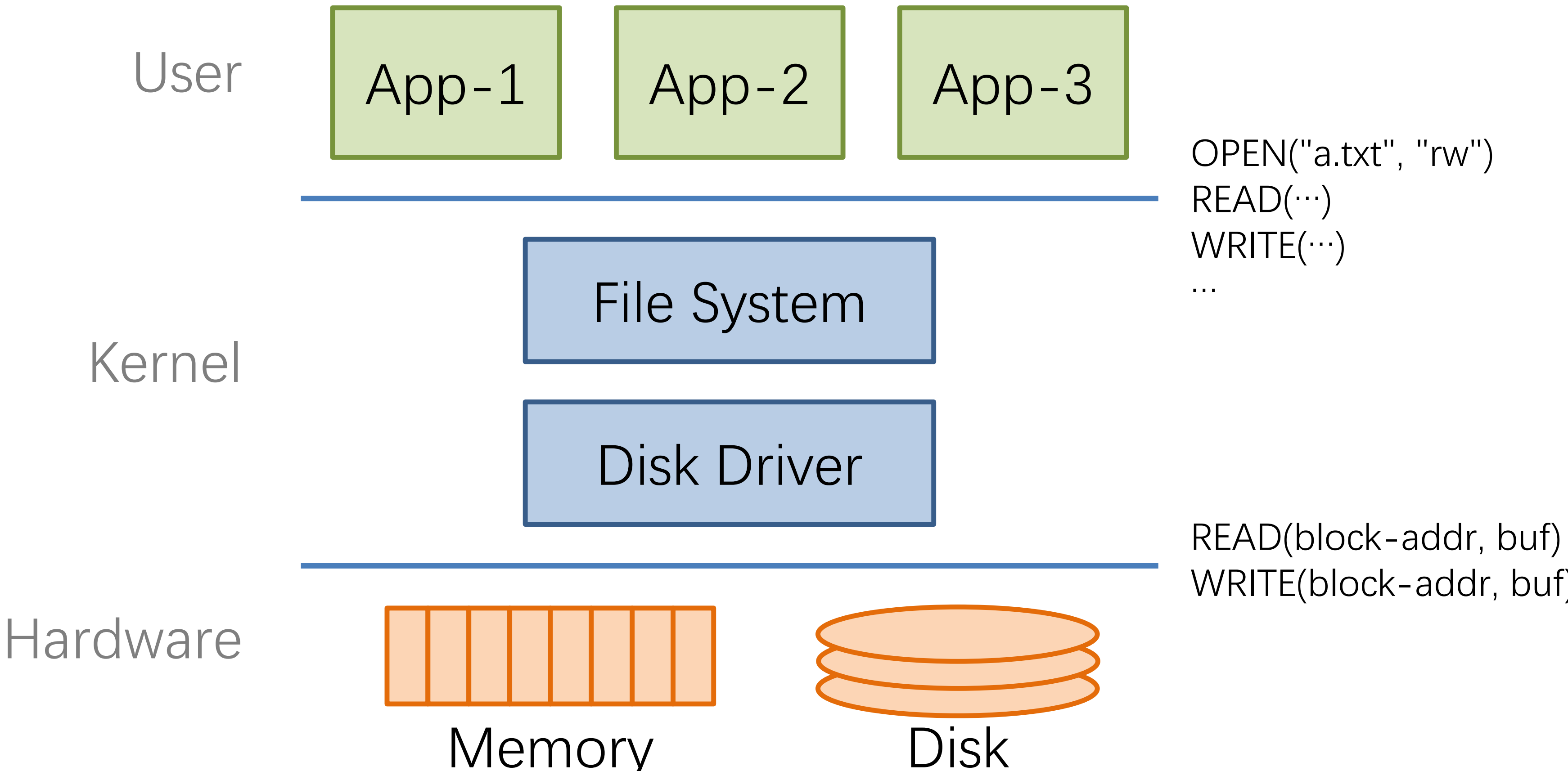
上海交通大學
Shanghai Jiao Tong University

File System

Yubin Xia

FILE SYSTEM BASIC

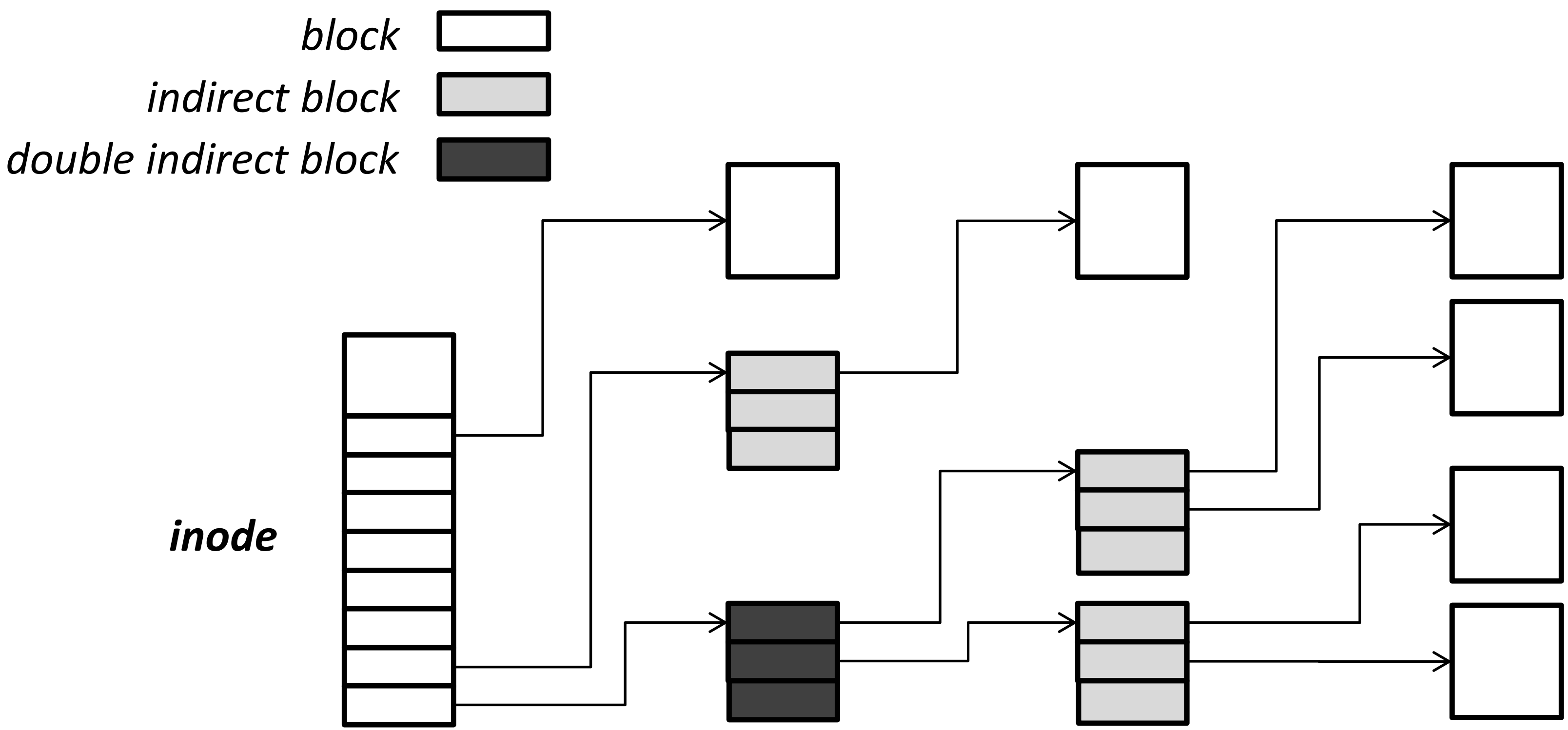
The Big Picture



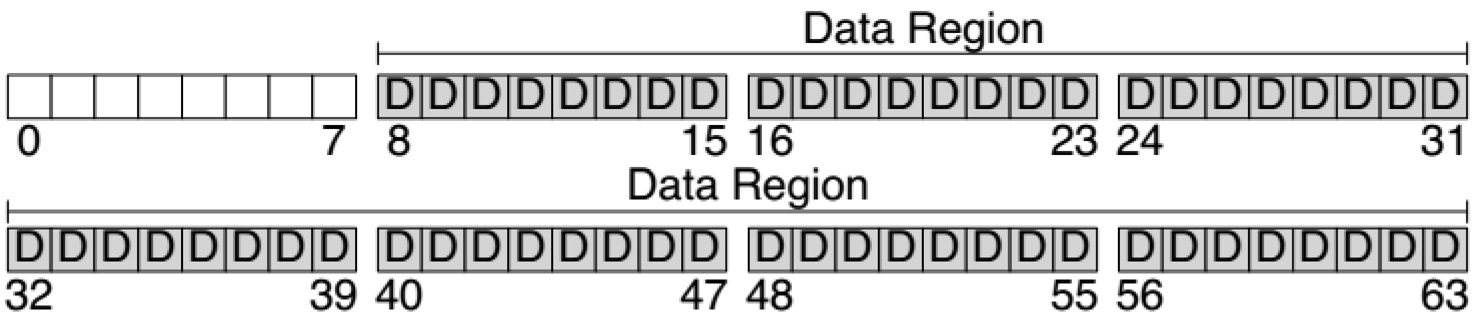
Abstraction: API of UNIX File System

- OPEN, READ, WRITE, SEEK, CLOSE
- FSYNC
- STAT, CHMOD, CHOWN
- RENAME, LINK, UNLINK, SYMLINK
- MKDIR, CHDIR, CHROOT
- MOUNT, UNMOUNT
- ….

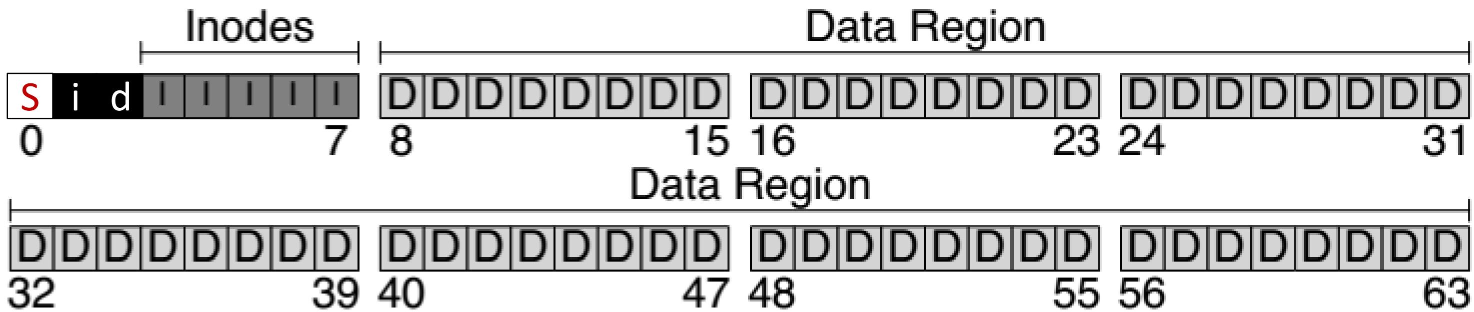
inode Structure



Disk Layout of a Simple File System



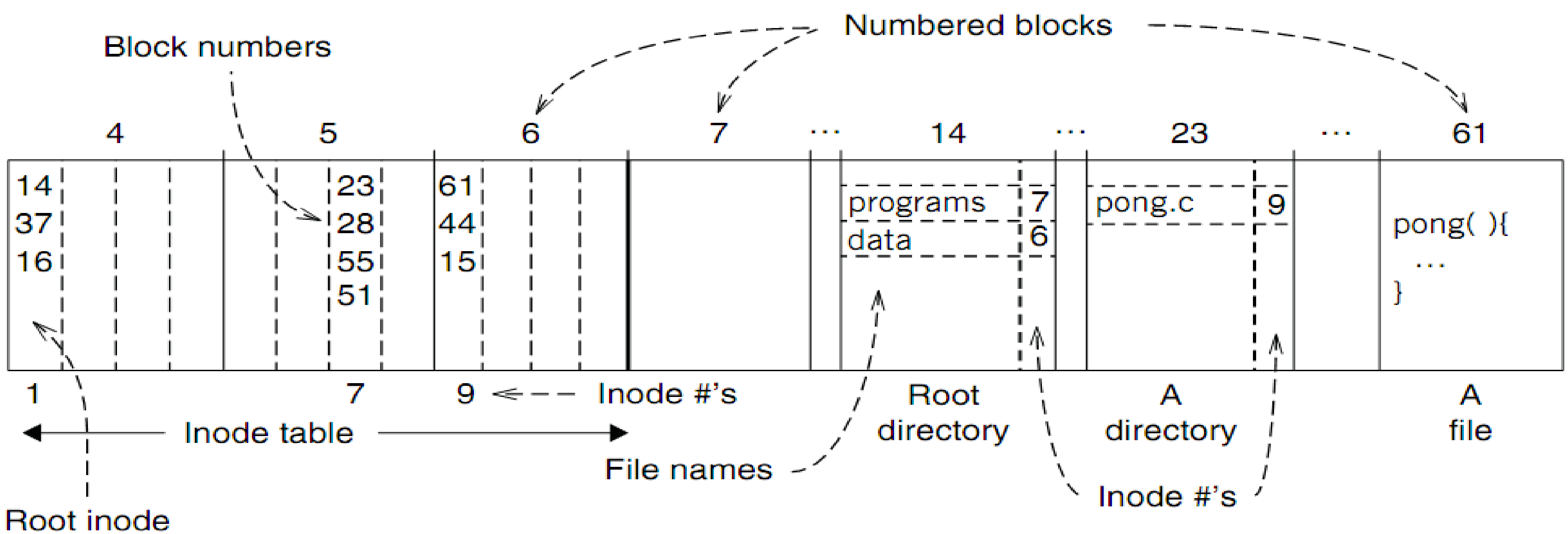
At the Head of a Disk Partition



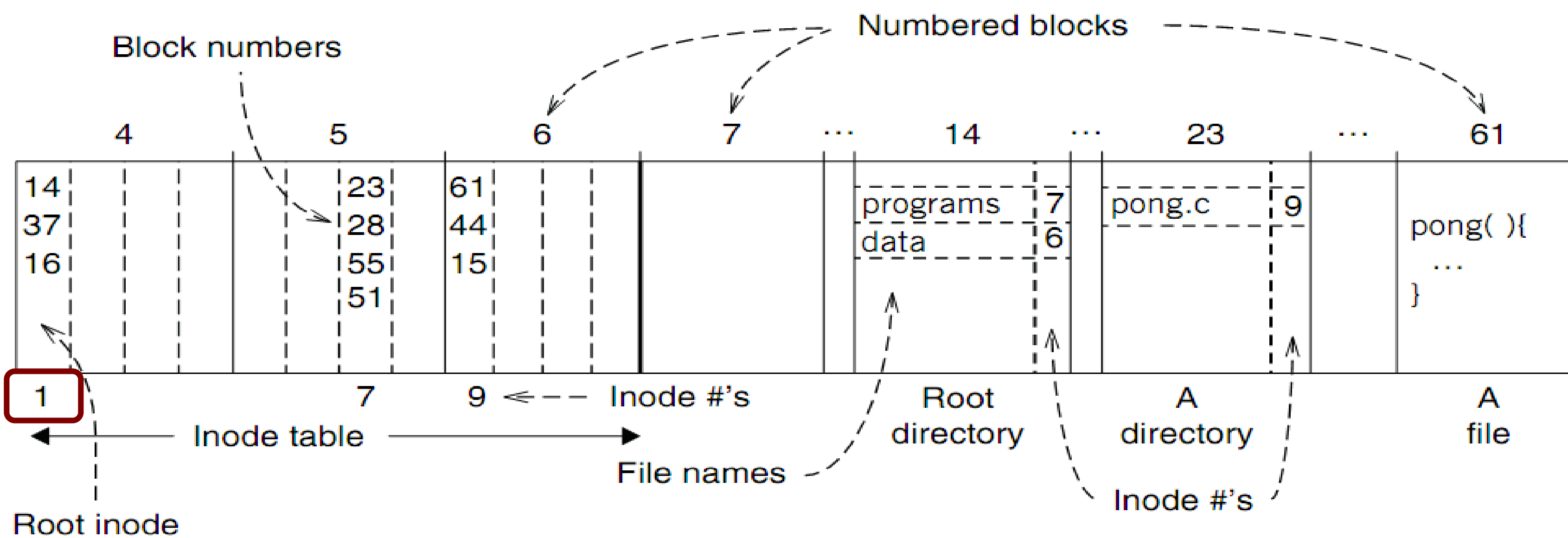
- i: inode free block bitmap
- d: data free block bitmap
- S: super-block
  - How many inodes: 80
  - How many data blocks: 56
  - Where the inode table begins: block 3
  - ...
  - The magic number to identify the file system type

The super-block is used when the file system is mounted

An example: find blocks of “/programs/pong.c”

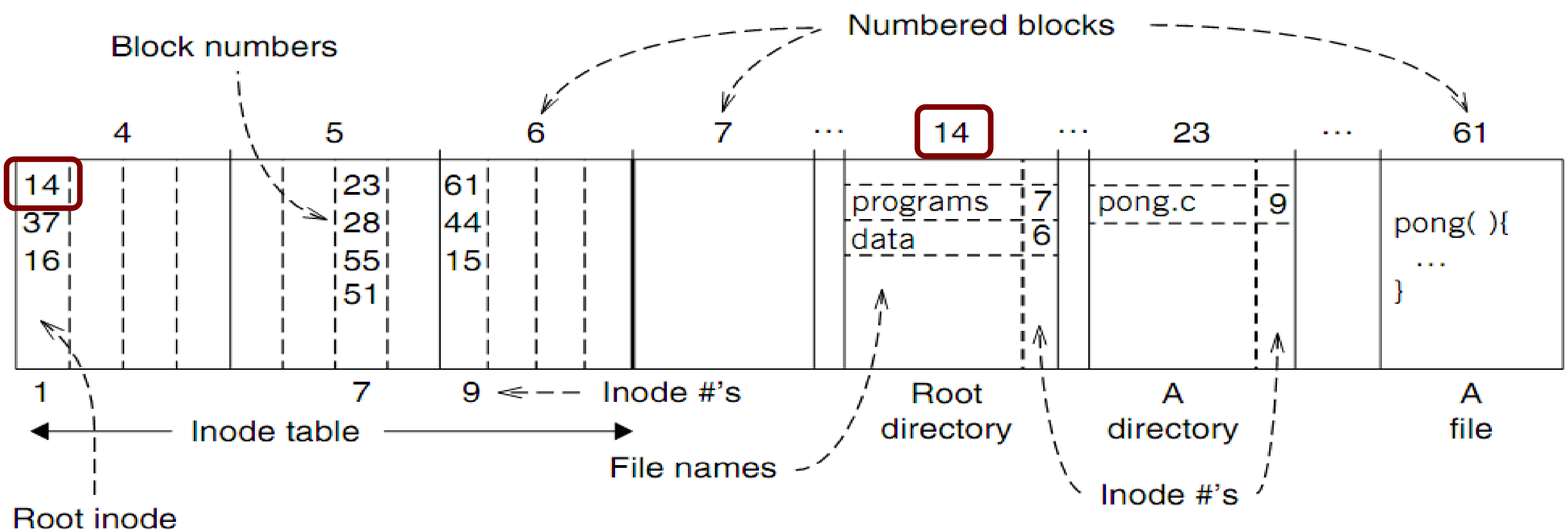


An example: find blocks of “/programs/pong.c”



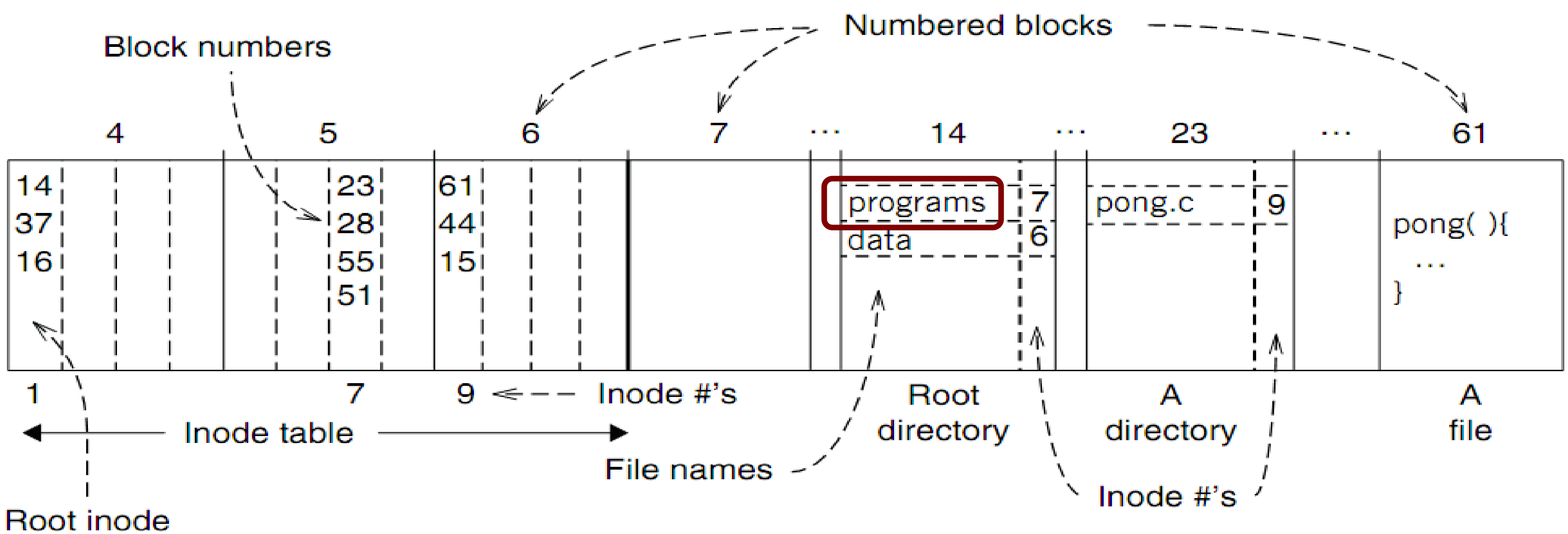
- ‘/’ root directory: inode is 1

An example: find blocks of "/programs/pong.c"



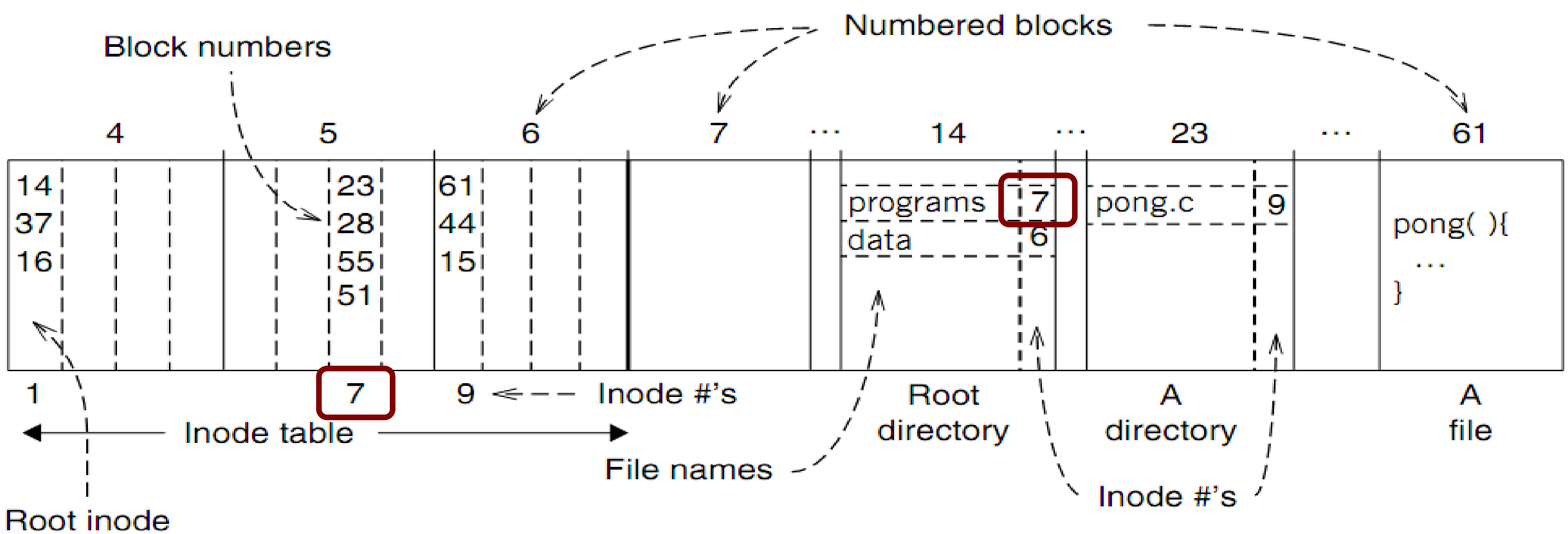
- Find the first directory in '/' by block number

An example: find blocks of “/programs/pong.c”



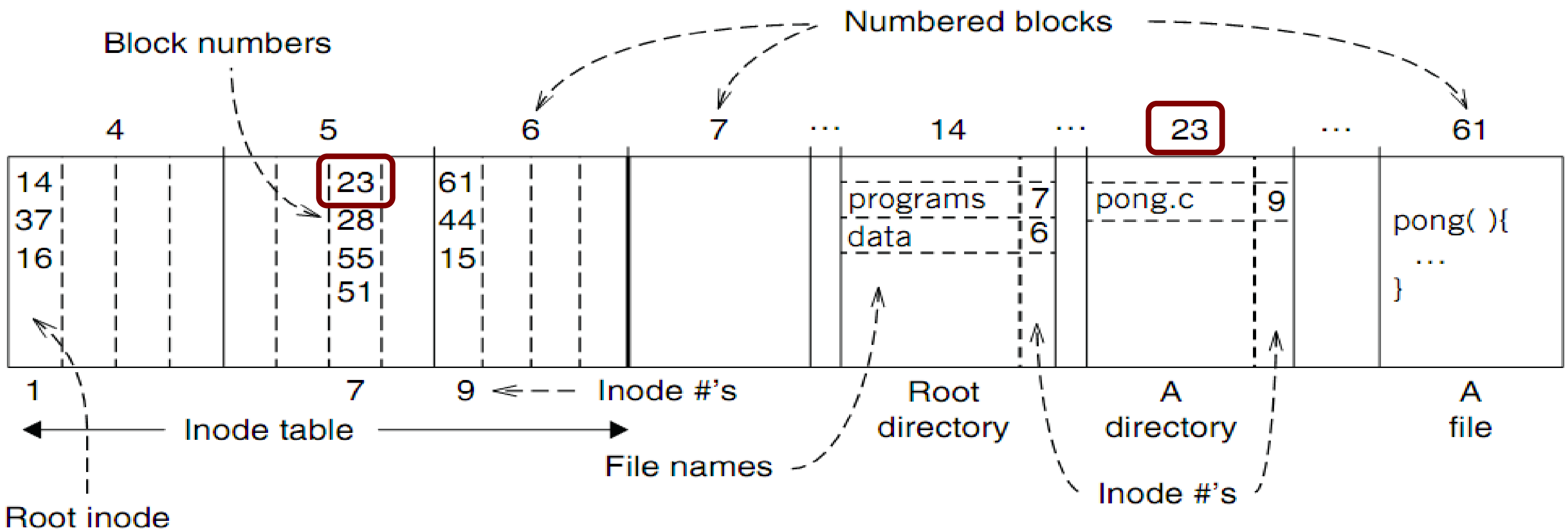
- Find ‘/programs’ by comparing name

An example: find blocks of “/programs/pong.c”



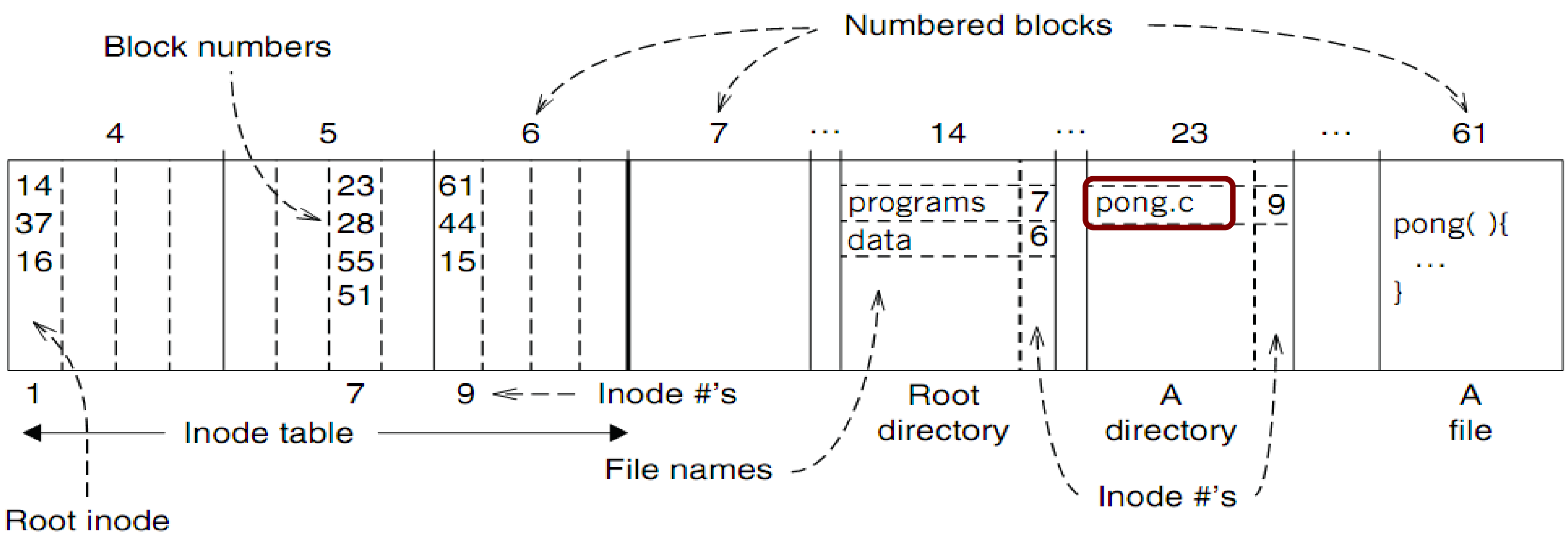
- Find ‘/programs’ inode by its inode number 7

An example: find blocks of “/programs/pong.c”



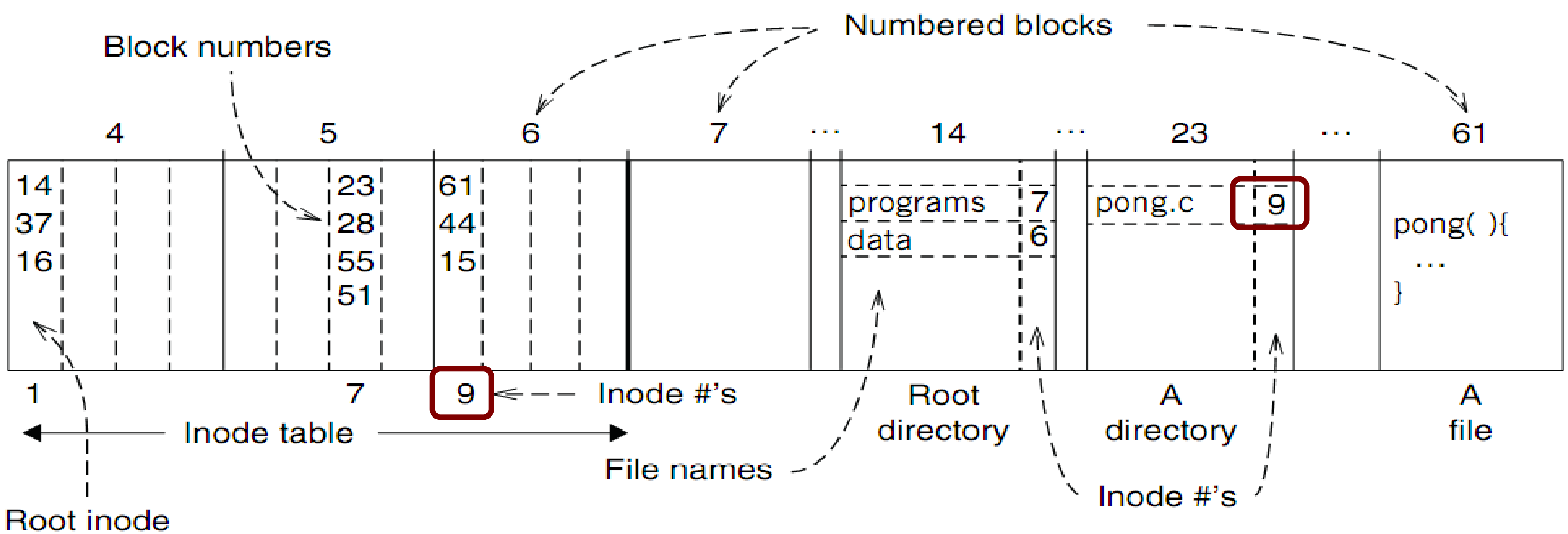
- Find the first file in ‘/programs/’

An example: find blocks of “/programs/pong.c”



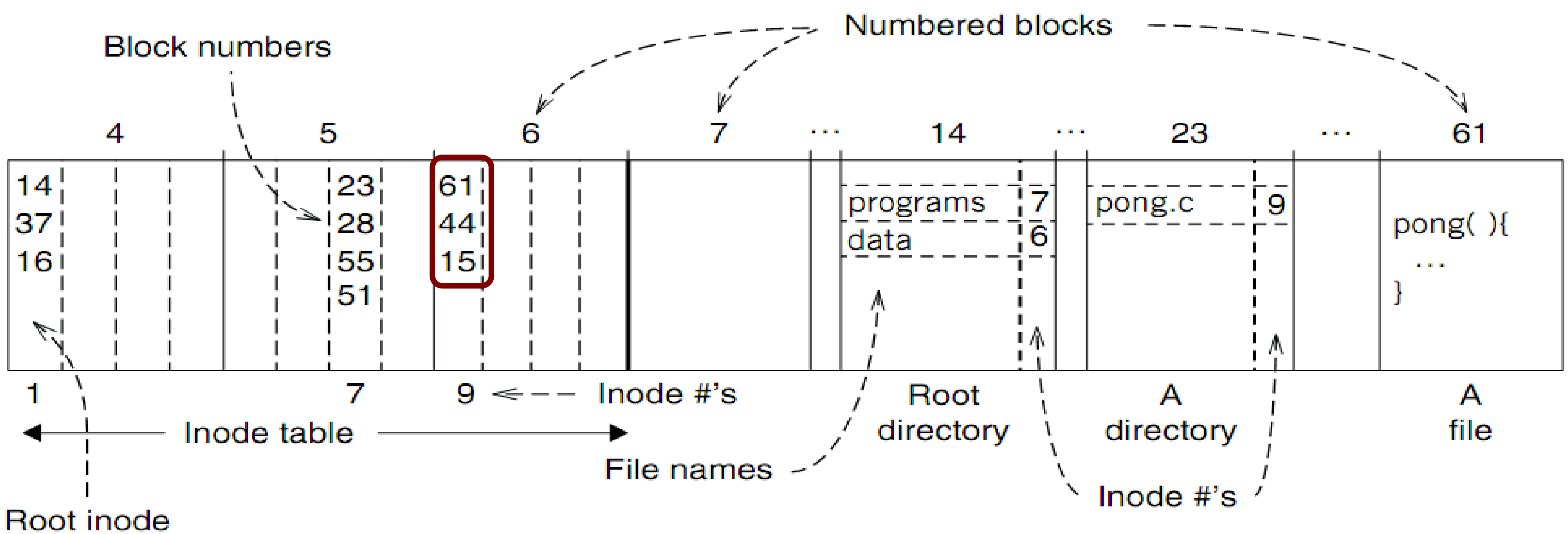
- Find ‘/programs/pong.c’ by comparing its name

An example: find blocks of “/programs/pong.c”



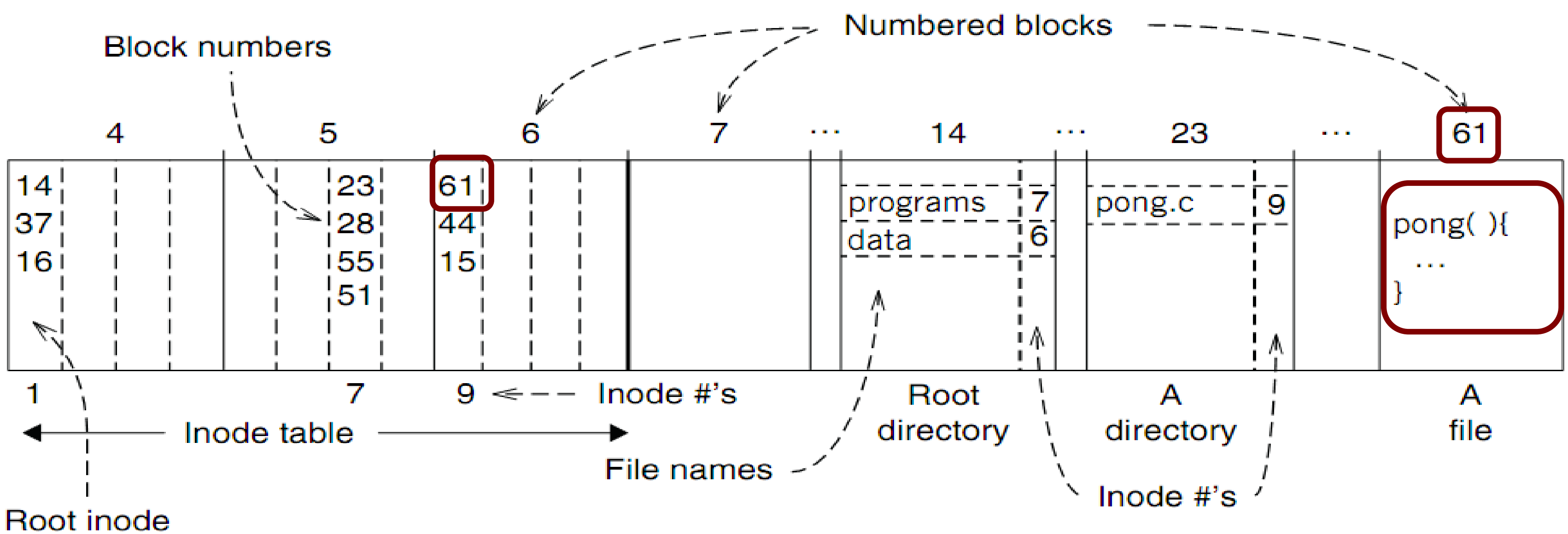
- Find inode of ‘/programs/pong.c’ by the inode number 9

An example: find blocks of “/programs/pong.c”



- Find block number of ‘/programs/pong.c’

An example: find blocks of “/programs/pong.c”



- Find data of block 61 by its block number
  - And data of block 44 & 15

Directly Dump a Directory

```
$ ls -ai temp
7536909 .  7530417 ..  7536939 a  7536940 b  7536941 c  7536942 d

$ echo “obase=16;7536909;7530417;7536939;7536940;7536941;7536942" | bc
73010D  72E7B1  73012B  73012C  73012D  73012E

$ sudo /sbin/debugfs /dev/sda1
debugfs 1.43.4 (31-Jan-2017)
debugfs:  dump temp temp.out
debugfs:  quit

$ xxd temp.out
0000000: 0d01 7300 0c00 0102 2e00 0000 b1e7 7200  ..s...........r.
0000010: 0c00 0202 2e2e 0000 2b01 7300 0c00 0101  ........+.s.....
0000020: 6100 0000 2c01 7300 0c00 0101 6200 0000  a...,.s.....b...
0000030: 2d01 7300 0c00 0101 6300 0000 2e01 7300  -.s.....c.....s.
0000040: c40f 0101 6400 0000 0000 0000 0000 0000  ....d...........
0000050: ...
```

Directly Dump a Directory

```
struct ext4_dir_entry {
  uint32_t inode_number;
  uint16_t dir_entry_length;
  uint8_t  file_name_length;
  uint8_t  file_type;
  char     name[EXT4_NAME_LEN];
}
```

```
0d01 7300 0c00 0102 2e00 0000
b1e7 7200 0c00 0202 2e2e 0000
2b01 7300 0c00 0101 6100 0000
2c01 7300 0c00 0101 6200 0000
2d01 7300 0c00 0101 6300 0000
2e01 7300 c40f 0101 6400 0000
```

```
File Type
  0x0: Unknown
  0x1: Regular file
  0x2: Directory
  0x3: Character device file
  0x4: Block device file
  0x5: FIFO
  0x6: Socket
  0x7: Symbolic link
```

```
0d01 7300 0c00 0102 2e00 0000
```

↓

```
0d01 7300: inode number
     0c00: entry length is 12 bytes
       01: file name length is 1 byte
       01: file type is regular file
2e00 0000: file name (2e -> ".")
```

Two Types of Links (Synonyms)

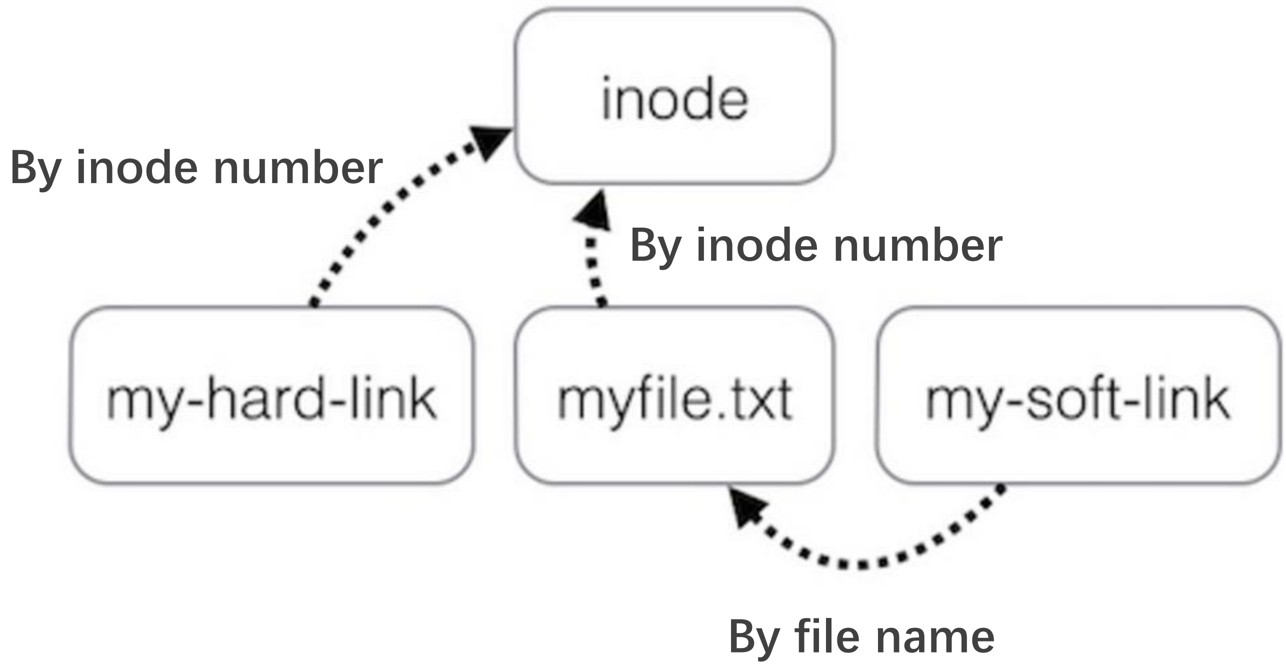
- Add link "assignment" to "Mail/new-assignment"
  - Hard link
    - No new file is created
    - Just add a binding between a string and an **existing** inode
    - Target inode reference count is increased
    - If target file is deleted, the link is still valid
  - Soft link
    - A new file is created, the data is the string "Mail/new-assignment"
    - Target inode reference count is not increased
    - If target file is deleted, the link is not valid
- Soft link can create cycle by SYMLINK("a", "a")

Directly Dump a Symbolic Link

```
$ ln -s "/tmp/abc" s-link

$ ls -l s-link
7536945 lrwxrwxrwx 1 xiayubin 8 Sep 20 08:01 s-link -> /tmp/abc

$ readlink s-link
/tmp/abc

$ cat s-link
cat: slink: No such file or directory

$ echo "hello, world" > /tmp/abc

$ cat s-link
hello, world
```

What does "8" means? **File size**

Two Types of Links (Synonyms)



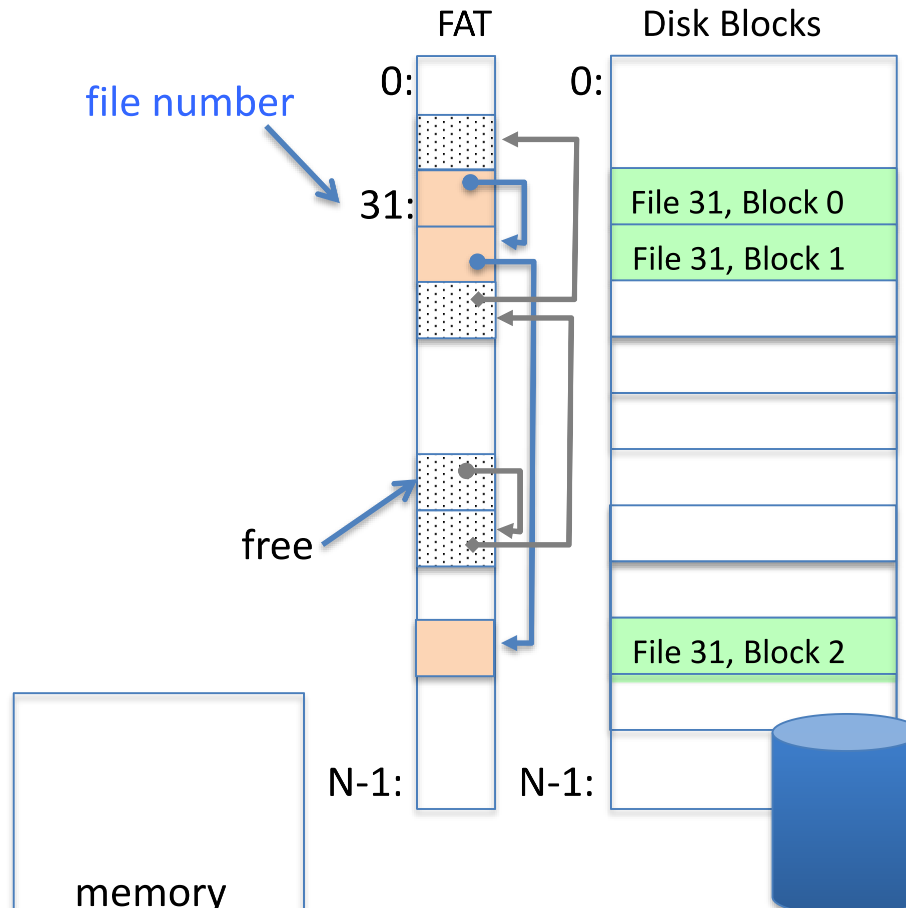
File Open & Read Timeline

	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					

```
open("/foo/bar", O_RDONLY)
```

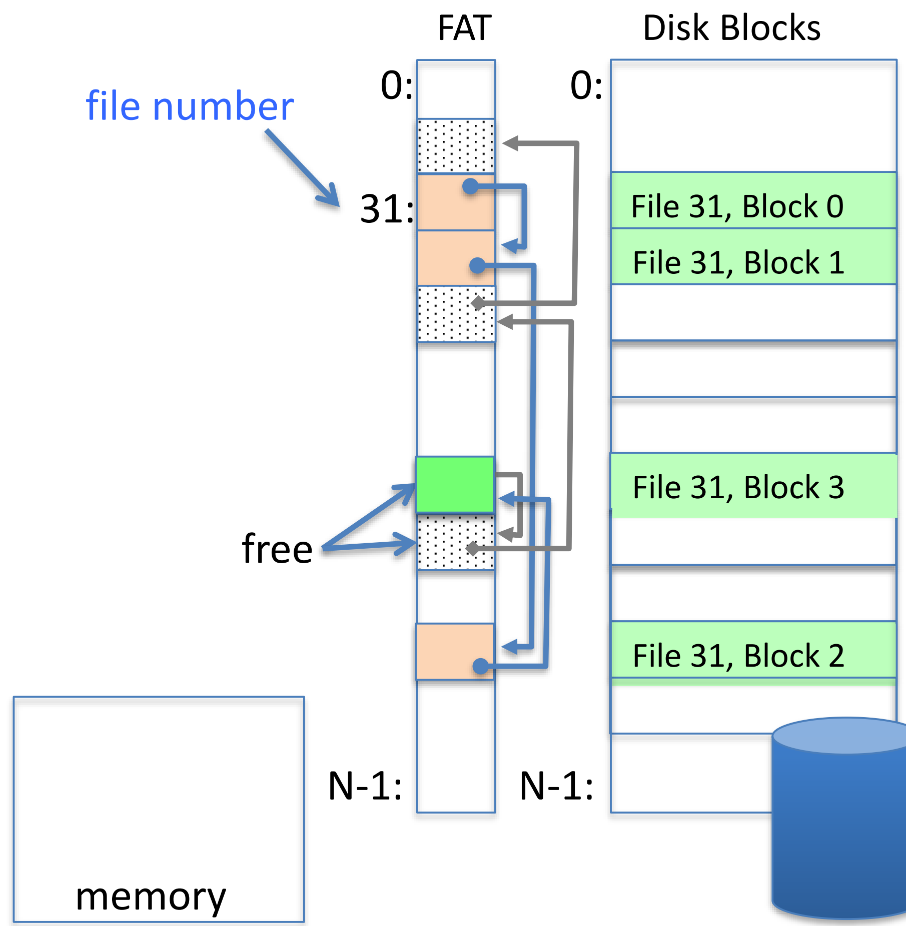
FAT (File Allocation Table) File System

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = < B, x >)
- Follow list to get block #
- Unused blocks ⇔ FAT free list

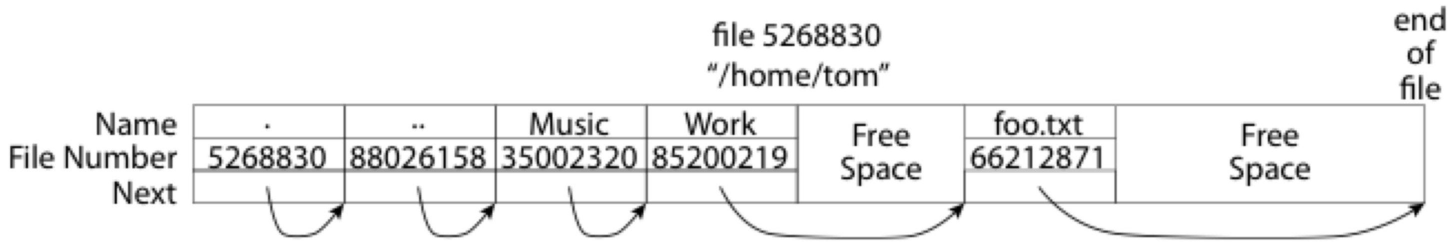


FAT Properties

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = < B, x >)
- Follow list to get block #
- Unused blocks ⬄ FAT free list
- Ex: file_write(31, < 3, y >)
  - Grab blocks from free list
  - Linking them into file



What about the Directory in FAT?



- Essentially a file containing `<file_name: file_number>` mappings
- Free space for new entries
- In FAT: file attributes are kept in directory (!!!)
- Each directory a linked list of entries
- Q: Where to find root directory ("/")?

FS CRASH CONSISTENCY

Crash Consistency Problem

- Single file-system operation updates multiple on-disk data structures
- System may crash in middle of updates
- File-system is partially (incorrectly) updated

File System Durability

Topic: tension between fs perf. and crash recovery

Disk performance is often a #1 bottleneck

"how many seeks will that take?”

Durability != Crash consistency

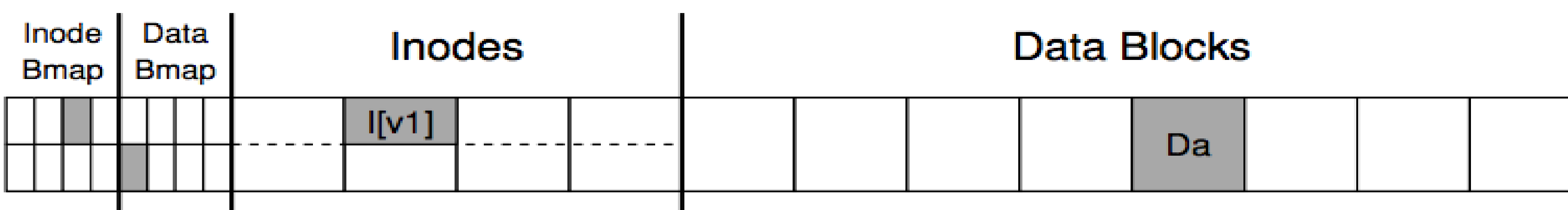
“Here is all of my data. But some of the metadata is wrong.”

Crash recovery is much harder than performance

"what if a crash occurred at this point?”

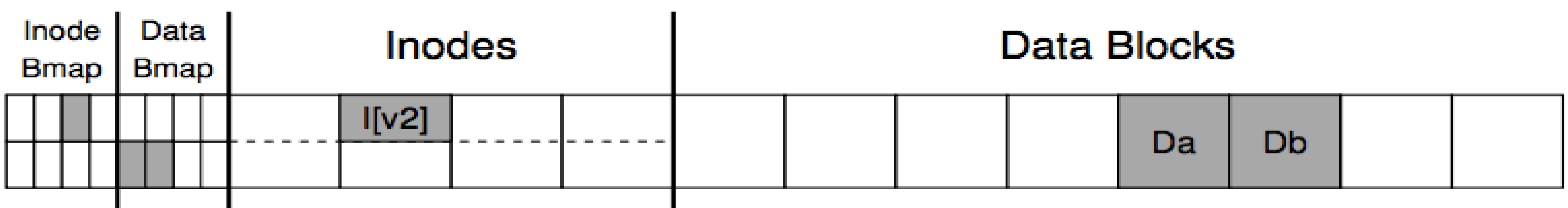
An Example: Append a File

- Inside of I[v1]:
  - owner : yubin
  - permissions : read-only
  - size : 1
  - pointer : 4
  - pointer : null
  - pointer : null



An Example: Append a File

- Inside of I[v2]:
  - owner : yubin
  - permissions : read-only
  - size : 2
  - pointer : 4
  - pointer : 5
  - pointer : null



Crash Scenarios: 1 Succeeds

- Imagine only a single write succeeds; there are thus three possible outcomes:

- 1. Just the data block (Db) is written to disk
  - What will happen?
- 2. Just the updated inode (I[v2]) is written to disk
  - What will happen?
- 3. Just the updated bitmap (B[v2]) is written to disk
  - What will happen?

Crash Scenarios: 2 Succeed

- Two writes succeed and the last one fails:

- 1. The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db)
- 2. The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2])
- 3. The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2])

Recovery Approach

- Synchronous meta-data update + fsck
  - Used in xv6-rev0
  - During check, synchronize meta-data, such as file size
- Soft update (FreeBSD fs modified on FFS)
  - Soft update, not covered in this course
- Logging (ext 3/4), xv6-rev6 and following versions
  - Before doing actual meta-data update, log the event
  - After crash, recover from log

SYNC METADATA UPDATE + FSCK

Typical Set of Tradeoffs

- FS ensures it can recover its meta-data
  - Internal consistency
  - No dangling references
  - Inode and block free list contain only used (not using) items
  - Unique name in one directory, etc.
- Weak semantic FS provided limited guarantees
  - Atomicity for creat, rename, delete
  - Often no durability for anything
  - (creat("a"), then crash, no a)
  - Often no order guarantees

What does fsck do?

- **1. Check superblock**
  - E.g., making sure the file system size is greater than the number of blocks allocated
  - If error, use an alternate copy of the superblock
- **2. Check free blocks**
  - Scans the inodes, indirect blocks, double indirect blocks, etc.
  - Uses this knowledge to produce a correct version of the allocation bitmaps
  - Same for the inode bitmap

What does fsck do?

- **3. Check inode states**
  - Check type: regular file, dir, symbolic link, etc.
  - Clear suspect inodes and clear the inode bitmap
- **4. Check inode links**
  - Check link count by scanning the entire fs tree
  - If count mismatches, fix the inode
  - If inode is allocated but no dir contains it, lost+found
- **5. Check duplicates**
  - Two inodes refer to the same block
  - If one inode is obviously bad, clear it; otherwise, copy the block and give each a copy

What does fsck do?

- **6. Check bad blocks**
  - E.g., point to some out-of-range address
  - What should fsck do? Just remove the pointer
- **7. Check directories**
  - The only file that fsck know more semantic
  - Making sure that "." and ".." are the first entries
  - Ensure no dir is linked more than once
  - No same filename in one dir

Problem of fsck: Too Slow

- How long would fsck take?
  - an example server: fsck takes 10 minutes per 70GB disk w/ 2 million inodes
  - clearly reading many inodes sequentially, not seeking
  - still a long time, probably linear in disk size

- Consider the example before:
  - Scan the disk for only three disk block writes
  - Just like find a key by searching the entire house

What's the right order of synchronous writes?

- File creation
  - 1. mark inode as allocated
  - 2. create directory entry

- File deletion
  - 1. erase directory entry
  - 2. erase inode addrs[], mark as free
  - 3. mark blocks free

What about app-visible syscall semantics?

- Durable? Yes
  - Use write-through cache, sync I/O, O_SYNC
- Atomic? Often
  - Mkdir is an exception
- Ordered? Yes
  - If all writes are sync

Issues with Synchronous Write

- Main issue
  - Very slow during normal operation
  - Very slow during recovery

Ordinary perf. of sync meta-data update?

- Creating a file and writing a few bytes
  - Takes 8 writes, probably 80 ms
  - (ialloc, init inode, write dirent, alloc data block, add to inode, write data, set length in inode, xxx)
- Can create only about a dozen small files per second!
  - Think about `un-tar` or `rm *`

How to get better performance?

- Reality:
  - RAM is cheap
  - Disk sequential throughput is high, 50 MB/sec
- **Why not use a big write-back disk cache?**
  - No sync meta-data update operations
  - Only modify in-memory disk cache (no disk write)
  - So creat(), unlink(), write() &c return almost immediately
    - If cache is full, write LRU dirty block
    - Write all dirty blocks every 30 seconds, to limit loss if crash
  - This is how old Linux EXT2 file system worked

Write-back Cache

- Would write-back cache improve performance?
- Why performance would be improved?
  - After all, you have to write the disk in the end anyway

Barrier: Flush the Disk

- Disk's write buffer
  - Disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache
  - But the data is not on disk yet! No durability! No order!
- One solution: disable the buffer
- Another solution: using flush operation
  - Force the disk to write data to disk media
  - Aka., disk write barrier
- However, disks may not do as they claim…
  - Some disks just ignore the flush operation to be faster
  - "the fast almost always beats out the slow, even if it is wrong" --- Kahan

LOGGING / JOURNALING

Journaling Overview

Before updating file system, write note describing update

Make sure note is safely on disk

Once note is safe, update file system

- If interrupted, read note and redo updates

Journaling Overview

Workload: Creating and writing to a file

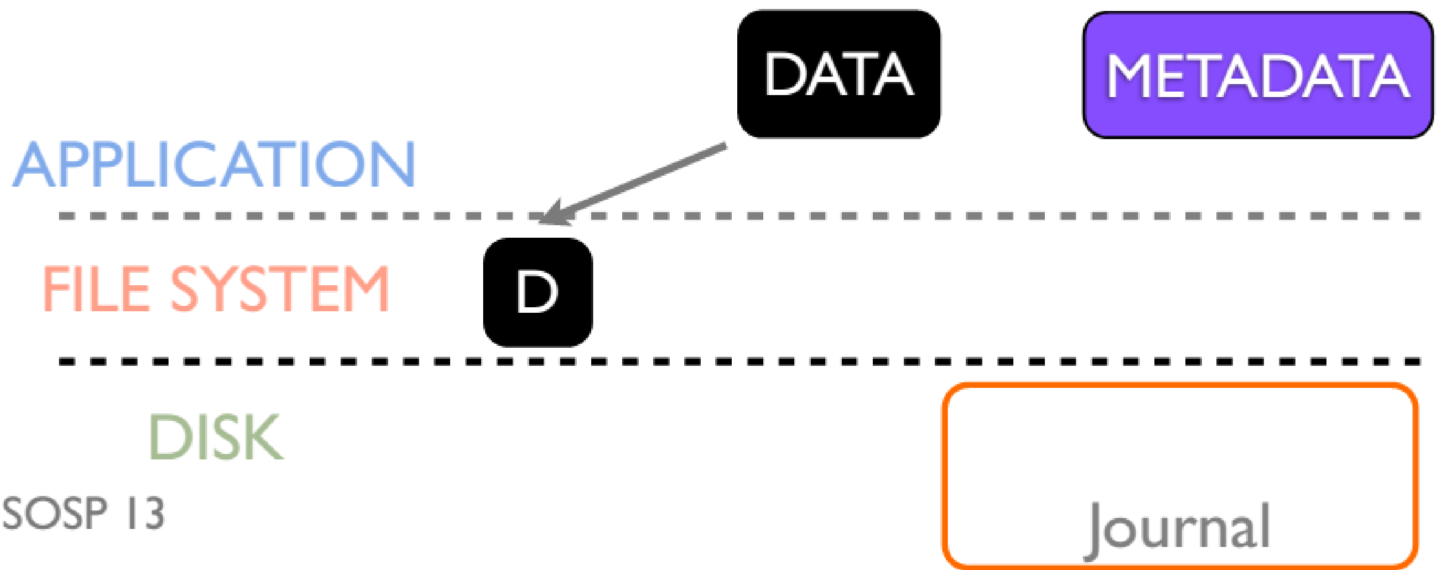
Journaling protocol (ordered journaling)

Journaling Overview

Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)

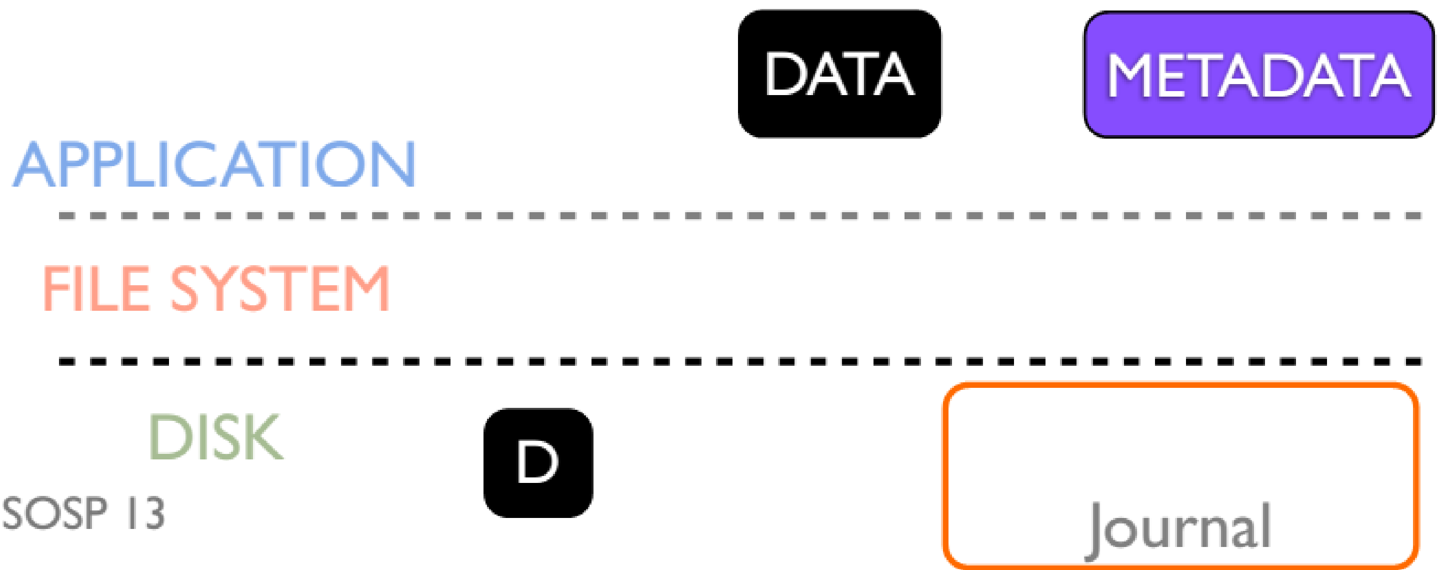


Journaling Overview

Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)

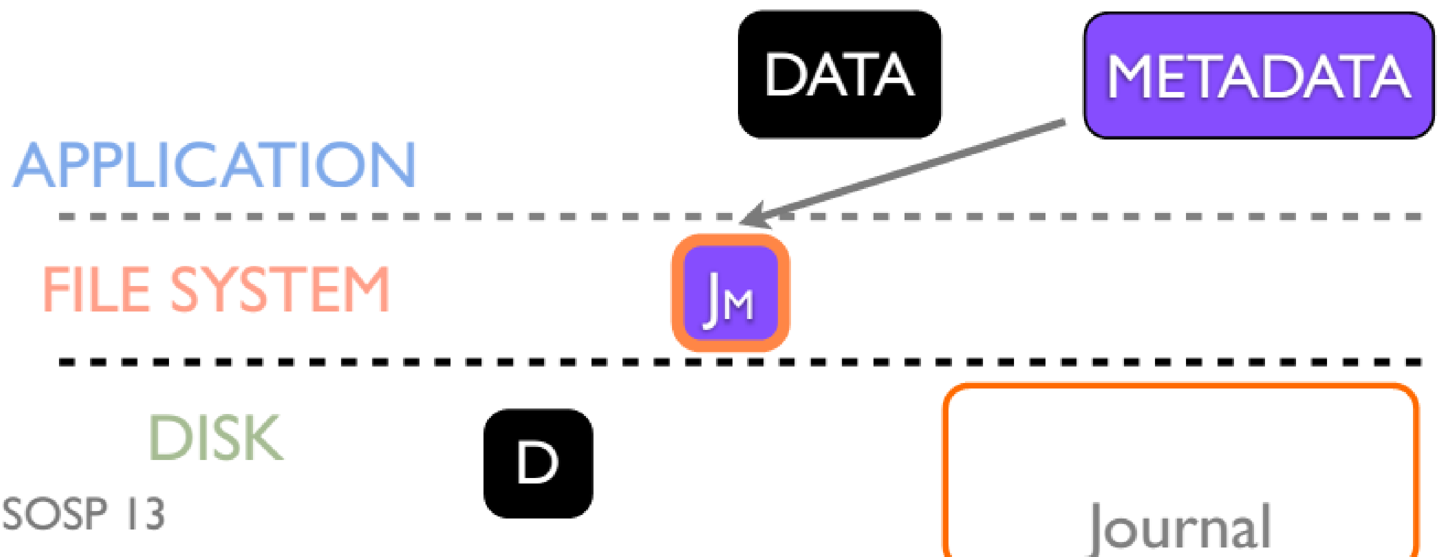


Journaling Overview

Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)
- Logging Metadata (J_M)

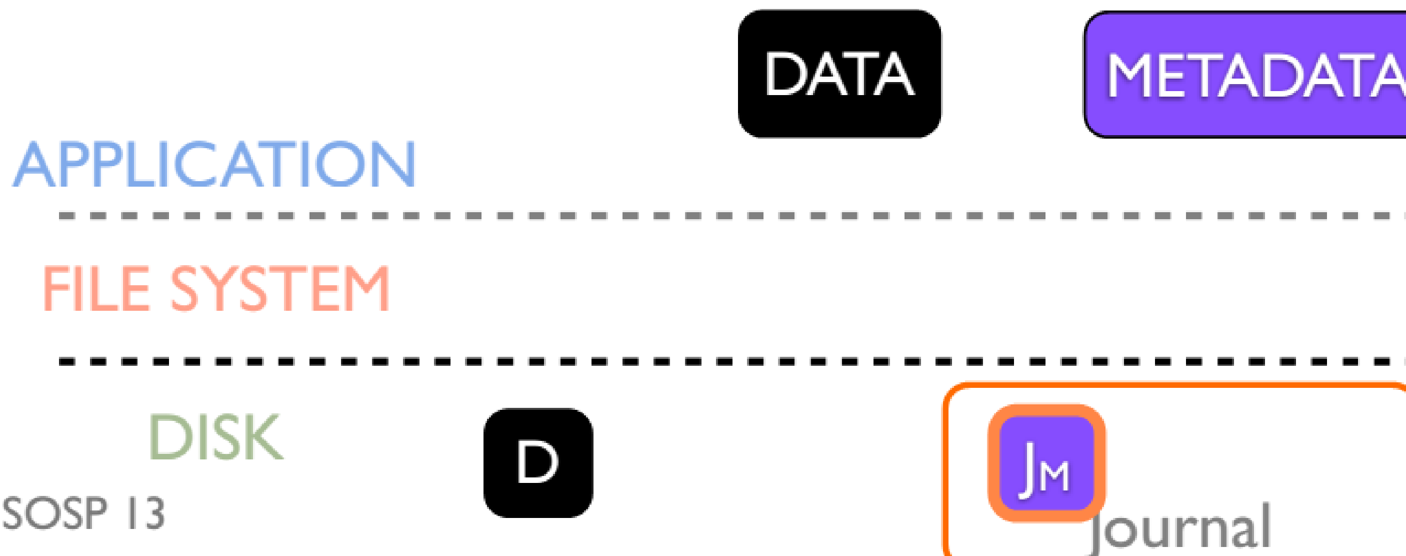


Journaling Overview

Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)
- Logging Metadata (J_M)

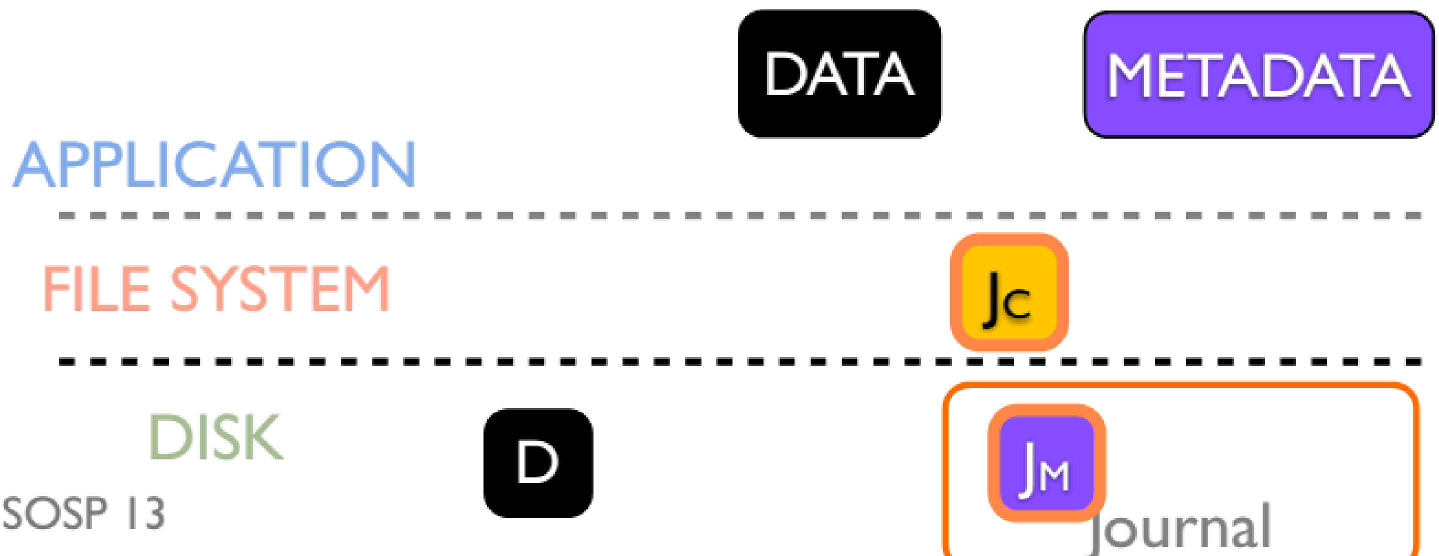


Journaling Overview

Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)

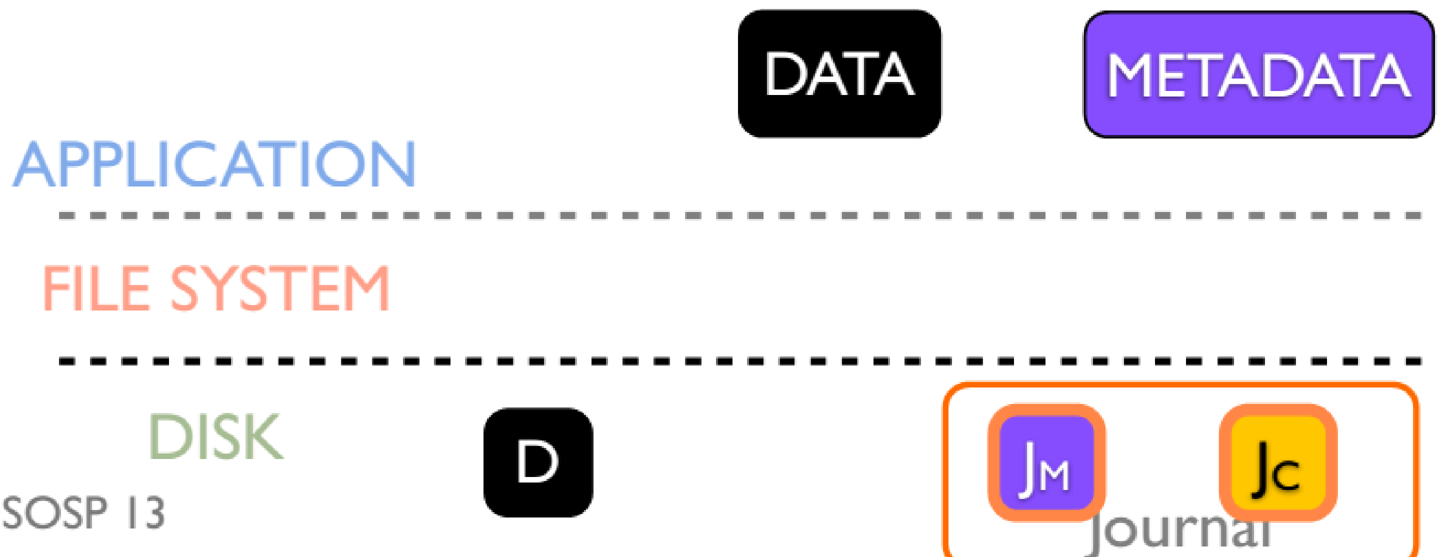


Journaling Overview

Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)

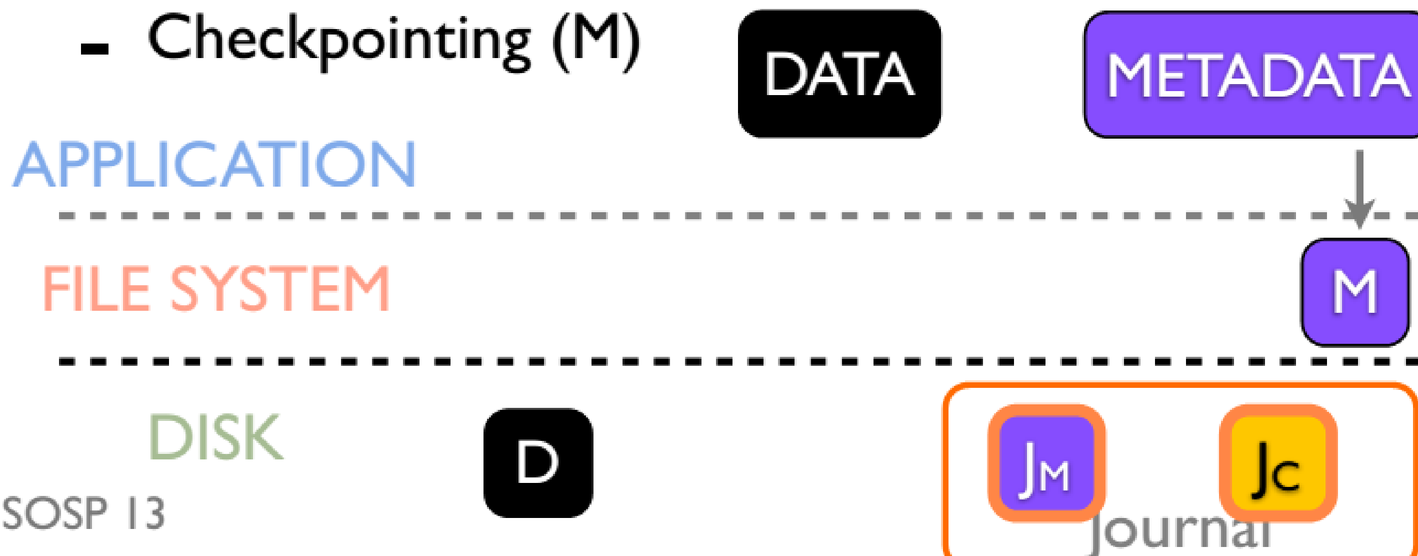


Journaling Overview

Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)
- Checkpointing (M)

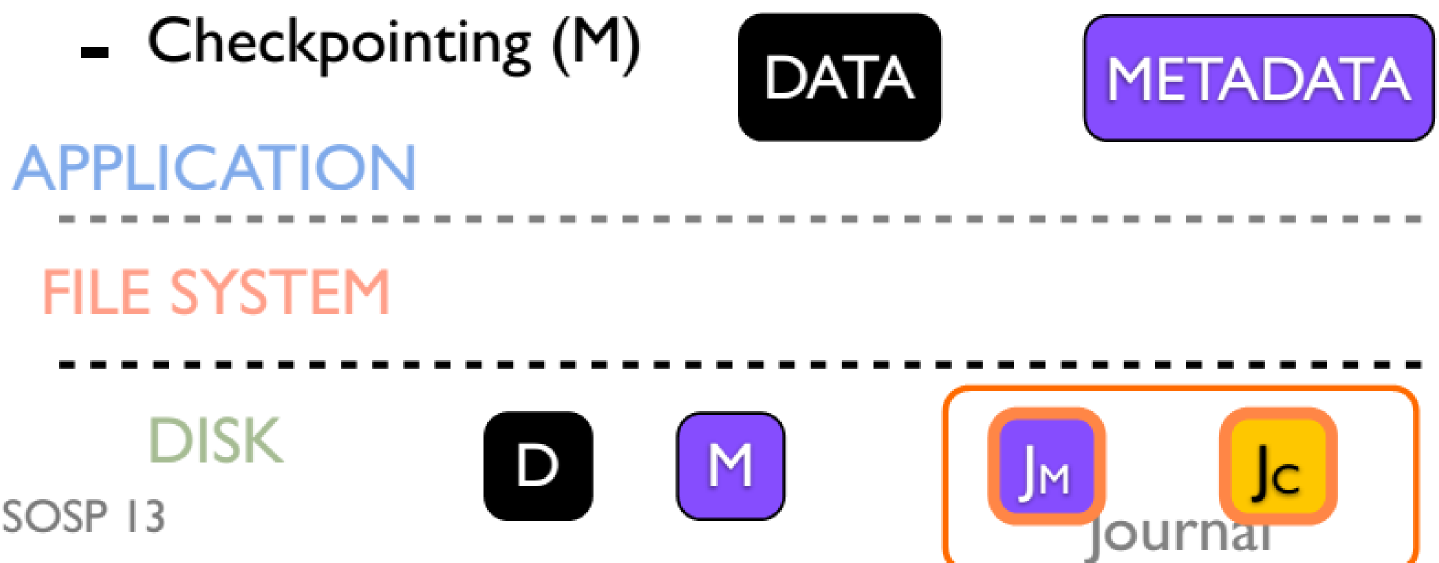


Journaling Overview

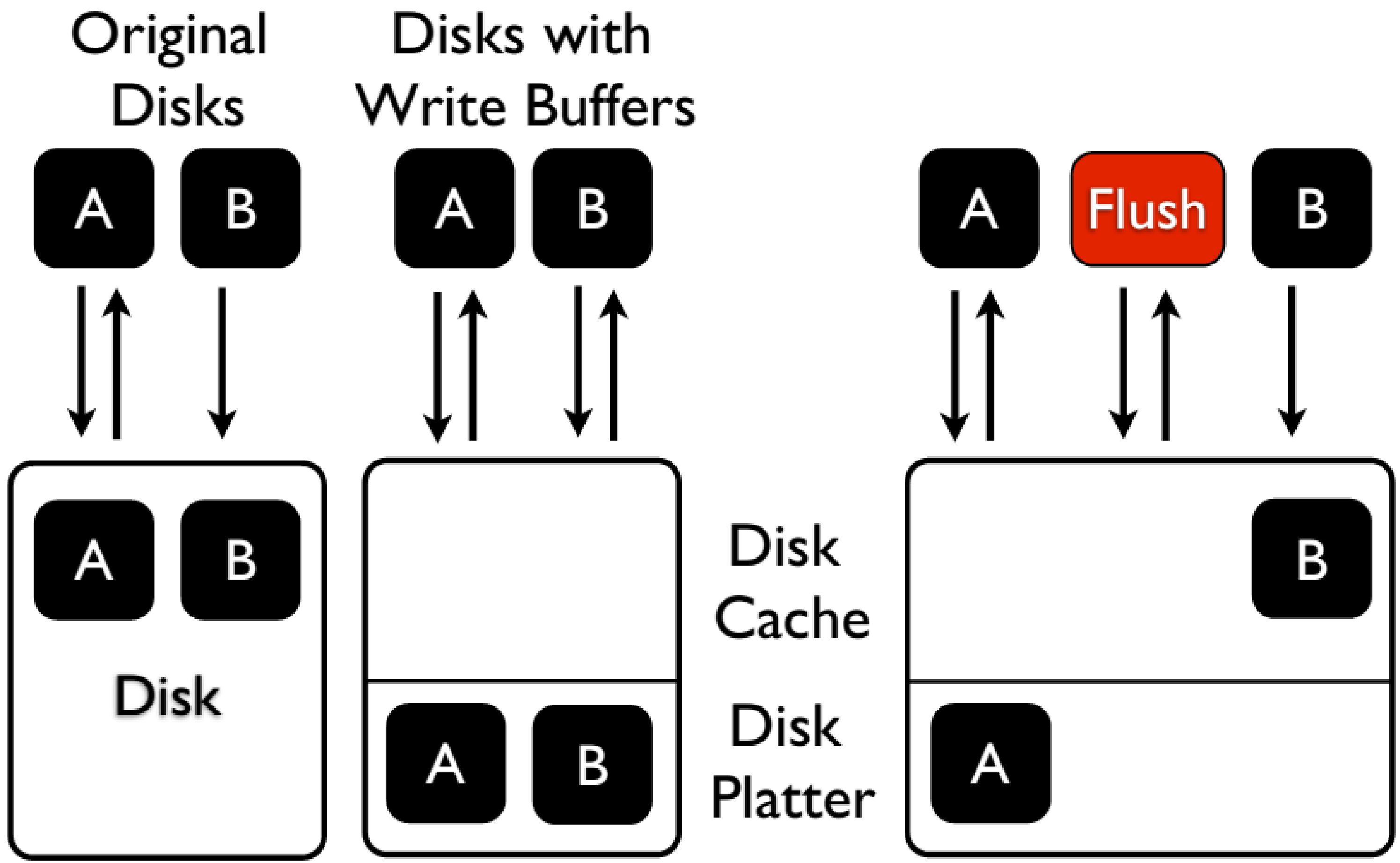
Workload: Creating and writing to a file

Journaling protocol (ordered journaling)

- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)
- Checkpointing (M)



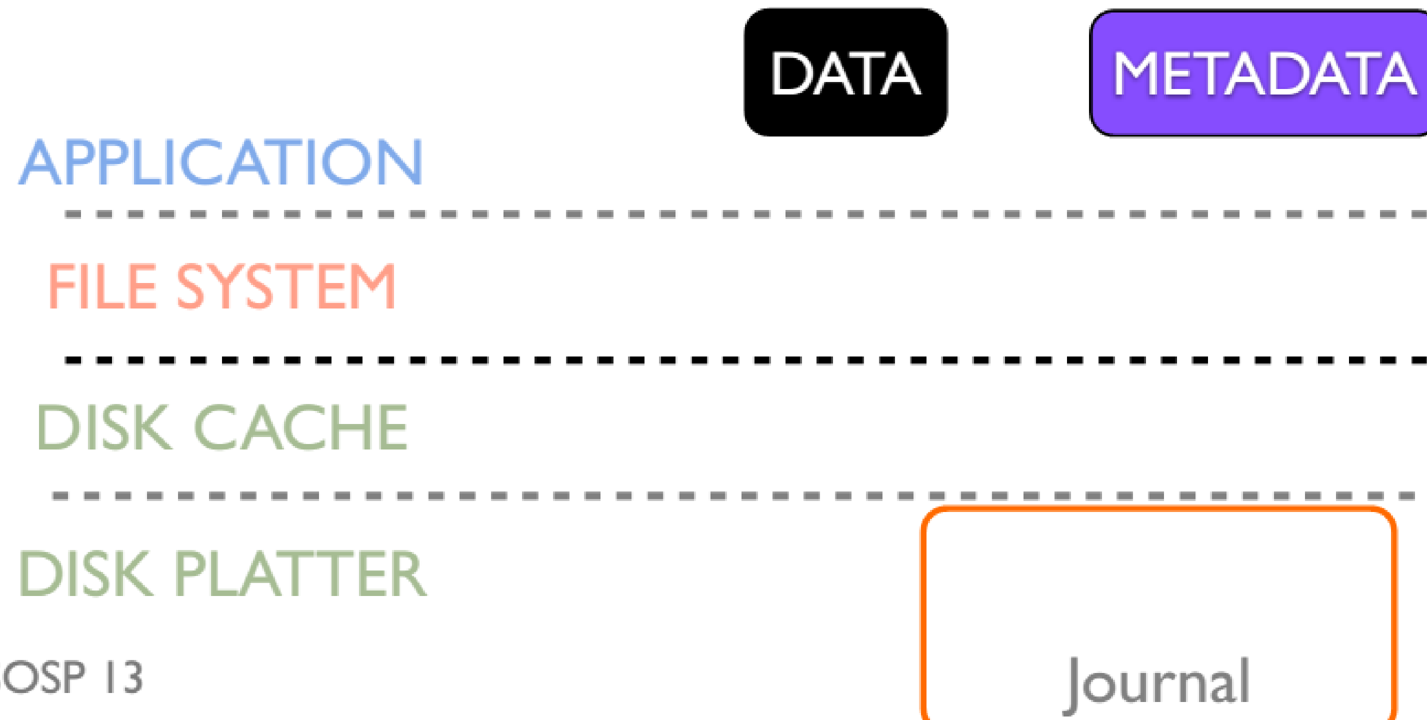
How Writes are Ordered



Journaling with Flushes

Journaling protocol

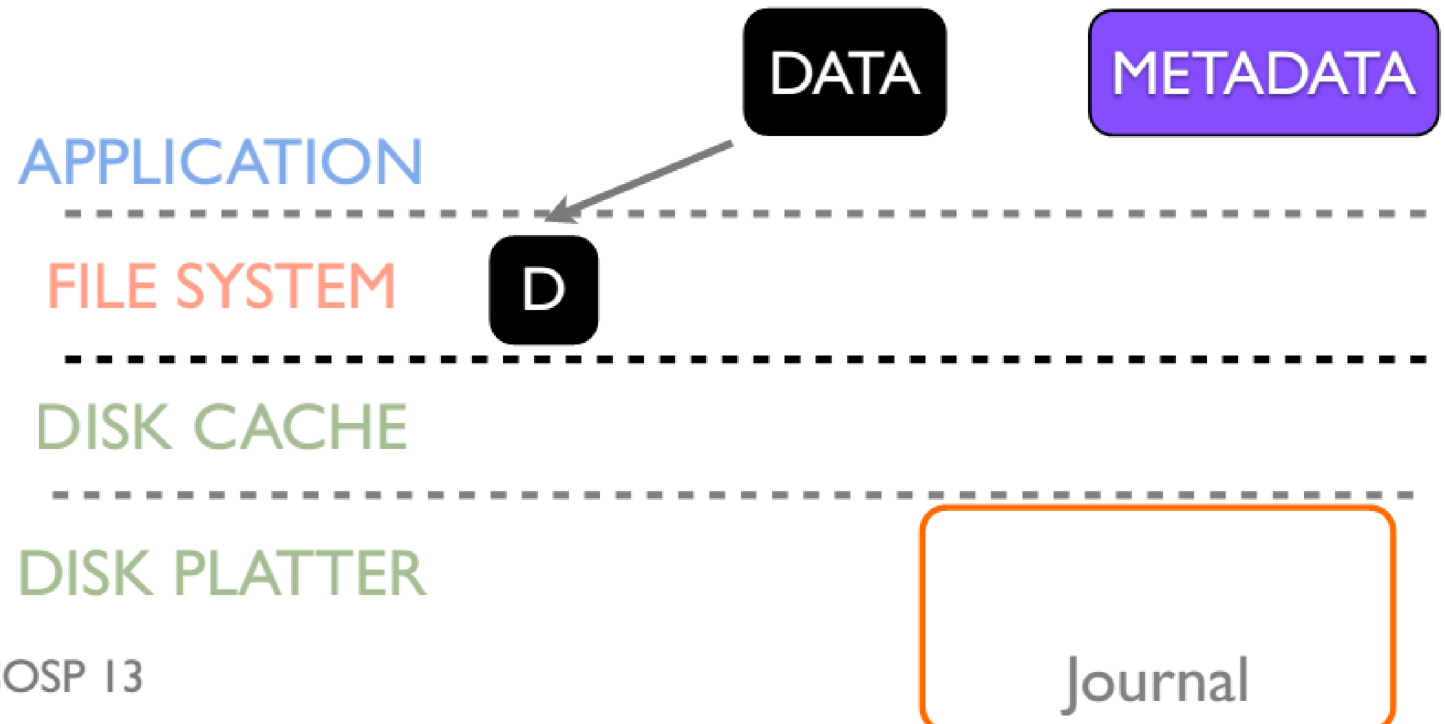
- Data write (D)



Journaling with Flushes

Journaling protocol

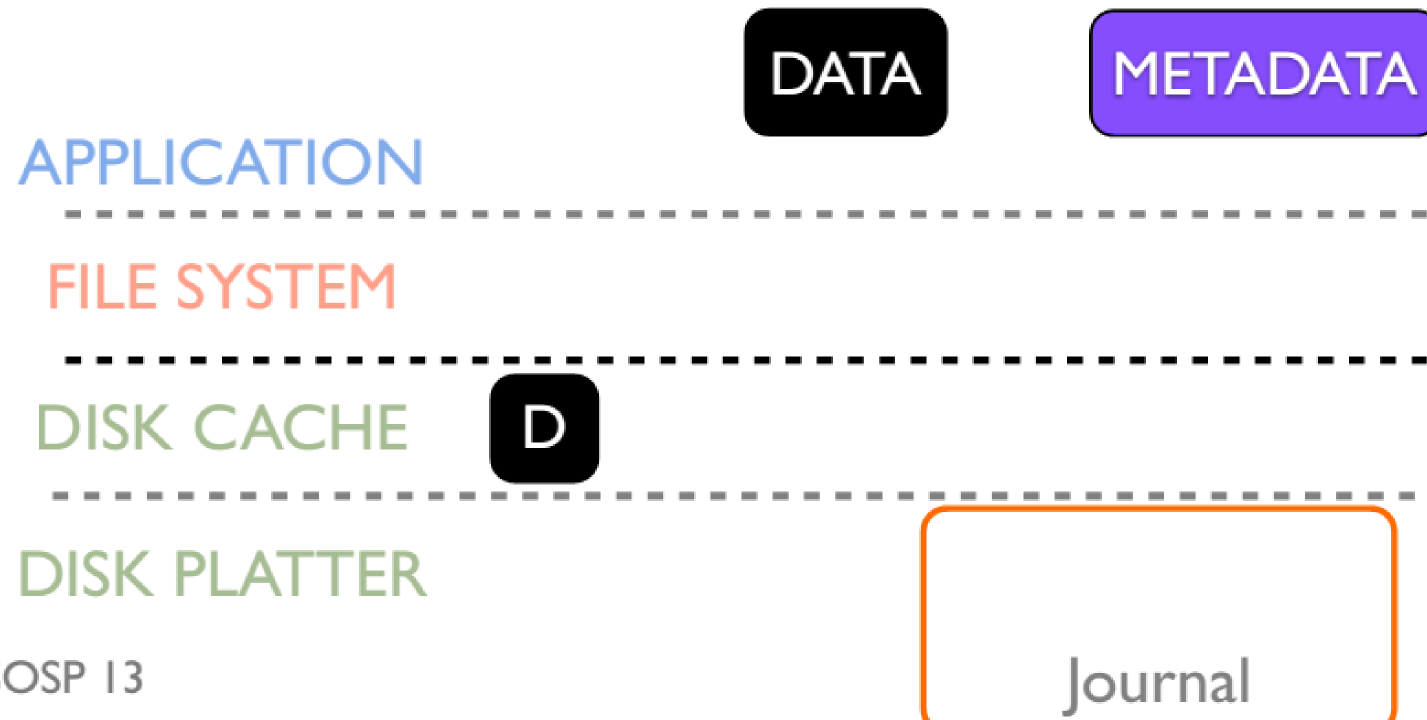
- Data write (D)



Journaling with Flushes

Journaling protocol

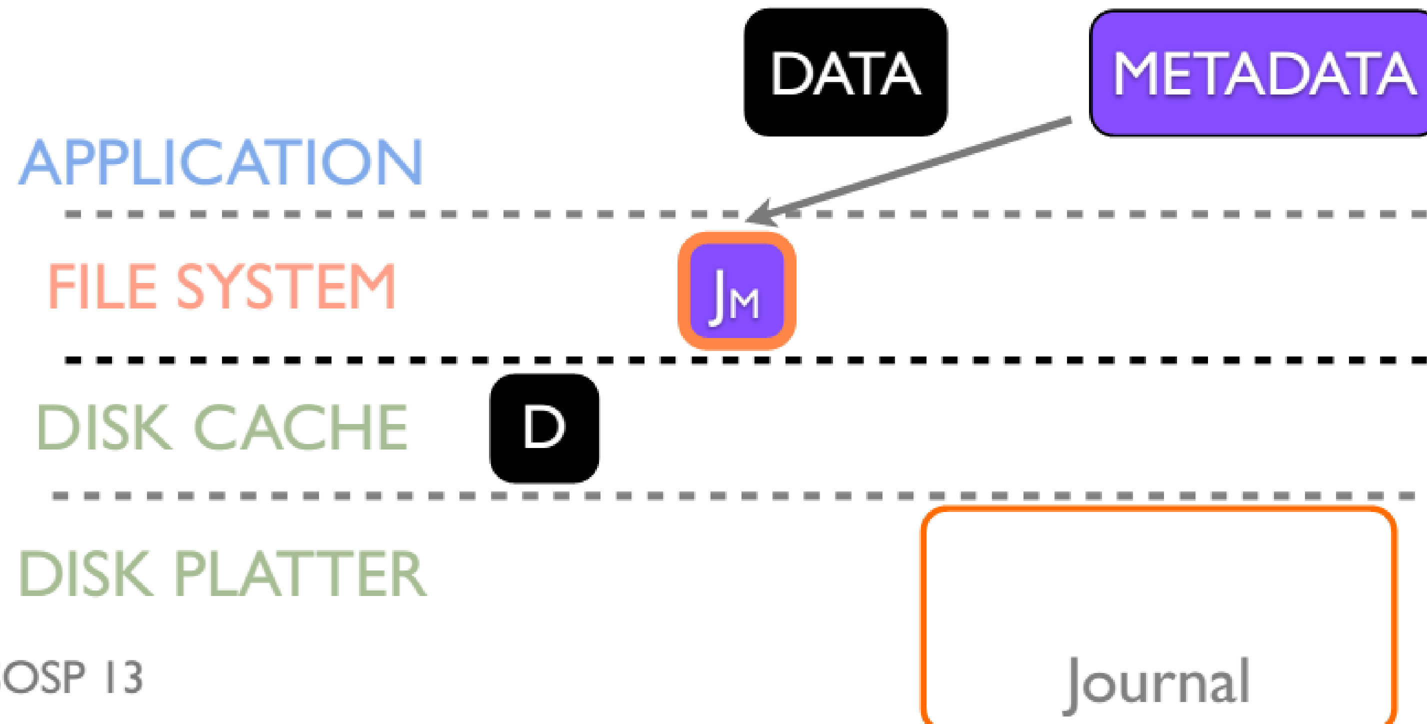
- Data write (D)



Journaling with Flushes

Journaling protocol

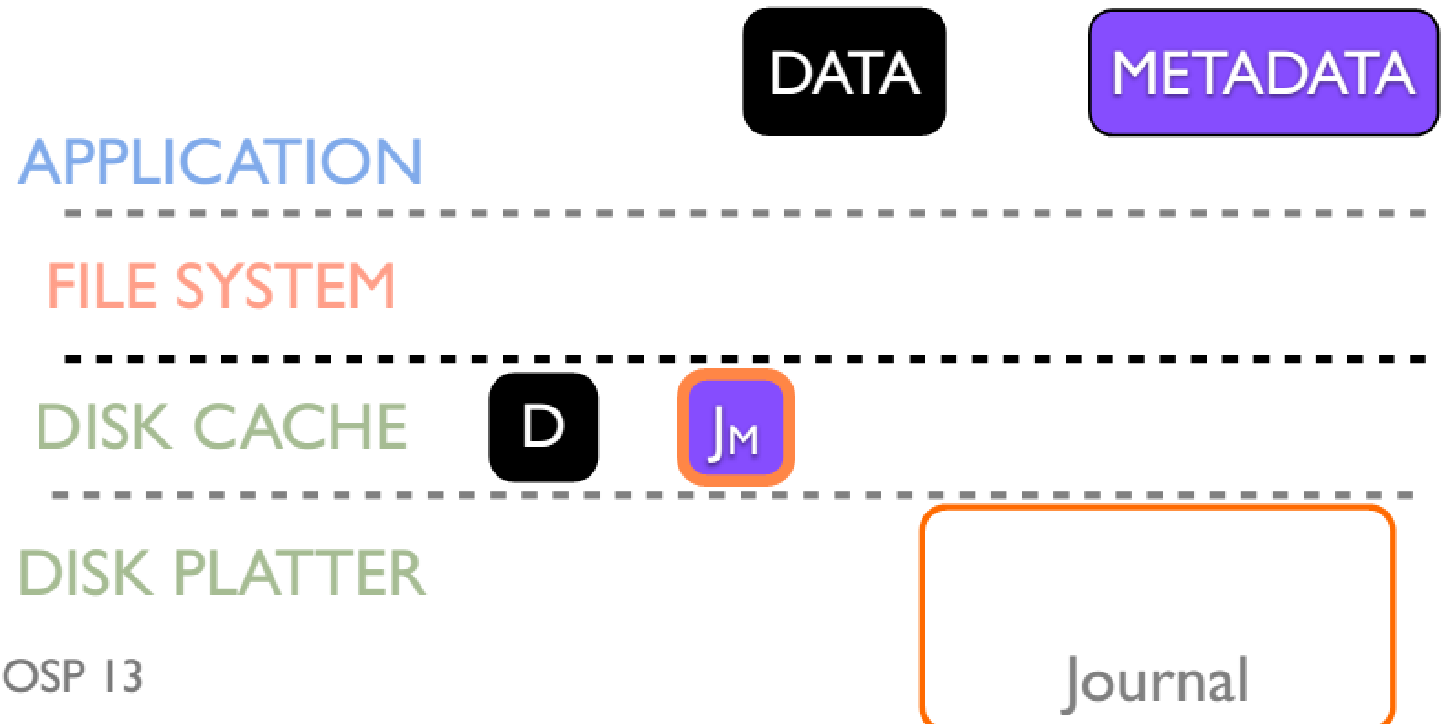
- Data write (D)
- Logging Metadata (J_M)



Journaling with Flushes

Journaling protocol

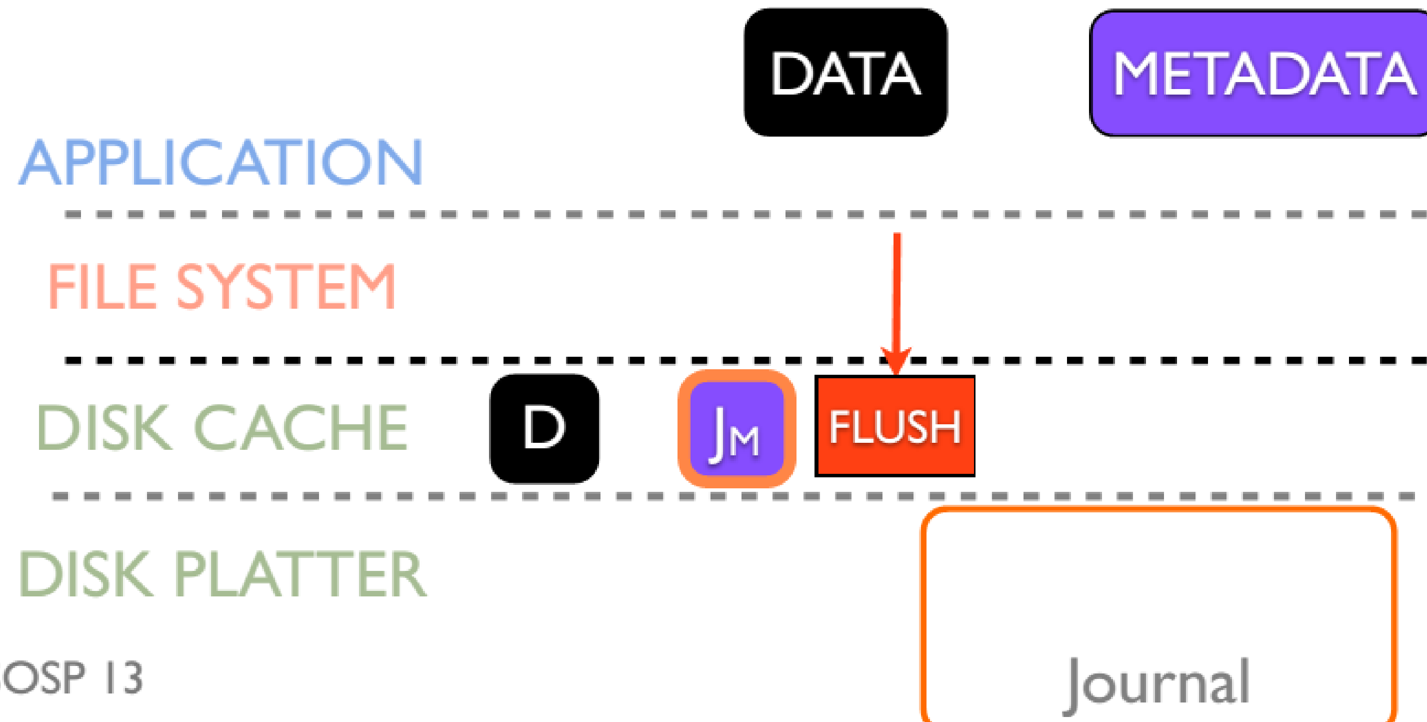
- Data write (D)
- Logging Metadata (J_M)



Journaling with Flushes

Journaling protocol

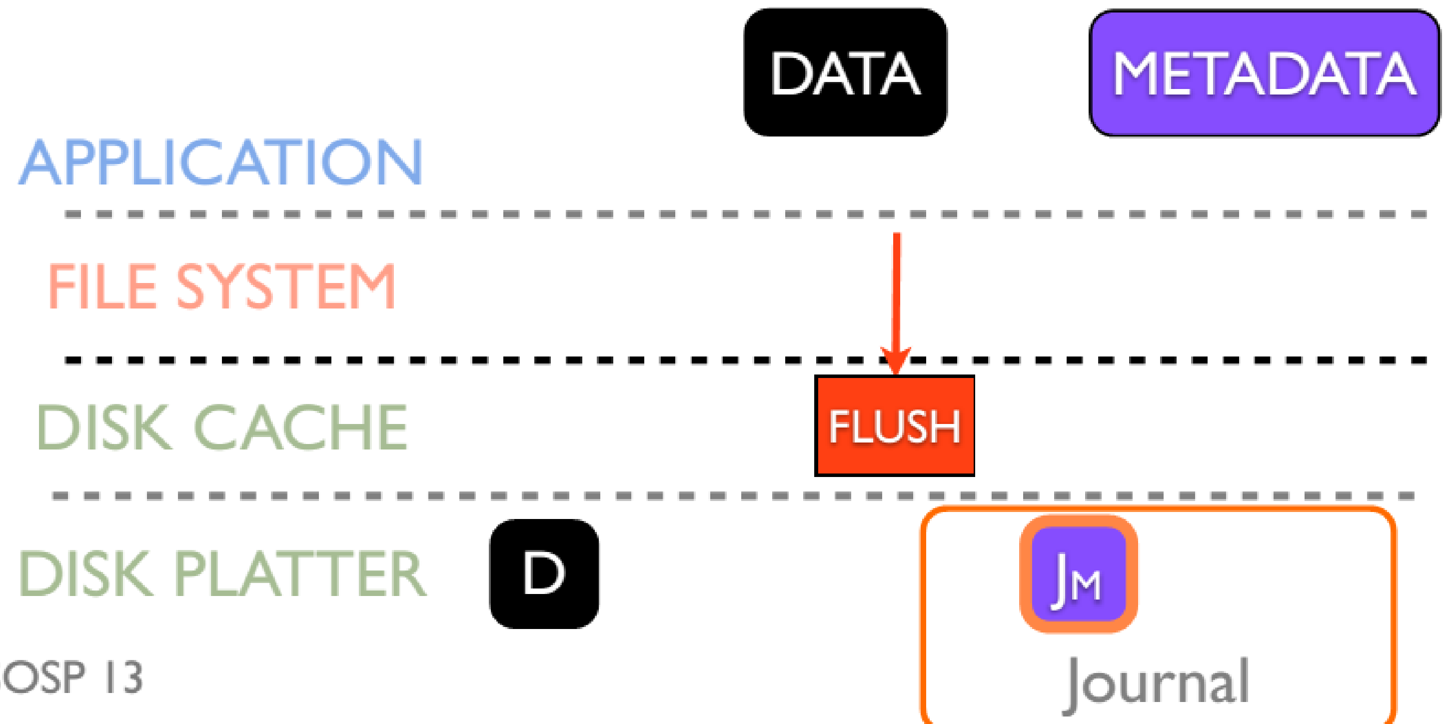
- Data write (D)
- Logging Metadata (J_M)



Journaling with Flushes

Journaling protocol

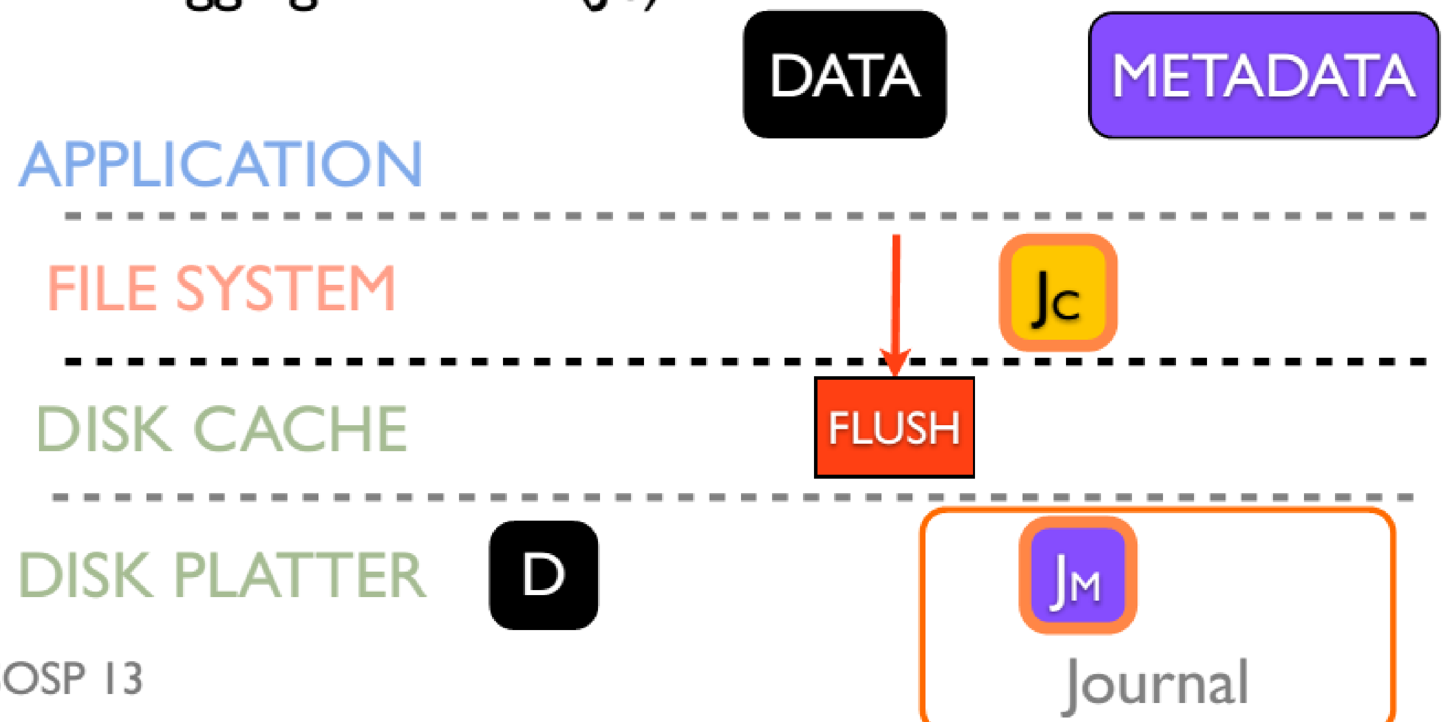
- Data write (D)
- Logging Metadata (J_M)



Journaling with Flushes

Journaling protocol

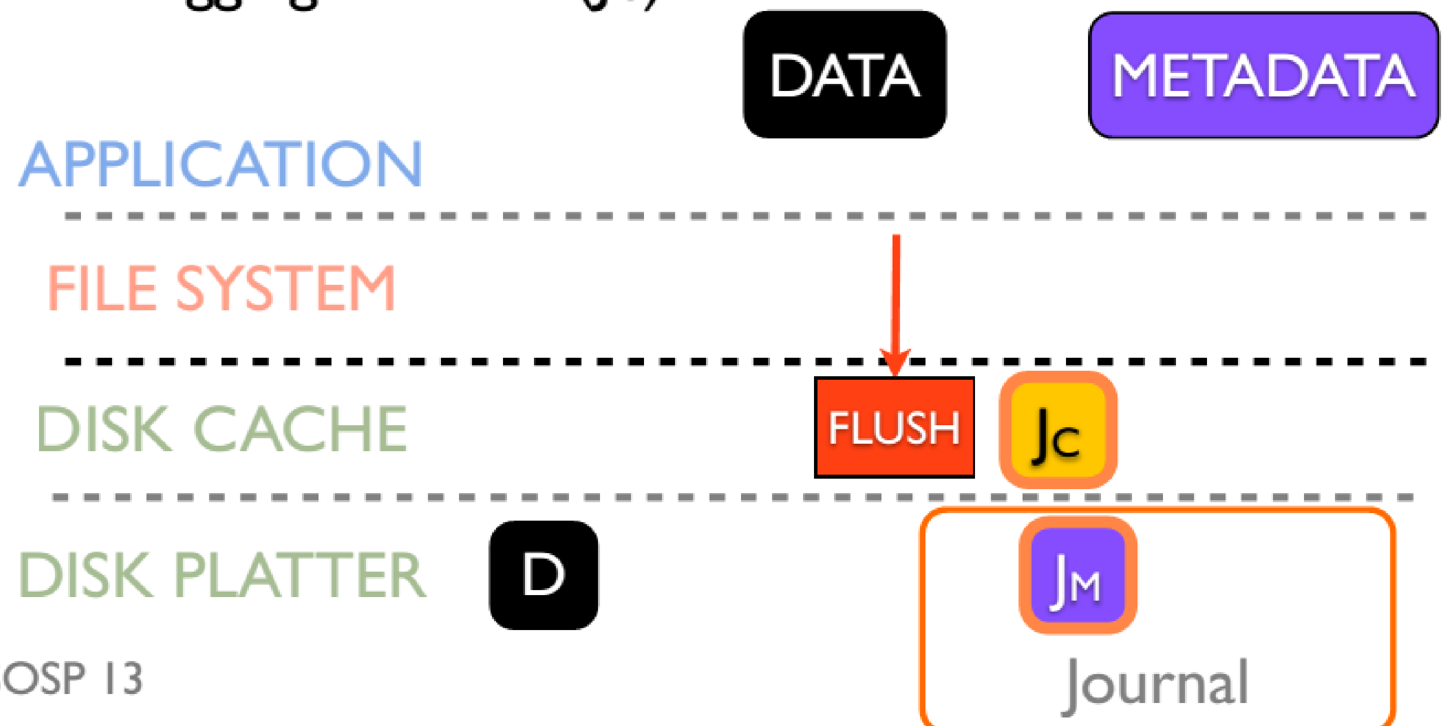
- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)



Journaling with Flushes

Journaling protocol

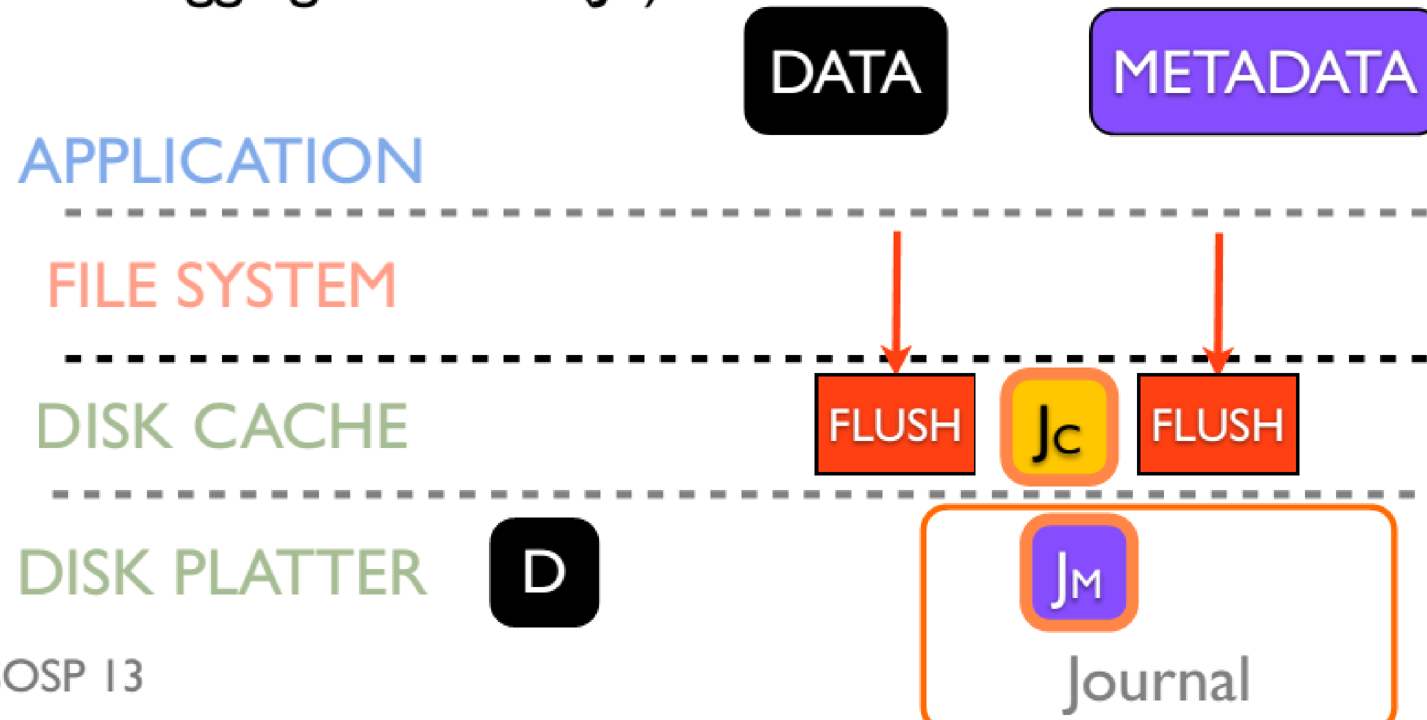
- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)



Journaling with Flushes

Journaling protocol

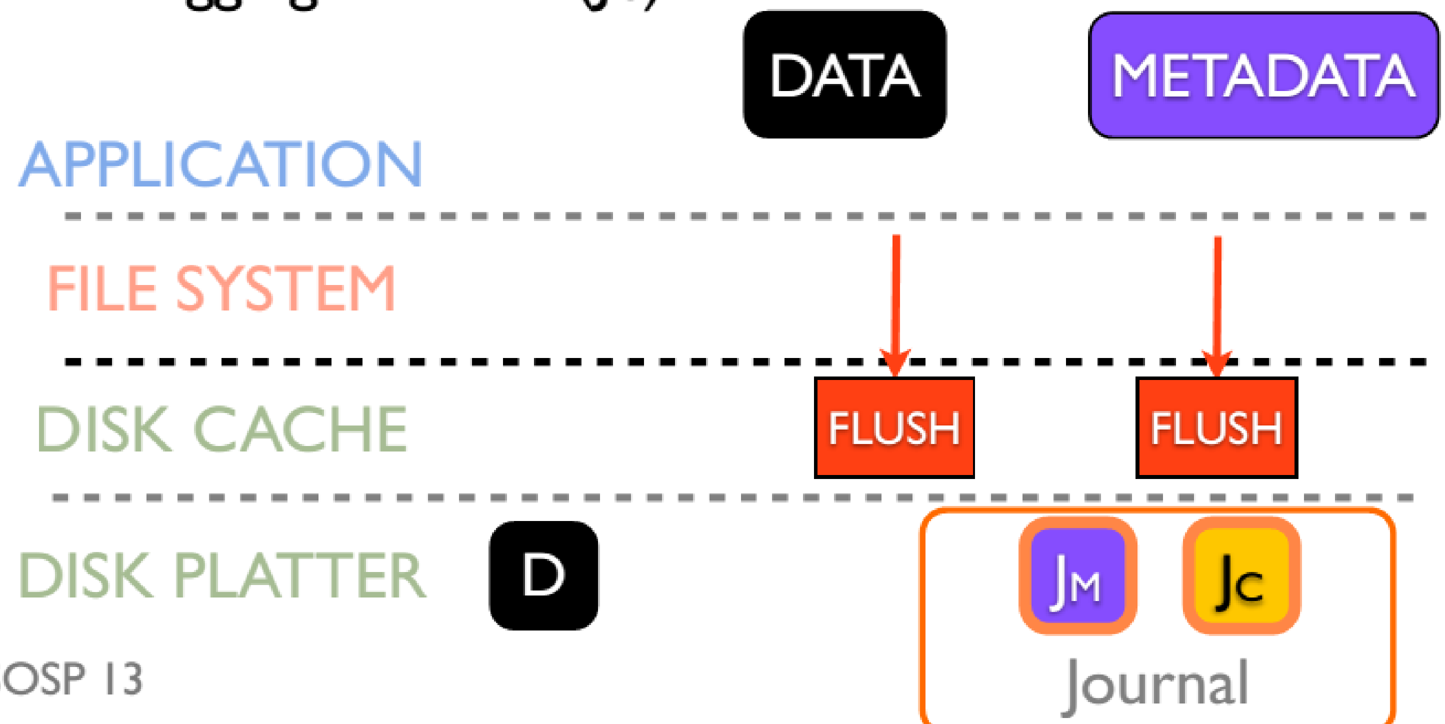
- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)



Journaling with Flushes

Journaling protocol

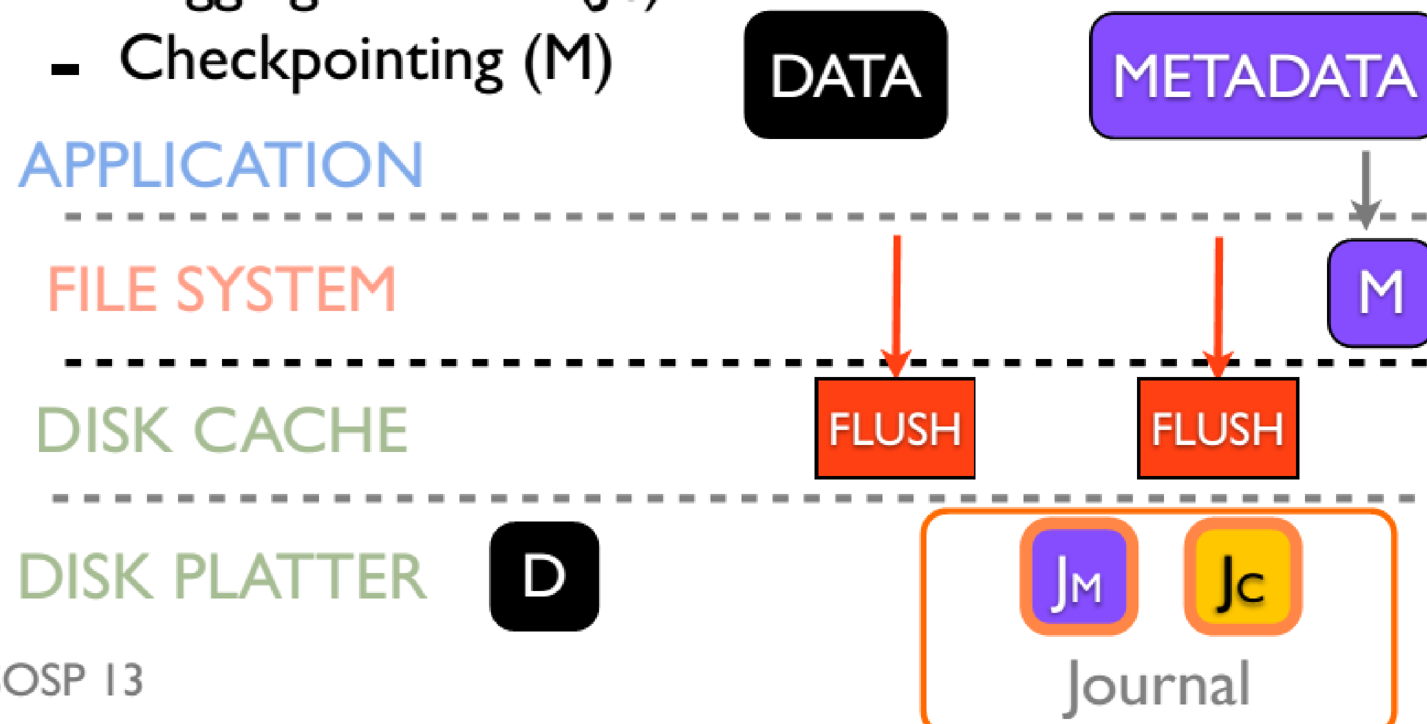
- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)



Journaling with Flushes

Journaling protocol

- Data write (D)
- Logging Metadata (J_M)
- Logging Commit (J_C)
- Checkpointing (M)



JOURNALING WITHOUT ORDERING

Journaling without Ordering

Practitioners turn off flushes due to performance degradation

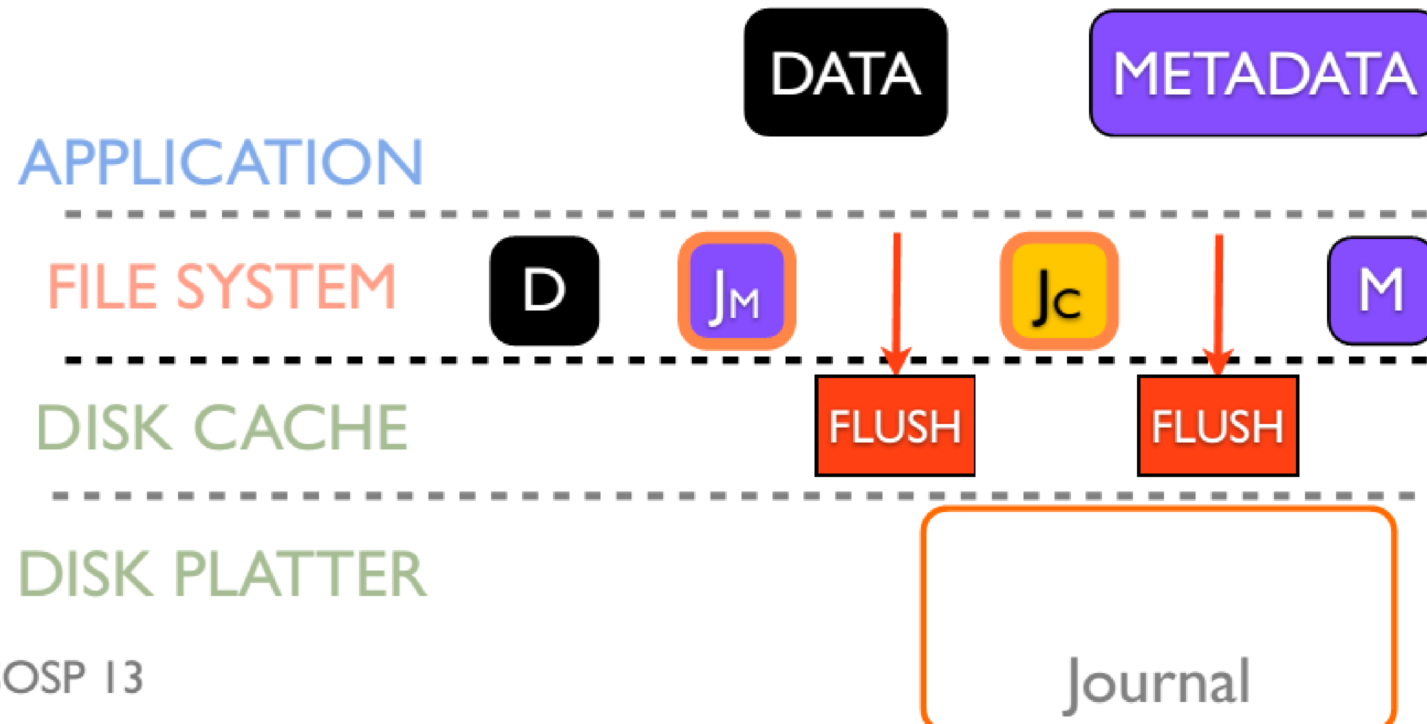
- Ex: ext3 by default did not enable flushes for many years

Observe crashes do not cause inconsistency for some workloads

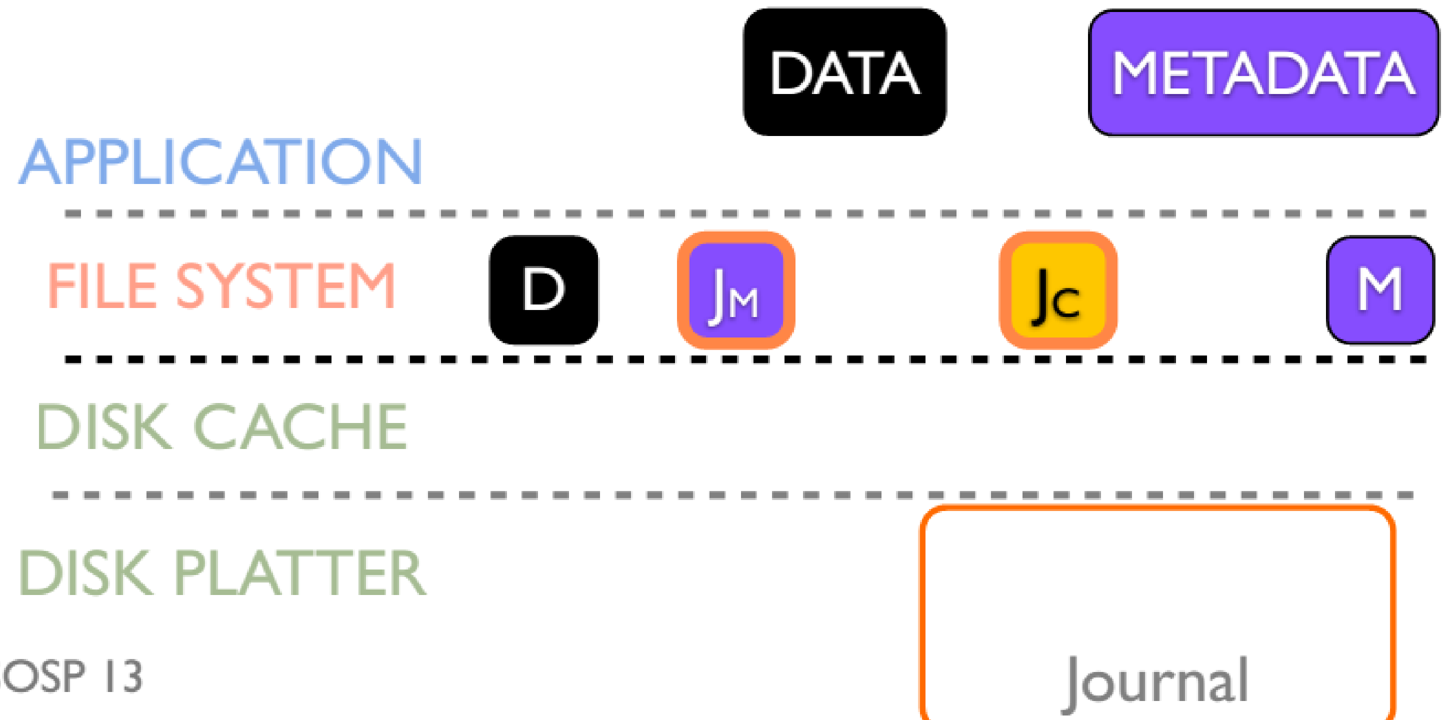
We term this probabilistic crash consistency

- Studied in detail

Journaling without Ordering

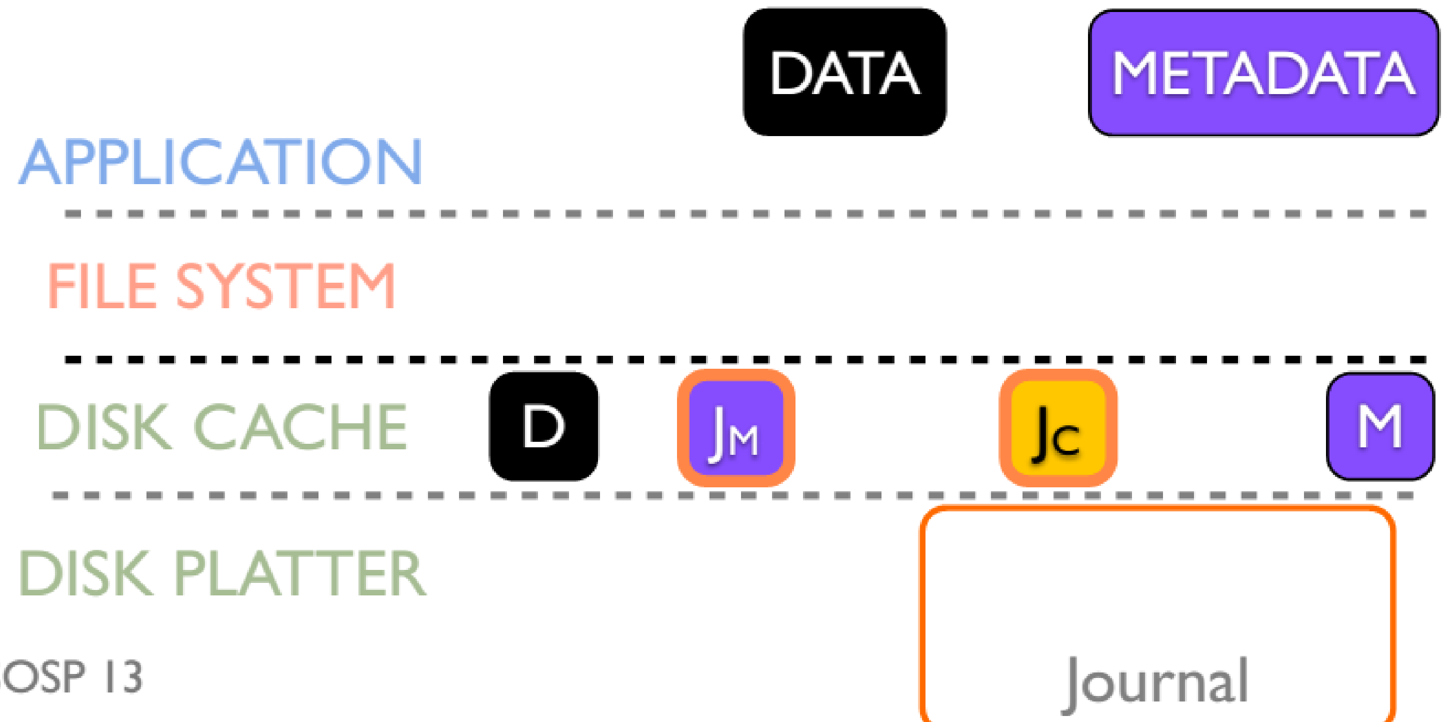


Journaling without Ordering



Journaling without Ordering

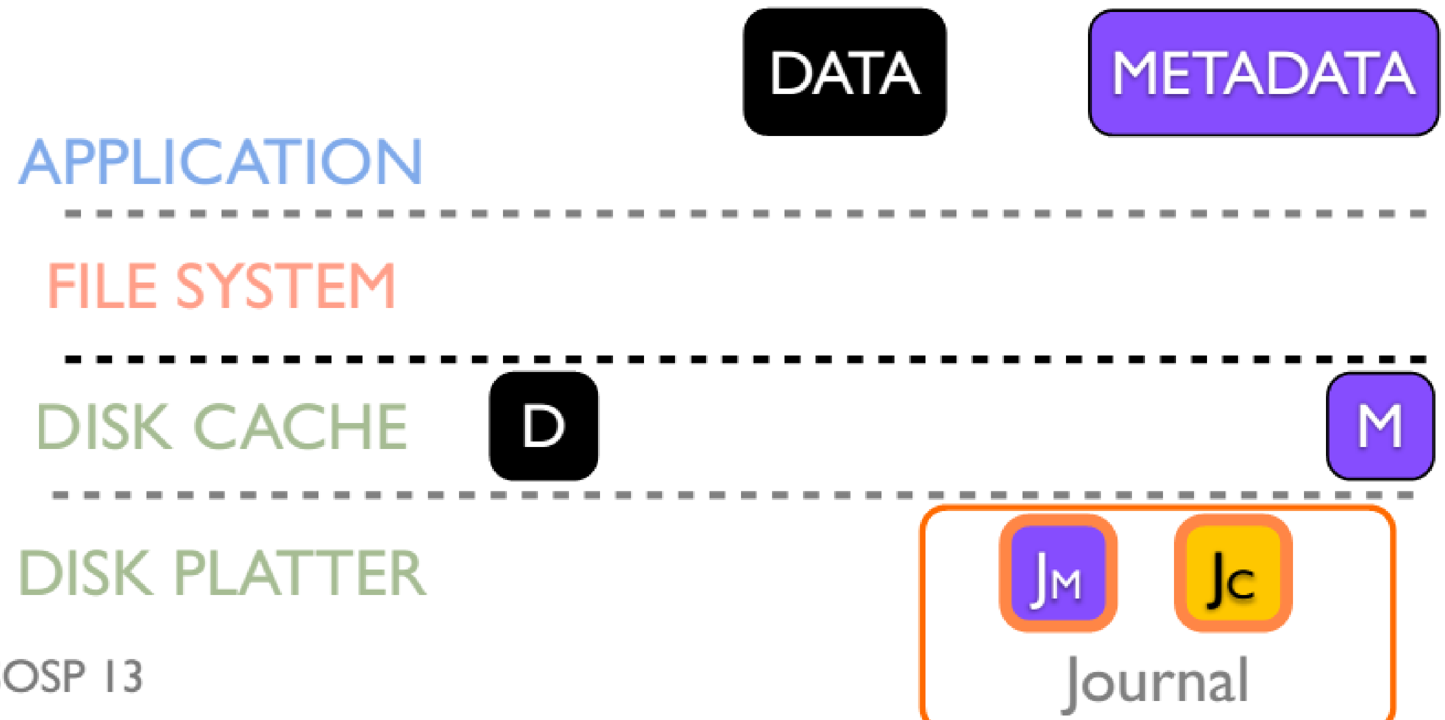
Without flushes, blocks may be reordered



Journaling without Ordering

Without flushes, blocks may be reordered

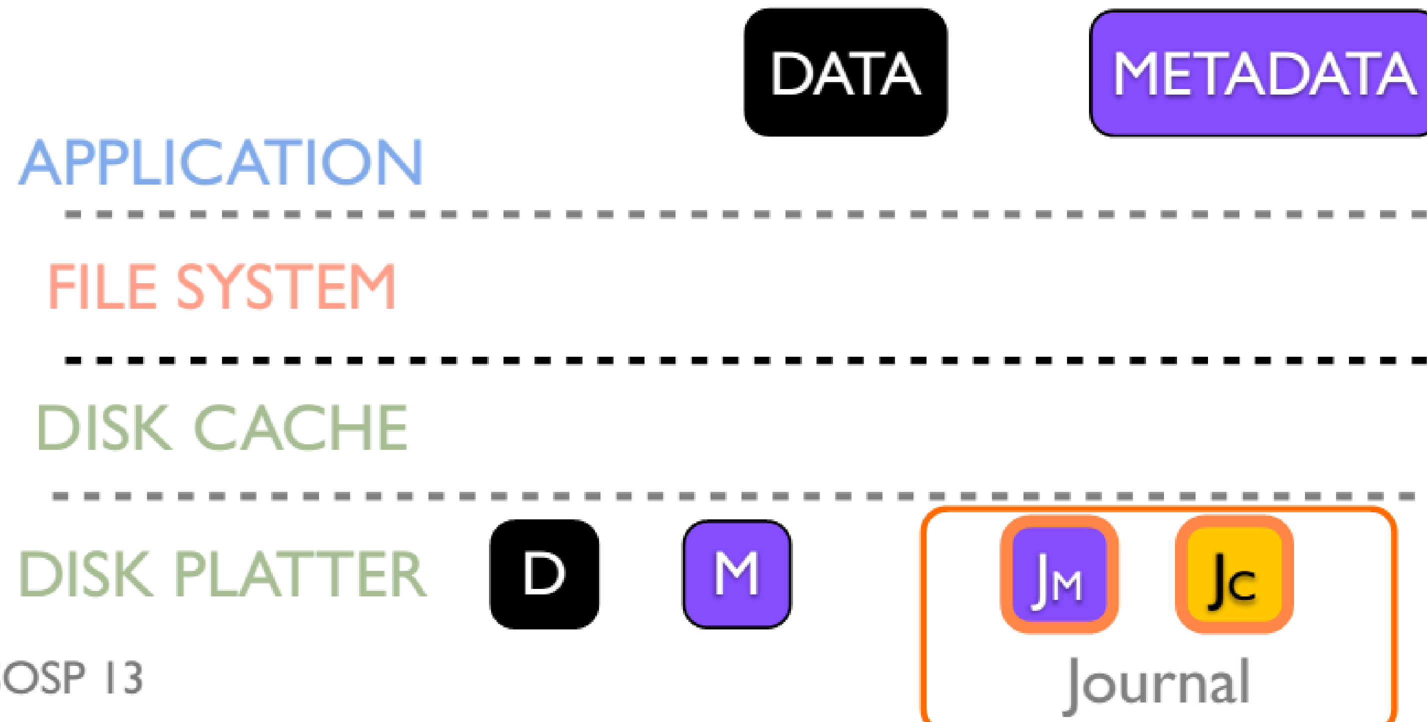
- Ex: J_C and J_M written first as disk head near journal



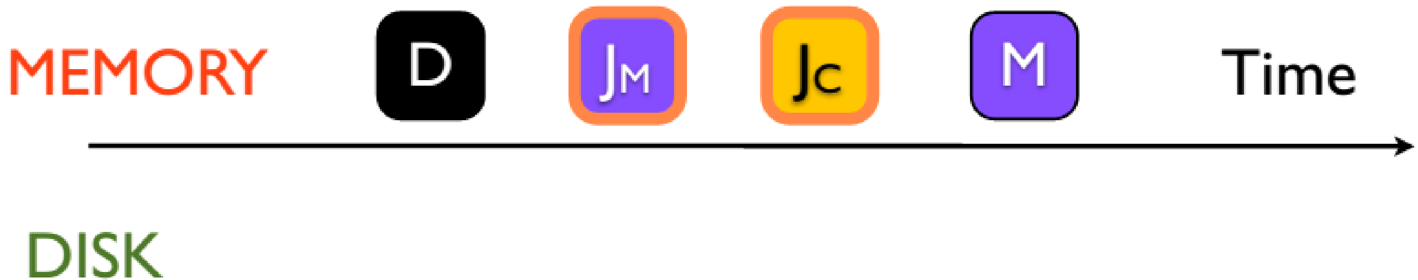
Journaling without Ordering

Without flushes, blocks may be reordered

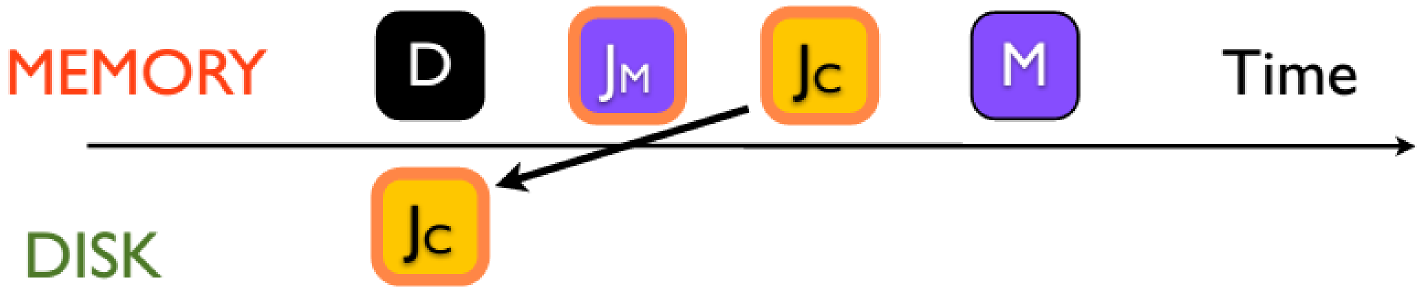
- Ex: J_C and J_M written first as disk head near journal



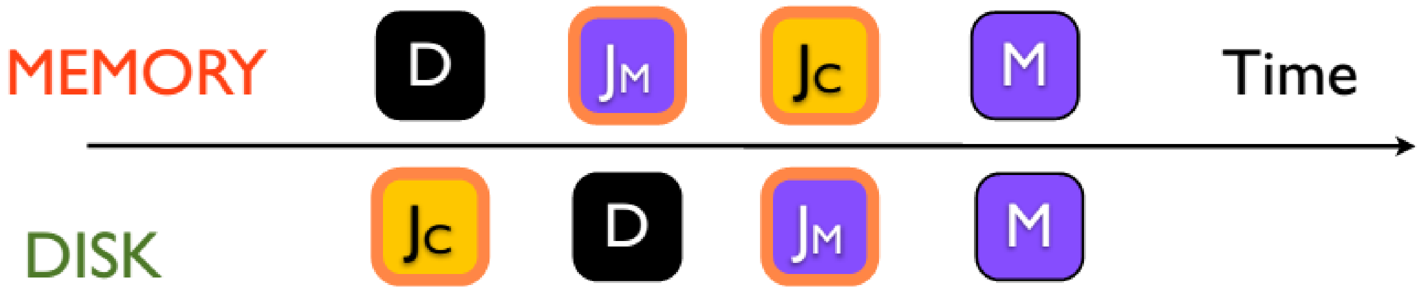
Probabilistic Crash Consistency



Probabilistic Crash Consistency

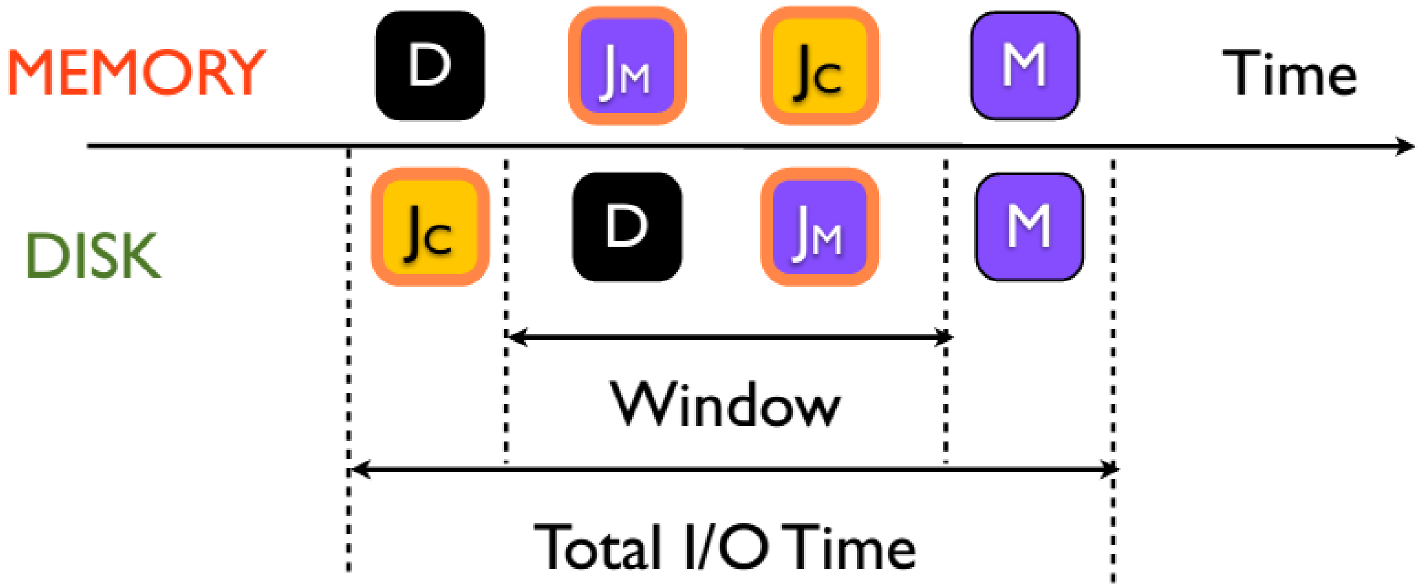


Probabilistic Crash Consistency



Probabilistic Crash Consistency

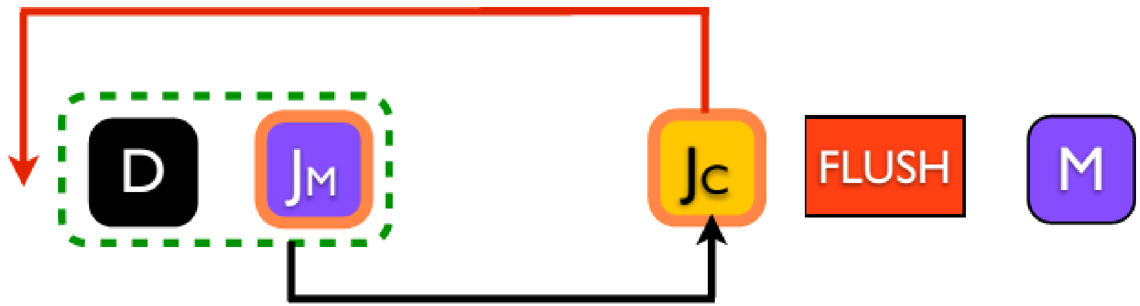
Re-ordering leads to windows of vulnerability



P-inconsistency = Time in window(s) / Total I/O Time

Technique #1: Checksums

J_C could be re-ordered before D or J_M



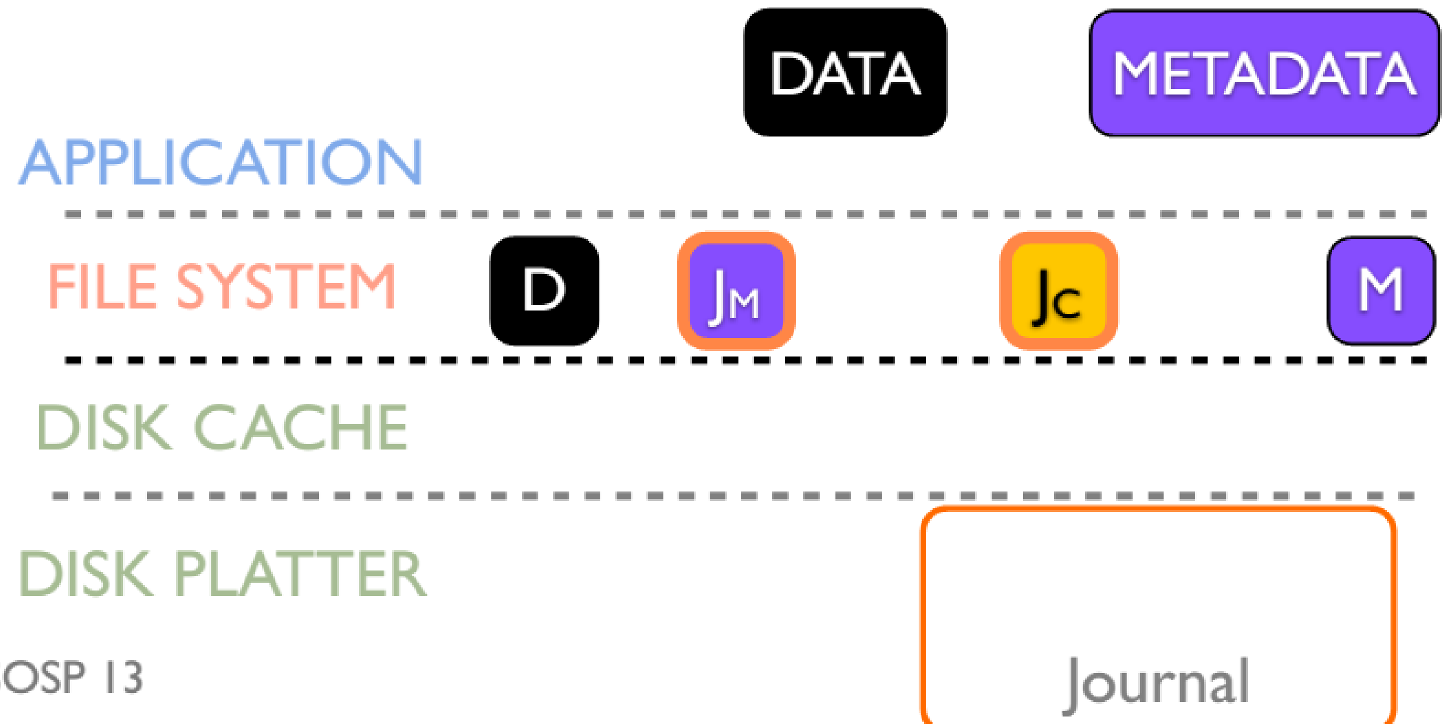
Re-ordering detected using checksums

- Computed over data and metadata
- Checked during recovery
- Mismatch indicates blocks were lost during crash

Handling Re-Ordering: Removing Flush #2

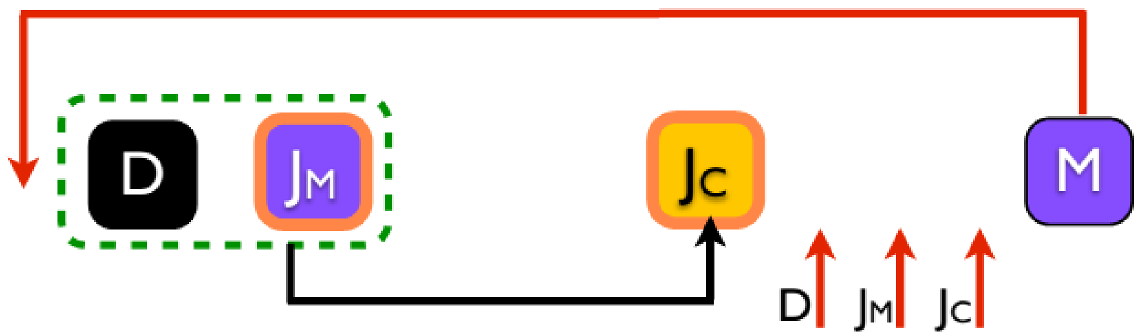
Flush after J_C is removed

- Delayed writes used to prevent reordering



Technique #2: Delayed Writes

M could be re-ordered before D or J_M or J_C

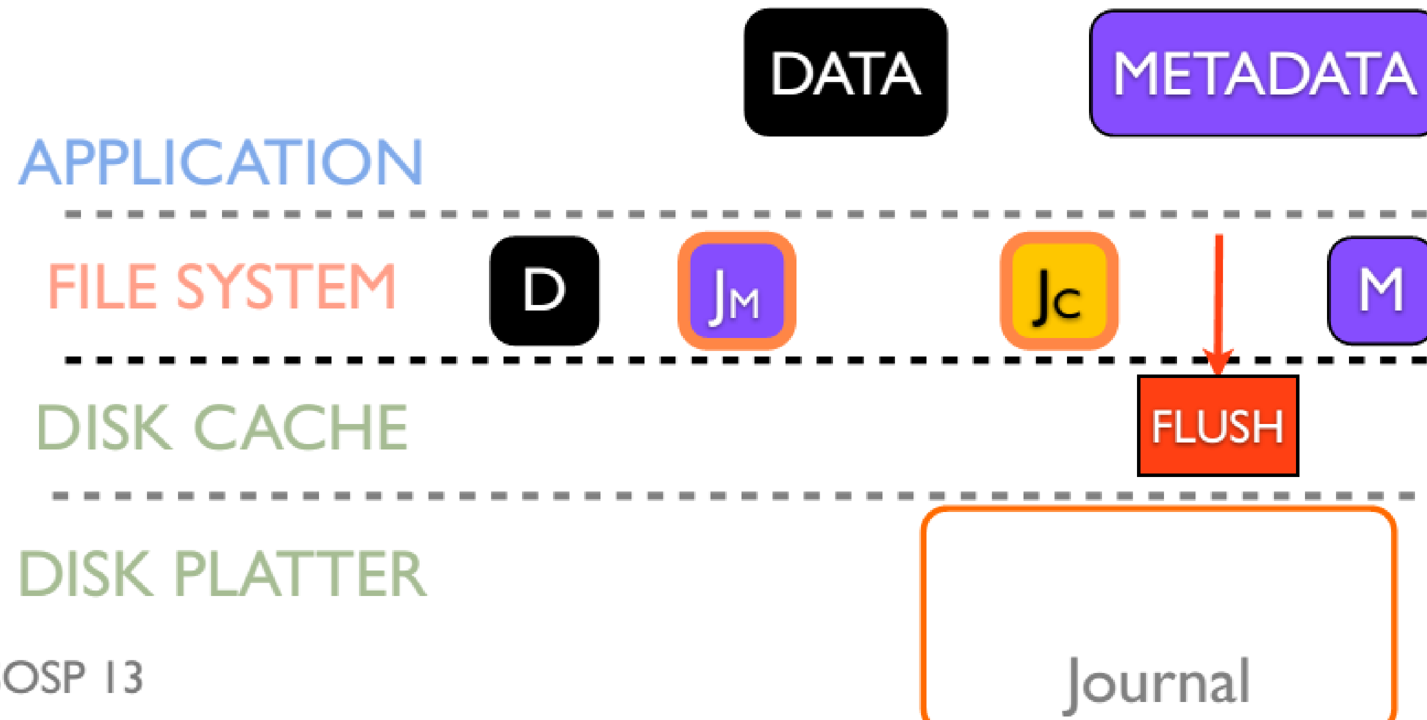


Re-ordering prevented using delayed writes

- Wait until ADN arrive for D, J_M, and J_C
- Then issue M to disk cache
- Invariant: D/J_M/J_C and M never dirty in cache together

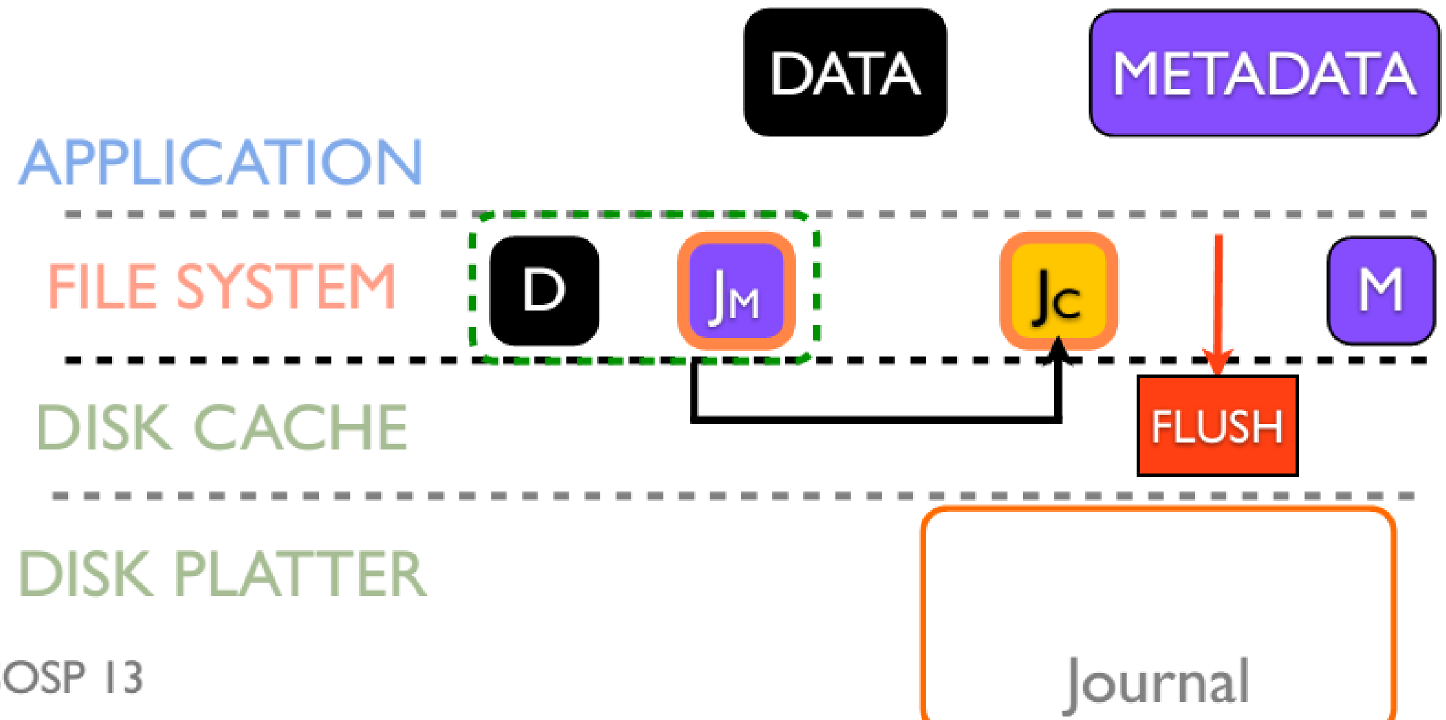
Optimistic Journaling

Checksums and Delayed Writes handle reordering from removing flushes



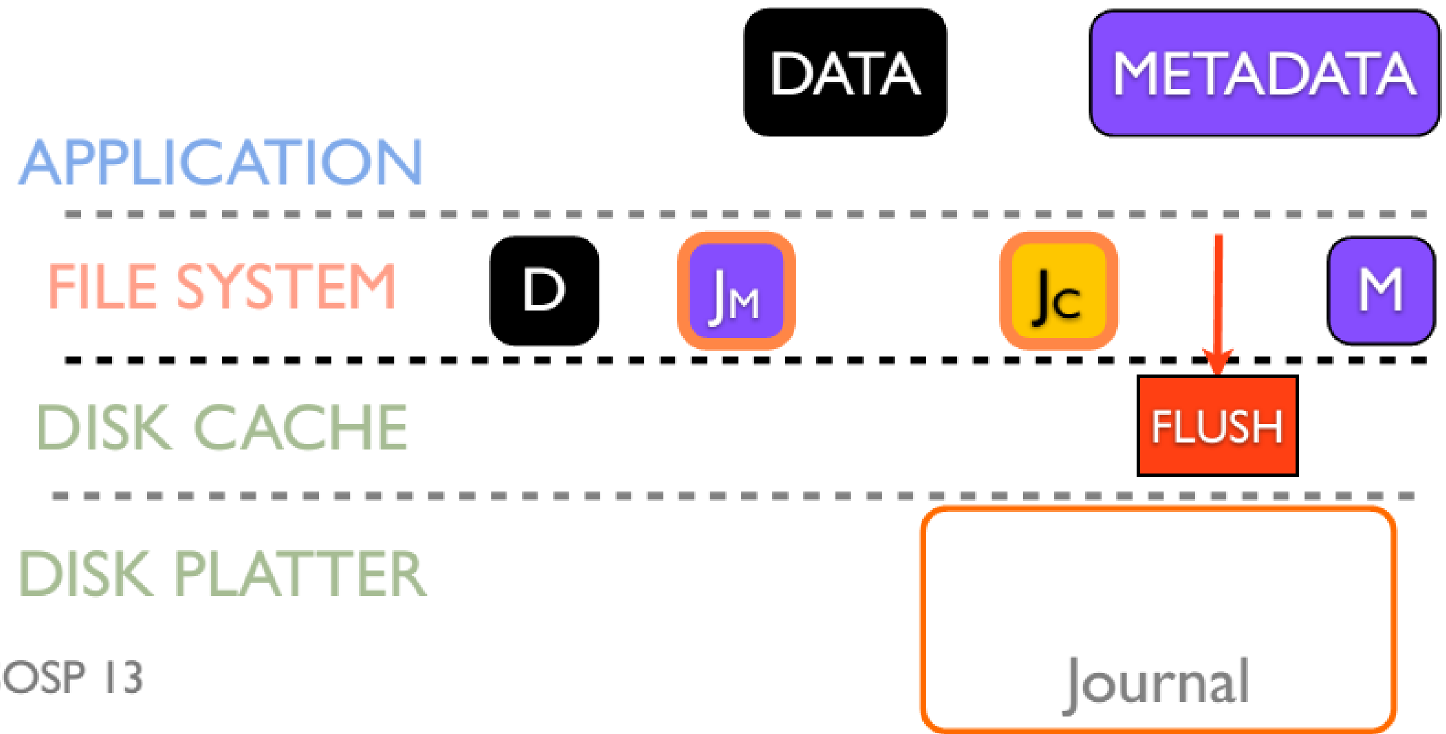
Optimistic Journaling

Checksums and Delayed Writes handle reordering from removing flushes



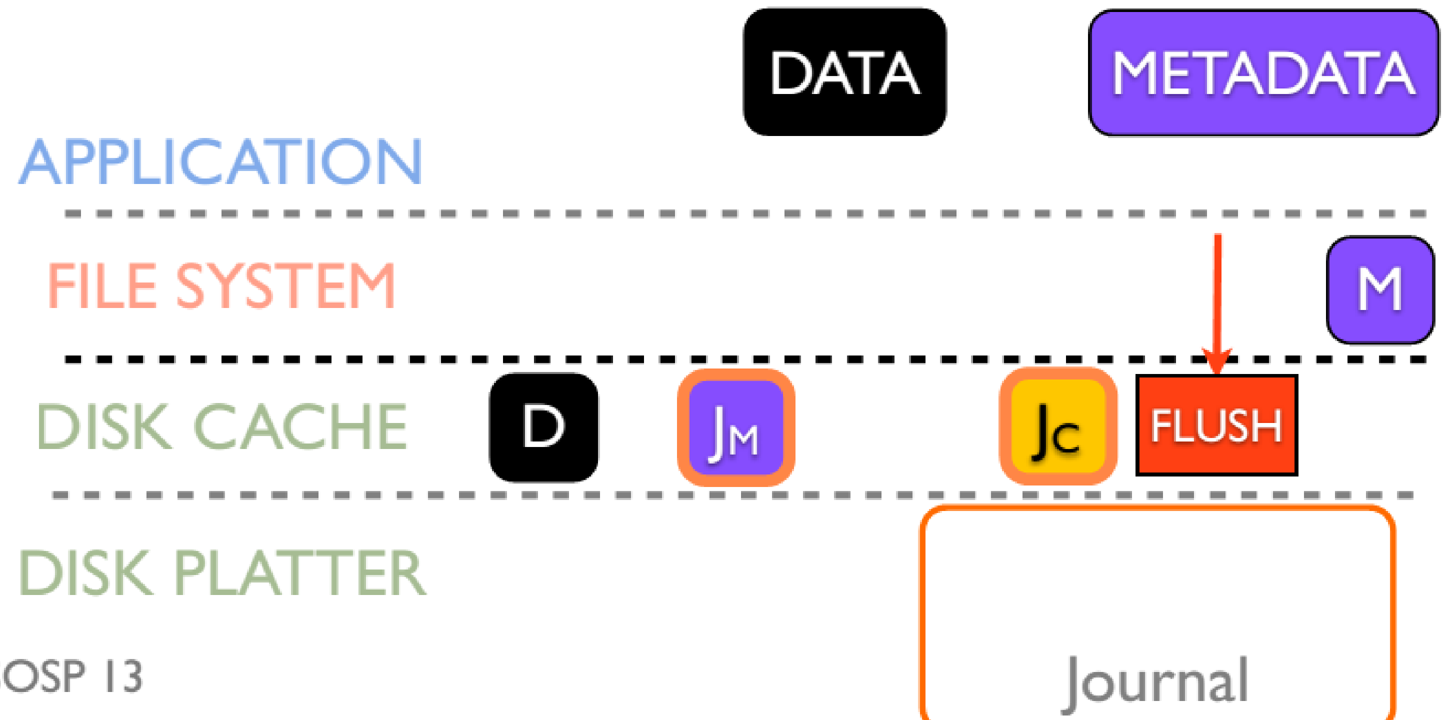
Optimistic Journaling

Checksums and Delayed Writes handle reordering from removing flushes



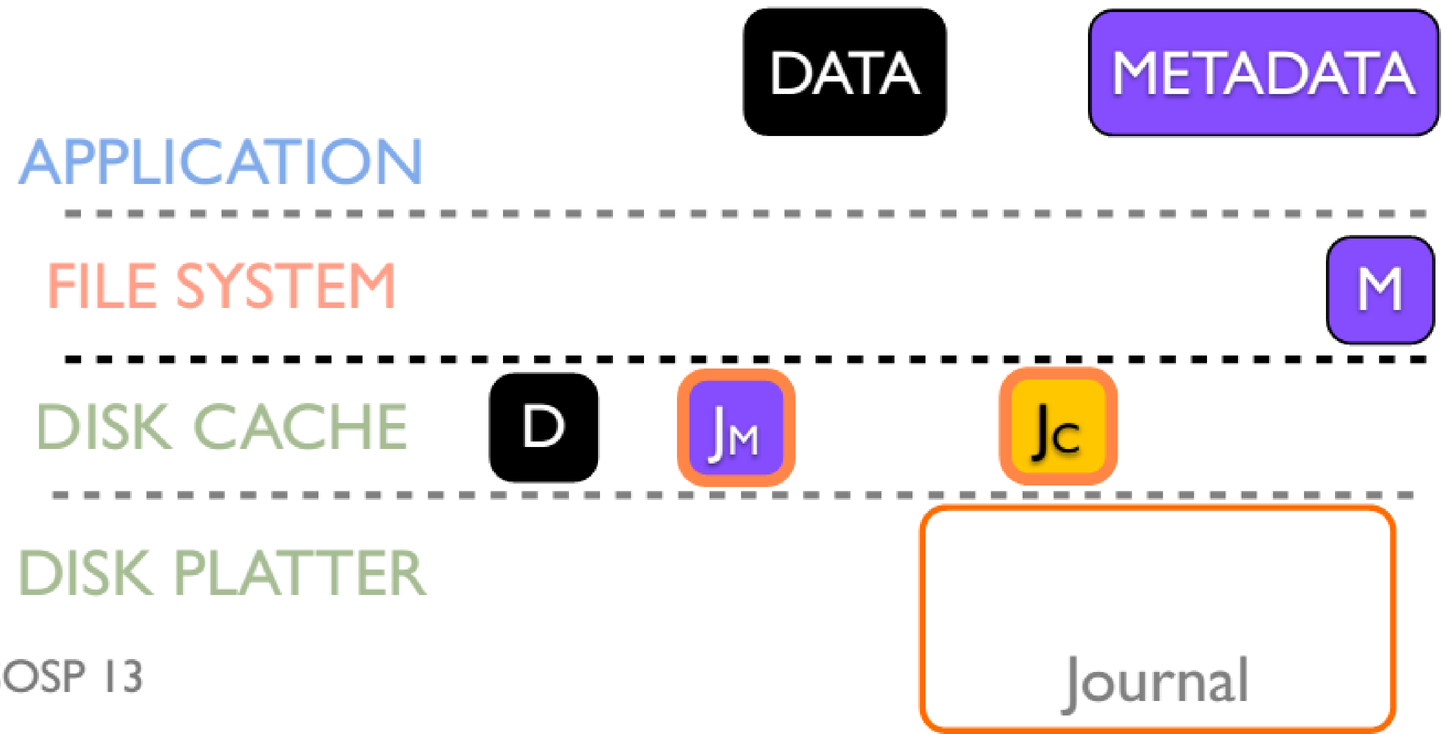
Optimistic Journaling

Checksums and Delayed Writes handle reordering from removing flushes



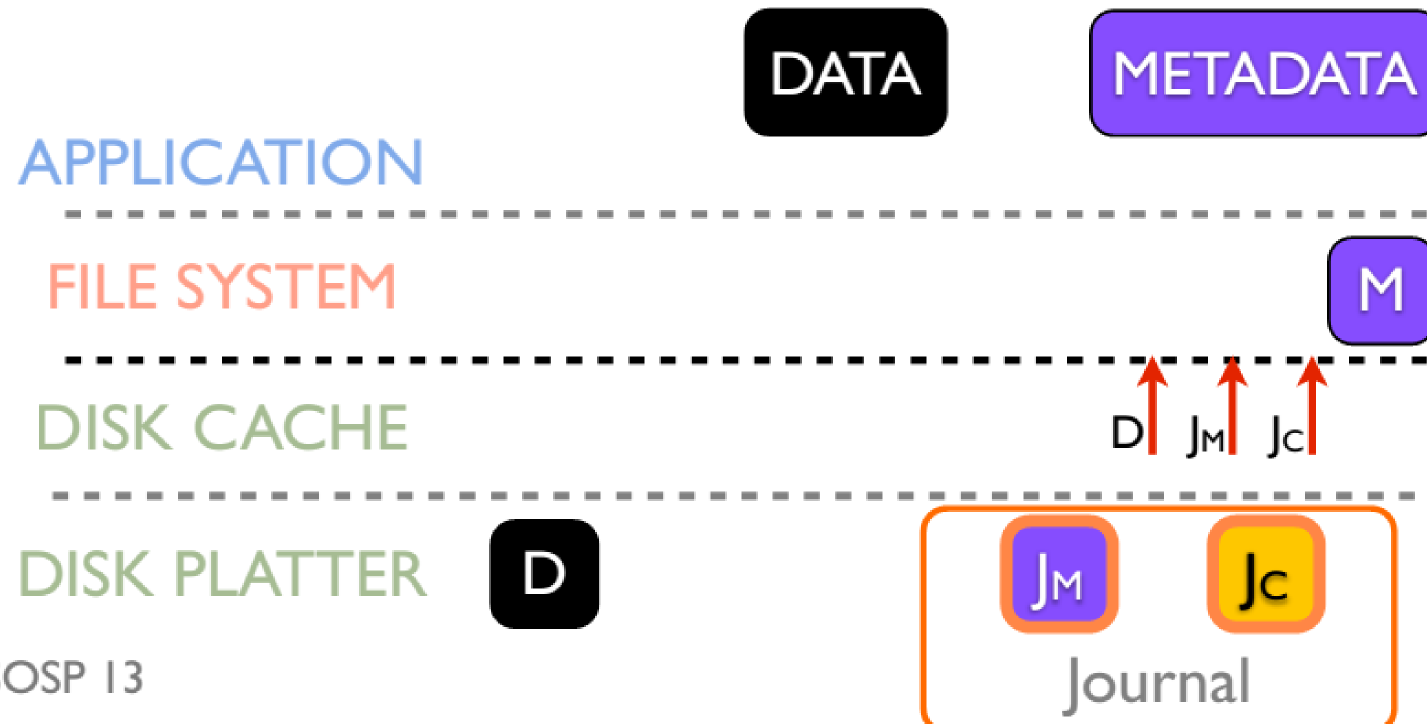
Optimistic Journaling

Checksums and Delayed Writes handle reordering from removing flushes



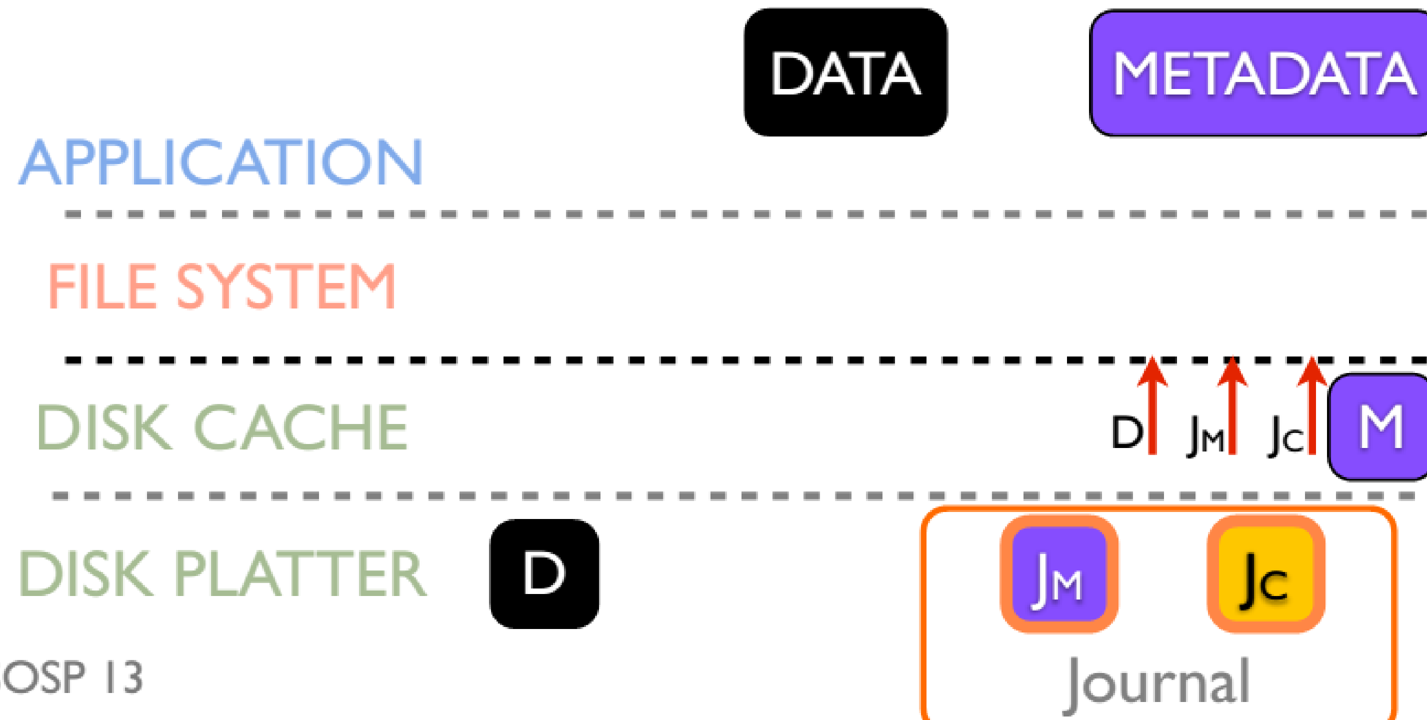
Optimistic Journaling

Checksums and Delayed Writes handle reordering from removing flushes



Optimistic Journaling

Checksums and Delayed Writes handle reordering from removing flushes



Evaluation

- Does OptFS preserve file-system consistency after crashes?
  - OptFS consistent after 400 random crashes
- How does OptFS perform?
  - OptFS 4-10x better than ext4 with flushes
- Can meaningful application-level consistency be built on top of OptFS?
  - Studied gedit and SQLite on OptFS