Lecture Note 7. Advanced File System

May 6, 2026

Jongmoo Choi

Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

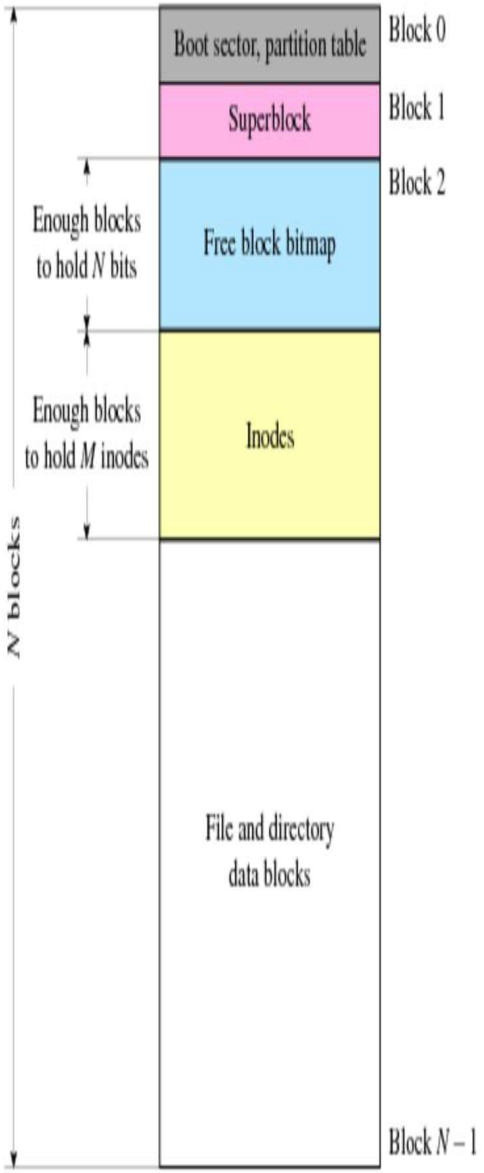
(본 교재는 2026년도 과학기술정보통신부 및 정보통신기획평가원의 'SW중심대학사업' 지원을 받아 제작 되었습니다.)

Contents

- From Chap 41~45 of the OSTEP
- Chap 41. Locality and the Fast File System
  - ✓ Performance requirement and Storage-aware performance enhancement
- Chap 42. Crash Consistency: FSCK and Journaling
  - ✓ Consistency requirement and Journaling mechanism
- Features of popularly used FS: Ext2/3/4, FAT, …
- Chap 43. Log-structured File Systems
  - ✓ Out-of-place updates and Details matter
- Chap 44. Flash-based SSDs
  - ✓ Flash Characteristics and FTL
- Chap 45. Data Integrity and Protection
- Summary and Lab3

Chap. 41 Locality and The Fast File System

- UFS (Unix File System)
  - ✓ Layout
    - Boot sector
    - Superblock
      - how big FS is, how many inodes, where is inode, …
    - Bitmap + Inode + User data
    - ➔ Simple and easy-to-use
  - ✓ Access method
    - Inode & data are accessed alternately
      - Look not bad, but consider disk geometry (see chapter 37) and multiple I/Os per a write (see chapter 40)
    - Concerns: 1) Long seek time, 2) Consistency
      - Performance issue ➔ This chapter
      - Consistency issue ➔ Next chapter

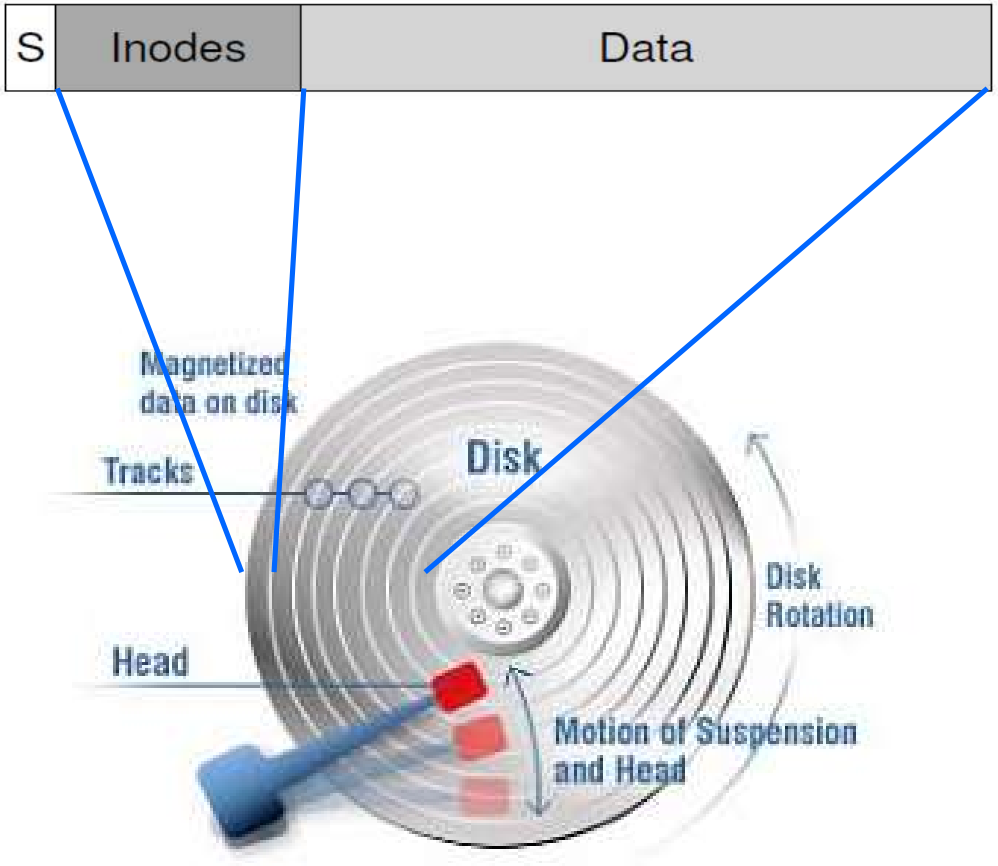


41.1 Poor Performance

- UFS (also our VSFS)
  - ✓ poor performance
  - ✓ 1) Inode and User data are located in different tracks 2) A file is fragmented as time goes (external fragmentation) ➔ long seek

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read			read				
				read						
							read			
					read					
read()					read					
					write					
read()					read					
					write					
read()					read					
					write					

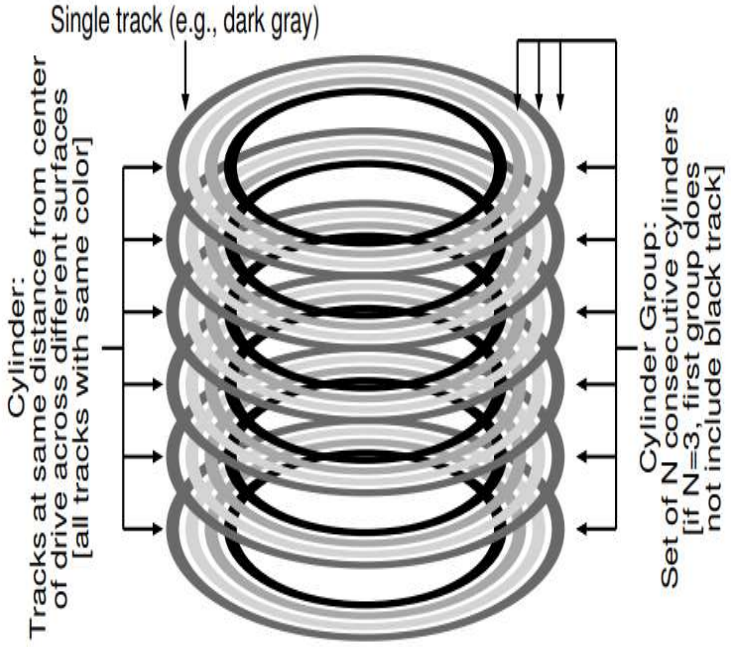
Figure 40.3: File Read Timeline (Time Increasing Downward)

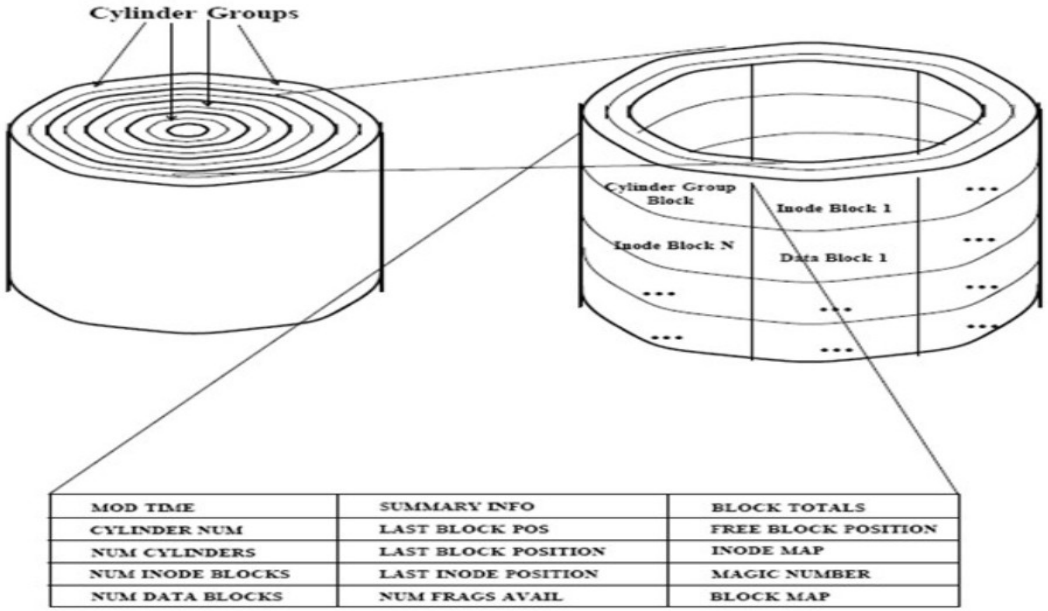


☞ **How to overcome this problem?**

41.2 FFS: Disk Awareness

- New proposal: FFS (Fast File System from BSD OS)
  - ✓ Place inodes and user data blocks as close as possible
  - ✓ Disk-awareness
    - Data in the same cylinder ➔ no seek distance (or closer cylinder ➔ less seek distance)
      - Cylinder group is defined as a set of tracks on different surfaces that are the same distance from the center
  - ✓ This idea is also used in Ext2/3/4 File system

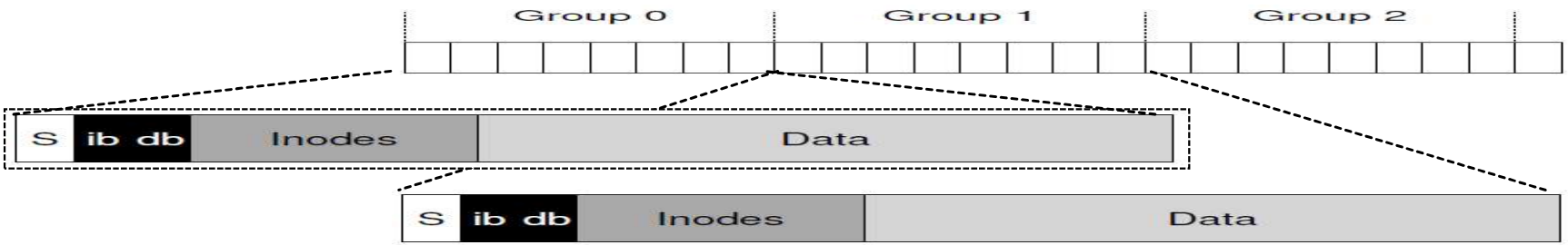




(Source: https://slideplayer.com/slide/8117044/)

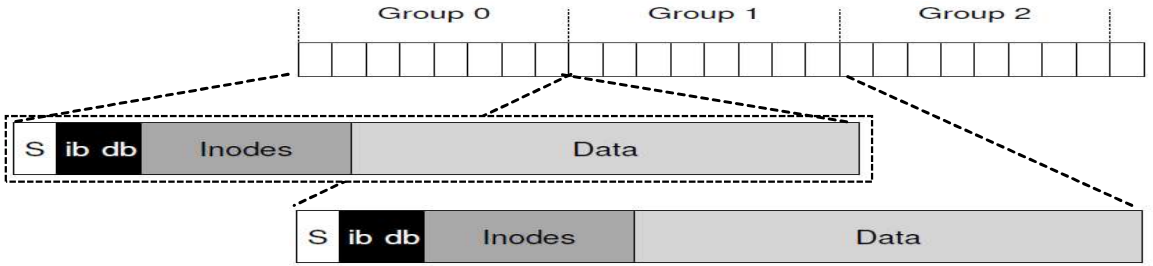
41.3 Organizing Structure: The Cylinder Group

- FFS in detail
  - ✓ Partition(or a disk): divided into a number of cylinder groups
  - ✓ Cylinder group
    - N consecutive cylinders
    - Structure of each cylinder group
      - Superblock (duplication for reliability)
      - Per-group bitmap, inode and data blocks
    - Management
      - Allocate an inode and data at the same group: e.g. Inode and data blocks for file A in Group 0, those for file B in group 1, … ➔ Small seek distance
      - Ext2: similar approach called block group
  - ✓ Feature of FFS: Different internal implementation, but same external interfaces



41.4 Policies: How to Allocate Files and Directories

- Allocation in FFS
  - ✓ Idea: keep related stuff together
    - Data and related inode, file and its related directory, …
  - ✓ Allocation issue
    - E.g.) Create a file A. which group does it allocate?
    - E.g.) Create a directory B. which group does it allocate?



  - ✓ Allocation rules
    - Rule 1. Directory: place it into a cylinder group with a high number of free inodes (a low number of allocated directories)
      - To balance directories across groups
    - Rule 2. File: 1) put files in the cylinder group of the directory they are in, 2) allocate data blocks of a file in the same group as its inode
      - To allocate inode, data blocks and directory as close as possible

41.4 Policies: How to Allocate Files and Directories

- Allocation in FFS
  - ✓ Allocation rules
    - E.g.) create three directories (/, /a, /b) and four files (/a/c /a/d, /a/e, /b/f)
      - Assumption: 1) Directory: 1 block, 2) file: 2 blocks
    - FFS allocates three directories at different group (rule 1, load balancing), allocate files in the same directory (rule 2, namespace locality)

```
group inodes     data
    0 /--------- /---------
    1 a--------- a---------
    2 b--------- b---------
    3 c--------- cc--------
    4 d--------- dd--------
    5 e--------- ee--------
    6 f--------- ff--------
    7 ---------- ----------
    ...
```

(Even allocation)

```
group inodes     data
    0 /--------- /---------
    1 acde------ accddee---
    2 bf-------- bff-------
    3 ---------- ----------
    4 ---------- ----------
    5 ---------- ----------
    6 ---------- ----------
    7 ---------- ----------
    ...
```

(FFS allocation)

  - ✓ Analysis
    - "ls –l" in the "a" directory
      - Within one group in FFS allocation vs Access 4 groups in even allocation
    - User usage pattern: strong namespace locality

41.6 The Large-File Exception

- How to handle a large file for allocation in FFS?
  - ✓ Large file ➔ fill up a cylinder group with its own data ➔ undesirable with the consideration of the namespace locality
  - ✓ Rule 3. For a large file
    - Allocate a limited number of blocks (called as chunks) in a group. Then, go to another group and allocate a limited number of blocks there. Then, move another one. …
    - Pros) locality among files, Cons) locality in a file
  - ✓ E.g.): 1) file A: 30 blocks, 2) limited number of blocks in a group: 5

```
group inodes     data
    0 /a-------- /aaaaa---- ---------- ---------- ----------
    1 ---------- aaaaa----- ---------- ---------- ----------
    2 ---------- aaaaa----- ---------- ---------- ----------
    3 ---------- aaaaa----- ---------- ---------- ----------
    4 ---------- aaaaa----- ---------- ---------- ----------
    5 ---------- aaaaa----- ---------- ---------- ----------
    6 ---------- ---------- ---------- ---------- ----------
    ...
```

(FFS allocation)

```
group inodes     data
    0 /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
    1 ---------- ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ---------- ----------
    ...
```

(Without Rule 3)

41.6 The Large-File Exception

- How to handle a large file for allocation in FFS?
  - ✓ Analysis of Rule 3
    - How much is the seek overhead for accessing a large file?
      - Seek and Transfer alternatively due to the Rule 3 in FFS
  - ✓ Example
    - Assumption: Seek=10ms, Bandwidth = 40MB/s
    - Example 1) limited number of blocks (chunks) in a group = 4MB
      - Transfer time: 4MB / (40MB/s) = 100ms vs. seek time = 10ms ➔ 90%(100 / 110) bandwidth is used for data transfer
    - Example 2) limited number of blocks (chunks) in a group = 400KB
      - Transfer time: 0.4MB / (40MB/s) = 10ms vs. seek time = 10ms ➔ 50%(10 / 50) bandwidth is used for data transfer
    - ➔ Large chunks can amortize the seek overhead

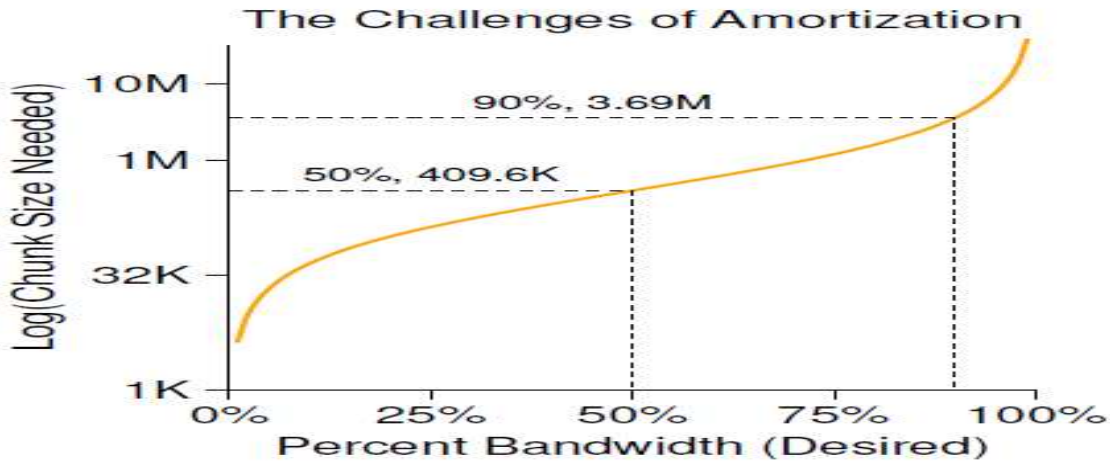


Figure 41.2: Amortization: How Big Do Chunks Have To Be?

41.7 A Few Other Things about FFS

- Another features in FFS
  - ✓ Larger disk block size: 512B (sector) in UFS ➔ 4KB (disk block) in FFS
    - Pros) Larger size ➔ Less seek and more transfer ➔ Higher Bandwidth usage in disk
    - Cons) Internal fragmentation
      - Waste space (e.g. half when a file is 2KB)

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

  - ✓ Sub-blocks (fragment) allocation
    - To overcome the internal fragmentation
  - ✓ Parameterization
    - Sequential block requests: 1, 2, 3, …., (request 1, transfer, request 2, transfer, …) ➔ But when the request 2 is arrived in disk, the head has already passed the location of 2 ➔ solution: parameterized placement
    - c.f.) Modern disk: use track buffer

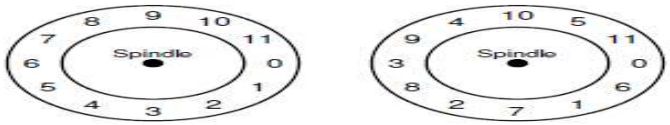
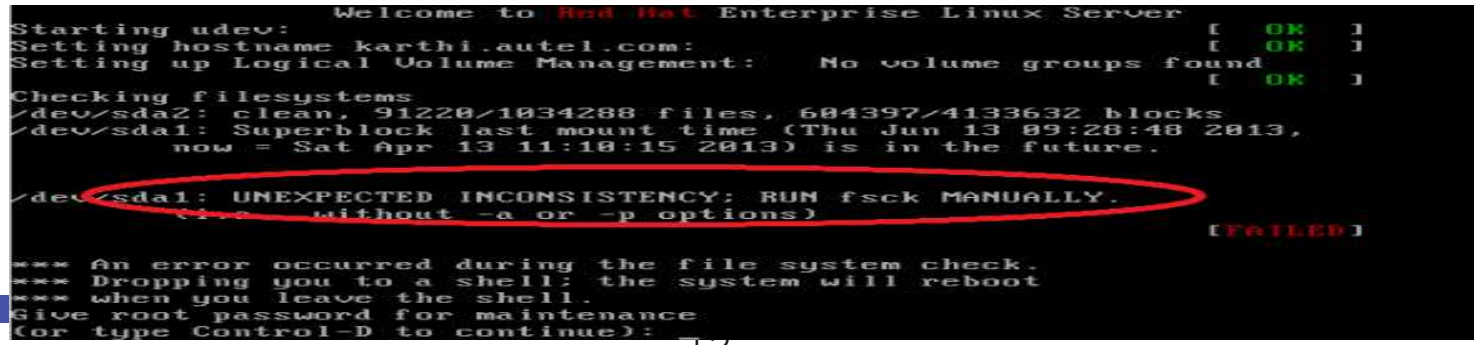


Figure 41.3: FFS: Standard Versus Parameterized Placement

  - ✓ Others: Symbolic link (link across multiple file systems), atomic rename(), long file name, …

Chap. 42 Crash Consistency: FSCK and Journaling

- Non-volatility: no-free lunch
  - ✓ Pros: retain data while power-off
  - ✓ Cons: requires maintaining file system consistency
- Consistency definition
  - ✓ Changes in a file system are guaranteed from a valid state to another valid state
    - E.g.) inconsistent state: bitmap says that a block is free even though it is used by a file
  - ✓ What happen if, right in the middle of creating a file, a system loses power? ➔ have a probability to be an inconsistent state
- Solutions
  - ✓ FSCK (File System Check)
  - ✓ Journaling: employed many file systems such as Ext3/4, JFS, …
  - ✓ Others: Soft update, COW, Integrity checking, Optimistic, …



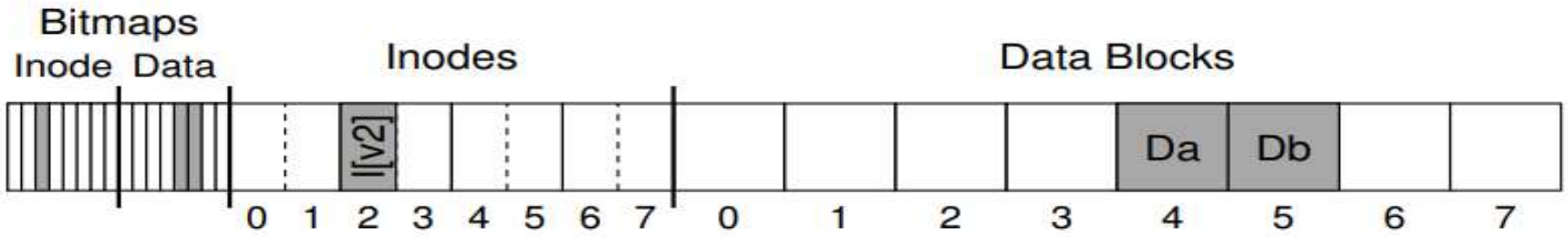
42.1 A Detailed Example

- Example
  - ✓ Simple FS: 8 inodes, 8 disk blocks, i-bitmap, d-bitmap
  - ✓ One file: size=4KB, owner =Remzi


```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

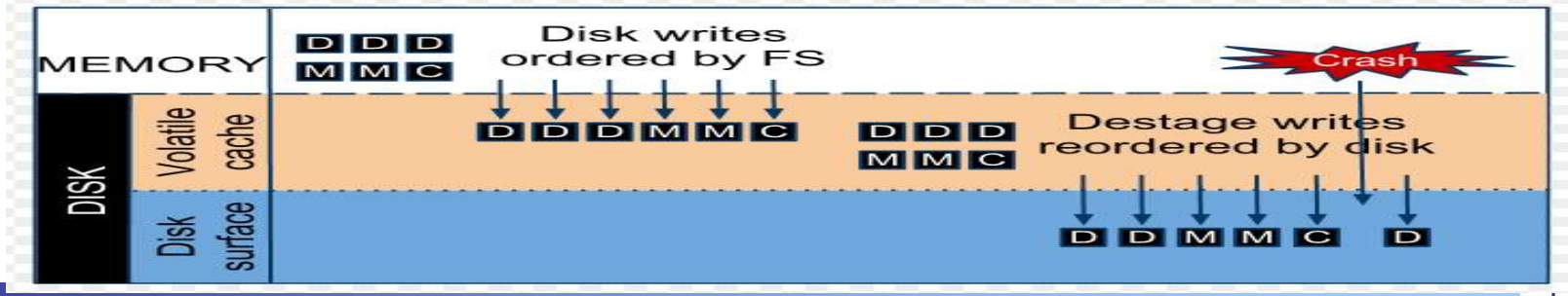
```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

  - ✓ Modify the file: appending, size=8KB
    - Note that we need to change three locations ➔ need three writes



42.1 A Detailed Example

- Crash scenario
  - ✓ Looks one write, but indeed three writes: Db, I[v2], B[v2]
  - ✓ Delayed write using cache (or queuing) ➔ Unexpected power loss or system crash ➔ Some writes can be done while others are not.
    - Db only is written to disk: no problem
    - B[v2] only is written to disk: 1) space leak, 2) inconsistency: inode vs. bitmap
    - I[v2] only is written to disk: 1) garbage read, 2) inconsistency
    - Db and B[v2] are written to disk (except I[v2]): inconsistency
    - Db and I[v2] are written to disk (except B[v2]): inconsistency
    - I[v2] and B[v2] are written to disk (except Db): Garbage read
  - ✓ Need consistency: write all modifications or nothing (a kind of atomicity)



42.2 Solution #1: The File System Checker

- Traditional solution: fsck (file system checker)
  - ✓ Consist of several passes
    - Superblock: metadata for FS, usually sanity check
    - Free blocks: check all inodes and their used blocks. If there is an inconsistent case in bitmaps, correct it (usually follow inode info.)
    - Inode state: validity check in each inode. reclaim wrong inodes
      - Inode links: link counts check by scanning the entire directory tree. Move the missed file (there is an inode but no directory entry points it) into the lost+found directory
      - Duplicate pointers: find blocks which are pointed by two or more inodes
      - Bad blocks: pointer that points outside its valid ranges
    - Directory checks: fs-specific knowledge based directory check (e.g. "." and ".." are the first entries
  - ✓ Issue: too slow
    - Remzi says that "the fsck looks like that, even though you drop the key in your bedroom, you start a search-the-entire-house-for-key algorithm, scanning from the basement, kitchen, and every room."

42.3 Solution #2: Journaling (or WAL)

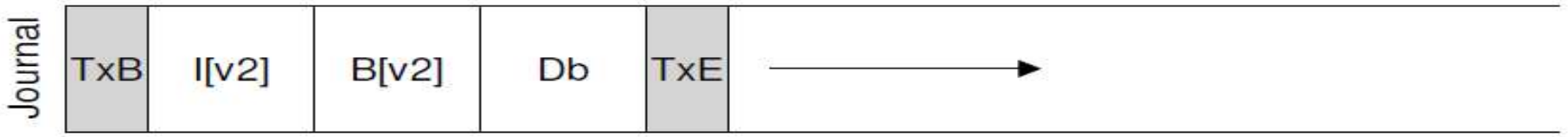
- Journaling
  - ✓ A Kind of WAL (Write-ahead logging)
  - ✓ Key idea: When updating disks, before overwriting the structure in place, first write down a little note to somewhere in a well-known location, describing what you are about to do.
  - ✓ Crash occur ➔ The note can say what you intended ➔ redo or undo
- Journaling FS
  - ✓ Linux Ext3/4, IBM JFS, SGI XFS, NTFS, Reiserfs, …
  - ✓ Features of Ext3 file system
    - Integrate journaling into ext2 file system
    - Three types: 1) journal (data journal), 2) ordered (metadata journal, ordered, default), 3) writeback (metadata journal, non-ordered)

(Ext2 disk layout, like FFS)

(Ext3 disk layout: Ext3 + Journaling)

42.3 Solution #2: Journaling (or WAL)

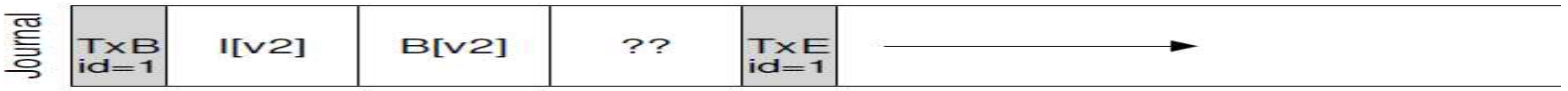
- Data Journaling
  - ✓ Assume we want to do three writes (I[v2], B[v2], and Db)
  - ✓ Before writing them to their final locations, we first write them to the log ➔ step 1: journaling.



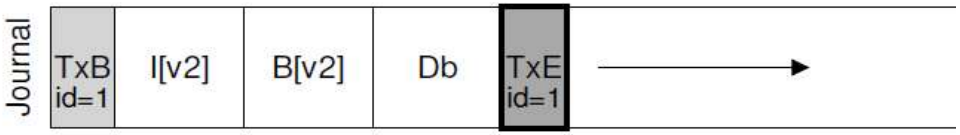
    - TxB: Transaction begin, include Tid and writes information
    - Log
      - Physical logging: same contents to the final locations
      - Logical logging: intent (save space, but more complex)
    - TxE: End with Tid
  - ✓ After making this transaction safe on disk, we are ready to update the original data ➔ step 2: checkpointing
  - ✓ Recovery (fault handling)
    - In the case of failures btw journaling and checkpointing, we can replay journal (redo) ➔ can go into the next consistent state
    - In the case of failures before journaling (before TxE), we can remove journal (undo) ➔ can stay in the previous consistent state

42.3 Solution #2: Journaling (or WAL)

- Issue 1: How to reduce journaling performance overhead?
  - ✓ For journaling, we need to write a set of blocks
    - e.g. TxB, i[v2], B[v2], Db, TxE
  - ✓ Approach 1: issue all writes at once
    - Unsafe, might be loss some requests



    - Transaction looks valid (it has begin and end). Thus, replaying journal leads wrong data to be updated.
  - ✓ Approach 2: issue each request at a time, wait for each to complete, then issuing the next (e.g. fsync() at each write)
    - Too slow
  - ✓ Approach 3: employ commit
    - Separate TxE from all other writes (e.g. fsync() before TxE)
    - Recovery: 1) not committed ➔ undo, 2) committed, but not in the original locations ➔ redo logging



  - ✓ Approach 4: issue all writes at once and apply checksum using all contents in the journal (integrity example)

42.3 Solution #2: Journaling (or WAL)

- Issue 2: How to reduce journaling write volume overhead?
  - ✓ Data journaling writes data twice, which increases I/O traffic (reducing performance), especially painful for sequential large writes
- Metadata Journaling
  - ✓ Journal Metadata Only
    - User data is not written to the journal (I and B, except D)

  - ✓ Question?
    - Does the writing order btw user data and journal become matter? ➔ Yes, writing journal before user data causes problems (garbage read)
  - ✓ Conclusion: ordered journaling
    - 1) Data write ➔ 2) Journal metadata write ➔ 3) Journal commit ➔ 4) Checkpoint ➔ 5) Free
  - ✓ Real world
    - Ext3: support both ordered and writeback(non-ordered)
    - Windows NTFS and SGI's XFS use non-ordered metadata journaling

42.3 Solution #2: Journaling (or WAL)

- Timeline
  - ✓ Data journaling vs. Metadata Journaling
    - Horizontal dashed line is “write barrier”
    - Note that, in Figure 42.2, the order btw Data and Journaling is not guaranteed (writeback mode in the ext3.)

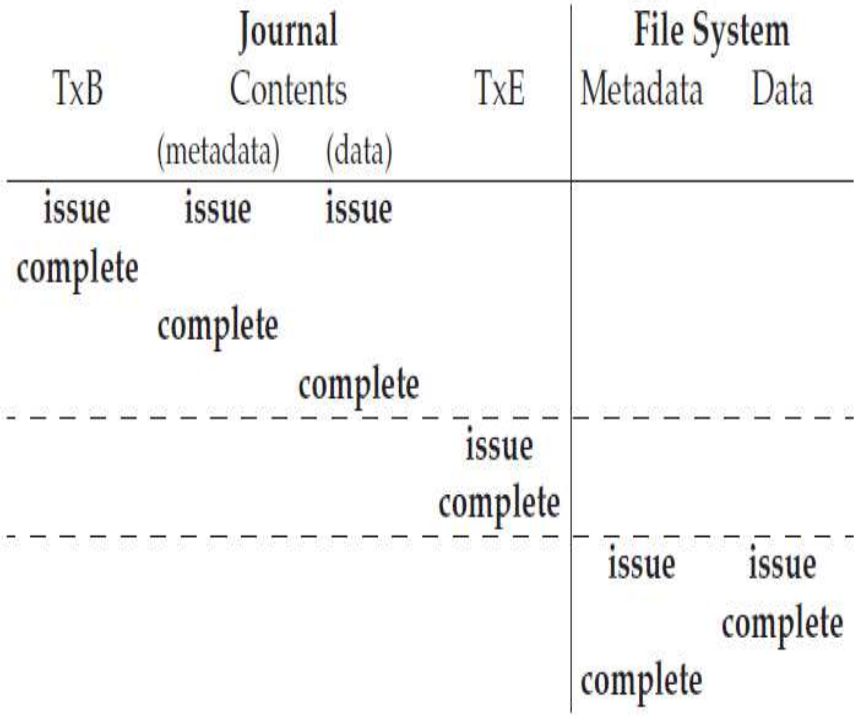


Figure 42.1: Data Journaling Timeline

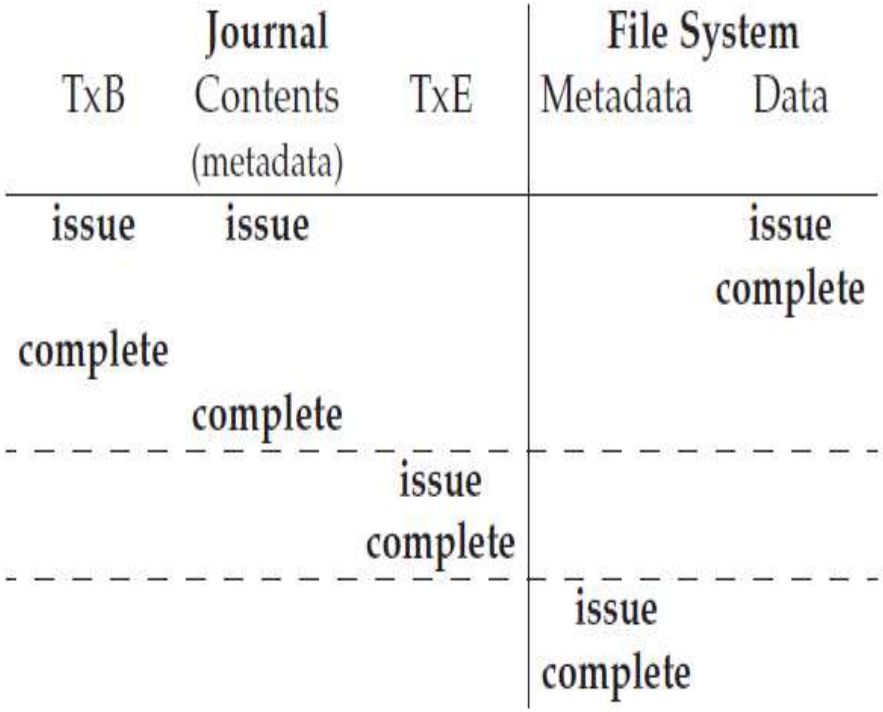
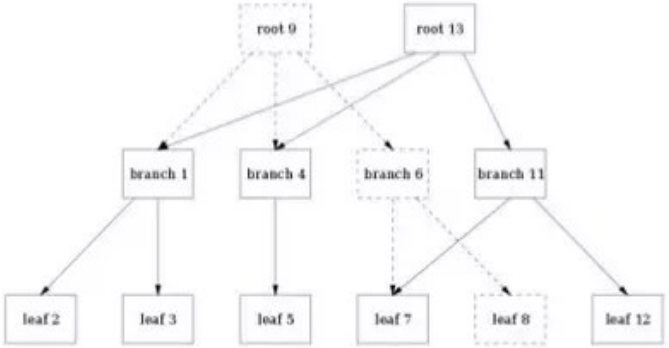


Figure 42.2: Metadata Journaling Timeline

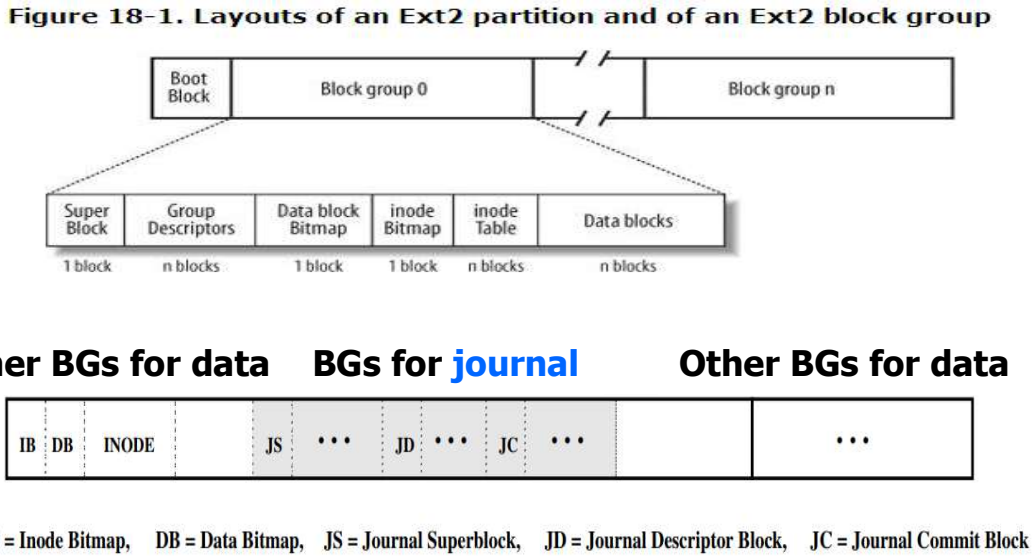
42.4 Solution #3: Other Approaches

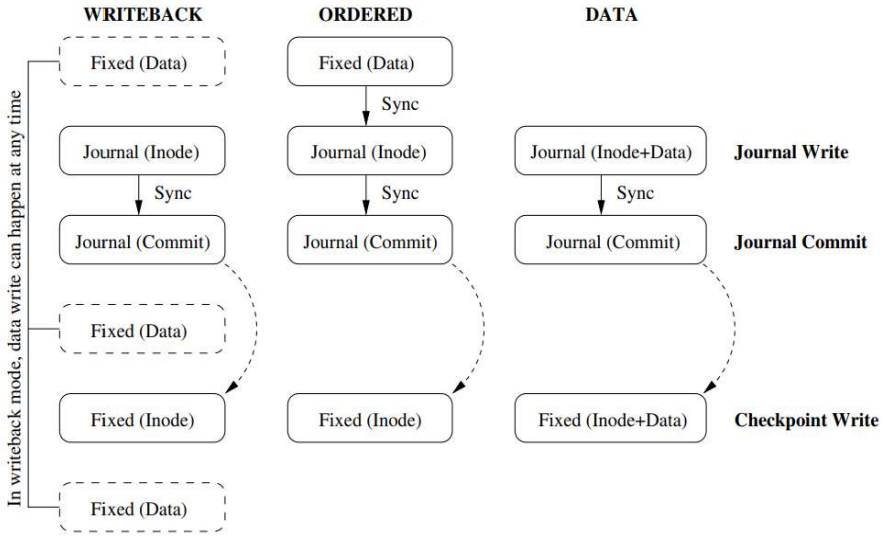
- Summary
  - ✓ fsck: A lazy approach
  - ✓ Journaling: An active approach
    - Ext3, Reiserfs, IBM's JFS, …
  - ✓ Soft update
    - Suggested by G. Ganger and Y. Patt
    - Carefully order all writes so that on-disk structures are never left in an inconsistent state (e.g. data block is always written before its inode)
    - Soft update is not easy to implement since it requires intricate knowledge about file system (On contrary, journaling can be implemented with relatively little knowledge about FS)
  - ✓ COW (Copy on Write)
    - Used in Btrfs and Sun's ZFS
  - ✓ Optimistic crash consistency
    - Enhance performance by issuing as many writes to disk as possible
    - Exploit checksum as well as a few other techniques



Features of popularly used FS: Ext2/3/4 FS

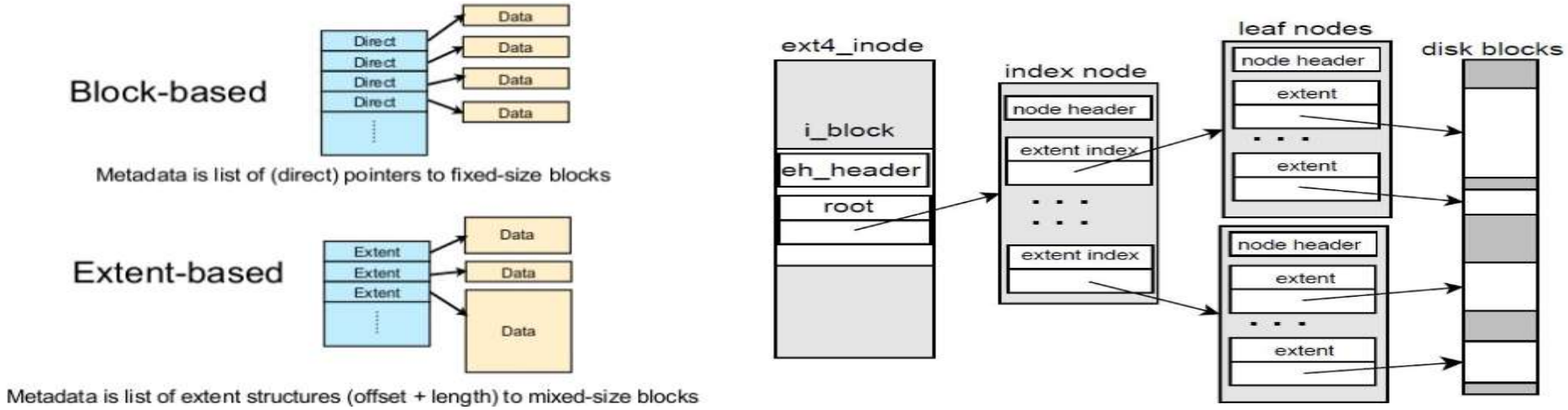
- Ext2
  - ✓ R. Card, T. Ts'o and S. Tweedie, "Design and Implementation of the second extended FS", http://e2fsprogs.sourceforge.net/ext2intro.html
  - ✓ Performance enhancement: 1) cylinder group, 2) pre-allocation: usually 8 adjacent blocks, 3) Read-ahead during sequential reads
    - Employ FFS approach
- Ext3
  - ✓ Ext2 + Journaling
  - ✓ Use a block group (or groups) for journaling
  - ✓ Three types: 1) data journal, 2) ordered, 3) writeback





Features of popularly used FS: Ext2/3/4 FS

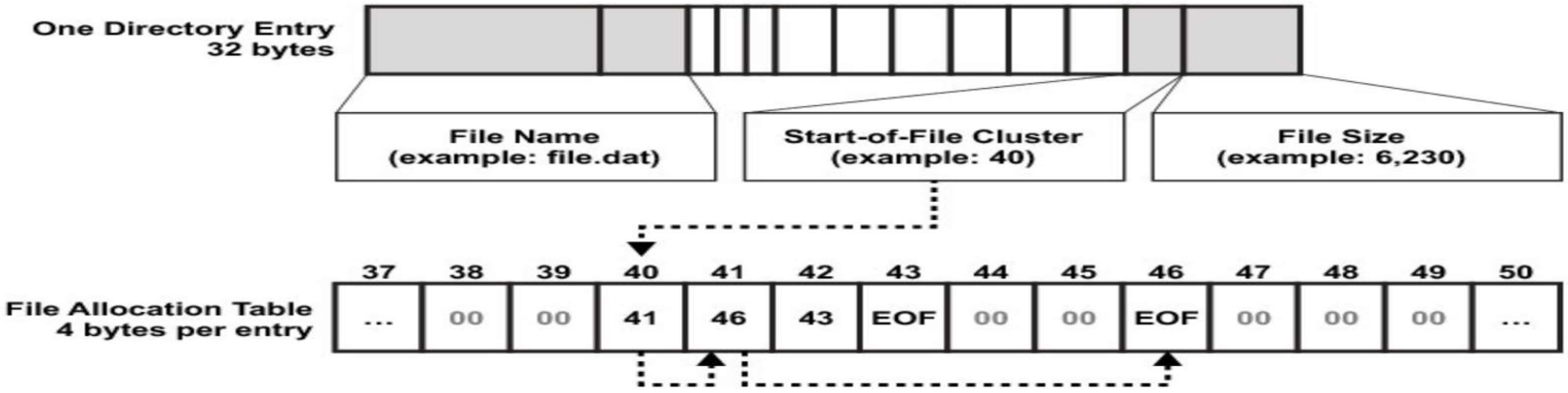
- Ext4
  - ✓ Ext3 + Larger file system capacity with 64-bit
    - Supports huge file size (e.g. 16TB) and file system (e.g. 2^64 blocks)
    - Directory can contain up to 64,000 subdirs
  - ✓ Extent-based mapping
    - Extent: Variable size (c.f. Inode: fixed size (4KB))
      - E.g. Contiguous 16KB ➔ need one mapping vs need 4 mappings
      - Ext4, BtrFS, ZFS, NTFS, XFS, …
    - Need split/merge in a tree structure (extent tree)
  - ✓ Hash based directory entries management



(Source: https://www.slideshare.net/relling/s8-filesystemslisa13, https://blog.naver.com/PostView.nhn?blogId=jalhaja0&logNo=221536636378)

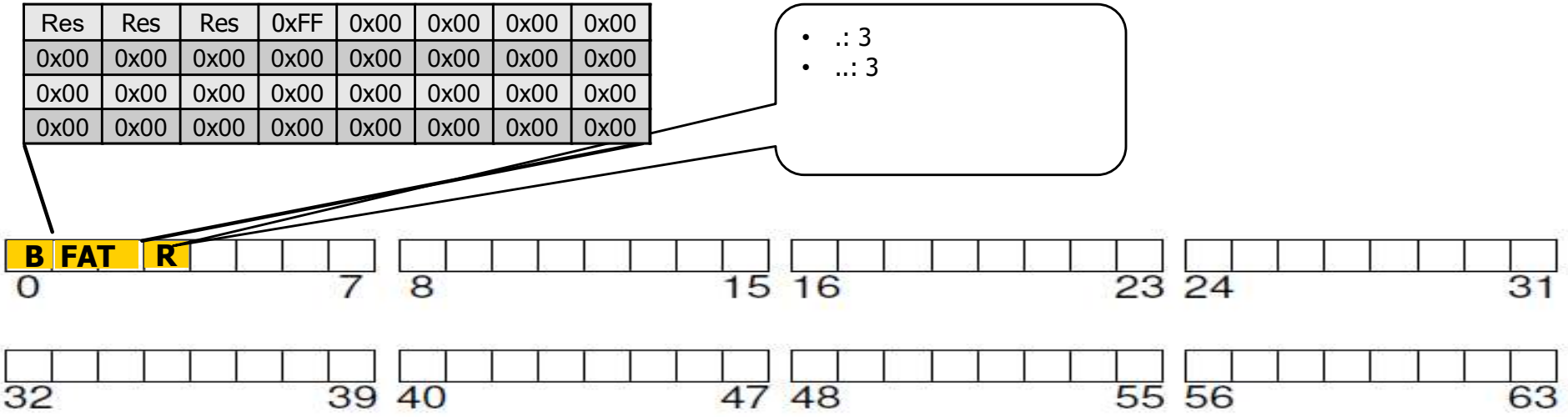
Features of popularly used FS: FAT FS

- Why?
  - ✓ Large vs Small storage (USB, Memory card, IoT device)
    - Space for Metadata is quite expensive
- Solution: FAT file system
  - ✓ Originated by Microsoft
  - ✓ Idea: Bitmap, Inode ➔ FAT (File Allocation Table)
    - 1) Used for used/free, 2) data location (link for next block)
      - c.f.: inode: per file metadata vs. FAT: for all files (one in a file system)
    - Directory entry: point to the first index for FAT
    - Metadata (size, time, permission, ..) in directory entry



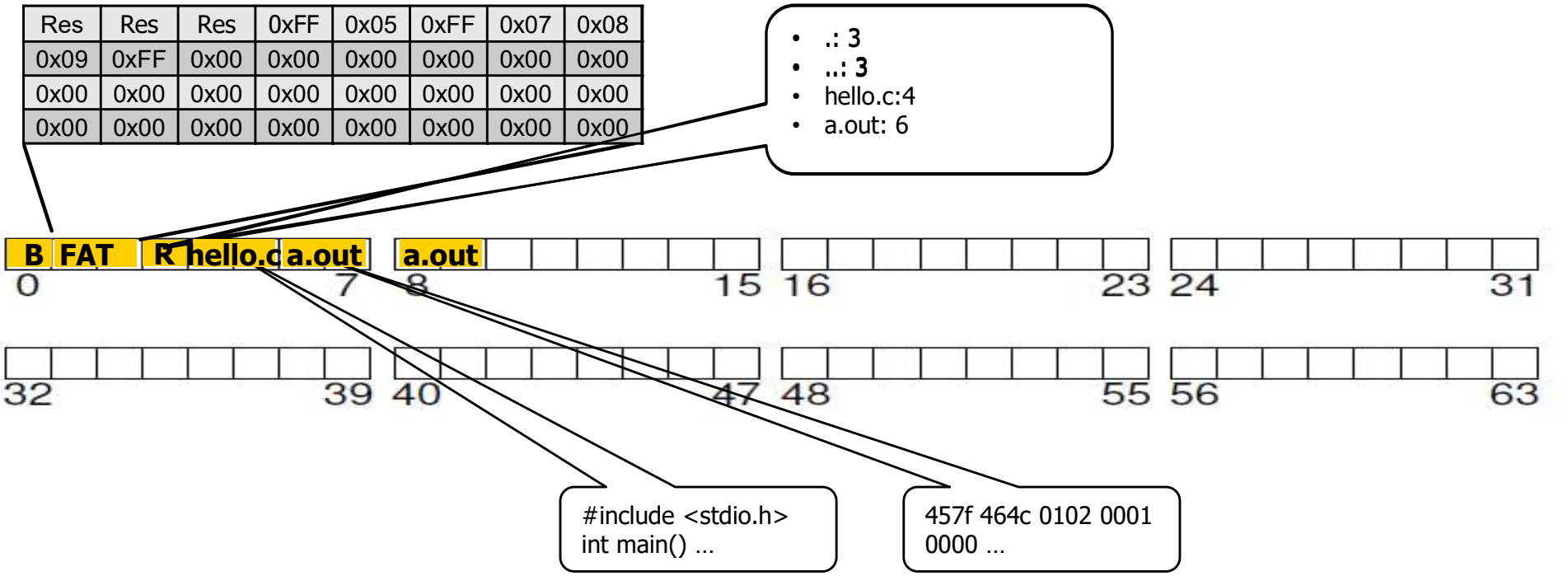
Features of popularly used FS: FAT FS (optional)

- Example
  - ✓ Layout assumption
    - 1 block for Boot Sector, 2 blocks FAT, 1 block for root directory
  - ✓ Working scenario
    - After initialization



Features of popularly used FS: FAT FS (optional)

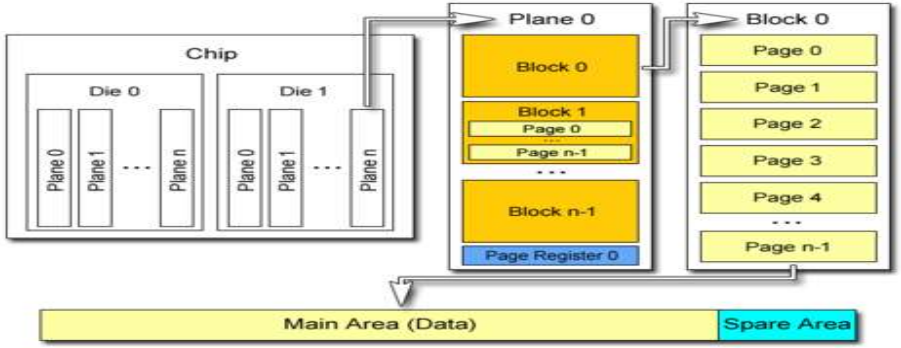
- Example
  - ✓ Layout assumption
    - 1 block for Boot Sector, 2 blocks FAT, 1 block for root directory
  - ✓ Working scenario (similar example in 40.3 The inode)
    - When we create a new file (named hello.c whose size is 7KB) in a root directory?
    - Then, we compile it? (a.out whose size is 15KB)



Chap. 44 Flash-based SSDs

- Representative storage devices
  - ✓ Disk (or Hard Disk Drive) vs Flash memory (or Solid Stage Drive)
- Characteristic comparison
  - ✓ Same: non-volatility
  - ✓ Differences
    - 1) No mechanical parts in Flash memory: good for mobile devices
    - 2) Flash is faster than Disk (us vs ms)
    - 3) Overwrite limitation (Erase-before-Write)
      - Disk: read, write vs. Flash memory: read, write, erase (from EEPROM)
    - 4) Read/write vs. Erase granularity
      - Read/write: page unit (e.g. 4KB)
      - Erase: block unit (e.g. 2MB)
    - 5) Endurance: limited number of erasures (wear out issue)

<Read/Write> <Read/Write + Erase>

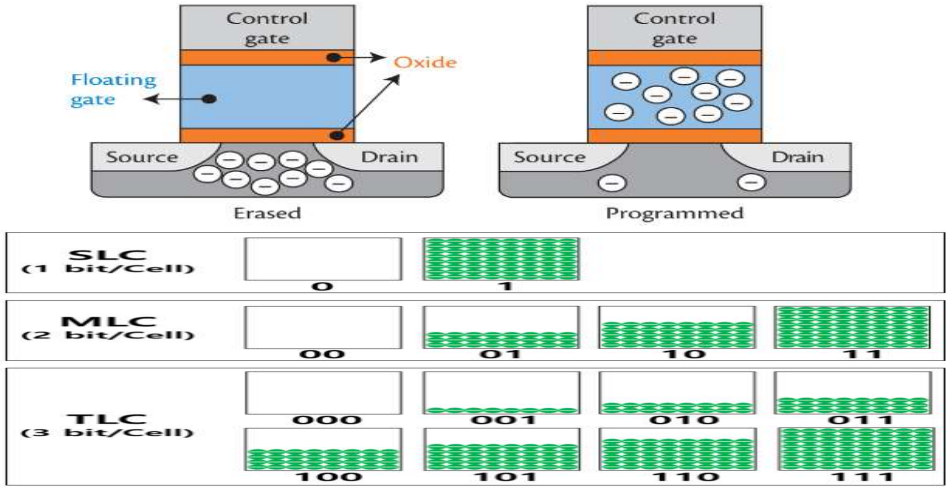


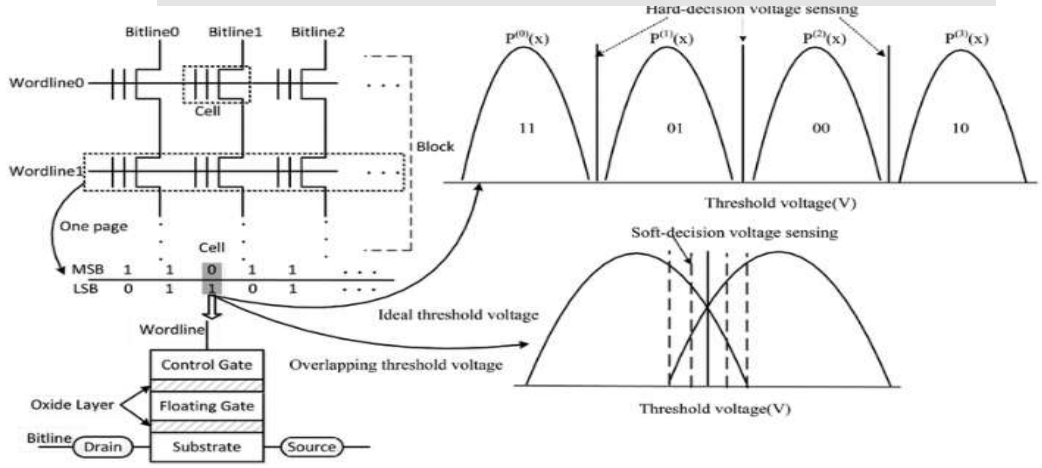
<page vs. block >

44.1 Storing a Bit / 44.2 From Bits to Pages/Blocks

- Flash memory structure
  - ✓ Basic unit: cell
    - Consist of gates (floating and control gate)
    - Can trap charges (support non-volatility)
    - Differentiate bits based on the charges: SLC, MLC, TLC, QLC, …
  - ✓ Page and Block
    - Page: a set of cells (read/write unit)
    - Block: a set of pages (erase unit)
    - Plane (or Bank): a set of Blocks
    - Chip: a set of plane

> **TIP: BE CAREFUL WITH TERMINOLOGY**
> You may have noticed that some terms we have used many times before (blocks, pages) are being used within the context of a flash, but in slightly different ways than before. New terms are not created to make your life harder (although they may be doing just that), but arise because there is no central authority where terminology decisions are made. What is a block to you may be a page to someone else, and vice versa, depending on the context. Your job is simple: to know the appropriate terms within each domain, and use them such that people well-versed in the discipline can understand what you are talking about. It's one of those times where the only solution is simple but sometimes painful: use your memory.





The structure of MLC NAND flash memory and hard-decision/soft-decision voltage sensing.

(Source: https://www.researchgate.net/figure/The-structure-of-MLC-NAND-flash-memory-and-hard-decision-soft-decision-voltage-sensing_fig1_314034986)

44.3 Basic Flash Operations

- Flash operations
  - ✓ Simple flash chip assumption
    - 4 pages per a block: Figure 44.1
  - ✓ Operations
    - Read a page: sensing charges (threshold voltage) to identify bits, fast (same speed regardless locations which are different from disks)
    - Erase a block: set each bit to 1 (less charge, lower threshold voltage) in a block, expensive
    - Program (or write) a page: change some of the 1's within a page to 0's (more charge, higher threshold voltage), middle speed
      - Note1: programing is allowed only on the erase pages (reprogramming makes error) ➔ erase-before-write property

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure 44.2: **Raw Flash Performance Characteristics**

(Operation latency)

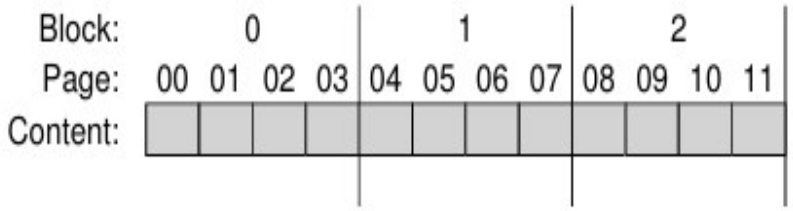


Figure 44.1: **A Simple Flash Chip: Pages Within Blocks**

(Simple Flash: 4 pages per a block)

		iiii	*Initial: pages in block are invalid (i)*
Erase()	→	EEEE	*State of pages in block set to erased (E)*
Program(0)	→	VEEE	*Program page 0; state set to valid (V)*
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	VVEE	*Program page 1*
Erase()	→	EEEE	*Contents erased; all pages programmable*

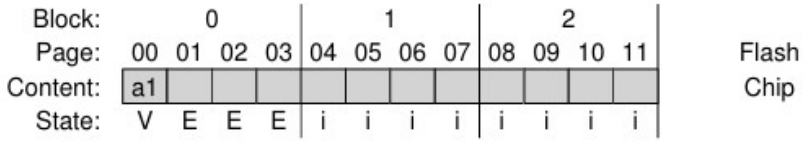
(Operation example, reads do not affect states)

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

(After the first Program(0))

44.3 Flash performance and reliability

- Issues
  - ✓ Performance issue
    - 1) It requires erase before program ➔ make write performance bad
    - 2) Erase has to be conducted in a block unit (larger granularity) ➔ if there are valid pages, they have to be copied before erase ➔ make write performance worse

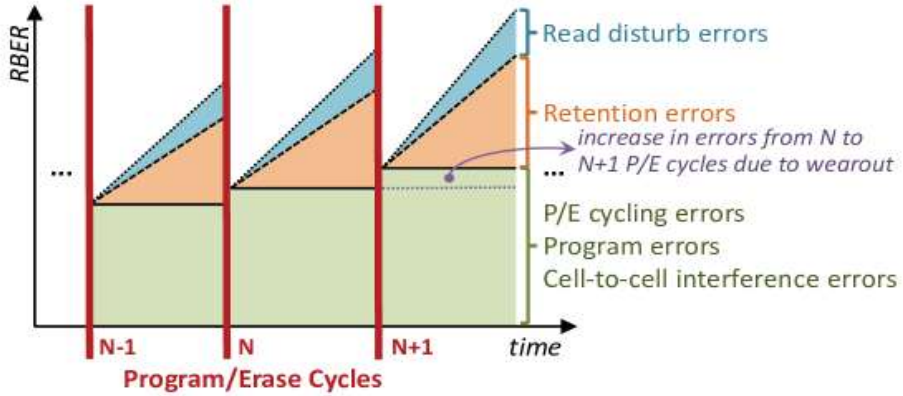


  - ✓ Reliability issue

☞ **How to handle these issues? ➔ FTL**

    - 1) Endurance error: number of P/E cycles are limited ➔ more erasures, less reliable, often become bad blocks (wear out)
    - 2) Other errors: retention error, disturbance error, …

	SLC	MLC	TLC
Bits per cell	1	2	3
P/E Cycles	100,000	3,000	1,000
Read Time	25 µs	50 µs	~75 µs
Program Time	200-300 µs	600-900 µs	~900-1350 µs
Erase Time	1.5-2 ms	3 ms	4.5 ms



(Source: Errors in Flash-Memory-Based SSDs: Analysis, Mitigation, and Recovery, Proceeding of IEEE, 2017)

44.5 From Flash to SSDs

- SSDs
  - ✓ Flash memory + Processing capability (Logic + Memory) + Interface
  - ✓ Three features:
    - 1) make SSDs look like HDDs (backward compatibility)
    - 2) handle Flash characteristics + performance and reliability issues
    - 3) make use of concurrency: multiple chips (multi-channels/ways)
  - ✓ Employ SW called FTL (Flash Translation Layer)
    - That's why SSDs need processing capability (ARM CPUs and SRAM)
    - Interface: like HDDs (read LBA, write LBA), no erase interface
    - Functionalities: 1) mapping (translation: indirection from LBA to pages), 2) GC (migration and erase, try to reduce write amplification), 3) wear-leveling, 4) bad block handling and reliability enhancement mechanisms, 5) support concurrency (e.g. super pages)

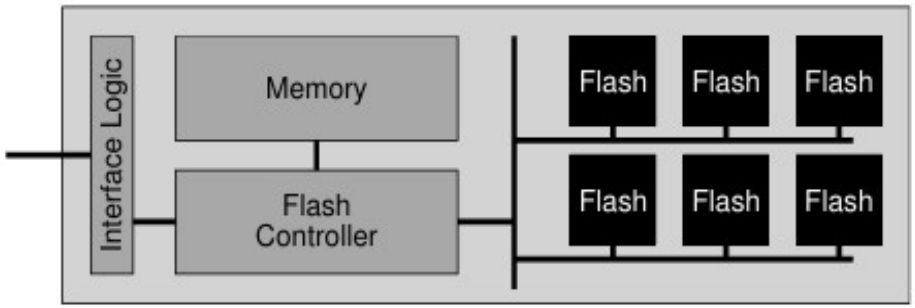
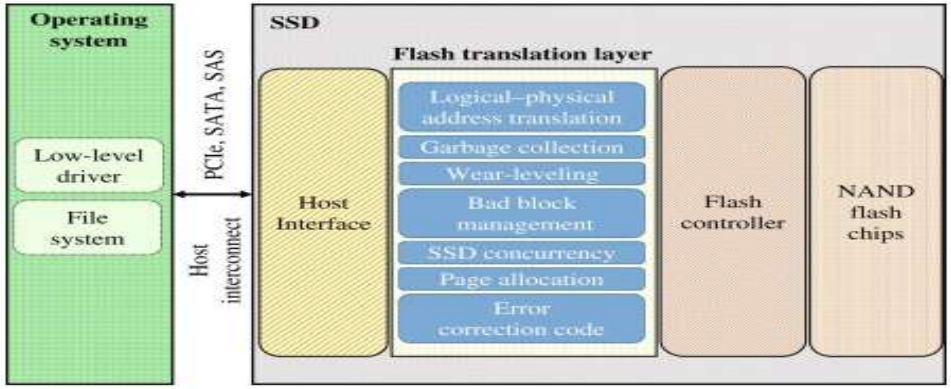


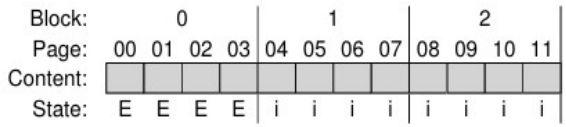
Figure 44.3: **A Flash-based SSD: Logical Diagram**



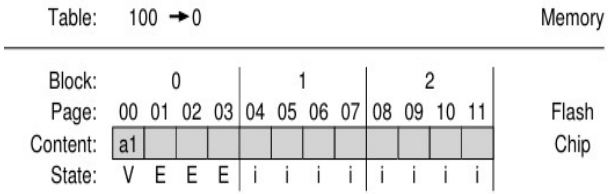
44.6 Simple FTL / 44.7 Log-structured FTL

- FTL Organization
  - ✓ Simple and Naïve: No indirection (direct mapping)
    - E.g.) LBA 0 is mapped to Block 0, Page 0
    - Do erase per each write ➔ Costly approach (performance, write amplification, reliability, …)
  - ✓ Log-Structured FTL: Out-of-place update with mapping table
    - E.g.)
      - Write(100) with contents a1
      - Write(101) with contents a2
      - Write(2000) with contents b1
      - Write(2001) with contents b2

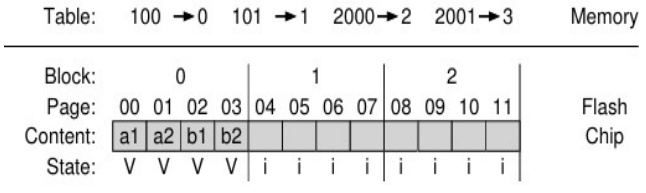
      where 100, 101, 2000, 2001 are LBAs
    - Internal changes
      - 1) receive first write, 2) erase block 0
      - 3) program page 0 and make a mapping in the table
        - Subsequent reads use this table to translate LBAs to blocks/pages
      - 4) program other writes sequentially



(after erase block 0)



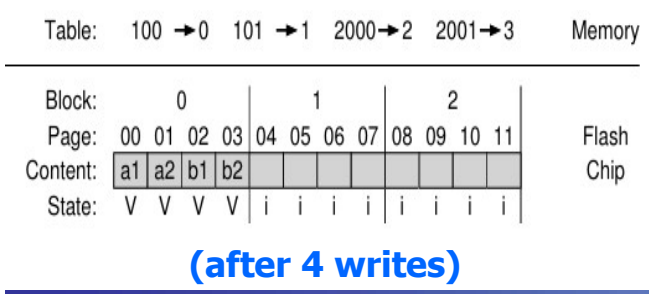
(after write (100))



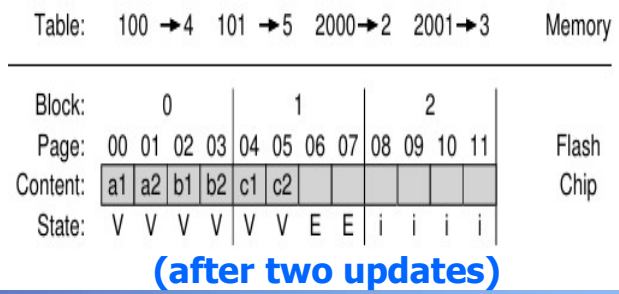
(after 4 writes)

44.8 Garbage Collection

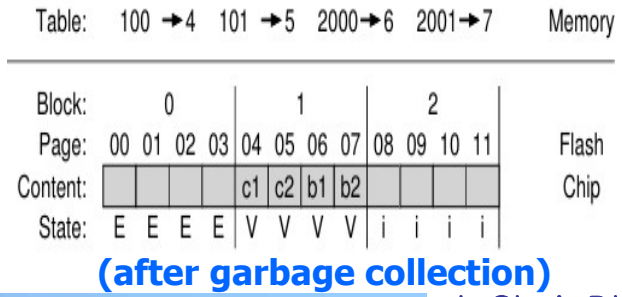
- Garbage collection
  - ✓ Log-structured FTL use the out-of-place update
    - Updates always go to new locations ➔ old locations become invalid
    - E.g.) after 4 writes, write(100) with contents c1, write(101) with contents c2
      - Erase block 1, Write page 4 and 5, Mapping table are also changed
      - Old page 0 and 1 are now obsolete (not valid)
  - ✓ Garbage collection
    - Shortage of free space ➔ reclaim obsolete pages to make free space
      - 1) find a block that contains one or more garbage pages, 2) read in the live (non-garbage) pages from that block, 3) write out those live pages to new locations, 4) update mapping table, and 5) erase the block
    - Note: 6 writes from user, 8 writes in Flash chip (two additions for GC) ➔ write amplification
    - Many SSDs have over-provision area to boost garbage collection



(after 4 writes)



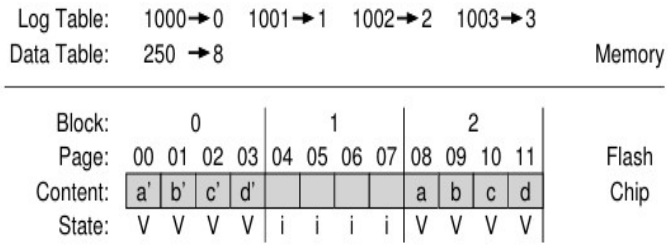
(after two updates)



(after garbage collection)

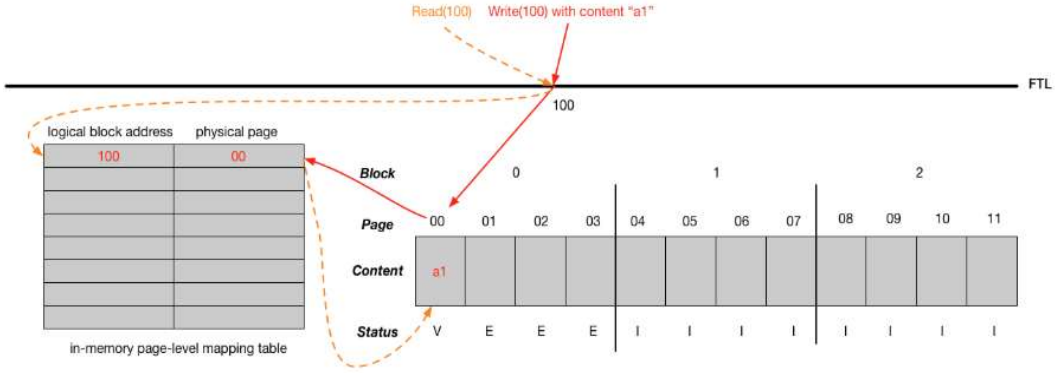
44.9 Mapping table size (Optional)

- 3 approached for Mapping management
  - ✓ Page-level FTL
    - Translate LBAs to Flash pages
      - E.g.) LBA 100 ➔ page 0, LBA 2000 ➔ page 2
    - Pros) flexible, Cons) larger mapping table size

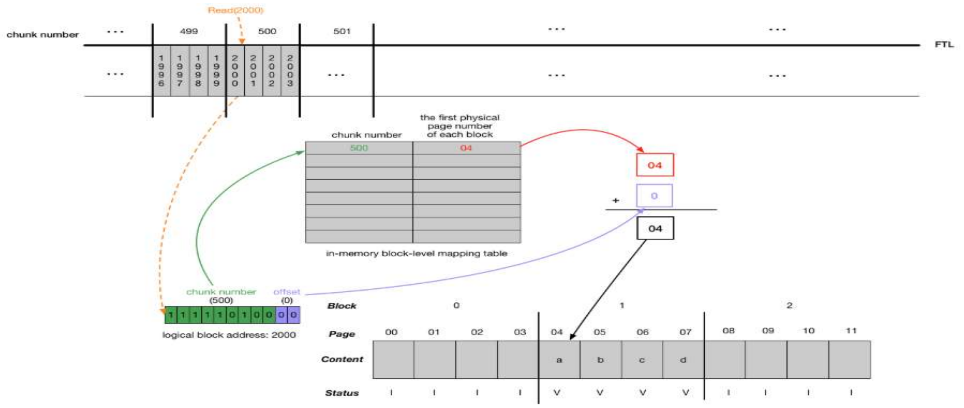


(Hybrid mapping FTL)

  - ✓ Block-level FTL
    - Calculate logical Flash-block # (chuck #), Translate chuck # to Flash blocks
      - E.g.) 1) LBA 2000, calculate: chunk # = 2000/4=500, page # = 2000%4 = 0. 2) translate 500 ➔ 4, 3) go 4 block, read the first page of that block
    - Pros) smaller mapping table (N_{block} vs. N_{page}), Cons) less flexible (fixed location in a block), higher GC overhead (E.g. update LBA 2000, 2 times)
  - ✓ Hybrid-level FTL
    - Data block: block-level, Log block: page-level
    - Trade-offs between table size and garbage collection overhead



Page-level FTL write & read example : A write to logical block address 100 with content "a1" to SSD followed by a read from the same logical block address.

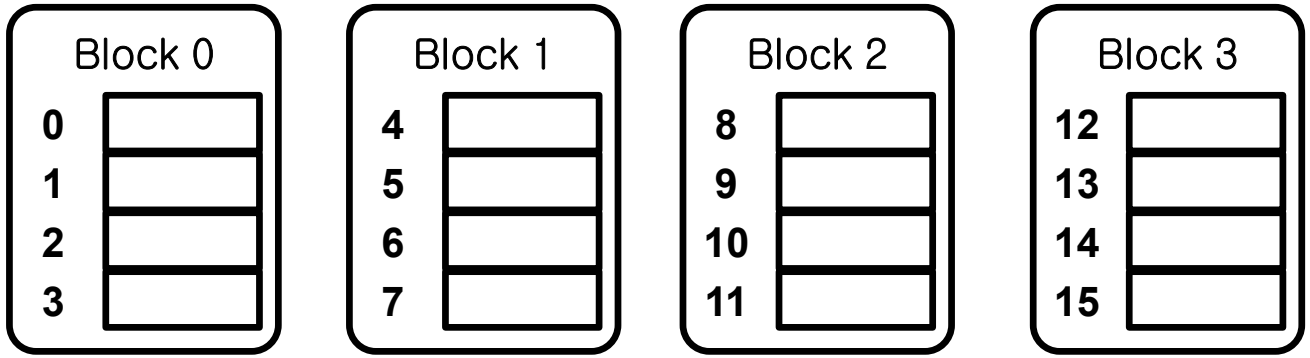


Block-level FTL read example : the logical block address consists of a chunk number (500) and a offset (0).

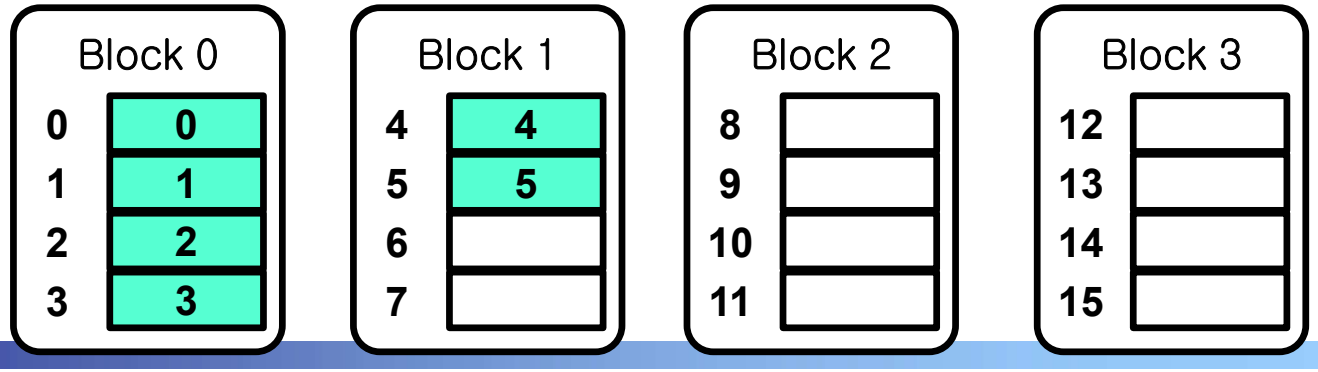
(Source: https://zhu45.org/posts/2018/Jul/01/flash-based-ssd-basics/)

44.9 Mapping table size (Optional)

- Realization example of Page-level FTL
  - ✓ 1. SSD assumption
    - 1) 4 pages per block, 2) 4 blocks in a chip, 3) GC trigger only 1 free block



  - ✓ 2. Workload assumption
    - write LBAs: 0,1,2,3,4,5,1,3,4,1,5,6
  - ✓ 3. Mapping example
    - For write LBAs: 0,1,2,3,4,5

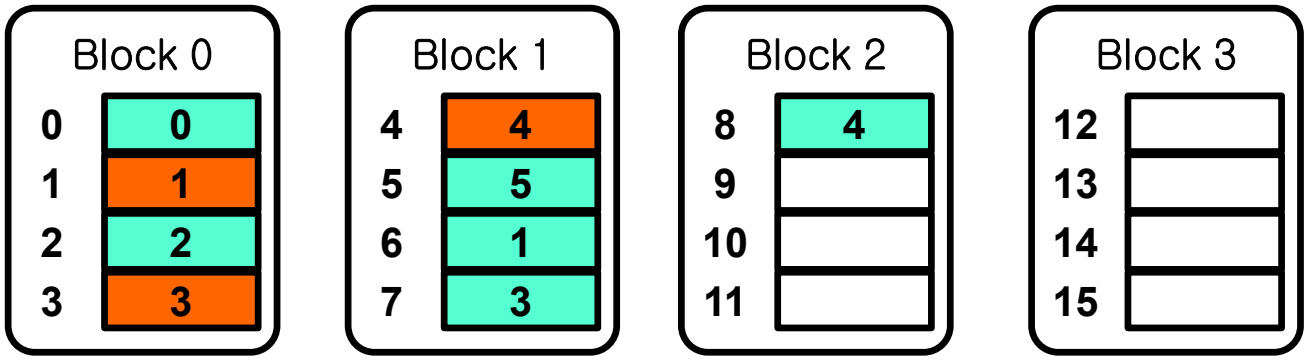


Mapping table

LBA	PPN
0	0
1	1
2	2
3	3
4	4
5	5
6	-
7	-
8	-
9	-
10	-
11	-

44.9 Mapping table size (Optional)

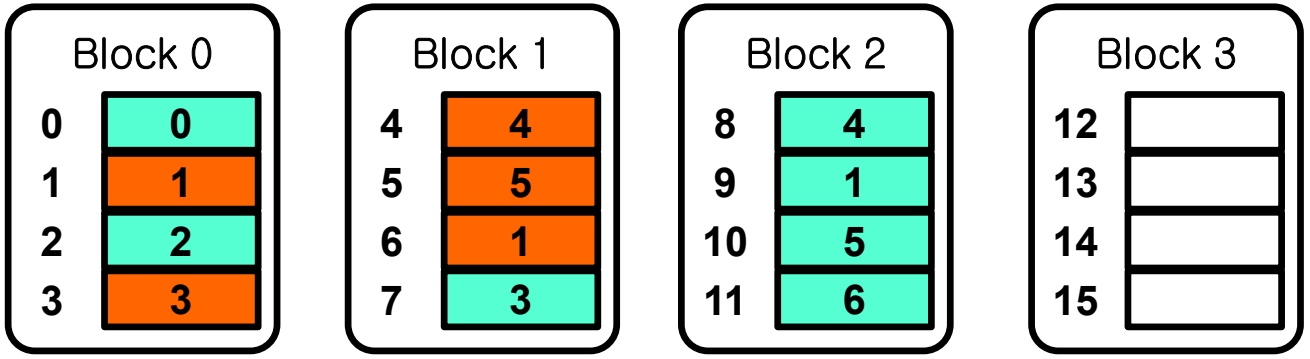
- Realization example of Page-level FTL (cont')
  - ✓ 3. Mapping example
    - For write LBAs: 0,1,2,3,4,5,1,3,4 ➔ consider overwrite limitation ➔ need to employ out-of-place updates



Mapping table

LBA	PPN
0	0
1	6
2	2
3	7
4	8
5	5
6	-
7	-
8	-
9	-
10	-
11	-

    - For write LBAs: 0,1,2,3,4,5,1,3,4,1,5,6

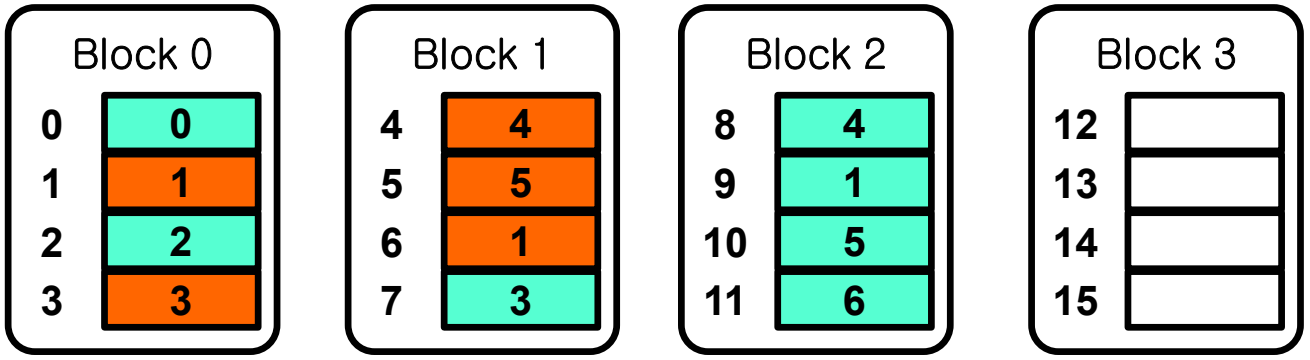


LBA	PPN
0	0
1	9
2	2
3	7
4	5
5	10
6	11
7	-
8	-
9	-
10	-
11	-

    - Only one free block ➔ GC trigger

44.9 Mapping table size (Optional)

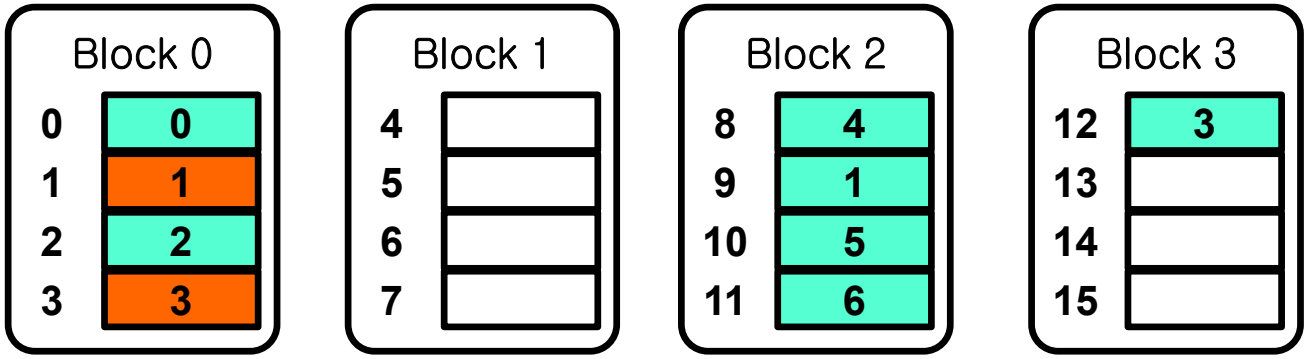
- Realization example of Page-level FTL (cont')
  - ✓ 4. Garbage collection
    - Policies: 1) When: 1 free block, 2) which victim: Greedy, 3) how many: 1 victim block (differ from Lab3)



Mapping table

LBA	PPN
0	0
1	9
2	2
3	7
4	5
5	10
6	11
7	-
8	-
9	-
10	-
11	-

    - After GC
      - GC step: 1) copy valid pages, 2) erase, 3) set as free



LBA	PPN
0	0
1	9
2	2
3	12
4	5
5	10
6	11
7	-
8	-
9	-
10	-
11	-

    - Some reporting: GC copies = 1, erasures = 1, WA = 12/13
    - ☞ **More details: Semi-conductor SW class in the next Semester!!**

44.10 Wear Leveling / 44.11 SSD Perf. and Cost

- Wear-leveling
  - ✓ Goal: spread out writes evenly across blocks
    - all blocks will wear out at roughly the same time, instead of a few “popular” blocks quickly becoming unusable
  - ✓ How to: 1) allocate less-used blocks, 2) migrate long-lived data from less-used to more-used blocks, 3) hot-cold detection and wear-aware management, …
- SSD performance and cost
  - ✓ Performance
    - 1) random is fast compared to disks (in other word, disks performs well for sequential), 2) among randoms, writes relatively better than reads (due to log-structured writes), …
  - ✓ SSD cost: 10X compared to disks
    - E.g.) an SSD costs 60 cents per GB, while an HDD costs 5 cents per GB
  - ✓ Implication: tiered storage
    - SSDs + HDDs (Google, Meta, Naver, …)
      - Performance sensitive: SSDs
      - Big data: HDDs

Device	Random		Sequential	
	Reads (MB/s)	Writes (MB/s)	Reads (MB/s)	Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure 44.4: SSDs And Hard Drives: Performance Comparison

44.12 Summary

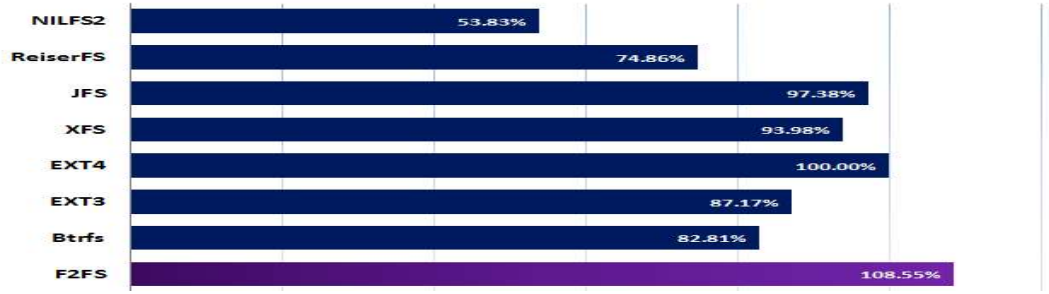
- Key SSD terms

- A **flash chip** consists of many banks, each of which is organized into **erase blocks** (sometimes just called **blocks**). Each block is further subdivided into some number of **pages**.
- Blocks are large (128KB–2MB) and contain many pages, which are relatively small (1KB–8KB).
- To read from flash, issue a read command with an address and length; this allows a client to read one or more pages.
- Writing flash is more complex. First, the client must **erase** the entire block (which deletes all information within the block). Then, the client can **program** each page exactly once, thus completing the write.
- A new **trim** operation is useful to tell the device when a particular block (or range of blocks) is no longer needed.
- Flash reliability is mostly determined by **wear out**; if a block is erased and programmed too often, it will become unusable.
- A flash-based **solid-state storage device** (**SSD**) behaves as if it were a normal block-based read/write disk; by using a **flash translation layer** (**FTL**), it transforms reads and writes from a client into reads, erases, and programs to underlying flash chips.
- Most FTLs are **log-structured**, which reduces the cost of writing by minimizing erase/program cycles. An in-memory translation layer tracks where logical writes were located within the physical medium.
- One key problem with log-structured FTLs is the cost of **garbage collection**, which leads to **write amplification**.
- Another problem is the size of the mapping table, which can become quite large. Using a **hybrid mapping** or just **caching** hot pieces of the FTL are possible remedies.
- One last problem is **wear leveling**; the FTL must occasionally migrate data from blocks that are mostly read in order to ensure said blocks also receive their share of the erase/program load.

Summary

- File basic
  - ✓ Layout: superblock, bitmap, inode, data blocks
  - ✓ Access methods: open(), read(), write(), …
- Optimization
  - ✓ Performance: FFS, Ext2, …
    - Storage-awareness, simple but effective techniques
  - ✓ Consistency: Ext3/4, JFS, …
    - Change from valid state to another valid state
  - ✓ Make sequential: LFS, WAFL, …
    - Successfully realization of the out-of-place update concept
  - ✓ Flash-Awareness: Generic FS on FTL, F2FS, …
    - Handle Flash Characteristics, SSD Characteristics, Semi-conductor SW
  - ✓ For mobile devices: FAT, QNX FS, …
    - Small metadata, Robustness, …

- ext2 – Great implementation of a "classic" file system
- ext3 – Add a journal for faster crash recovery and less risk of data loss
- ext4 – Scale to bigger data sets, plus other features



(Source: https://www3.cs.stonybrook.edu/~porter/courses/cse506/f14/slides/ext4.pdf)

Lab3: Ext2 Analysis

- What to do?
  - ✓ Goal
    - Lab3: Analyze Ext2 file system internal (a kind of digital forensic!!)
    - Note: last year we conducted an Lab3 about FTL (why not this year? ➔ We have a Semiconductor SW class in the fall semester)
  - ✓ How to do?
    - 1. create ramdisk
    - 2. make ext2 file system on ramdisk and mount the ext2 file system
    - 3. run the script on the mount directory (./create.sh) ➔ will generate dirs. and files
    - 4. find two files assigned to you and find blocks allocated for the files
      - Assigned files: last three digits of a student number ➔ directory + file name
      - (e.g. *****123 ➔ directory name is 1, file name is 23 and 32)
      - How to: dump ramdisk (using xxd), examine Ext2 (or make a program that parsing Ext2)
      - Superblock ➔ Group descriptor ➔ root inode ➔ root data ➔ dir. inode ➔ dir. data ➔ … ➔ file inode ➔ file data (eventually block number)
  - ✓ Requirement
    - Report: 1) goal, 2) analysis results and snapshots, 3) discussion
  - ✓ How to submit? upload at Google form
  - ✓ Due: two weeks later
  - ✓ Bonus
    - Print your name and student id while mounting Ext2
    - Ext2 source modification + make + module insert (e.g. insmod) + mkfs + mount

Lab3: Ext2 Analysis Details (1/3)

- Main steps

```
sys32153550@ESL-LeeJY:~/workspace/2020_1/OS_Lab3$ ls
append.c  create.sh  Makefile  ramdisk.c
sys32153550@ESL-LeeJY:~/workspace/2020_1/OS_Lab3$ sudo su
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# make
make -C /lib/modules/5.3.0-42-generic/build M=/home/sys32153550/workspace/2020_1/OS_Lab3 modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-42-generic'
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/ramdisk.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/sys32153550/workspace/2020_1/OS_Lab3/ramdisk.mod.o
  LD [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/ramdisk.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-42-generic'
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# ls
append.c  create.sh  Makefile  modules.order  Module.symvers  ramdisk.c  ramdisk.ko  ramdisk.mod  ramdisk.mod.c  ramdisk.mod.o  ramdisk.o
```

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# insmod ramdisk.ko
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# lsmod | grep ramdisk
ramdisk                16384  0
```

(1) make a ramdisk and insmod it

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# mkfs.ext2 /dev/ramdisk
mke2fs 1.44.1 (24-Mar-2018)
Creating filesystem with 262144 4k blocks and 65536 inodes
Filesystem UUID: 5f361a67-3aaf-48aa-9013-7e3ab1080ffd
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# mount /dev/ramdisk ./mnt
```

(2) mkfs and mount

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# ./create.sh
create files ...
done
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# ls mnt
0  1  2  3  4  5  6  7  8  9  lost+found
```

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# ls mnt/0
0   12  16  2   23  27  30  34  38  41  45  49  52  56  6   63  67  70  74  78  81  85  89  92  96
1   13  17  20  24  28  31  35  39  42  46  5   53  57  60  64  68  71  75  79  82  86  9   93  97
10  14  18  21  25  29  32  36  4   43  47  50  54  58  61  65  69  72  76  8   83  87  90  94  98
11  15  19  22  26  3   33  37  40  44  48  51  55  59  62  66  7   73  77  80  84  88  91  95  99
```

(3) make file hierarchy by running script

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# xxd -g 4 -l 0x100 -s 0x38426000 /dev/ramdisk
38426000: 352f3530 2d310a00 00000000 00000000  5/50-1..........
38426010: 00000000 00000000 00000000 00000000  ................
```

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# xxd -g 4 -l 0x100 -s 0x10bc6000 /dev/ramdisk
10bc6000: 352f3530 2d320a00 00000000 00000000  5/50-2..........
10bc6010: 00000000 00000000 00000000 00000000  ................
```

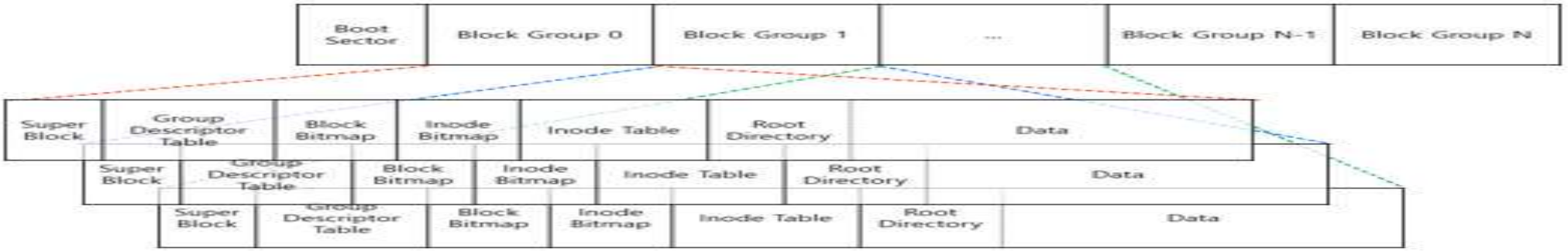
```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# xxd -g 4 -l 0x100 -s 0x10c67000 /dev/ramdisk
10c67000: 352f3530 2d330a00 00000000 00000000  5/50-3..........
10c67010: 00000000 00000000 00000000 00000000  ................
```

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# xxd -g 4 -l 0x100 -s 0x1102a000 /dev/ramdisk
1102a000: 352f3530 2d31330a 00000000 00000000  5/50-13.........
1102a010: 00000000 00000000 00000000 00000000  ................
```

(4) Explore file system layout

Lab3: Ext2 Analysis Details (2/3)

- Key structures



(layout)

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	inode count				block count				res block count				free block count			
10	free inode count				first data block				log block size				log frag size			
20	block per group				frag per group				inode per group				mtime			
30	wtime				mount count		max mount size		magic		state		errors		minor version	
40	last check				check interval				creator OS				major version			
50	def_res uid		def_res gid		first non-reserved inode				inode size		block grp num		compatible feature flag			
60	incompatible feature flag				feature read only compat				uuid (16 byte)							
70									volume name (16 byte)							
80																
90	last mounted (64 byte)												prealloc dir block			
a0																
b0													prealloc block			
c0									algorithm usage bitmap							padding
d0	journal uuid															
e0	journal inode number				journal device				last orphan							
f0	hash seed (16 byte)													pad		padding
100	default mount option				first meta block								default hash version			

(superblock)

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	block bitmap				inode bitmap				inode table				free blk cnt		free ino cnt	
10	used dir cnt		padding		reserved (padding)											

(group descriptor table)

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	mode		uid		size				access time				change time			
10	modification time				deletion time				gid		link count		blocks			
20	flags				OS description 1											
30	block pointer (60 byte)															
40																
50																
60					generation				file access control list				dir access control list			
70	fragmentation blk addr				OS description 2											

(inode)

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	inode				record len		name len	file type	name (~255 byte)							

(directory entry)

Lab3: Ext2 Analysis Details (3/3)

- Bonus

```
static int ext2_fill_super(struct super_block *sb, void *data, int silent)
{
        struct dax_device *dax_dev = fs_dax_get_by_bdev(sb->s_bdev);
        struct buffer_head * bh;
        struct ext2_sb_info * sbi;
        struct ext2_super_block * es;
        struct inode *root;
        unsigned long block;
        unsigned long sb_block = get_sb_block(&data);
        unsigned long logic_sb_block;
        unsigned long offset = 0;
        unsigned long def_mount_opts;
```

(modify ext2 source: just add your name)

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2# insmod os_ext2.ko
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2# lsmod | grep os_ext2
os_ext2                73728  0
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2# cd ..
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# ls
append.c   Makefile  modules.order   os_ext2    ramdisk.ko   ramdisk.mod.c  ramdisk.o
create.sh  mnt       Module.symvers  ramdisk.c  ramdisk.mod  ramdisk.mod.o
```

(insmod: os_ext2)

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2# make
make -C /lib/modules/5.3.0-42-generic/build M=/home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-42-generic'
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/balloc.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/dir.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/file.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/ialloc.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/inode.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/ioctl.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/namei.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/super.o
  CC [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/symlink.o
  LD [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/os_ext2.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/os_ext2.mod.o
  LD [M]  /home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2/os_ext2.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-42-generic'
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3/os_ext2# ls
acl.c     dir.c   file.o    inode.o  Makefile       namei.o       os_ext2.mod.o  symlink.c  xattr.h
acl.h     dir.o   ialloc.c  ioctl.c  modules.order  os_ext2.ko    os_ext2.o      symlink.o  xattr_security.c
balloc.c  ext2.h  ialloc.o  ioctl.o  Module.symvers os_ext2.mod   super.c        tags       xattr_trusted.c
balloc.o  file.c  inode.c   Kconfig  namei.c        os_ext2.mod.c super.o        xattr.c    xattr_user.c
```

(make module: kernel loadable module)

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# mkfs.ext2 /dev/ramdisk
mke2fs 1.44.1 (24-Mar-2018)
Creating filesystem with 262144 4k blocks and 65536 inodes
Filesystem UUID: 820655ee-13bb-4475-8b87-1c951acaff33
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# mount -t os_ext2 /dev/ramdisk ./mnt
```

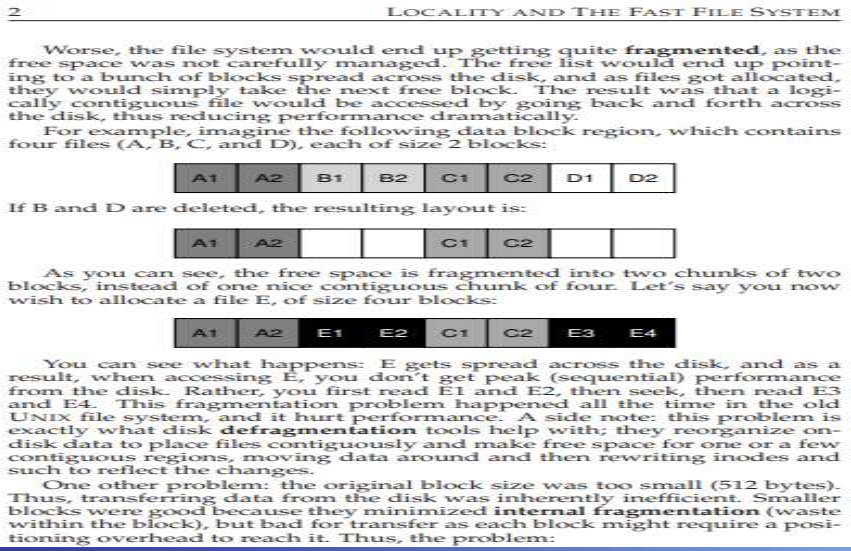
(mkfs and mount)

```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# dmesg | grep os_ext2
[2510165.993926] os_ext2 : Lee Jeyeon OS Lab3
```

(Your name and student ID are printed out at the kernel level NOT at the user level!!)

Quiz for this Lecture

- Quiz
  - ✓ 1. Read page 2 in Chap. 41 of OSTEP and explain why fragmentation (external fragmentation) happens and what is the benefit of a defragmentation tool?
  - ✓ 2. Explain why FFS makes use of the rule 2 using the term of namespace locality.
  - ✓ 3. Discuss the “amortization” using the rule 3 of FFS (hint: compare when limited number of blocks in a group is 4MB vs. 400KB)
  - ✓ 4. We want to create a file whose size is 4KB, as shown in the below right figure. Using the figure, explain the terms of “1) space leak“, “2) garbage read“, “3) dangling reference”, and “4) inconsistent”.



2 LOCALITY AND THE FAST FILE SYSTEM

Worse, the file system would end up getting quite **fragmented**, as the free space was not carefully managed. The free list would end up pointing to a bunch of blocks spread across the disk, and as files got allocated, they would simply take the next free block. The result was that a logically contiguous file would be accessed by going back and forth across the disk, thus reducing performance dramatically.

For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:

A1	A2	B1	B2	C1	C2	D1	D2

If B and D are deleted, the resulting layout is:

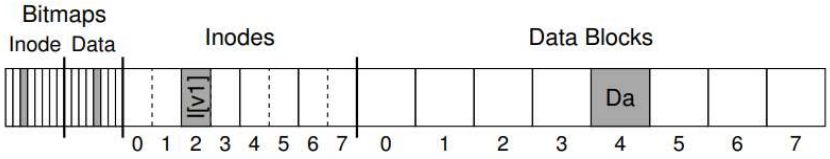
A1	A2			C1	C2		

As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say you now wish to allocate a file E, of size four blocks:

A1	A2	E1	E2	C1	C2	E3	E4

You can see what happens: E gets spread across the disk, and as a result, when accessing E, you don't get peak (sequential) performance from the disk. Rather, you first read E1 and E2, then seek, then read E3 and E4. This fragmentation problem happened all the time in the old UNIX file system, and it hurt performance. A side note: this problem is exactly what disk **defragmentation** tools help with; they reorganize on-disk data to place files contiguously and make free space for one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes.

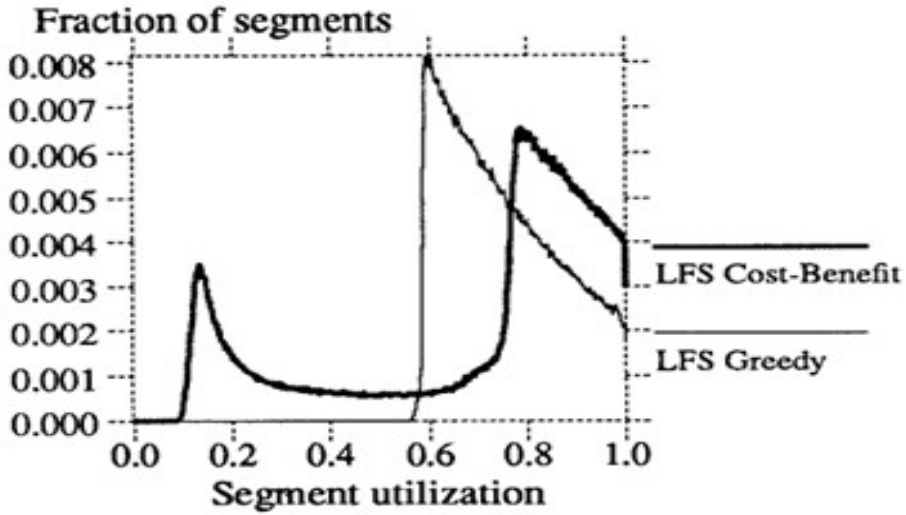
One other problem: the original block size was too small (512 bytes). Thus, transferring data from the disk was inherently inefficient. Smaller blocks were good because they minimized **internal fragmentation** (waste within the block), but bad for transfer as each block might require a positioning overhead to reach it. Thus, the problem:

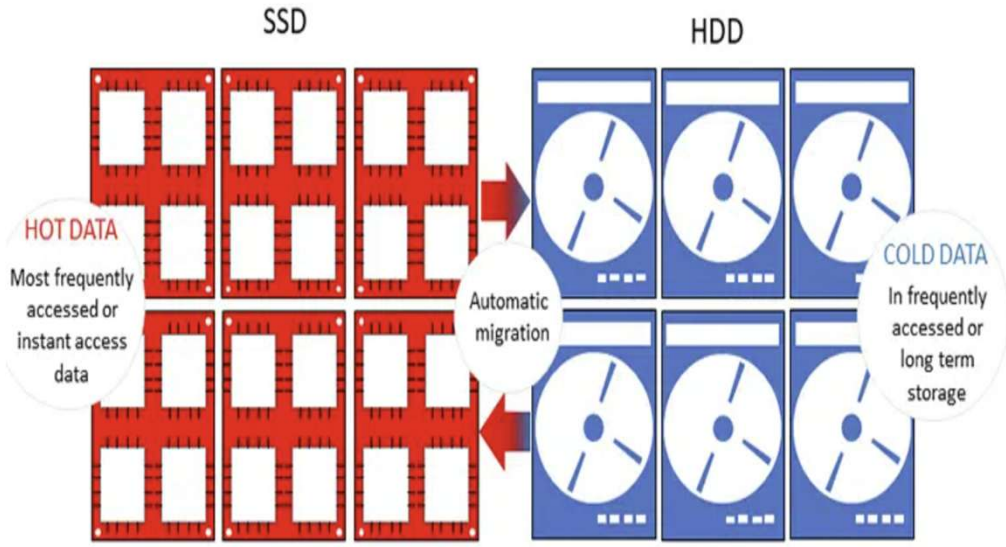



```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

Quiz for this Lecture

- Quiz
  - ✓ 5. Explain the concept of the log-structure approach. Why it requires indirection?
  - ✓ 6. Discuss the differences between greedy and cost-benefit segment selection policies using the below left figure.
  - ✓ 7. FTL supports an abstraction, hiding details of Flash memory characteristics and providing it looks like a Disk. Describe key functionalities of FTL.
  - ✓ 8. Many bigdata companies makes use of hybrid storage, consisting of SSDs and HDDs. Discuss why such a storage hierarchy is actively adopted by the companies?





Appendix 1: Some details (1/2)

- 41.5 Measuring File Locality: FFS relies on Common Sense (What CS stands for ^^)
  - ✓ Files in a directory are often accessed together (namespace locality)
  - ✓ Measurement: Fig. 41.1
    - Using real trance called SEER traces
    - Path difference: how far up the directory tree you have to travel to find the common ancestor btw the consecutive opens in the trace
      - E.g.) same file: 0, /a/b and /a/c: 1, /a/b/e and /a/d/f: 2, …
    - Observation: 60% of opens in the trace ➔ less than 2.
      - E.g.) OSproject/src/a.c, OSproject/include/a.h, OSproject/obj/a.o, …

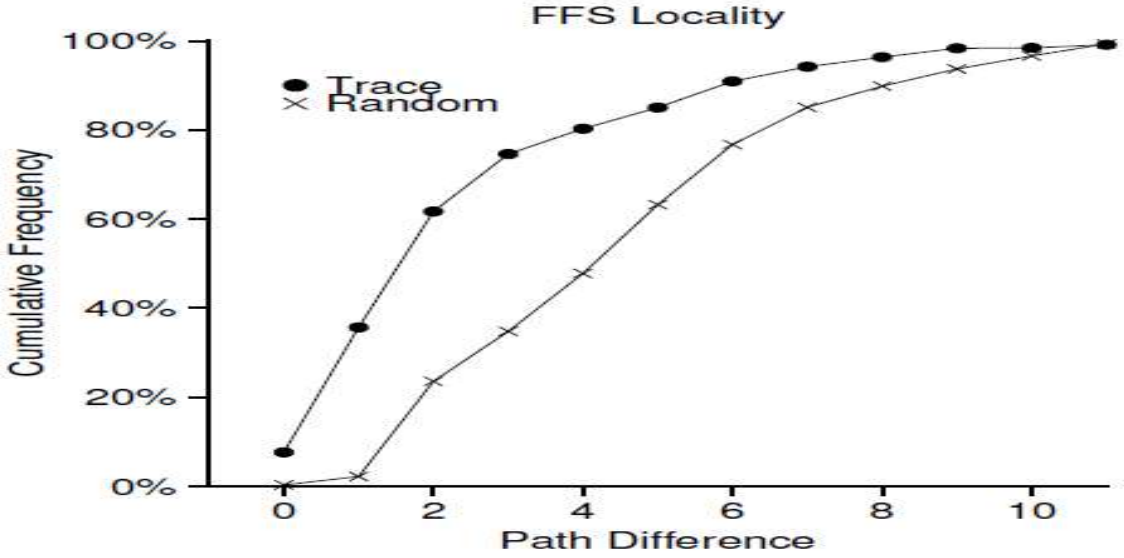


Figure 41.1: FFS Locality For SEER Traces

Appendix 1: Some details (2/2)

- 42.3 Solution #2: Journaling (or WAL): Revoke record in journal: for block reuse handling
  - ✓ Scenario: 1) there is a directory called foo, 2) a user adds an entry to foo (create a file), 3) foo's contents are written to block 1000, 4) log are like the following figure (note that directory is metadata, which is also logged)

  - ✓ 5) The user deletes the foo (and its subfiles), 6) The user creates another file (say foobar), which uses the block 1000, 7) Writes for foobar are logged (note that file contents themselves are not logged)

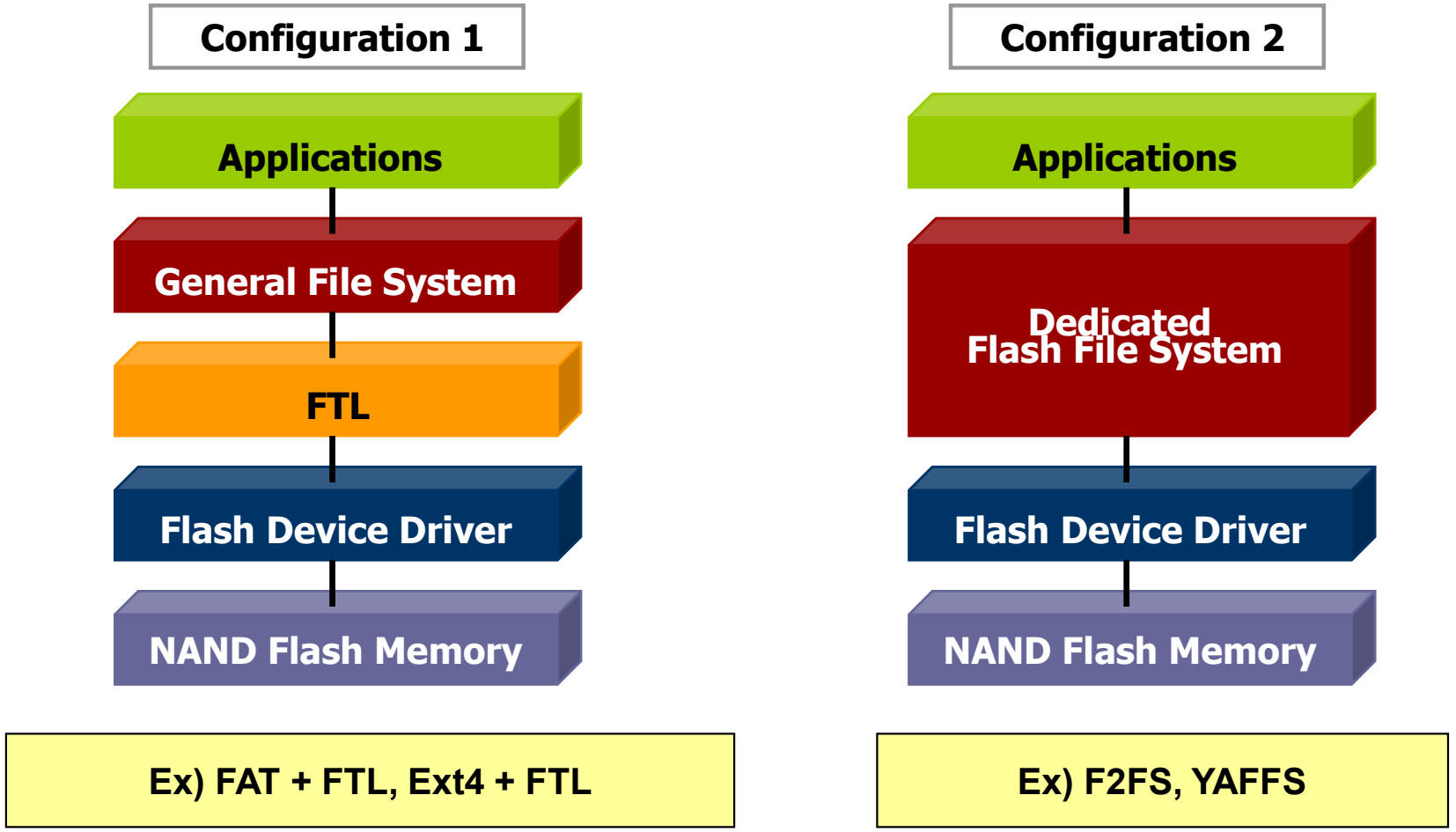
  - ✓ 8) At this point, a crash occurs. 9) recovery performs "redo" from the beginning of the log. 10) overwrites the user data of the file foobar with the old directory contents.
  - ✓ Solution
    - Ext3 adds a new type of record, a revoke record, for the deleted file or directory. When do replaying, any revoked records are not redo

Appendix 2: Alternative Lab3 (1/2)

- What to do?
  - ✓ Goal
    - Lab3: Make a FTL based on a Simple Flash Chip
    - Assumption (Refer to Realization example of Page-level FTL)
      - Page size: 4KB, Pages per a block: 4 ➔ Block size: 16KB
      - Blocks per a chip: 10 ➔ Chip capacity: 160KB
      - Mapping: Page mapping
      - GC trigger: when free blocks become less than or equal to 2
      - GC victim selection: greedy and cost-benefit (select 2 victims at each trigger)
  - ✓ How to do?
    - 1. Make a FTL including mapping and garbage collection (no wear-leveling)
    - 2. Run a given workload
    - 3. After finishing the workload, showing Mapping table
    - 4. Also reporting the number of copies and erasures during GC, and WA
    - 5. Do the 3 and 4 with different workloads
    - See Lab. 3 in https://github.com/DKU-EmbeddedSystem-Lab/2025_DKU_OS
  - ✓ How to submit?
    - 1) Report (Sections: Goal, Design, Result, Discussion), 2) Source code (with Makefile) ➔ upload at Google Form
  - ✓ Due: two weeks later
  - ✓ Bonus
    - Employ cache for reducing overwrites to Flash memory (4 DRAM pages)

Appendix 2: Alternative Lab3 (2/2)

- FTL vs. Flash-aware file system
  - ✓ Abstract Flash memory like a Disk by using 1) out-of-place update and mapping, 2) garbage collection, 3) wear-leveling, …



☞ **More details: Semi-conductor SW in the next Semester!!**

사사

- 본 교재는 2026년도 과학기술정보통신부 및 정보통신기획평가원의 ‘SW중심대학사업’ 지원을 받아 제작 되었습니다.
- 본 결과물의 내용을 전재할 수 없으며, 인용(재사용)할 때에는 반드시 과학기술정보통신부와 정보통신기획평가원이 지원한 ‘SW중심대학’의 결과물이라는 출처를 밝혀야 합니다.

IITP 정보통신기획평가원

디지털인재양성단 SW인재팀