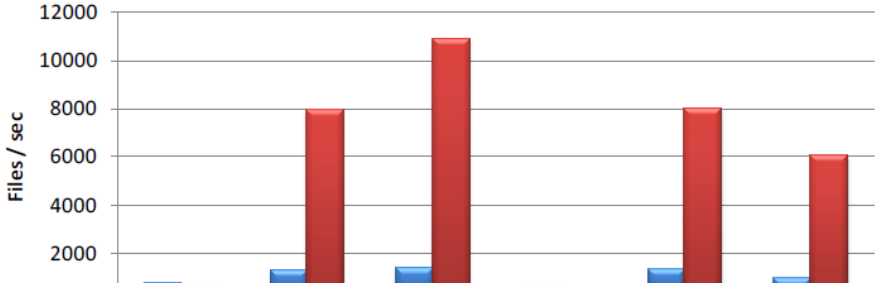
	seq. Read	seq. Write	rand. Read	rand. Write
EXT4	30.753	17.066	5.06	4.15
F2FS	30.71	16.906	5.073	15.204

[iozone]



	5000	10000	15000	20000	25000
ext4	5675.9	4741.6	3065	2439.5	2038.6
f2fs	6167.3	5207.9	4815.9	4544.4	4382.9

[fs_mark]



	seq.create	seq.stat	seq.delete	rand.create	rand.stat	rand.delete
EXT4	692	1238	1370	663	1250	915
F2FS	631	7871	10832	620	7962	5992

[bonnie++]

Today

- File System Basics
- Traditional Flash File Systems
- SSD-Friendly Flash File Systems
- **Reference**

Reference

- Rosenblum, Mendel, and John K. Ousterhout. "The design and implementation of a log-structured file system." *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992): 26-52.
- Woodhouse, David. "JFFS2: The Journalling Flash File System, Version 2, 2001.“
- YAFFS2 Specification. http://www.yaffs.net/yaffs-2-specification
- Lee, Changman, et al. "F2FS: A New File System for Flash Storage.“ *USENIX FAST*. 2015.
- Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. *Operating systems: Three easy pieces*. Vol. 151. Wisconsin: Arpaci-Dusseau Books, 2014.
- Hunter, Adrian. "A brief introduction to the design of UBIFS." *Rapport technique, March* (2008).