Ext4 file system and crash consistency

Dongyoon Lee

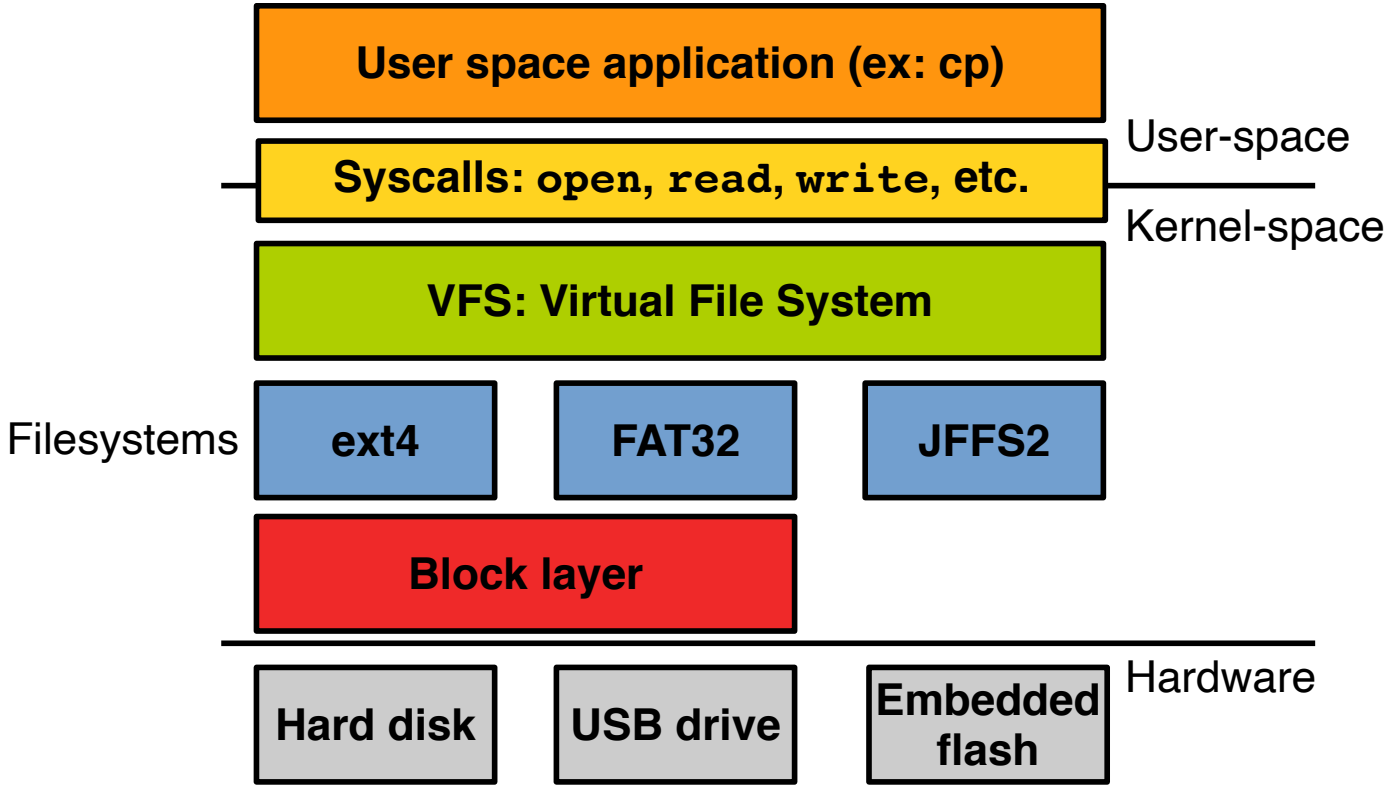
Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management
- Virtual file system
- Page cache and page fault

Today: ext4 file system and crash consistency

- File system in Linux kernel
- Design considerations of a file system
- History of file system
- On-disk structure of Ext4
- File operations
- Crash consistency

File system in Linux kernel



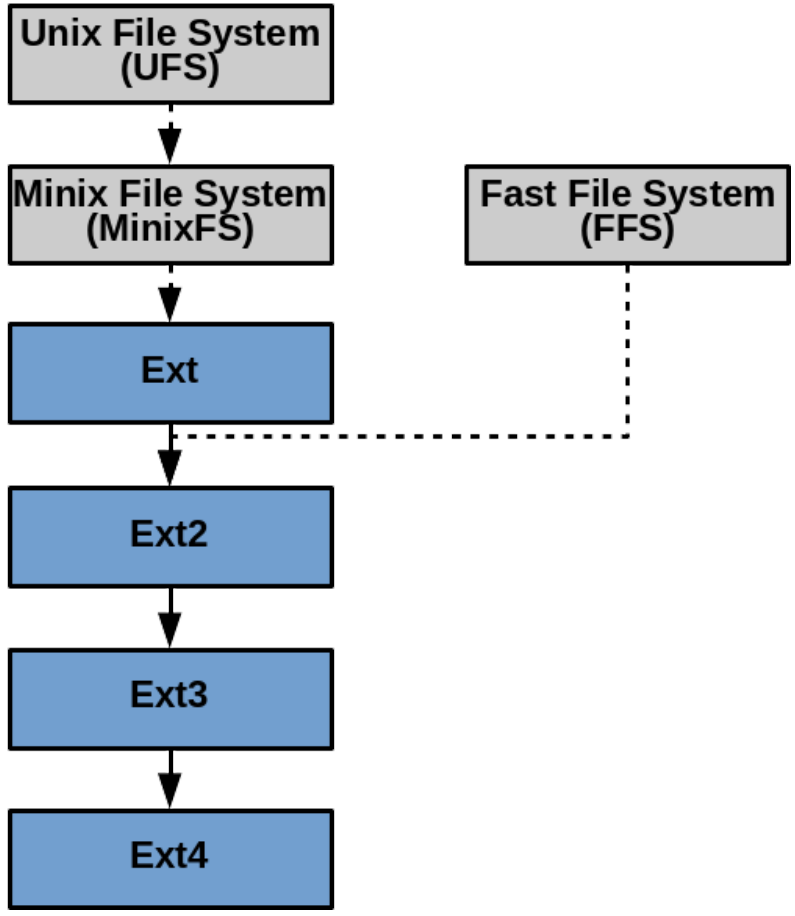
What is a file system fundamentally?

```
int main(int argc, char *argv[])
{
    int fd;
    char buffer[4096];
    struct stat_buf;
    DIR *dir;
    struct dirent *entry;

    /* 1. Path name -> inode mapping */
    fd = open("/home/lkp/hello.c" , O_RDONLY);

    /* 2. File offset -> disk block address mapping */
    pread(fd, buffer, sizeof(buffer), 0);

    /* 3. File meta data operation */
    fstat(fd, &stat_buf);
    printf("file size = %d\n", stat_buf.st_size);

    /* 4. Directory operation */
    dir = opendir("/home");
    entry = readdir(dir);
    printf("dir = %s\n", entry->d_name);
    return 0;
}
```

Why do we care EXT4 file system?

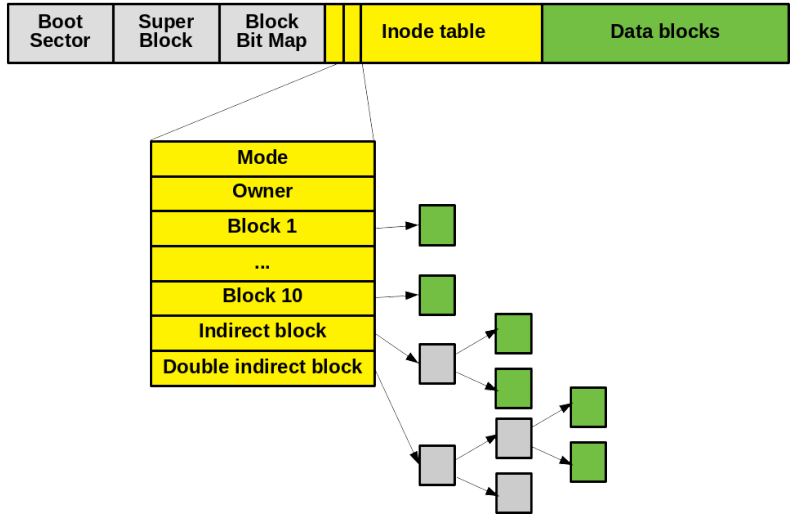
- **Most widely-deployed file system**
  - Default file system of major Linux distributions
  - File system used in Google data center
  - Default file system of Android kernel
- **Follows the traditional file system design**

History of file system design



UFS (Unix File System)

- The original UNIX file system
  - Design by Dennis Ritche and Ken Thompson (1974)
- The first Linux file system (ext) and Minix FS has a similar layout



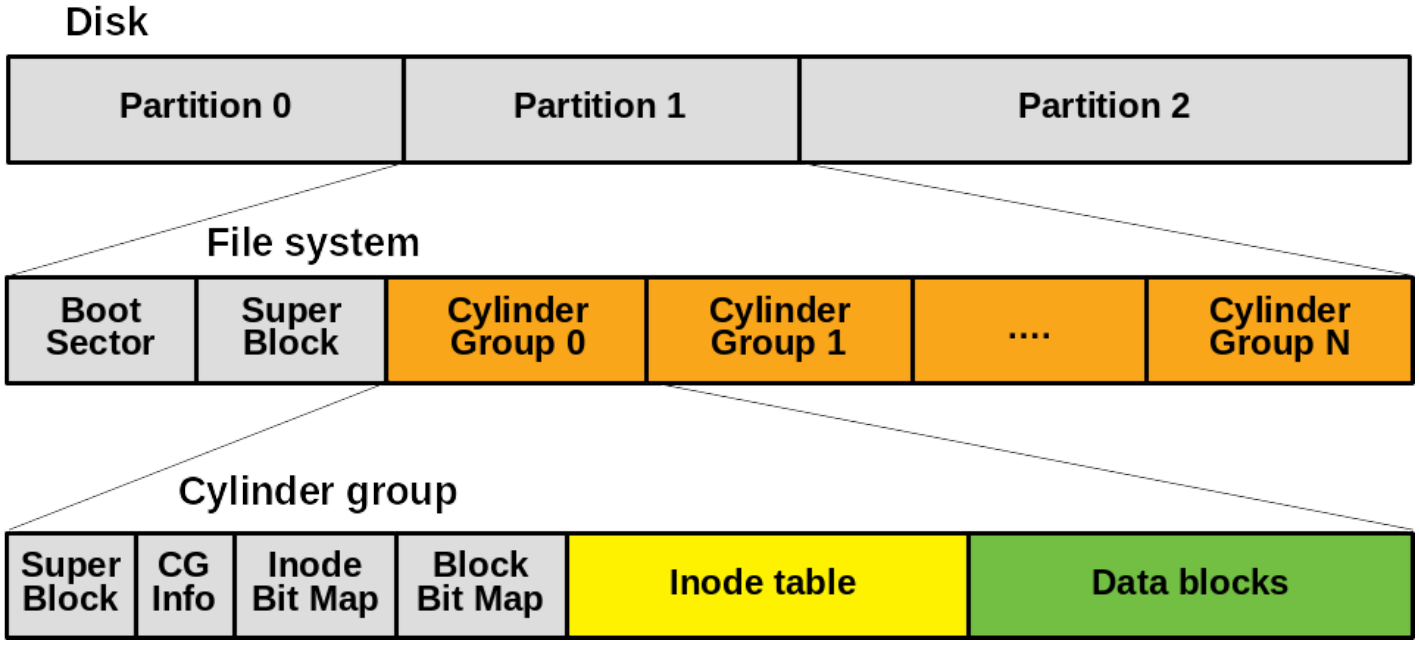
UFS (Unix File System)

- Performance problem of UFS (and the first Linux file system)
  - Especially, long seek time between an inode and data block

FFS (Fast File System)

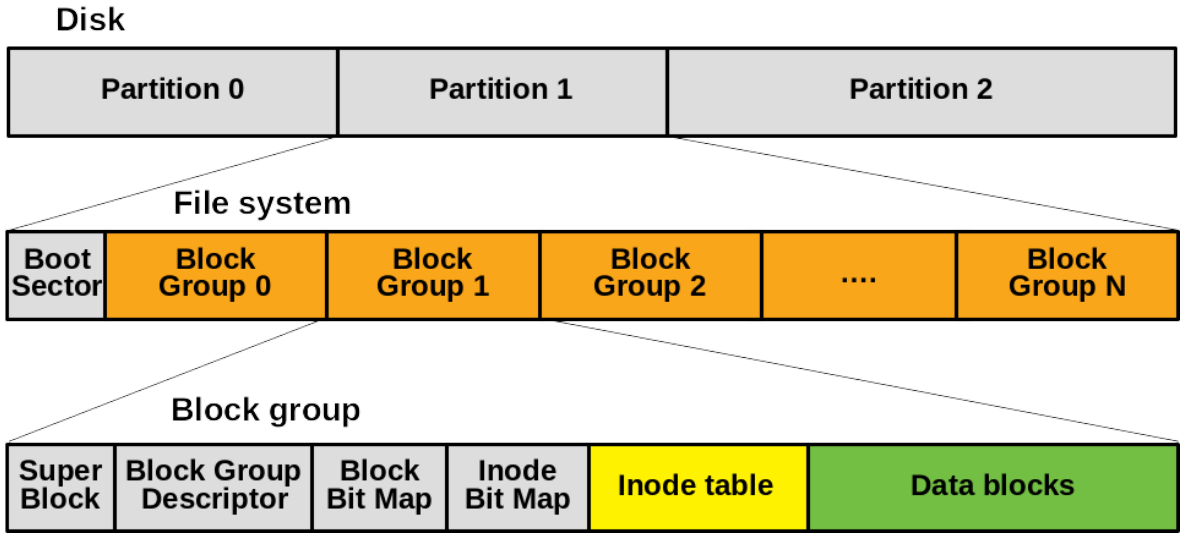
- The file system of BSD UNIX
  - Designed by Marshall Kirk McKusick, et al. (1984)
- Cylinder group: one or more consecutive cylinders
  - Disk block allocation heuristics to reduce seek time
  - Try to locate inode and associate data in the same cylinder group
- Many in-place-update file systems follows the design of FFS

FFS (Fast File System)

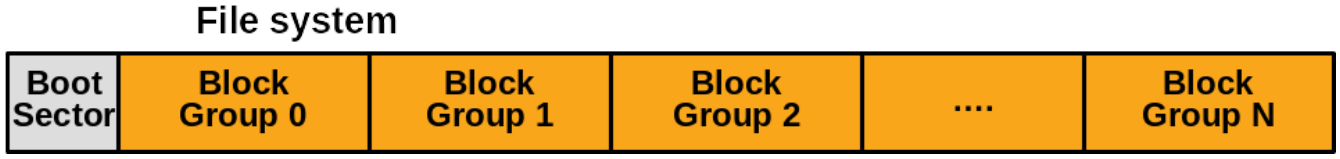


Ext2 file system

- The second extended version (ext2) of the first Linux file system (ext)
- Design is influenced by FFS in BSD UNIX
  - block group (similar to cylinder group) to reduce disk seek time

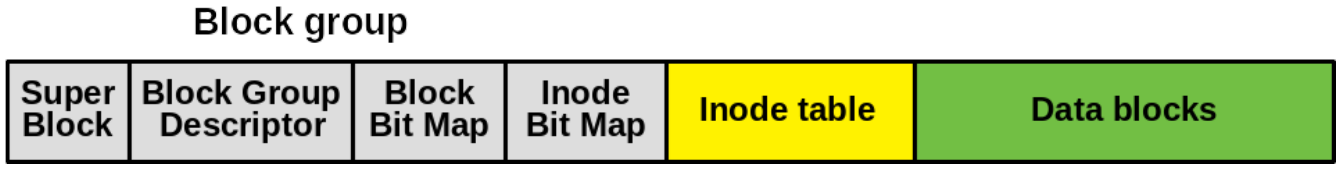


Ext2 file system



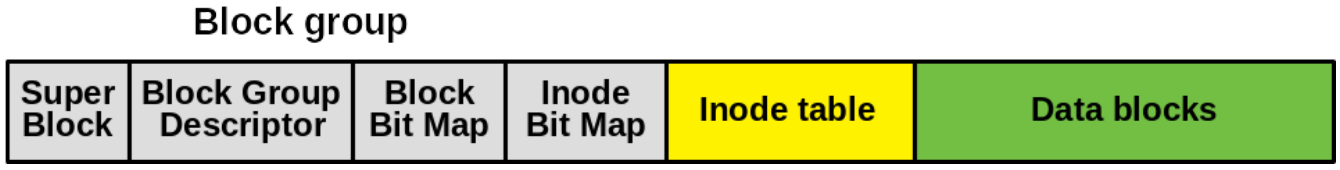
- The first block is reserved for the boot sector (not managed by file system)
- The rest of a Ext2 partition is split into block groups
- All the block groups have same size and are stored sequentially

Ext2 file system



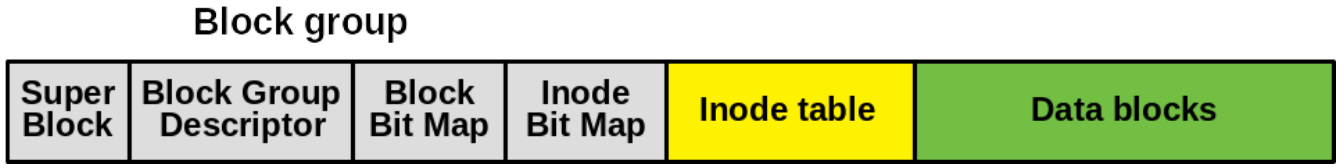
- **Superblock**: stores information describing filesystem
  - E.g., block size, number of inodes and blocks, number of free blocks in a file system, etc.
  - Each block group has a copy of superblock for recovery
- **Group Descriptor**: describe each block group information
  - E.g., number of free blocks in a block group, free inodes, and used directories in the group

Ext2 file system



- **Inode bitmap**: summarized inode usage information
  - 0 → free, 1 → in use
  - 100th bit in inode bitmap = usage of 100th inode in inode table
- **Block bitmap**: summarized data block usage information
  - 0 → free, 1 → in use
  - 100th bit in block bitmap = usage of 100th data block in a block group

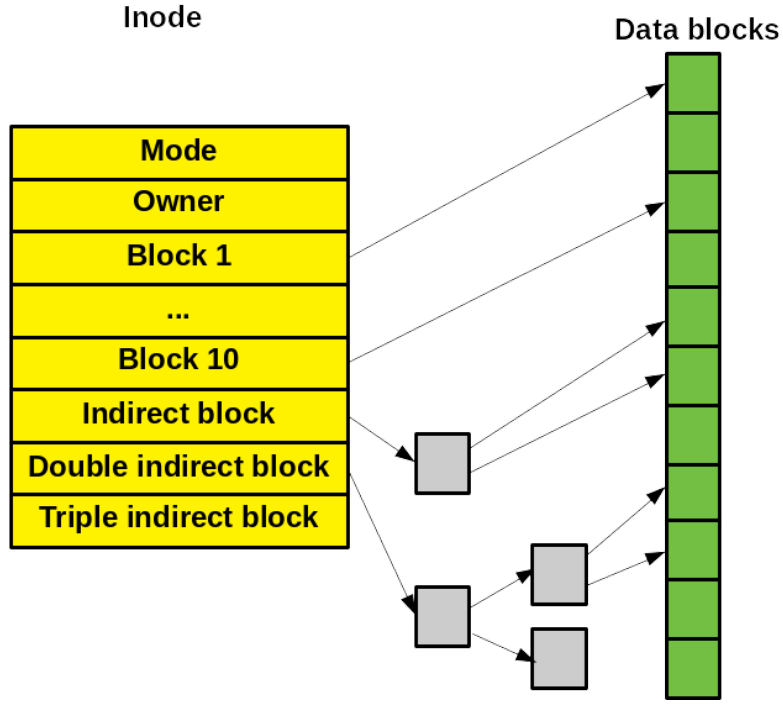
Ext2 file system



- **Inode table**: fixed size array of inodes
  - inode number = starting inode number of a block group + inode location in inode table
- **Data blocks**: actual data contents

Ext2 file system

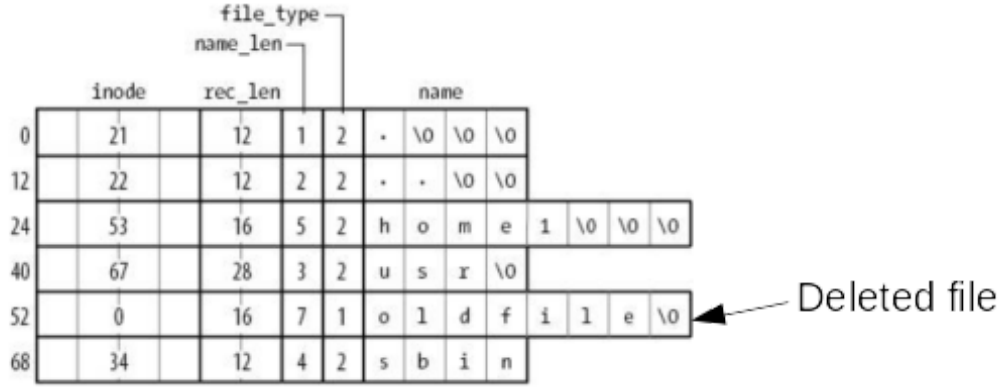
- **Inode indirect block map**: file offset → disk block address mapping



Ext2 file system

- **Directory: a liked list**
  - Directory access performance with many files is terrible

```
$> ls
.     home     sbin
..    usr
```



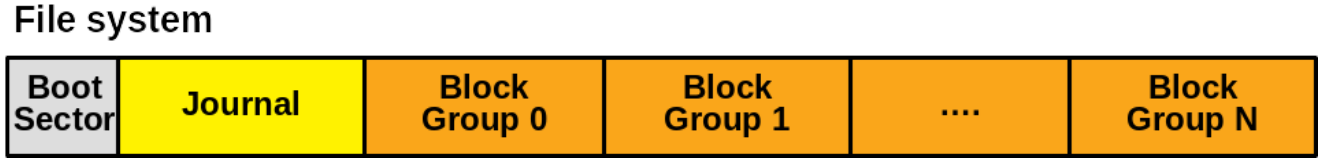
Ext2 file system

- **Q: How to open `/home/lkp/hello.c`**
  - `/` → `home/` → `lkp/` → `hello.c`
- **Q: Where to start?**

```
/* linux/fs/ext2/ext2.h */

/*
 * Special inode numbers
 */
#define EXT2_BAD_INO             1  /* Bad blocks inode */
#define EXT2_ROOT_INO            2  /* Root inode */
#define EXT2_BOOT_LOADER_INO     5  /* Boot loader inode */
#define EXT2_UNDEL_DIR_INO       6  /* Undelete directory inode */
```

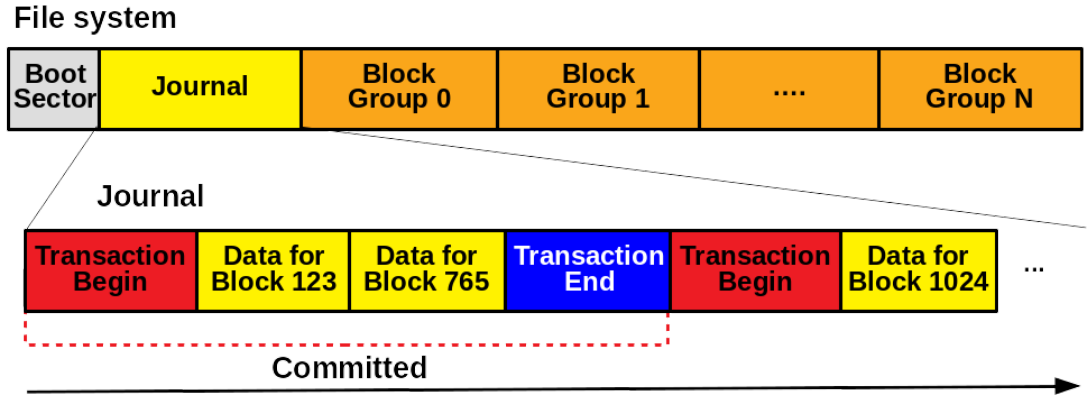
Ext3 file system

- Integrate **journaling** into ext2 file system to guarantee file system consistency after sudden system crash/power-off
- Journaling = WAL (Write-Ahead Loggin) in database systems
- When updating disks, before overwriting the structure in place, first write down what you are about to do in a well-known location.



Ext3 file system: journaling

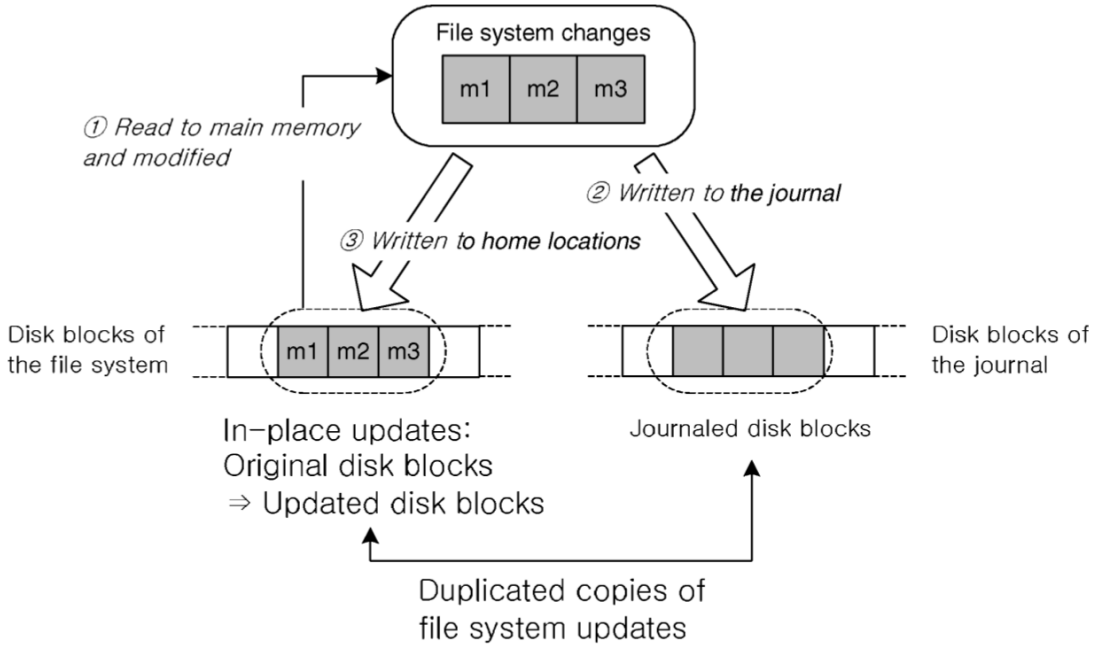
- Journal is logically a fixed-size, circular array
  - Implemented as a special file with a hard-coded inode number
  - Each journal transaction is composed of a begin marker, log, and end marker



Ext3 file system: journaling

- **Commit**
  1. Write a transaction begin marker
  2. Write logs
  3. **Write barrier**
  4. Write a transaction end marker
- **Checkpointing**
  - After ensuring this journal transaction is written to disk (i.e., commit), we are ready to update the original data

Ext3 file system: journaled write



- Source: JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory

Ext3 file system: journal recovery

- **Case 1: failure between journal commit and checkpointing**
  - There is a transaction end marker at the tail of the journal → logs between begin and end markers are consistent
  - Replay logs in the last journal transaction
- **Case 2: failure before journal commit**
  - There is no transaction end marker at the tail of the journal → there is no guarantee that logs between begin and end markers are consistent
  - Ignore the last journal transaction

Ext3 file system: journaling mode

- **Journal**
  - Metadata and content are saved in the journal.
- **Ordered**
  - Only metadata is saved in the journal. Metadata are journaled only after writing the content to disk. This is the default.
- **Writeback**
  - Only metadata is saved in the journal. Metadata might be journaled either before or after the content is written to the disk.

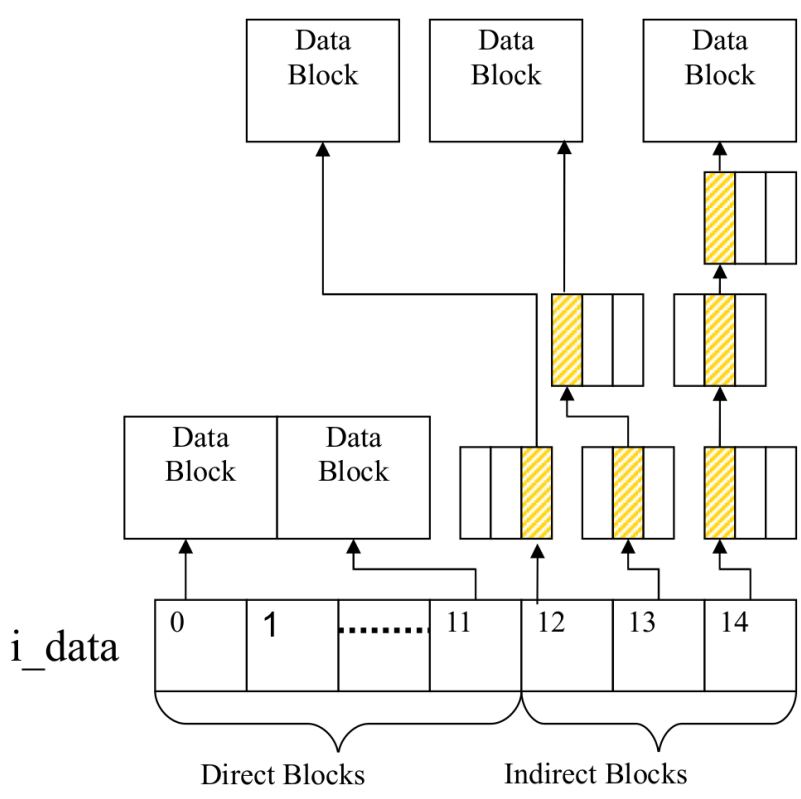
Ext4 file system

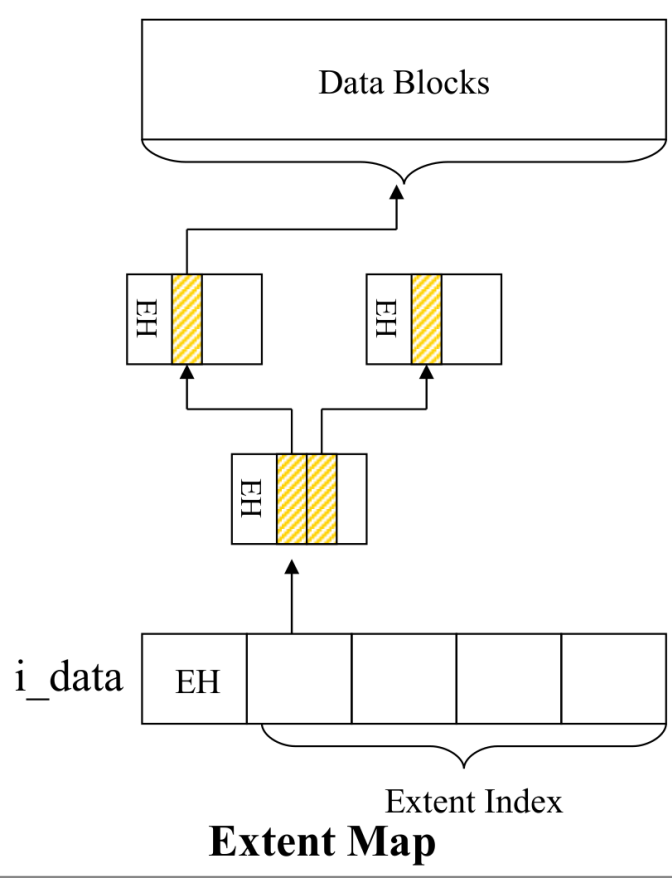
- Larger file system capacity with 64-bit
- Supports huge file size (from 16GB to 16TB) and file system (from 2^32 to 2^64 blocks)
  - **Indirect block map → extent tree**
- Directory can contain up to 64,000 subdirs (32,000 in ext3)
  - **List → HTree**

Indirect Block Map vs Extent map

- **Indirect block map**
  - Incredibly inefficient
  - One extra block read (and seek) every 1024 blocks
  - Really obvious when deleting large movie files
- **Extents**
  - An efficient way to represent large file
  - A single descriptor for a range of contiguous blocks
  - E.g., {file block offset, length, disk block address} = {100, 1000, 200}

Indirect Block Map vs Extent map





Extent related works

- Multiple block allocation
  - Allocate contiguous blocks together
  - Reduce fragmentation, reduce extent metadata
- Delayed allocation
  - Defer block allocation to writeback time
  - Improve chances allocating contiguous blocks, reducing fragmentation

HTree (hashed b-tree)

- Directory structure of ext3/4
- B-tree using hashed value of a file name in a directory as a key

Crash consistency in file system

- Crash consistency
- Journaling file system
- Log-structured file system
- Copy-on-Write file system

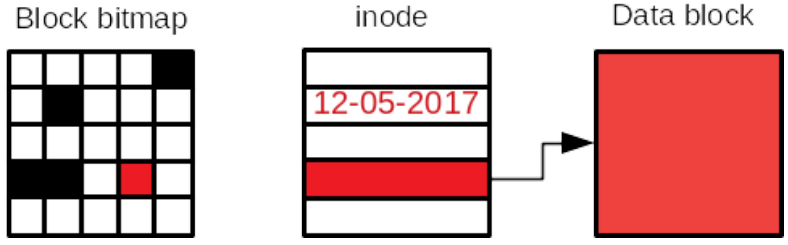
The problem: crash consistency

- Single file system operation updates **multiple** disk blocks
  - Classic storage device interface is block-based
  - While we can submit many writes, only one block is processed at a time
- System might crash in the **middle** of operation
  - Some blocks updated, some blocks not updated
  - Inconsistent file system
- After crash, file system need to be reparied
  - In order to restore **consistency** among blocks

Example: File Append

```
$ echo "hello" >> log.txt
```

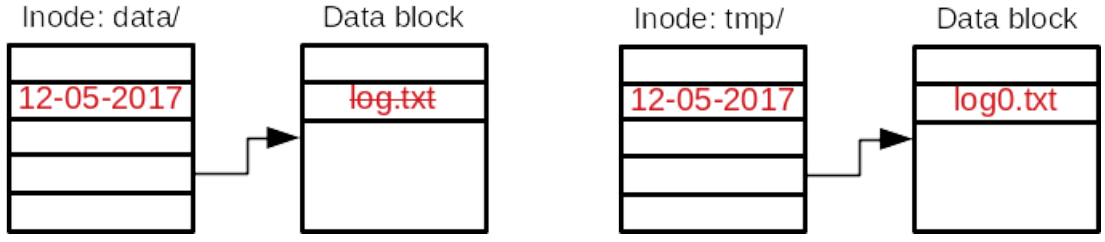
1. Allocates new data block (block bitmap)
2. Fills data block with user data (data block)
3. Records location of data block (inode)
4. Updates the last modified time (inode)



Example: Rename

```
$ mv /data/log.txt /tmp/log0.txt
```

1. Remove `log.txt` in `/data/` (directory)
2. Updates the last modified time of `/data` (inode)
3. Add `log0.txt` to `/tmp/` (directory)
4. Updates the last modified time of `/tmp` (inode)

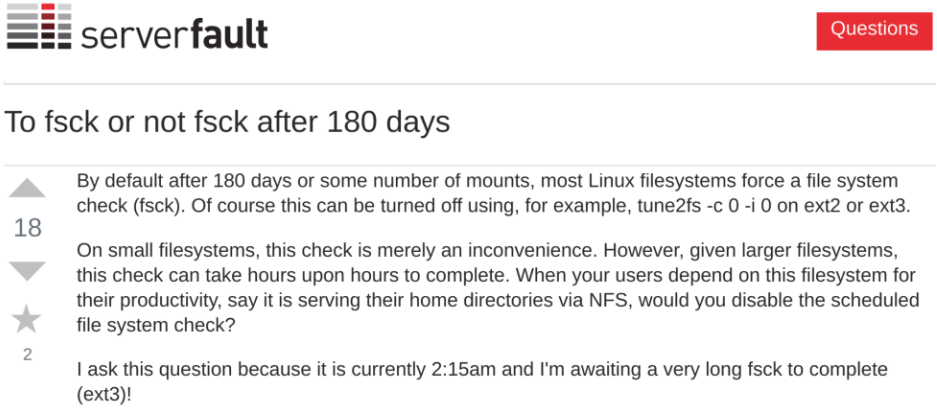


The root cause of the problem

- **Multiple block update is not atomic in a traditional storage device**
  - Single file system operation updates **multiple** disk blocks
  - System might crash in the **middle** of operation

Solution #1: Lazy, optimistic approach

- **Fix inconsistency upon reboot** → Directly updates the data
- Advantage: simple, high performance
- Disadvantage: expensive recovery
- Example: ext2 with `fsck`



Solution #2: Eager, pessimistic approach

- **Maintains the copy of data**
- Advantage: quick recovery
- Disadvantage: perpetual performance penalty
- Examples
  - Logging: ext3/4, XFS
  - Copy-on-write: btrfs, ZFS, WAFL
  - Log-structured writing: F2FS

Logging: UNDO vs. REDO

	UNDO logging	**REDO logging**
AKA	Rollback journaling	Write-ahead logging (WAL), journaling
Logging	How to undo a change (old copy)	How to reproduce a change (new copy)
Read	Read from original location	Read from log if there is a new copy

Journaling file system

- Use REDO logging to achieve atomic update despite crash
  - REDO logging = write-ahead logging (WAL) = journaling
- Record information about pending update (logging or journaling)
- If crash occurs during update, just replay what is in log to repair (recovery)
- **Turns multiple writes into atomic action**

Basic protocol

- **Logging**: write to journal
  - Record what is about to be written
- **Checkpointing**: write to main structures
  - Record information to final locations
- If crash occurs, **recover** from log
  - Replay log entries and finish update

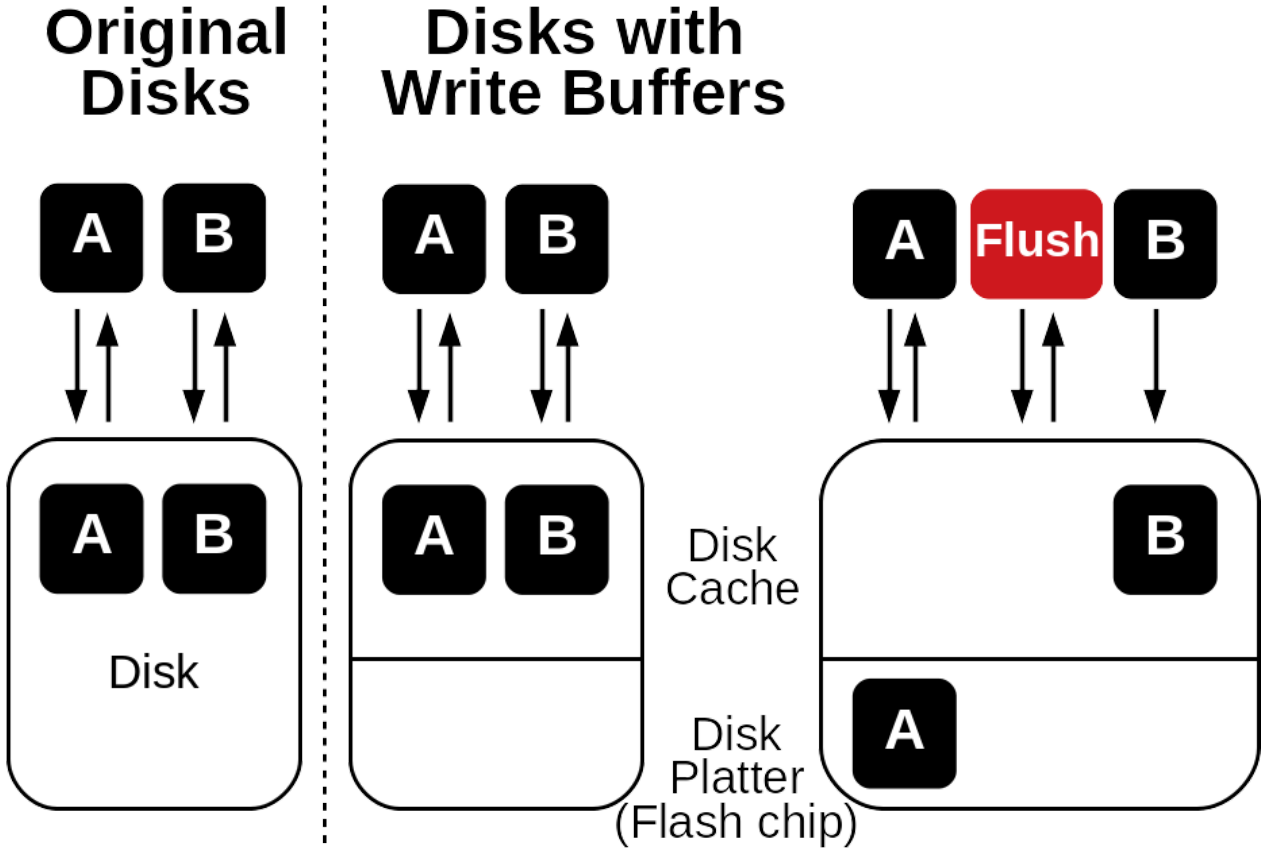
Basic (Buggy) Journaling protocol

- **Write protocol**
  - Transaction begin + contents
  - Transaction **commit**
  - **Checkpoint** contents
- **Read protocol**
  - **Design 1)** If a block is in the log, read the log. Otherwise, read contents from the original location.
  - **Design 2)** Prevent eviction of jorunaled pages until checkpointing. Read contents from page cache or the original location.

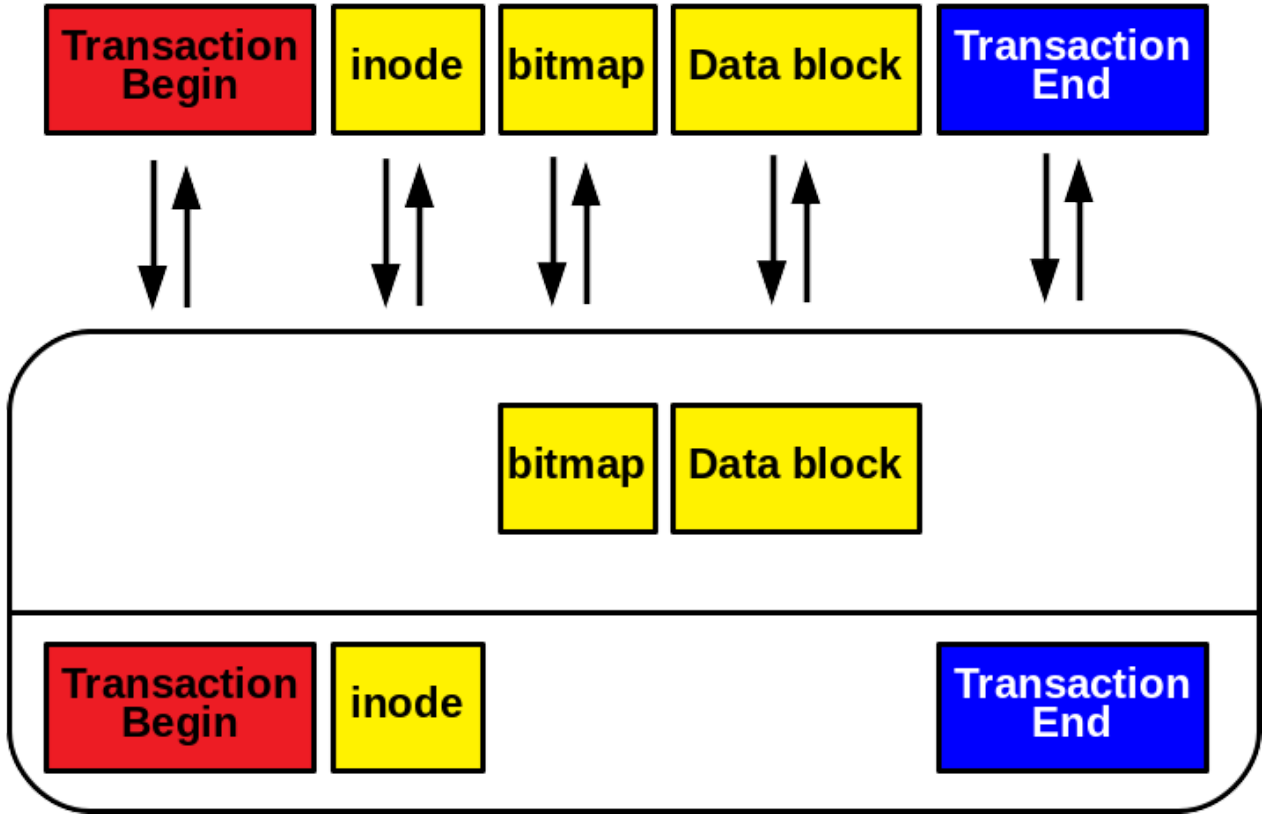
Modern disk caches

- **Disk caches**: critical for performance
  - cache tracks on reads
  - Buffer writes before committing to physical storage medium
  - HDD, SSD, SMR drive, etc.
- Example: why buffering matters
  - Performance: **factor of 2x** or greater (for some workloads)

How writes are ordered



(Buggy) Journaling protocol with disk cache

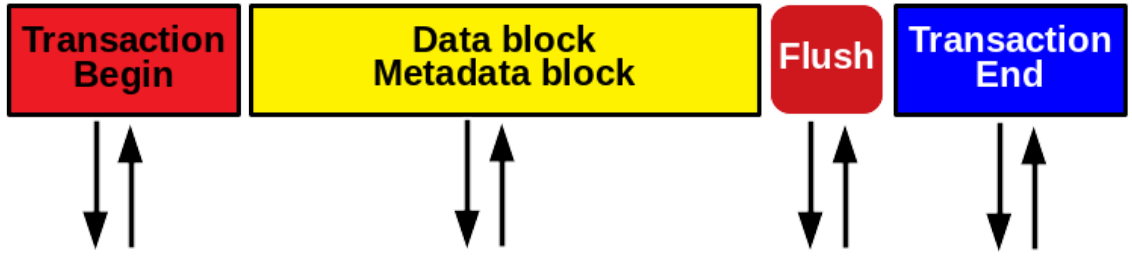


Correct journaling protocol

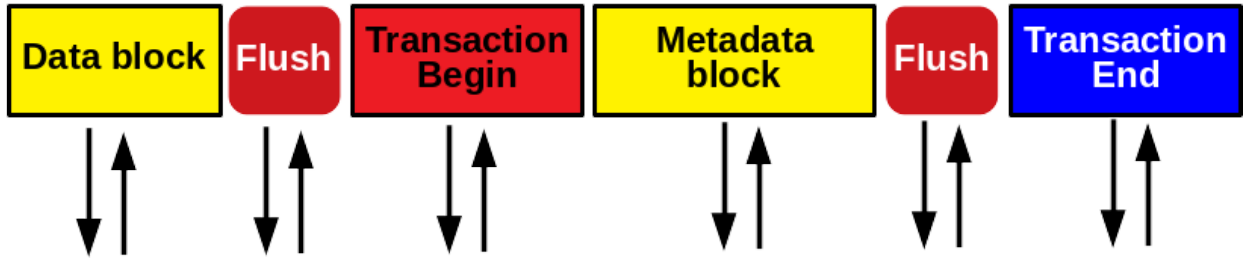
- **Write protocol**
  - **(Optional) Disk cache flush (write barrier)**
  - Transaction begin + contents
  - **Disk cache flush (write barrier)**
  - Transaction **commit**
  - **Checkpoint** contents

Correct journaling protocol

Data journaling



Ordered journaling



Journaling file system in Linux

- Almost all major file systems in Linux
- ext3, ext4, XFS

Log-structured file system (LFS)

- “The Design and Implementation of a Log-Structured File System”
  - Mendel Rosenblum and John K. Ousterhout
  - ACM TOCS 1992
- Key idea: **treats entire disk space as a log**

Motivation

- Most systems now have large enough memory to caches disk block to hold recently-accessed blocks
- Most reads are thus satisfied from the page cache
- From the point of view of the disk, most traffic is write traffic
- So to speed up disk I/O, we need to make writes go faster
- But disk performance is limited ultimately by disk head movement (seek time)
- With current file systems, adding a block takes several writes (to the file and to the metadata), requiring several disk seeks

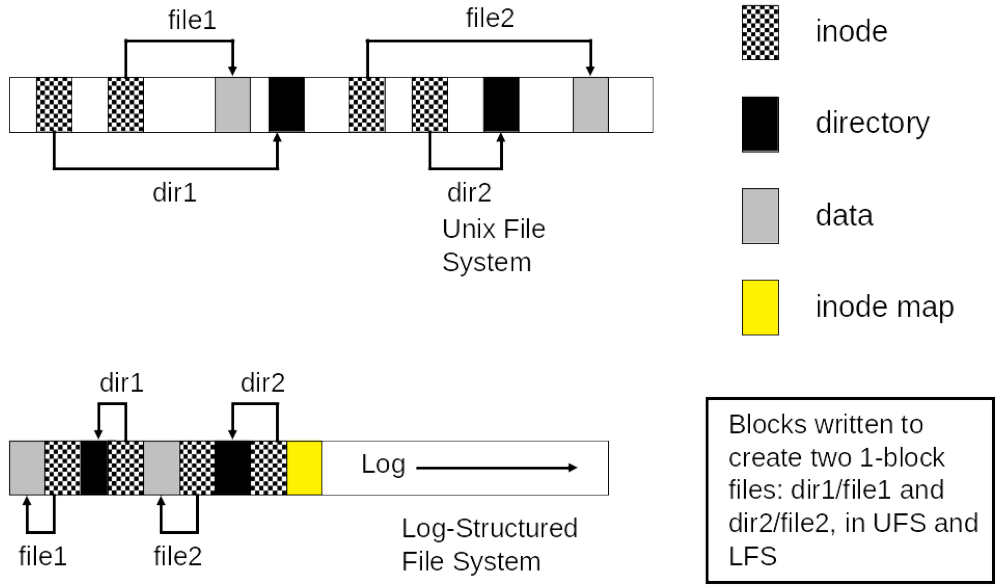
LFS: key idea

- An alternative is to use the disk as a **log**
- A log is a data structure that is written only at the head
- If the disk were managed as a log, there would be effectively no head seeks for write
- The **file** is always added to **sequentially**
- New data and metadata (inodes, directories) are accumulated in the buffer cache, then written all at once in large blocks (e.g., **segments** of .5M or 1M)
- This would greatly increase disk throughput

LFS data structures

- **inode**: as in UNIX, an inode maintain file offset to disk block mapping
- **inode map**: a table indicating where each inode is on the disk
  - inode map blocks are written as part of the segment
- **segment**: fixed size large chunk of block (e.g., .5M or 1M)
- **segment summary**: information on every block in a segment
- **segment usage table**: information on the amount of live data in a block
- **checkpoint region**: locates blocks of inode map and segment usage table, identifies last checkpoing in logl ; *located in a fixed location*
- **superblock**: static configuration of LFS; *located in a fixed location*

LFS vs. UFS


file1
file2
dir1
dir2
Unix File
System
inode
directory
data
inode map
dir1
dir2
Log
file1
file2
Log-Structured
File System
Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

LFS read

- Reads are not different that in Unix File System, once we find the inode for a file
  - checkpoint region in a fixed location → inode map → inode
  - inode: file offset → disk block mapping

LFS write

- Every write causes new blocks to be added to the current segment buffer in memory
  - When that segment is full, it is written to the disk
- Overwrite makes the overwritten, previous block obsolete
  - Live block ← newly written block
  - Dead block ← obsolete, overwritten block
- Over time, segments in the log become fragmented as we replace old blocks of files with new block
- In steady state, we need to have contiguous free space in which to write

Segment cleaning (or garbage collection)

- The major problem for a LFS is *cleaning*, i.e., producing contiguous free space on disk
- A **cleaner process** cleans old segments, i.e., takes several non-full segments and compacts them, creating one full segment, plus free space

How to choose victim segments to clean

- The cleaner choose segments on disk based on:
- **Utilization of a segment**
  - Higher utilization (more live blocks) → larger copy overhead
- **Age**
  - Recently written segment → more likely to change soon

Crash consistency in LFS

- Until disk block is cleaned by the cleaner process, the update history of a disk block is remained
- No special consistency guarantee mechanism is needed

Log-structured file system in Linux

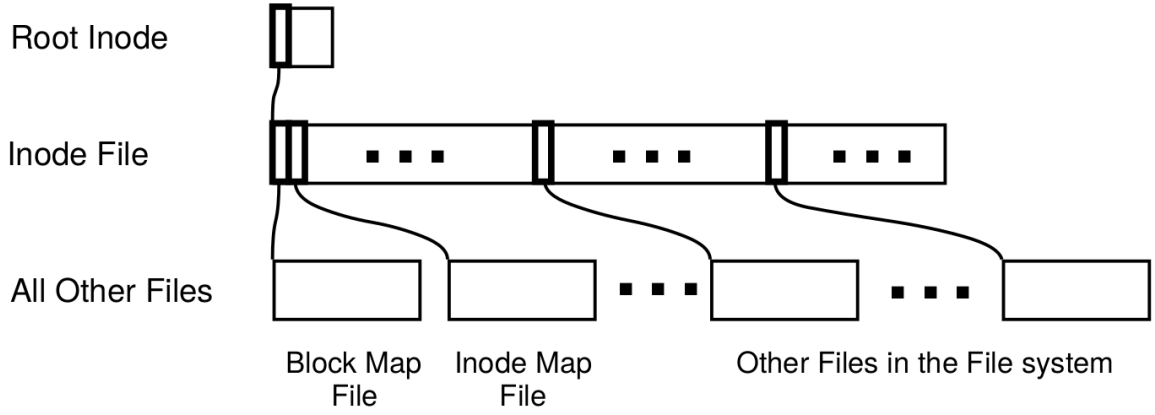
- **F2FS (Flash-Friendly File System)**
  - Optimized log-structured file system for NAND flash SSD
  - Used for mobile devices
- **NILFS**
  - Continuous snapshot file system
- **LFS design is widely adopted other than file systems**
  - Flash Translation Layer (FTL), which is a firmware for NAND flash SSD hiding complexity of NAND flash medium, is a kind of LFS managing a single file

Copy-on-Write (CoW) file system

- “File System Design for an NFS File Server Appliance”
  - Dave Hitz, James Lau, and Michael Malcolm, USENIX Winter 1994
  - **Write-Anywhere File Layout (WAFL)**: the core design of NetApp storage appliance
- **Inspired by LFS**
  - Never overwrite a block like LFS
  - No segment cleaning unlike LFS
- **Key idea**
  - Represent a filesystem as a single tree; never overwrite blocks (CoW)

WAFL layout: a tree of blocks

- **A root inode**: root of everything
- **An inode file**: contains all inodes
- **A block map file**: indicates free blocks
- **An inode map file**: indicates free inodes



Why keeping metadata in files

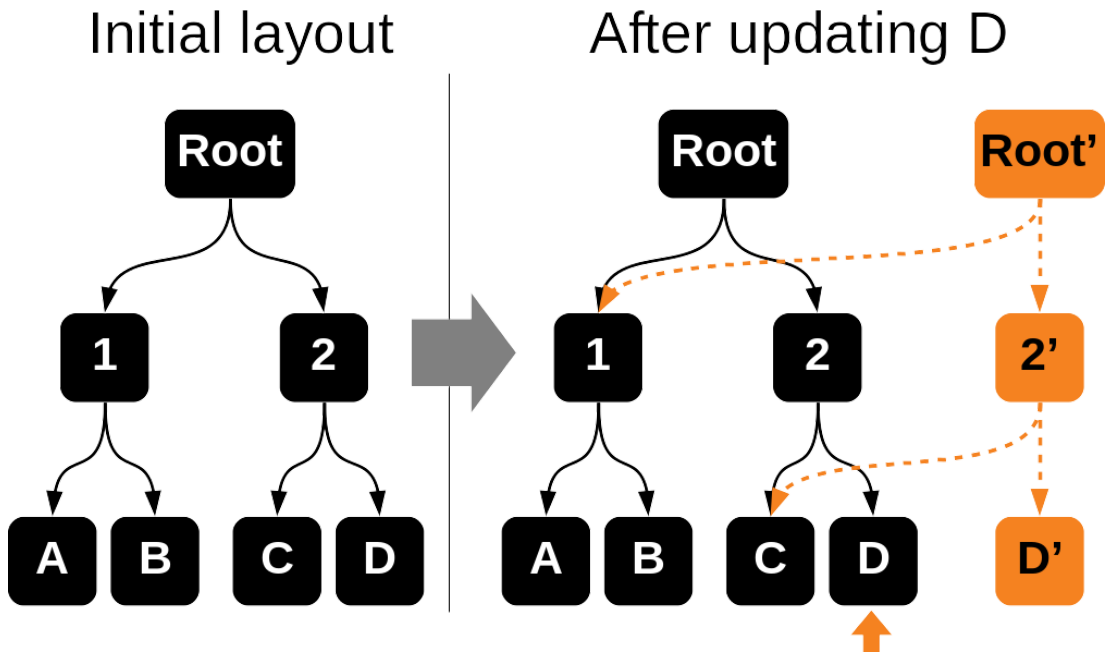
- **Allow meta-data blocks to be written anywhere on disk**
  - This is the origin of “Write Anywhere File Layout”
- **Easy to increase the size of the file system dynamically**
  - Add a disk can lead to adding i-nodes
- **Enable copy-on-write to create snapshots**
  - Copy-on-write new data and metadata on new disk locations
  - Fixed metadata locations are cumbersome

WAFL read

- Reads are not different that in Unix File System, once we find the inode for a file
  - root inode → inode file → inode
  - inode: file offset → disk block mapping

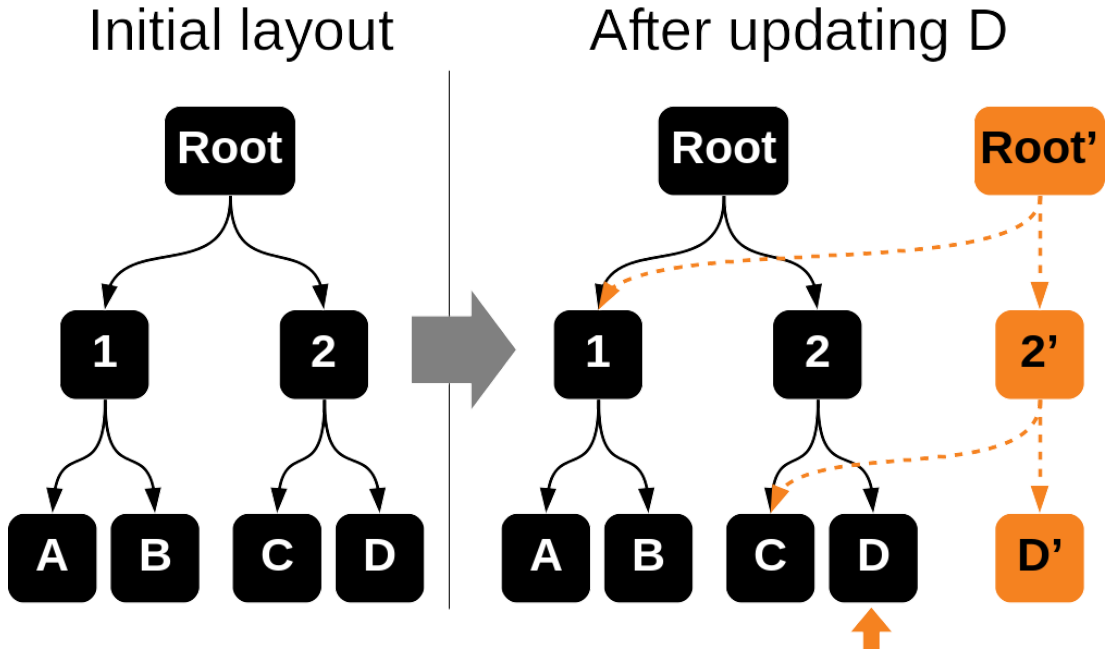
WAFL write

- WAFL filesystem is a tree of blocks
- Never overwrite blocks → Copy-on-Write



Crash consistency in LFS

- Each root inode represents a consistent snapshot of a file system



Copy-on-Write file system in Linux

- **btrfs (b-tree file system)**
  - A file system is a tree of four CoW-optimized B-trees
- **ZFS**
  - Default file system of Solaris

Further readings I

- Ext4: The Next Generation of Ext2/3
- Outline of Ext4 File System & Ext4 Online Defragmentation Foresight
- Speeding up file system checks in ext4
- Ext4 Disk Layout
- Chapter 9. The Extended Filesystem Family in Professional Linux Kernel Architecture
- Chapter 18. The Ext2 and Ext3 Filesystems in Understanding Linux Kernel
- An introduction to Linux's EXT4 filesystem

Further readings II

- What Used To Be Right Is Now Wrong
- Optimistic crash consistency
- Consistency Without Ordering
- Lightweight Application-Level Crash Consistency on Transactional Flash Storage
- SFS: Random Write Considered Harmful in Solid State Drives

Further readings III

- The design and implementation of a log-structured file system
- File System Design for an NFS File Server Appliance
- How Do SSDs Work?
- Wikipedia: disk buffer