Lecture Note 7. Advanced File System

May 4, 2021

Jongmoo Choi

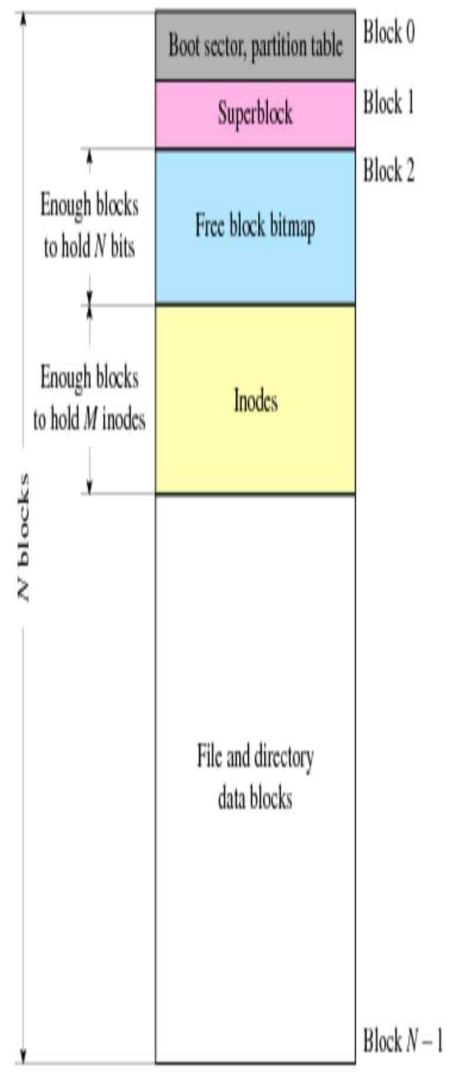
Dept. of software
Dankook University
http://embedded.dankook.ac.kr/~choijm

Contents

- From Chap 41~45 of the OSTEP
- Chap 41. Locality and the Fast File System
  - ✓ Performance requirement
  - ✓ Storage-aware performance enhancement
- Chap 42. Crash Consistency: FSCK and Journaling
  - ✓ Consistency requirement
  - ✓ Journaling mechanism
- Chap 43. Log-structured File Systems
- Chap 44. Flash-based SSDs
- Chap 45. Data Integrity and Protection
- Chap 46. Summary
- Summary. Features of Various FS: Ext2/3/4, FAT, Flash FS 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 access, data access alternately
      - Look good, 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

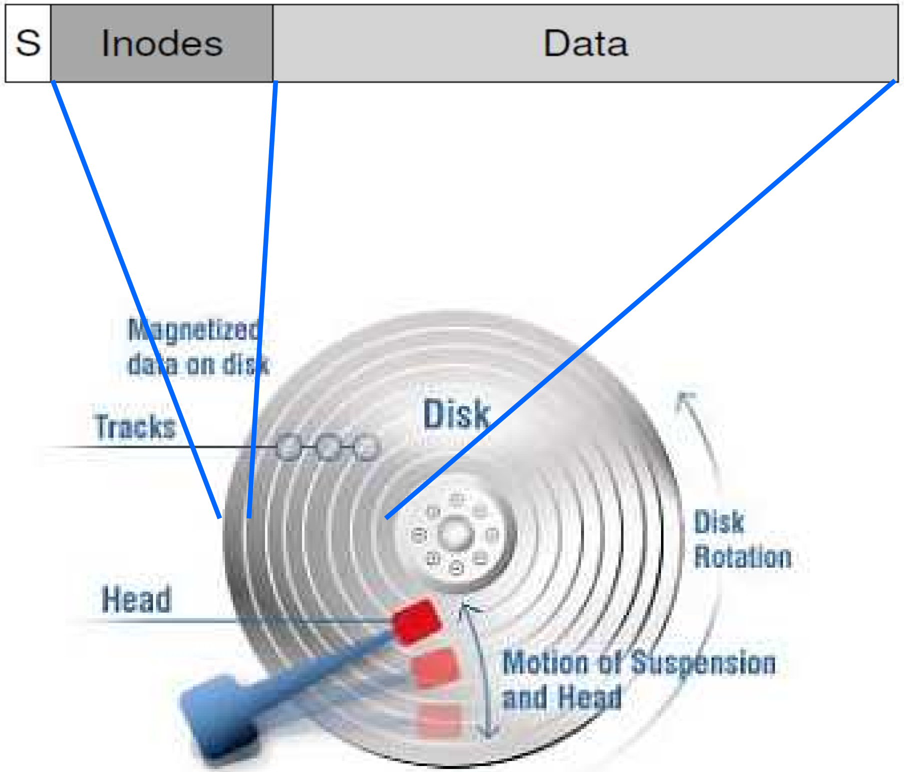


41.1 Poor Performance

- 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					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

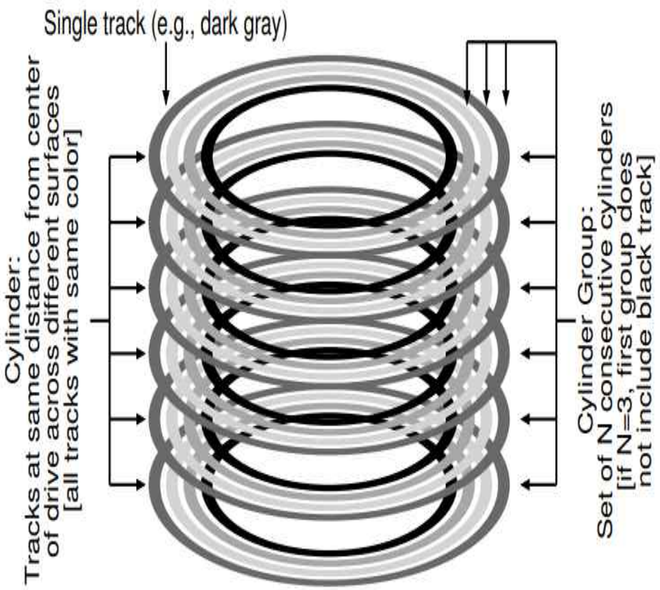
Figure 40.3: File Read Timeline (Time Increasing Downward)

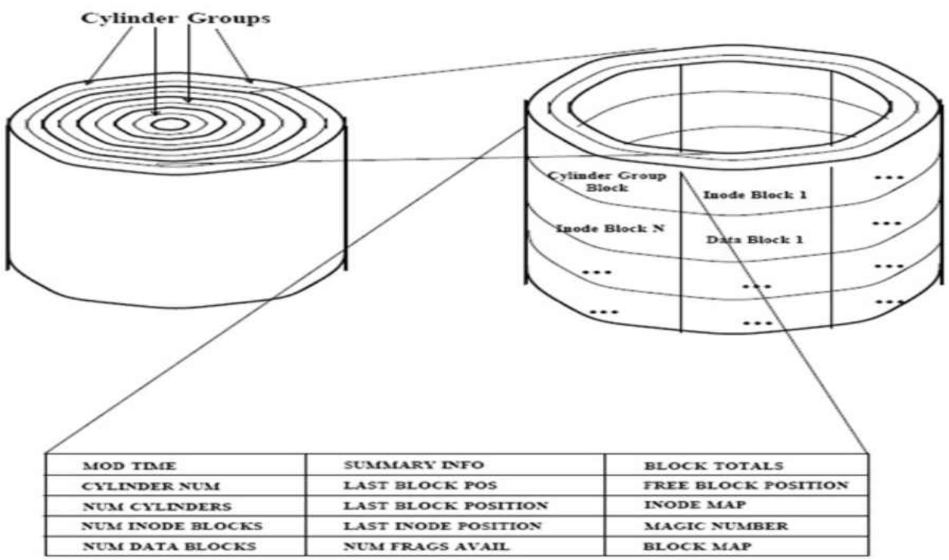


☞ **How to overcome this problem?**

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

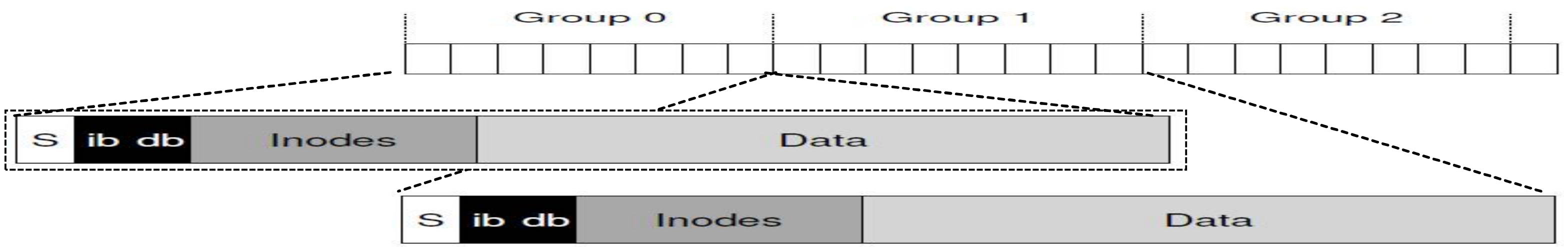




(Source: https://slideplayer.com/slide/8117044/)

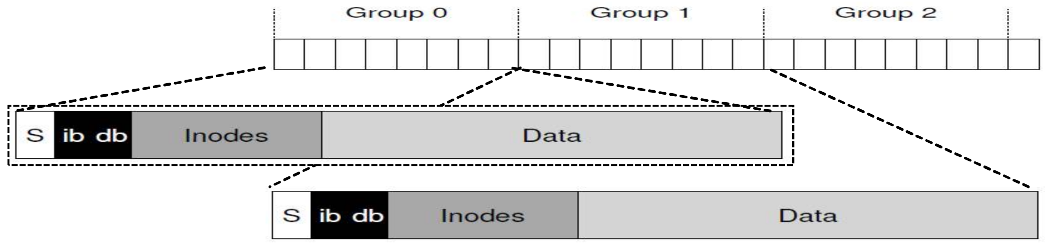
41.3 Organizing Structure: The Cylinder Group

- 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

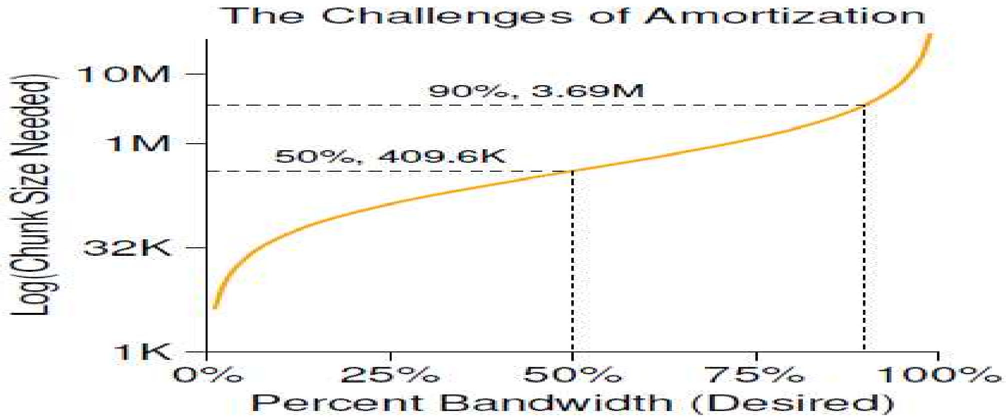


Figure 41.2: Amortization: How Big Do Chunks Have To Be?

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)
  - ✓ Sub-blocks (fragment) allocation
    - To overcome the internal fragmentation

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

  - ✓ 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

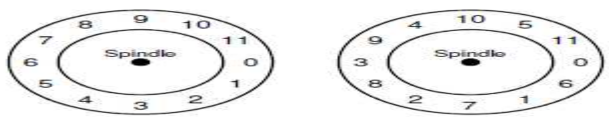
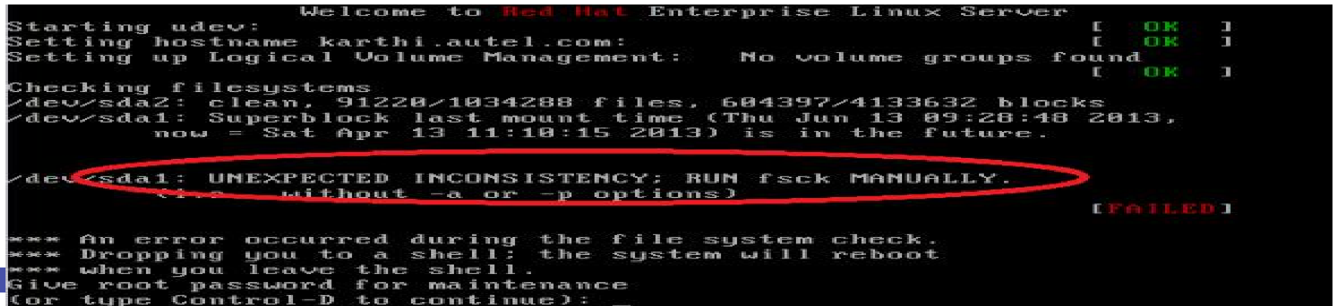


Figure 41.3: FFS: Standard Versus Parameterized Placement

  - ✓ 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
  - ✓ Can retain data while power-off
  - ✓ But, 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?
- Solutions
  - ✓ FSCK (File System Check)
  - ✓ Journaling: employed many file systems such as Ext3/4, JFS, …
  - ✓ Others: Soft update, COW, Integrity checking, Optimistic, …



Quiz for 11th-Week 1st-Lesson

- 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. Assume that we create four directories (/, /a, /b, /b/c) and four files (/a/d, /b/e, /b/c/f, /b/c/g). Also assume that each directory has 1 block while each file having 2 blocks (like 8 page). Discuss how FFS allocates these directories and files.
  - ✓ Due: until 6 PM Friday of this week (14th, May)

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:

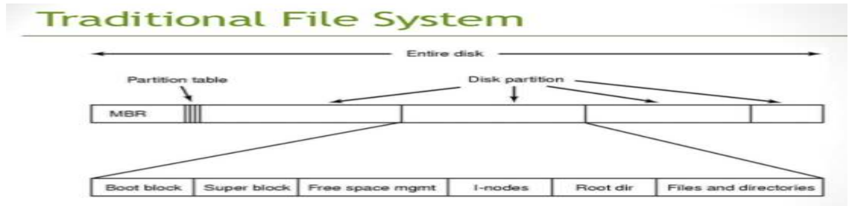
A1	A2			C1	C2		

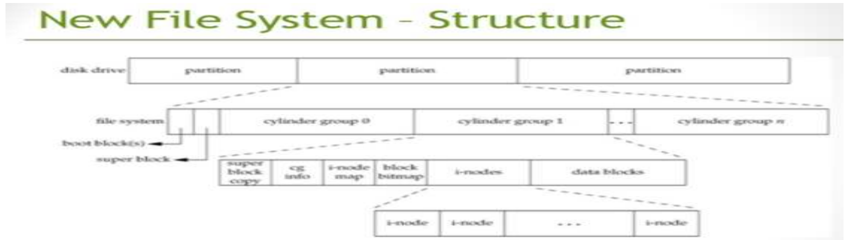
As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say you now wish to allocate a file E, of size four blocks:

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:

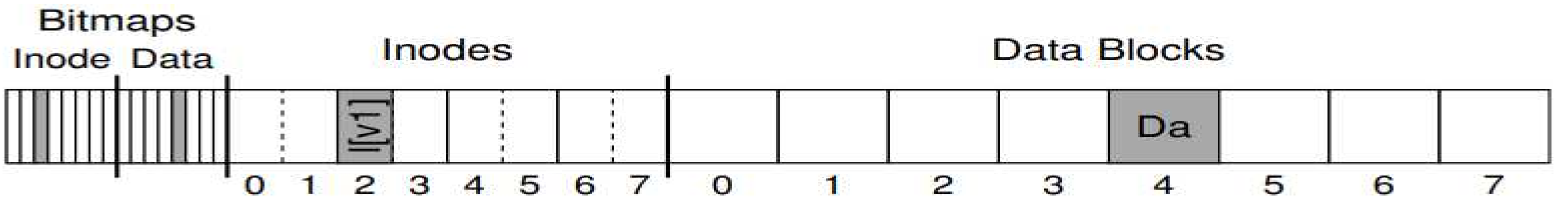




(Source: https://www.slideshare.net/parang.saraf/a-fast-file-system-for-unix-presentation-by-parang-saraf-cs5204-vt)

42.1 A Detailed Example

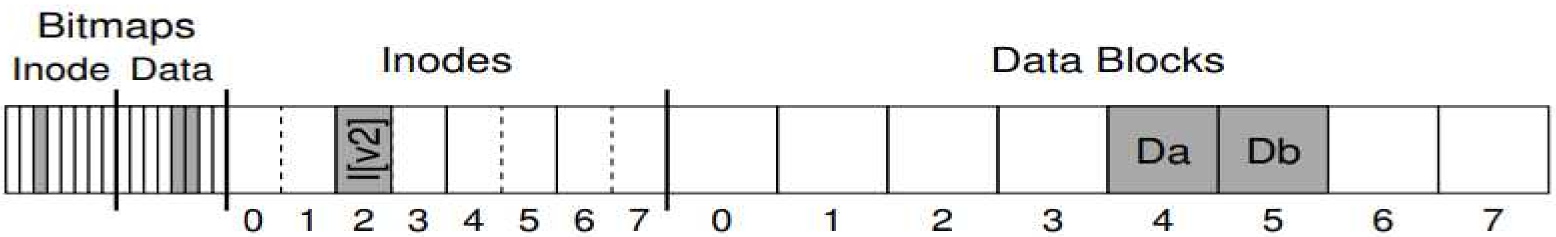
- Example
  - ✓ Simple FS: 8 inodes, 8 disk blocks, i-bitmap, d-bitmap
  - ✓ One file: size=4KB, owner =Remzi




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

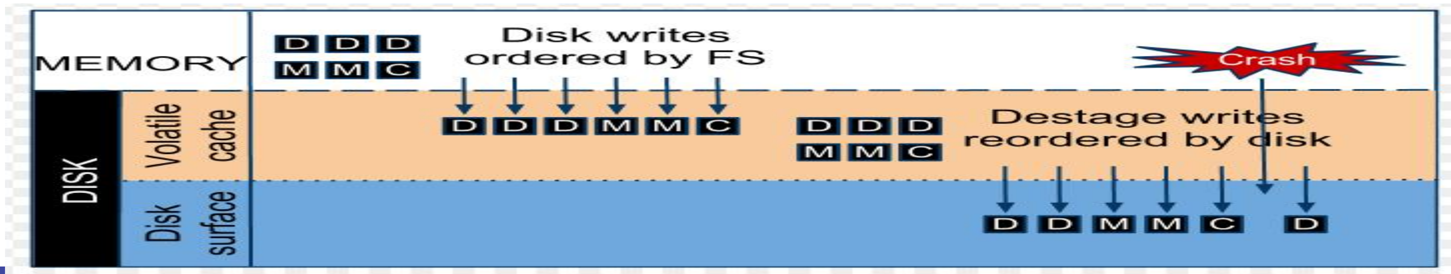
```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

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



42.1 A Detailed Example

- Crash scenario
  - ✓ 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: space leak
    - I[v2] only is written to disk: 1) garbage read, 2) inconsistency: inode vs. bitmap
    - 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)

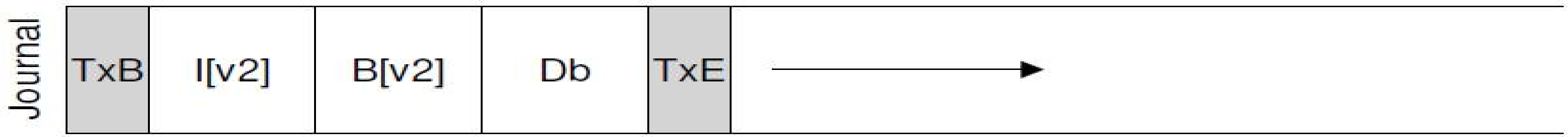
- Journaling
  - ✓ A Kind of WAL (Write-ahead logging)
  - ✓ Key idea: When updating disks, before overwriting the structure in place, first write down a little note to somewhere in a well-known location, describing what you are about to do.
  - ✓ Crash occur ➔ The note can say what you intended ➔ redo 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: Ext2 + 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.

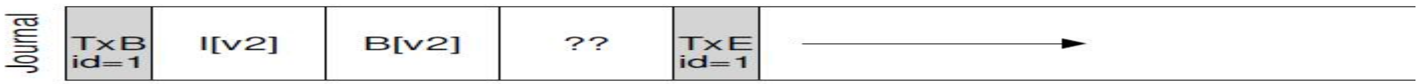


- TxB: Transaction begin, include Tid and writes information
- Log
  - Physical logging: same contents to the final locations
  - Logical logging: intent (save space, but more complex)
- TxE: End with Tid

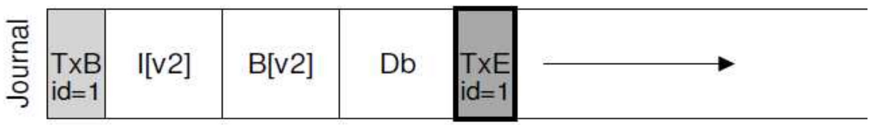
  - ✓ After making this transaction safe on disk, we are ready to update the original data ➔ step 2: checkpointing
  - ✓ Recovery (fault handling)
    - In the case of failures btw journaling and checkpointing, we can replay journal (redo) ➔ can go into the next consistent state
    - In the case of failures btw TxB and TxE, we can remove journal (undo) ➔ can stay in the previous consistent state

42.3 Solution #2: Journaling (or WAL)

- How to reduce journaling overhead? ➔ 1. performance
  - ✓ 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



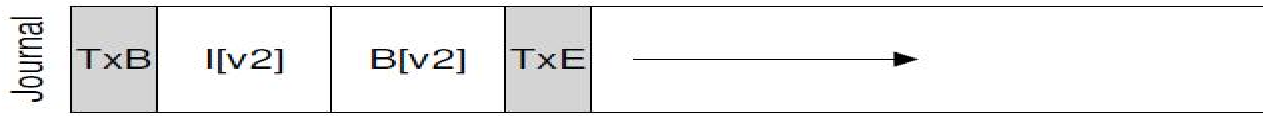
    - Transaction looks valid (it has begin and end). Thus, replaying journal leads wrong data to be updated.
  - ✓ Approach 2: issue each request at a time, wait for each to complete, then issuing the next (e.g. fsync() at each write)
    - Too slow
  - ✓ Approach 3: employ commit
    - Separate TxE from 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)

- How to reduce journaling overhead? ➔ 2. write volume
  - ✓ Data journaling writes data twice, which increases I/O traffic (reducing performance), especially painful for sequential 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 this figure, the order btw Data and Journaling is not guaranteed in the metadata journaling timeline (writeback mode in the ext3.)

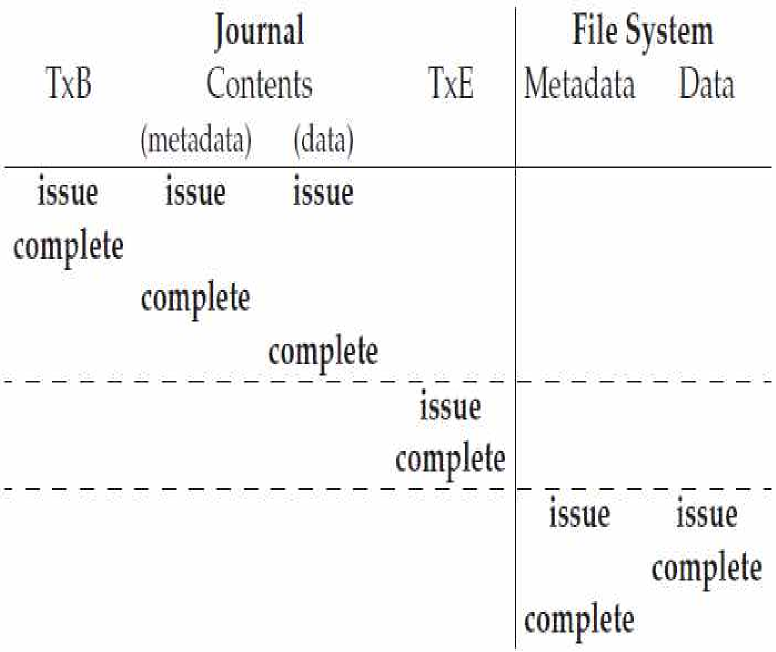


Figure 42.1: Data Journaling Timeline

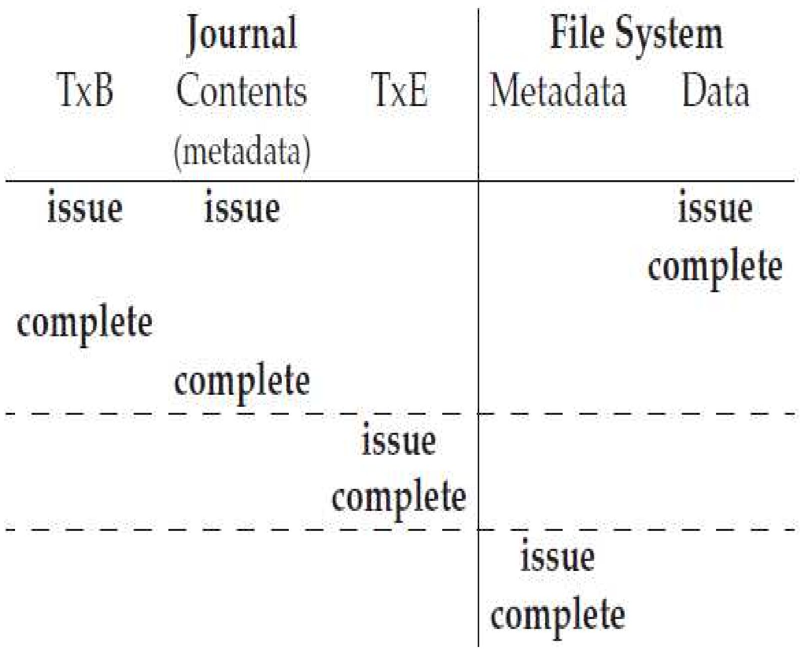
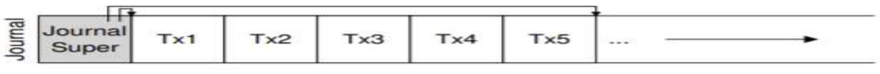


Figure 42.2: Metadata Journaling Timeline



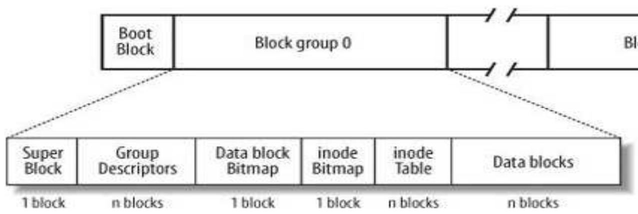
42.4 Solution #3: Other Approaches

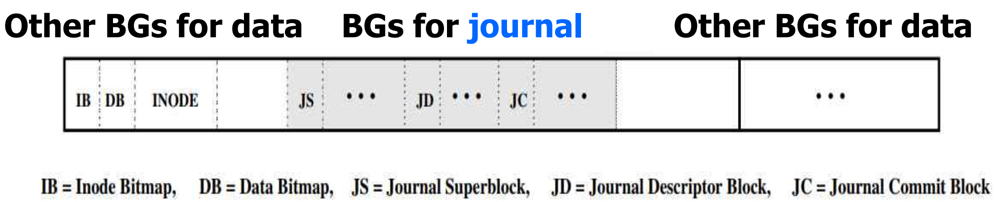
- Summary
  - ✓ fsck: A lazy approach
  - ✓ Journaling: An active approach
    - Ext3, Reiserfs, IBM's JFS, …
  - ✓ Soft update
    - Suggested by G. Ganger and Y. Patt
    - Carefully order all writes so that on-disk structures are never left in an inconsistent state (e.g. data block is always written 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

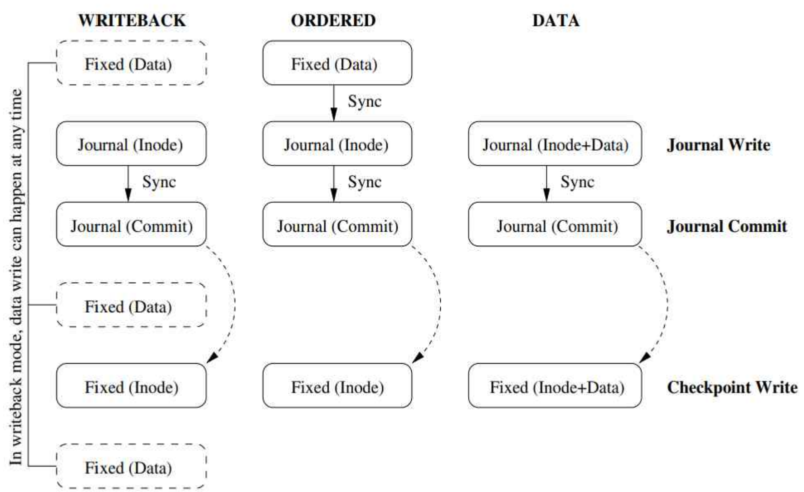
Features of Actual FS: Ext2/3/4 File System

- Ext2
  - ✓ Reference: 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
- Ext3
  - ✓ Ext2 + Journaling
  - ✓ Use a block group (or groups) for journaling
  - ✓ Three types: 1) data journal, 2) ordered, 3) writeback

Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group

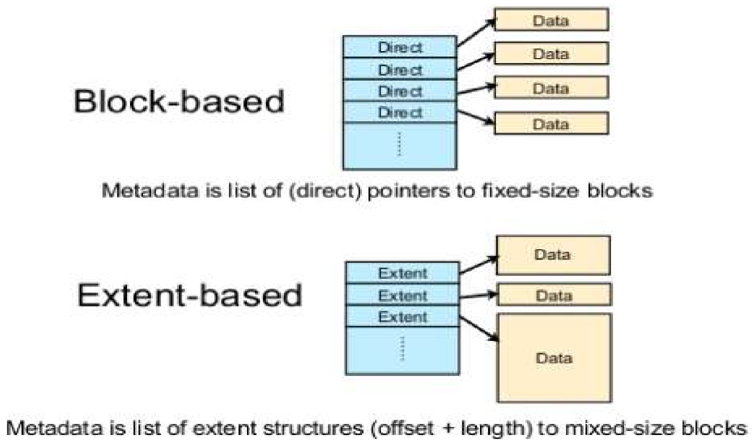


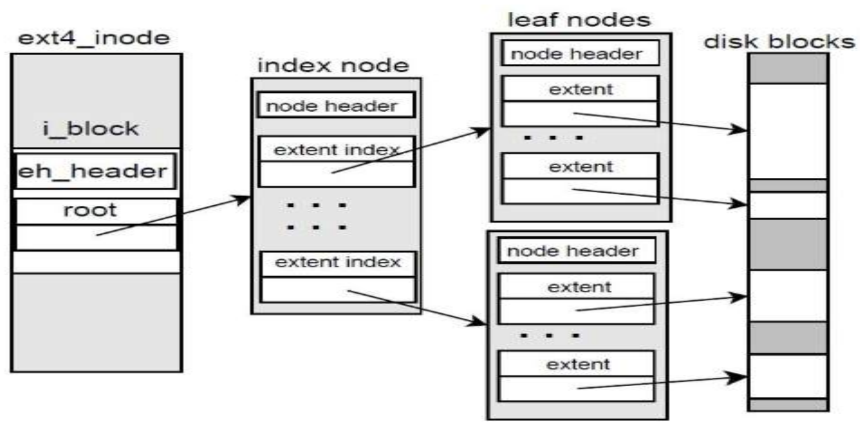




Features of Actual FS: Ext2/3/4 File System

- 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

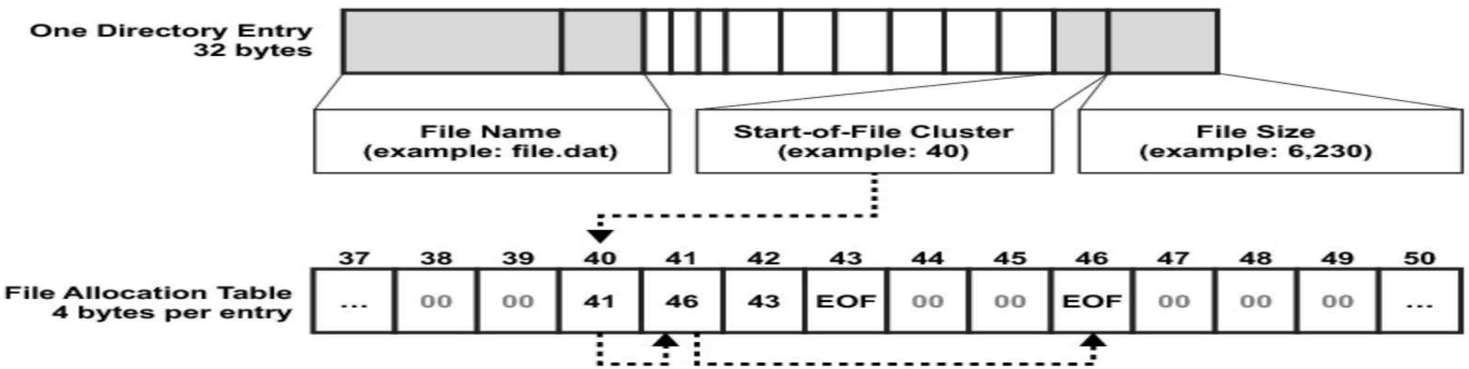




(Source: https://www.slideshare.net/relling/s8-filesystemslisa13, https://blog.naver.com/PostView.nhn?blogId=jalhaja0&logNo=221536636378)

Features of Actual FS: FAT File System

- 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

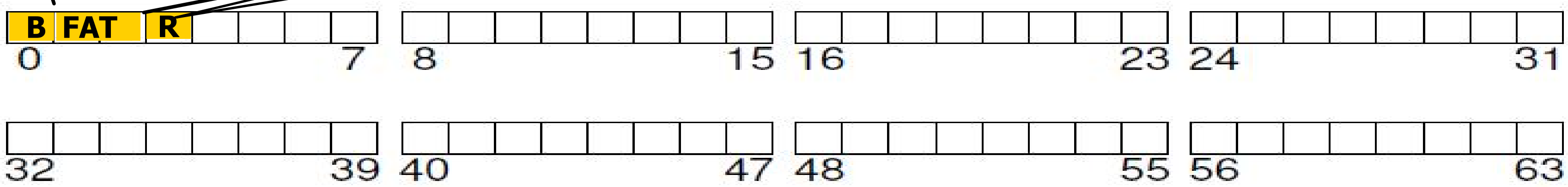


Features of Actual FS: FAT File System (optional)

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

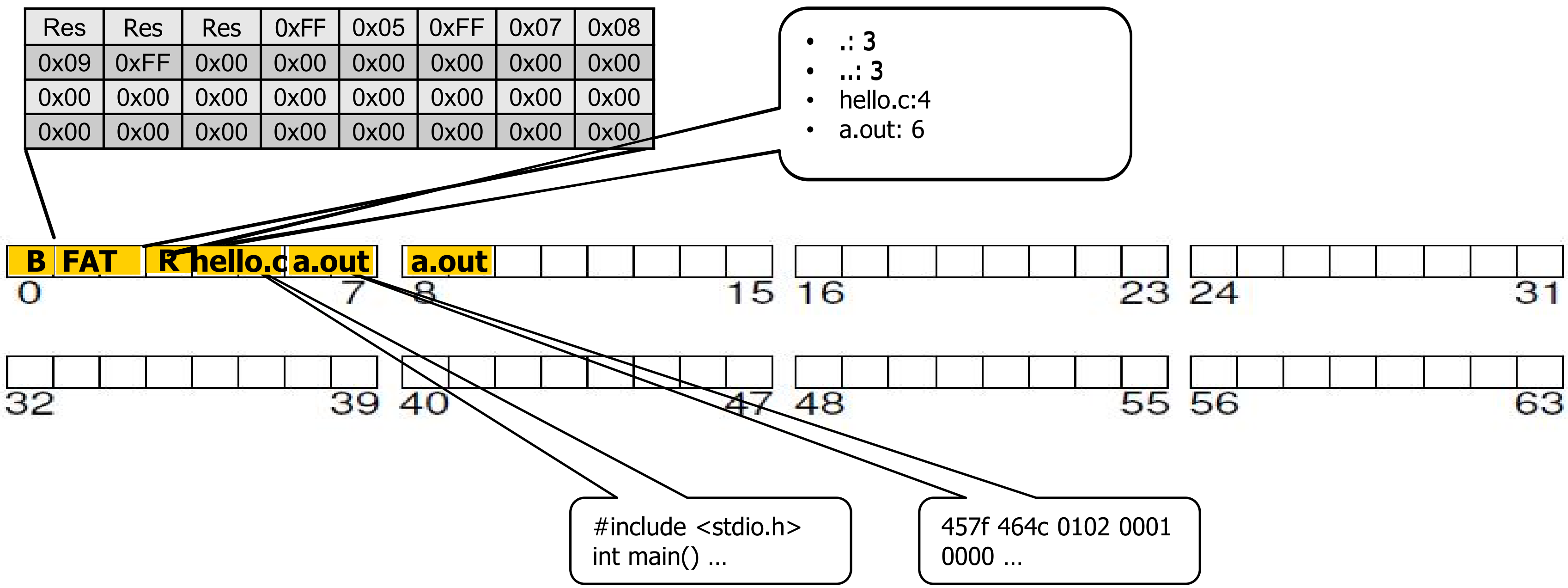
Res	Res	Res	0xFF	0x05	0xFF	0x07	0x08
0x09	0xFF	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

- .: 3
- ..: 3



Features of Actual FS: FAT File System (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)

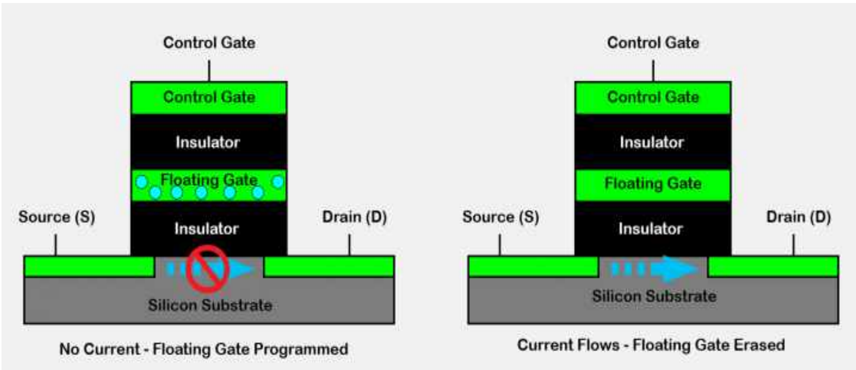


Res	Res	Res	0xFF	0x05	0xFF	0x07	0x08
0x09	0xFF	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

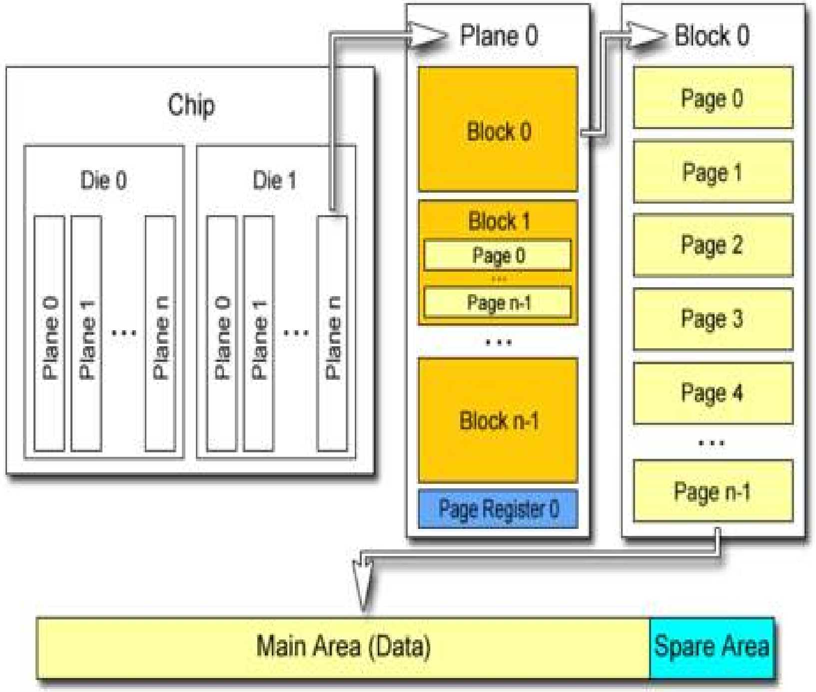
Features of Actual FS: Flash-aware FS

- Why?
  - ✓ Disk vs. Flash memory
    - Same: non-volatile
    - Different: 1) need erase operation in flash, 2) endurance, 3) read/write: small unit (4/8KB, usually called page), erase unit: large unit (512KB called block), 4) performance, mechanical, price, shock resistance, …

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



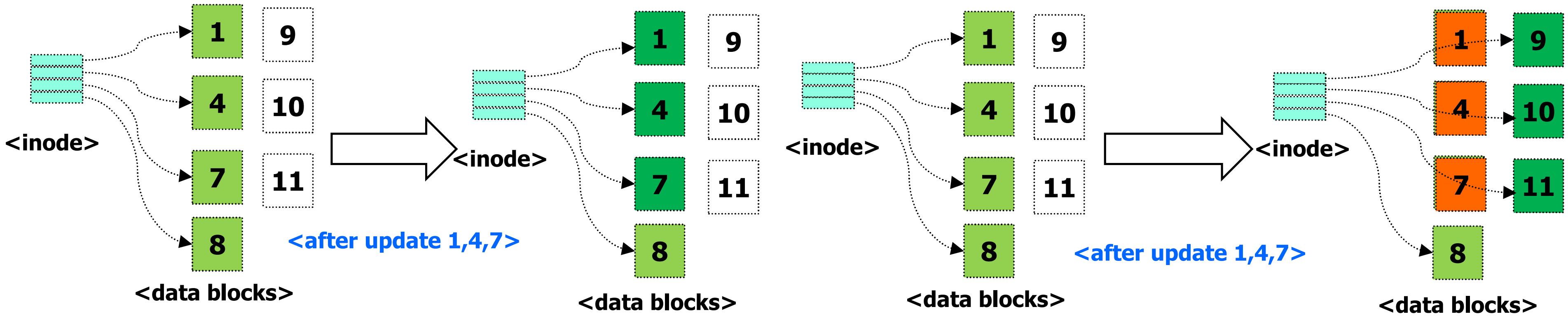
<What is erase?>



<page vs. block >

Features of Actual FS: Flash-aware FS

- Solution
  - ✓ Out place update (not in place update)
    - Allocate new disk blocks (erased) and write them ➔ mapping (address translation)
    - Reclaim the old invalidated disk blocks ➔ garbage collection
    - Example
      - A file whose size is 15KB ➔ 4 data blocks
      - Assume that disk blocks 1, 4, 7 and 8 are allocated for the file
      - A user modify the file ranging from 0 to 10KB
      - In place update ➔ write on the already allocated blocks
      - Out place update ➔ allocate new blocks and write on them
  - ✓ Real file systems: F2FS, LFS



<In place update> **<Out place update>**

Features of Actual FS: Flash-aware FS (Optional)

- F2FS: Flash Friendly FS by Samsung
  - ✓ Key idea
    - Use inode (like FFS) and Out-place update (like LFS)
    - Make new mapping in an inode and Invalidate old data (Translation)
    - Garbage collection to reclaim invalidated blocks
  - ✓ New features
    - 1) Multiple logging: for hot/cold separation, 2) NAT: to prevent wandering tree problem, 3) GC Optimization (Fore vs background, greedy vs cost-benefit)
  - ✓ Note: General FS (e.g. Ext4) + FTL (Flash Translation Layer)
    - FTL: Abstract flash memory like disks ➔ See Chapter 44 in OSTEP

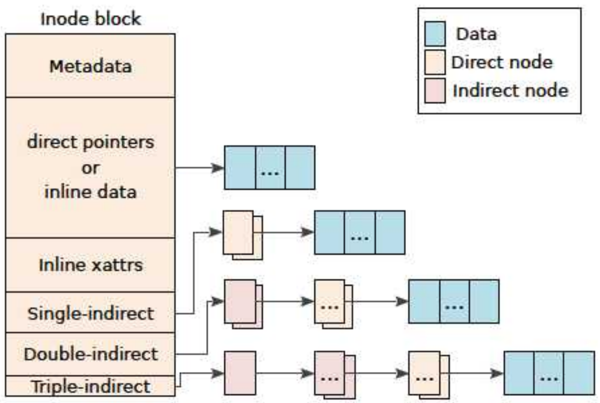
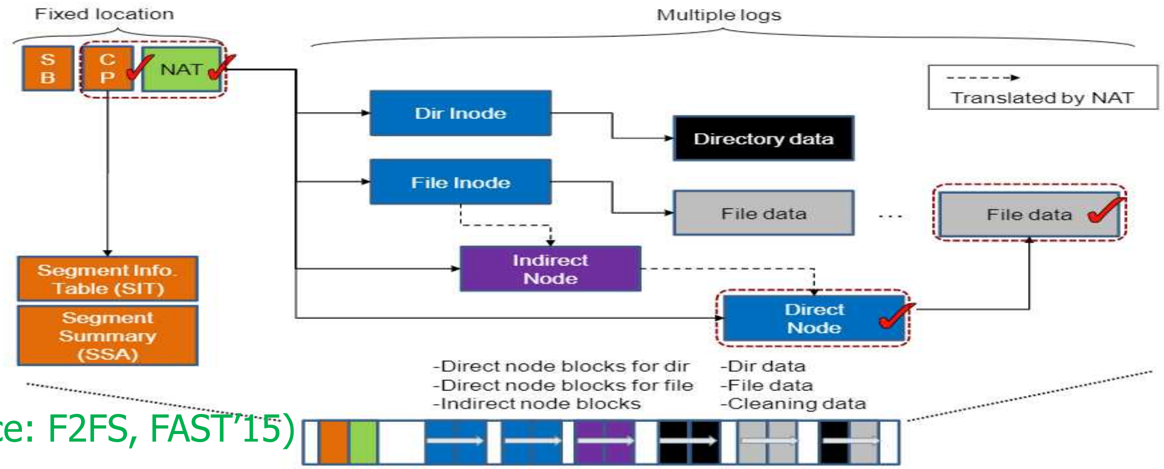


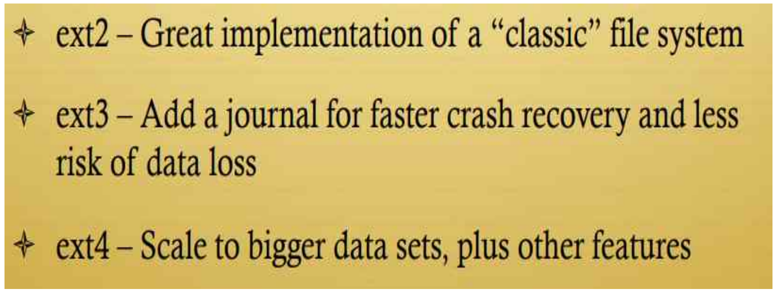
Figure 2: File structure of F2FS.

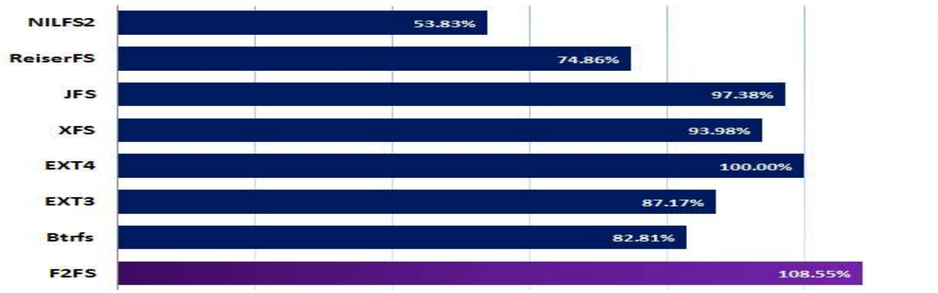


(Source: F2FS, FAST'15)

Summary

- File basic
  - ✓ Layout: superblock, bitmap, inode, data blocks
  - ✓ Access methods: open(), read(), write(), …
- Optimization
  - ✓ Performance: FFS, Ext2, …
    - A watershed moment in file system research
    - Storage-awareness, simple but effective techniques
  - ✓ Consistency: Ext3/4, JFS, …
    - Change from valid state to another valid state
    - Journaling: Performance and Reliability tradeoff
- Others
  - ✓ F2FS: for Flash memory file
  - ✓ FAT: for small storage

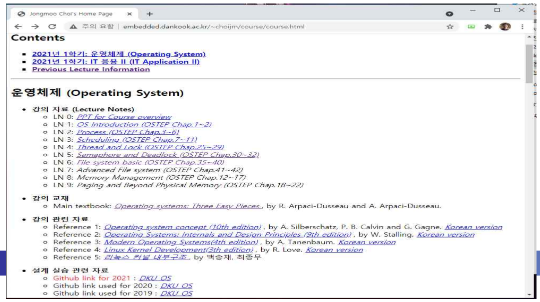


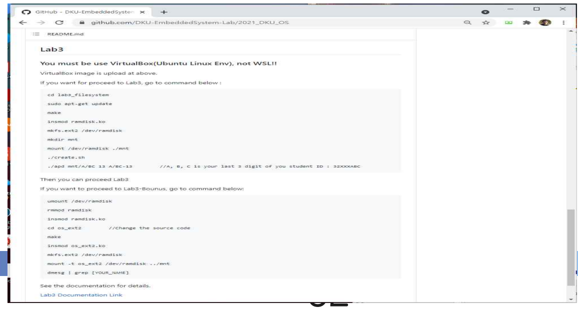


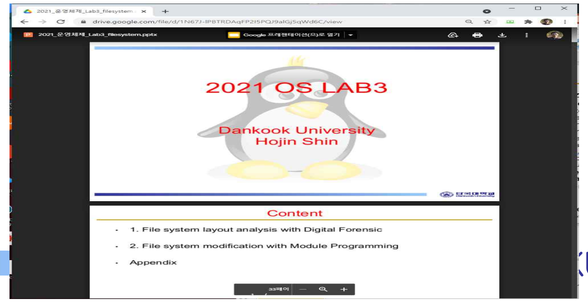
(Source: https://www3.cs.stonybrook.edu/~porter/courses/cse506/f14/slides/ext4.pdf)

Lab3 : Ext2 Analysis

- Lab3: Analyze Ext2 file system internal (a kind of digital forensic!!)
  - ✓ What we need to do
    - 1. create ramdisk
    - 2. make ext2 file system on ramdisk
    - 3. mount the ext2 file system
    - 4. run the script on the mount directory (./create.sh) ➔ will generate dirs. and files
    - 5. find a file assigned to you and find blocks allocated for the file
      - Assigned files: last three digits of a student number ➔ directory + file name
      - (e.g. *****236 ➔ directory name is 2, file name is 36)
    - 6. dump ramdisk (using xxd), examine Ext2 (or make a program that parsing Ext2)
      - Superblock ➔ Group descriptor ➔ root inode ➔ root data ➔ dir. inode ➔ dir. data ….
  - ✓ Requirement: report ➔ 1) goal, 2) analysis results and snapshots, 3) discussion
  - ✓ Submission: 1) upload e-learning campus, 2) email to TA
  - ✓ Due: 6pm, 28th May (Friday)
  - ✓ Bonus
    - Print your name and student id while mounting Ext2
    - Ext2 source modification + make + module insert (e.g. insmod) + mkfs + mount







Appendix 1: Lab3 details

- 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
```

(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
```

(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
```

(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  ................
```

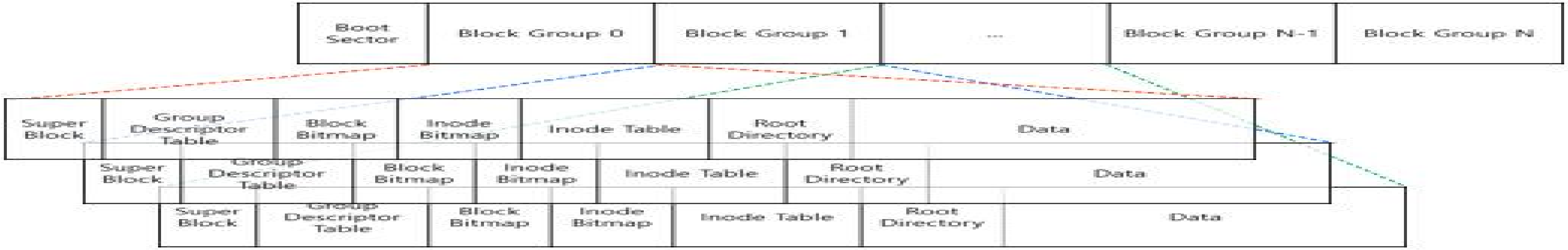
```
root@ESL-LeeJY:/home/sys32153550/workspace/2020_1/OS_Lab3# xxd -g 4 -l 0x100 -s 0x10c67000 /dev/ramdisk
10c67000: 352f3530 2d330a00 00000000 00000000  5/50-3..........
10c67010: 00000000 00000000 00000000 00000000  ................
```

```
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  ................
```

(Explore file system layout)

Appendix 1: Lab3 details

- 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)															
a0																
b0																
c0									algorithm usage bitmap				prealloc block	prealloc dir block	padding	
d0	journal uuid															
e0	journal inode number				journal device				last orphan							
f0	hash seed (16 byte)												default hash version	pad	padding	
100	default mount option				first meta block											

(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				block pointer (60 byte)							
30																
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)

■ 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# 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/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# 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
```

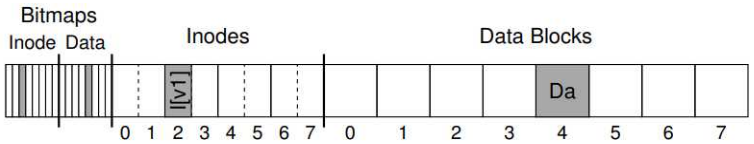
(mkfs and mount)

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

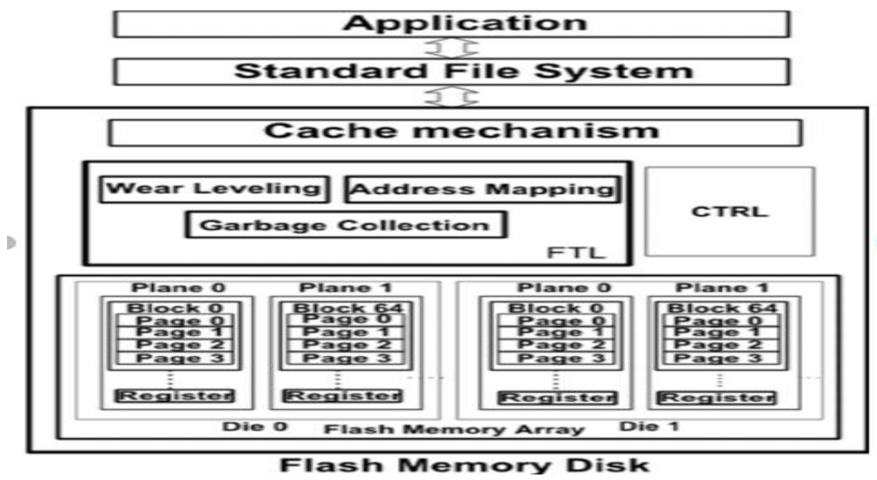
(Your name and student ID are printed out at the kernel level NOT at the user level!!)

Quiz for 11th-Week 2nd-Lesson

- Quiz
  - ✓ 1. We want to create a file whose size is 4KB, as shown in the below left figure. Using the figure, explain the terms of “inconsistent”, "space leak“, “garbage read“ and “dangling reference”.
  - ✓ 2. FTL (Flash Translation Layer) is a SW layer that abstracts flash memory like disks. Three key roles of FTL are 1) address mapping, 2) garbage collection and 3) wear-leveling. Explain these roles (refer to Chapter 44 in OSTEP).
  - ✓ Due: until 6 PM Friday of this week (14th, May)




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



(Source: https://www.secmem.org/blog/2020/01/17/FTL/)

Appendix 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, …

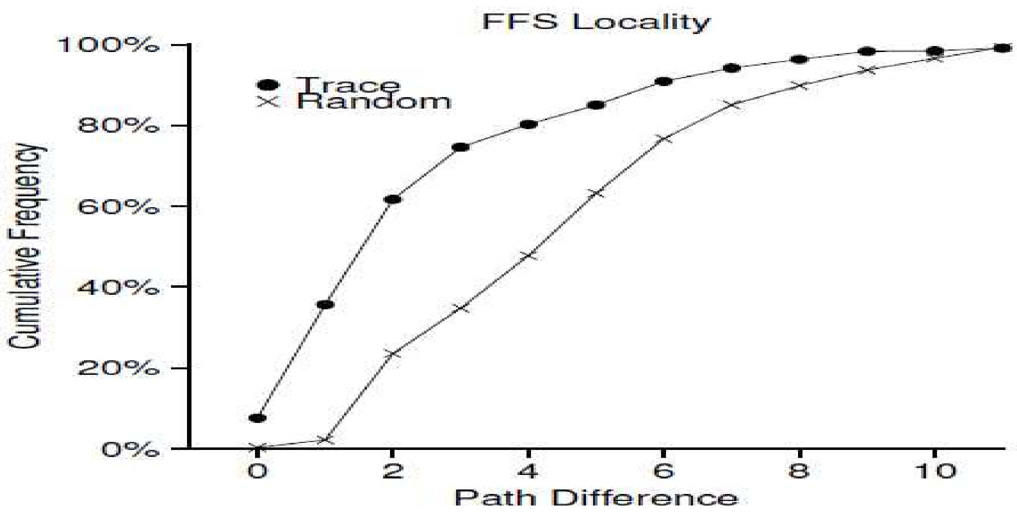
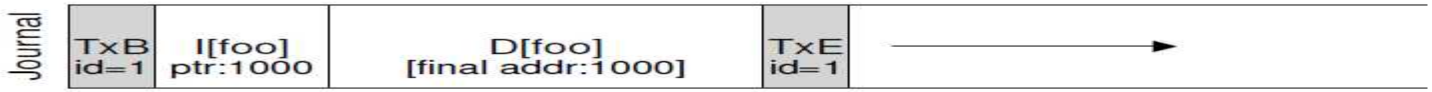


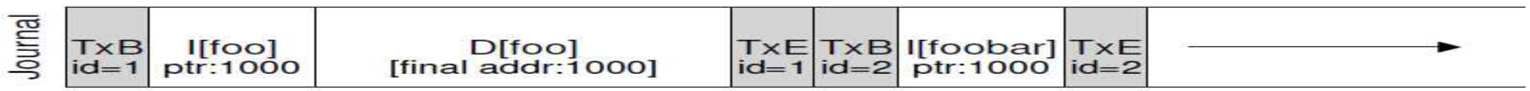
Figure 41.1: FFS Locality For SEER Traces

Appendix 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)



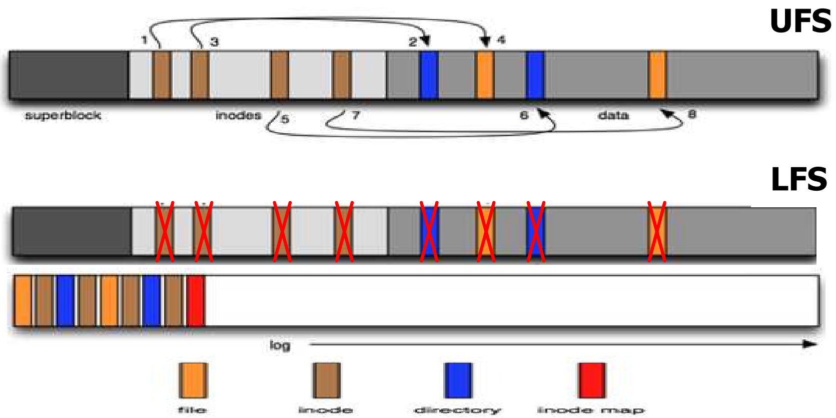
  - ✓ 5) The user deletes the foo (and its subfiles), 6) The user creates another file (say foobar), which uses the block 1000, 7) Writes for foobar are logged (note that file contents themselves are not logged)

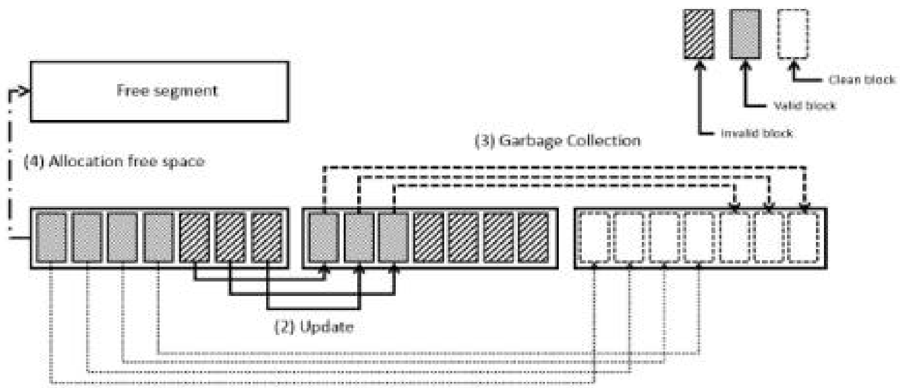


  - ✓ 8) At this point, a crash occurs. 9) recovery performs "redo" from the beginning of the log. 10) overwrites the user data of the file foobar with the old directory contents.
  - ✓ Solution
    - Ext3 adds a new type of record, a revoke record, for the deleted file or directory. When do replaying,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

Appendix 2

- Features of Actual FS: LFS (Log-Structured File System)
  - ✓ Why? How to reduce seek distance?
    - Allocate related data as close as possible: FFS, Ext2, …
    - But, eventually fragmentation occurs
      - E.g.) create a file 1 in a dir1, and a file 2 in a dir 2 ➔ 8 random writes in FFS
  - ✓ Proposal: write data sequentially in new place (log) instead of original place (out-place update vs in-place update)
    - Need to add new mapping information (inode map)
      - E.g.) create a file 1 in a dir1, and a file 2 in a dir 2 ➔ 8+1 sequential writes in LFS
      - Original data ➔ invalidate
    - Need garbage collection for reclaiming invalidated data





(Source: https://work.tinou.com/2012/03/log-structured-file-system-for-dummies.html, https://deepai.org/publication/ssdfs-towards-lfs-flash-friendly-file-system-without-gc-operation)