File Systems

Operating Systems

File Systems

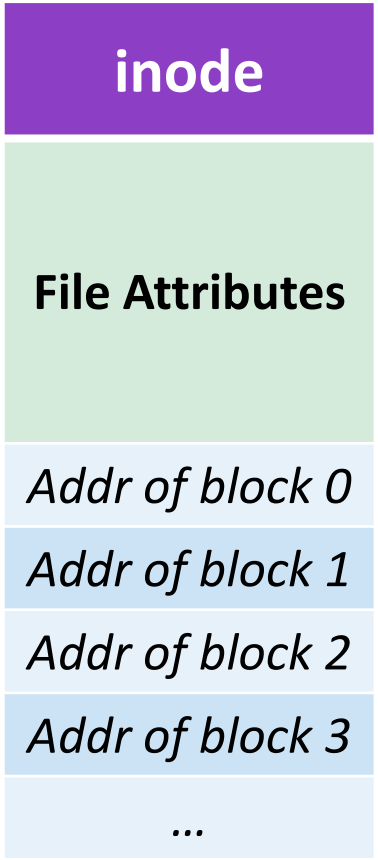
- What is a **file**? 
  - Array of **persistent** bytes that can be read/written

- **File system** 
  - Collection of files
  - Also, part of OS that manages those files

- Files need *names* to access correct file

inode

- **Kernel** structure for file-system object
  - e.g., file or directory
- Stores **metadata**
  - Type (file / directory)
  - UID, GID (owner / group)
  - rwx (permissions)
  - Size (in bytes)
  - Data blocks
  - Times (access / create)
  - Links count (# paths)
  - …



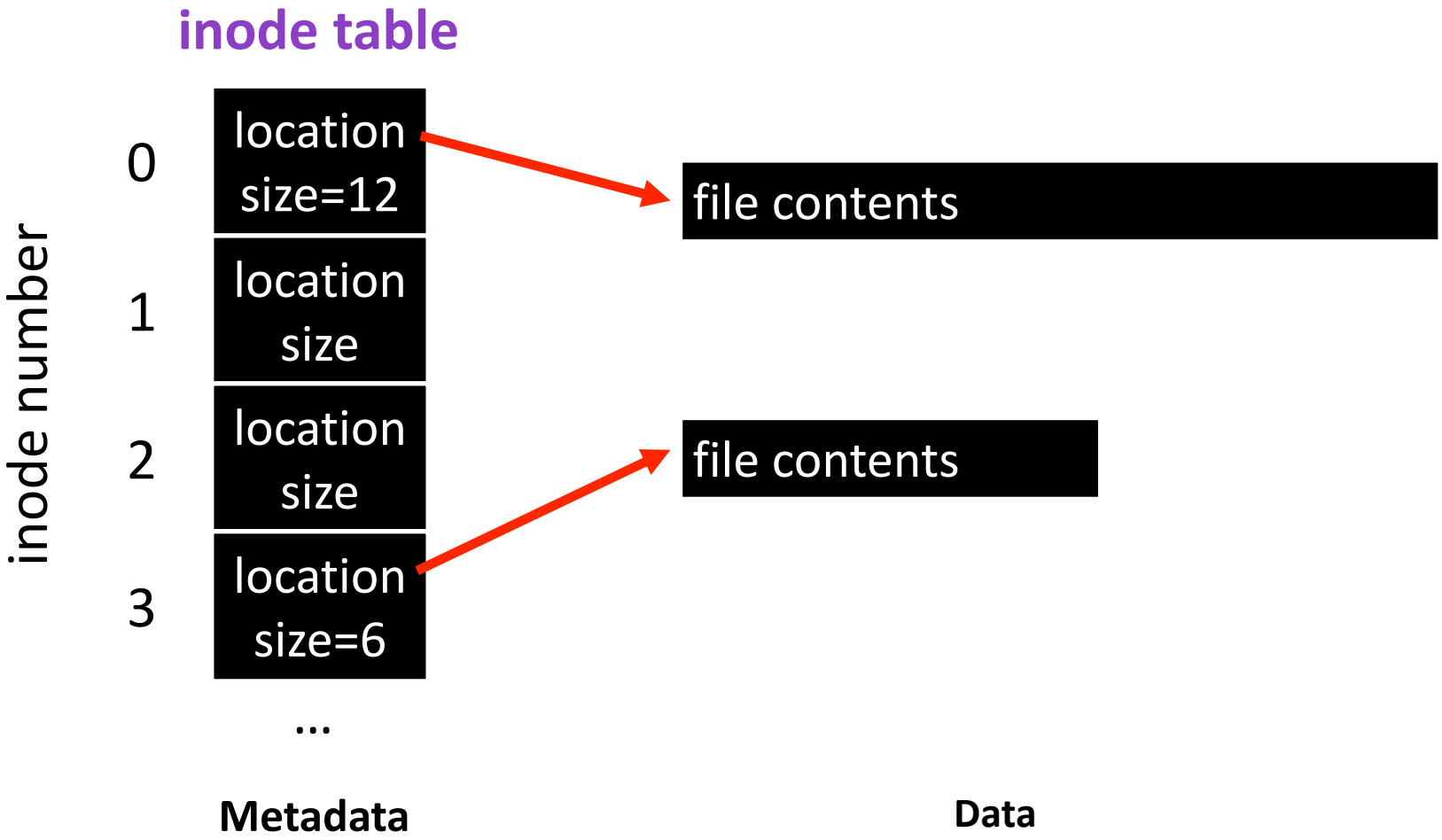
inode

- Each file has exactly one **inode** number

- Inodes are unique
  - At a given time
  - **Within the file system**
  - Stored in **inode table**

- See inodes via `ls -i`

inode


inode table
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
...
file contents
file contents
Metadata
Data

File API

- 1st attempt:
  - **read**(**int** *inode*, **void** **buf*, **size_t** *nbyte*)
  - **write**(**int** *inode*, **void** **buf*, **size_t** *nbyte*)
  - **seek**(**int** *inode*, **off_t** *offset*)

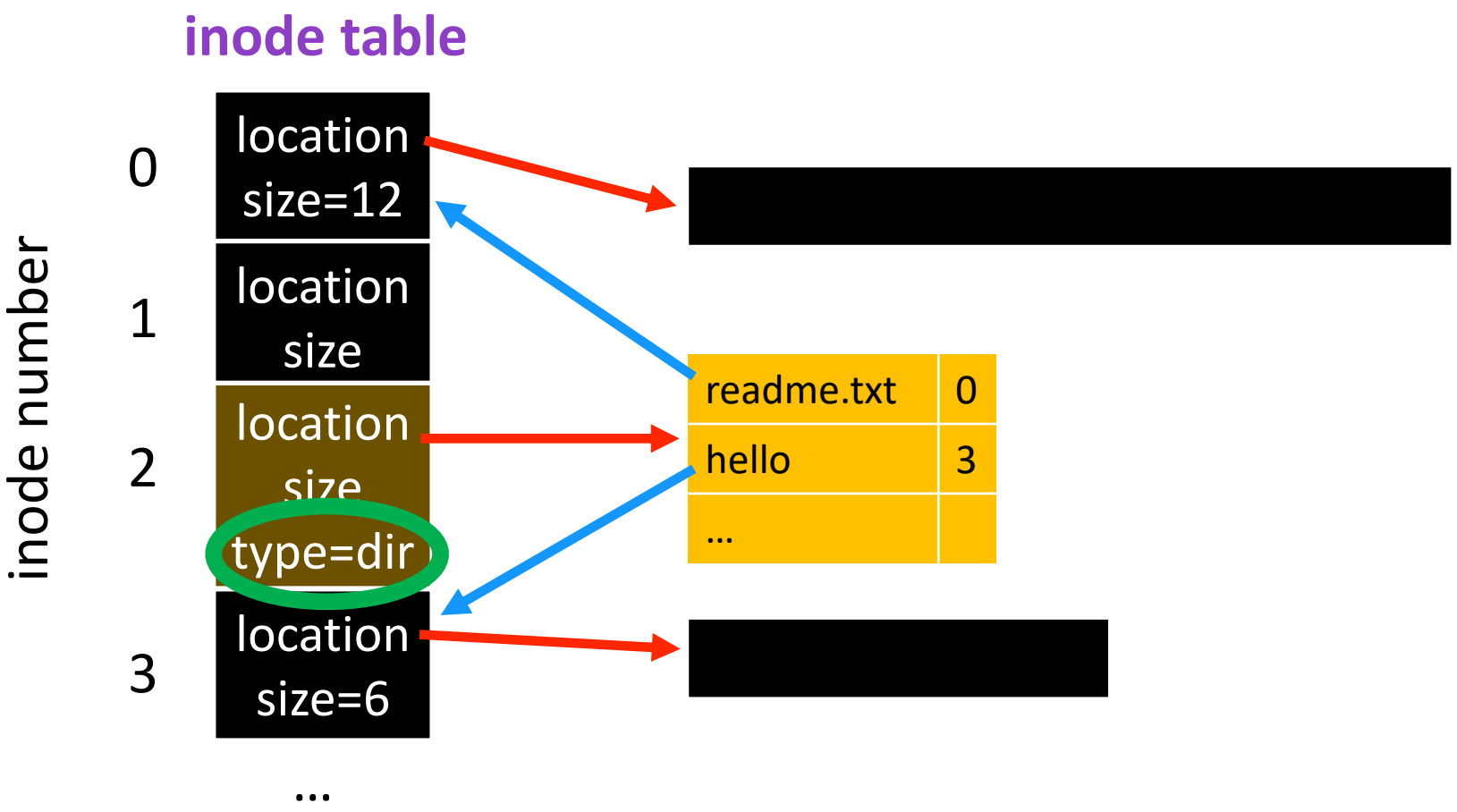
- Cons?
  - Names hard to remember
  - No organization or meaning to inode numbers
  - Offset across multiple processes?

File Names

- **File Names**
  - String names are friendlier
  - File system still interacts with *inode* number

- Store *name-to-inode* mapping in predetermined file
  - Typically, inode **2**

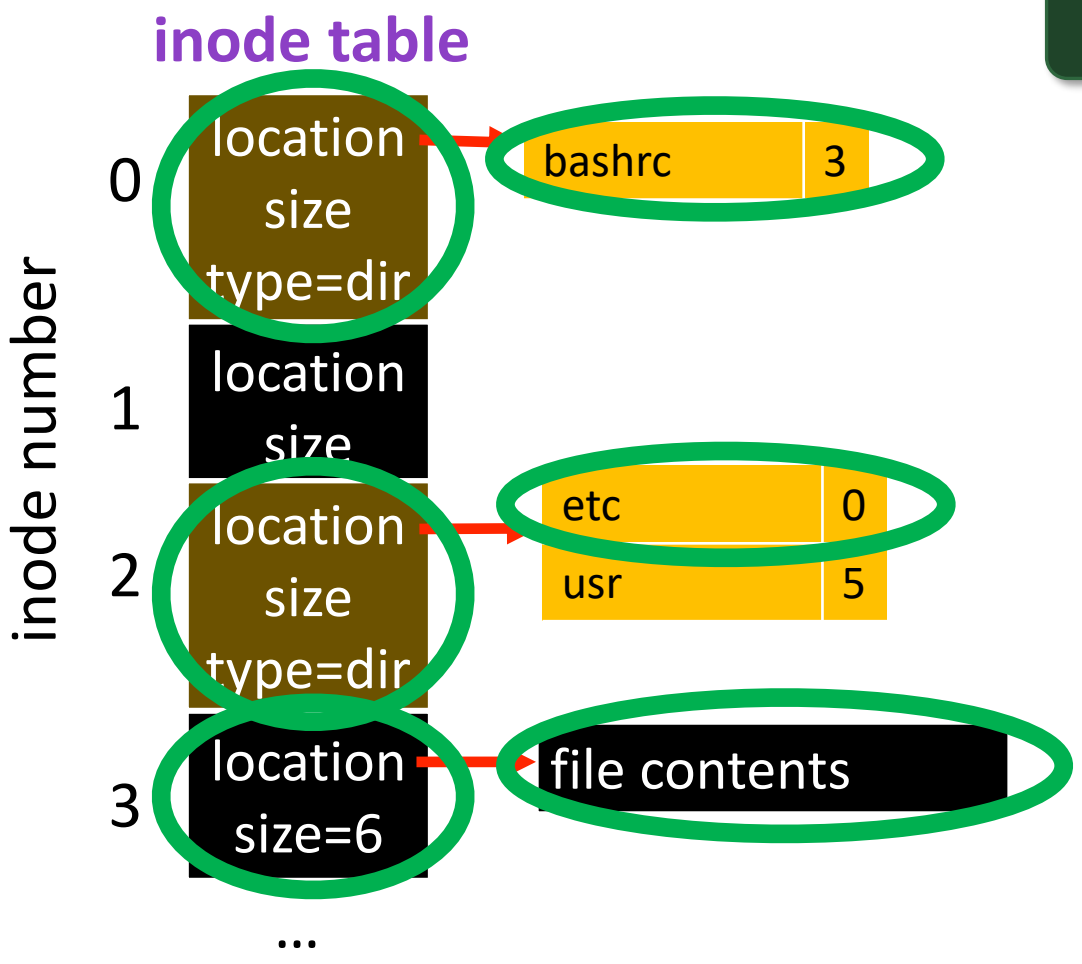
File Names


inode table
inode number
0
1
2
3
location
size=12
location
size
location
size
type=dir
location
size=6
...
readme.txt
0
hello
3
...

File Paths

- Let's generalize paths!
  - **Directory tree** instead of single root
  - Only **full paths** need to be unique
    - `/usr/myuser/file.txt`
    - `/tmp/file.txt`
  - Store *file-to-inode* mapping *for each* **directory**

File Paths



read /etc/bashrc

1. inode 2 (root)
2. root contents
3. inode 0 (*etc*)
4. *etc* contents
5. inode 3 (*bashrc*)
6. *bashrc* contents

Reads: **6**

File API

- 2nd attempt:
  - **read**(**char** *path, **void** *buf,
    off_t offset, **size_t** nbyte)
  - **write**(**char** *path, **void** *buf,
    off_t offset, **size_t** nbyte)

- Cons?
  - Expensive traversal
  - Goal: traverse *once*

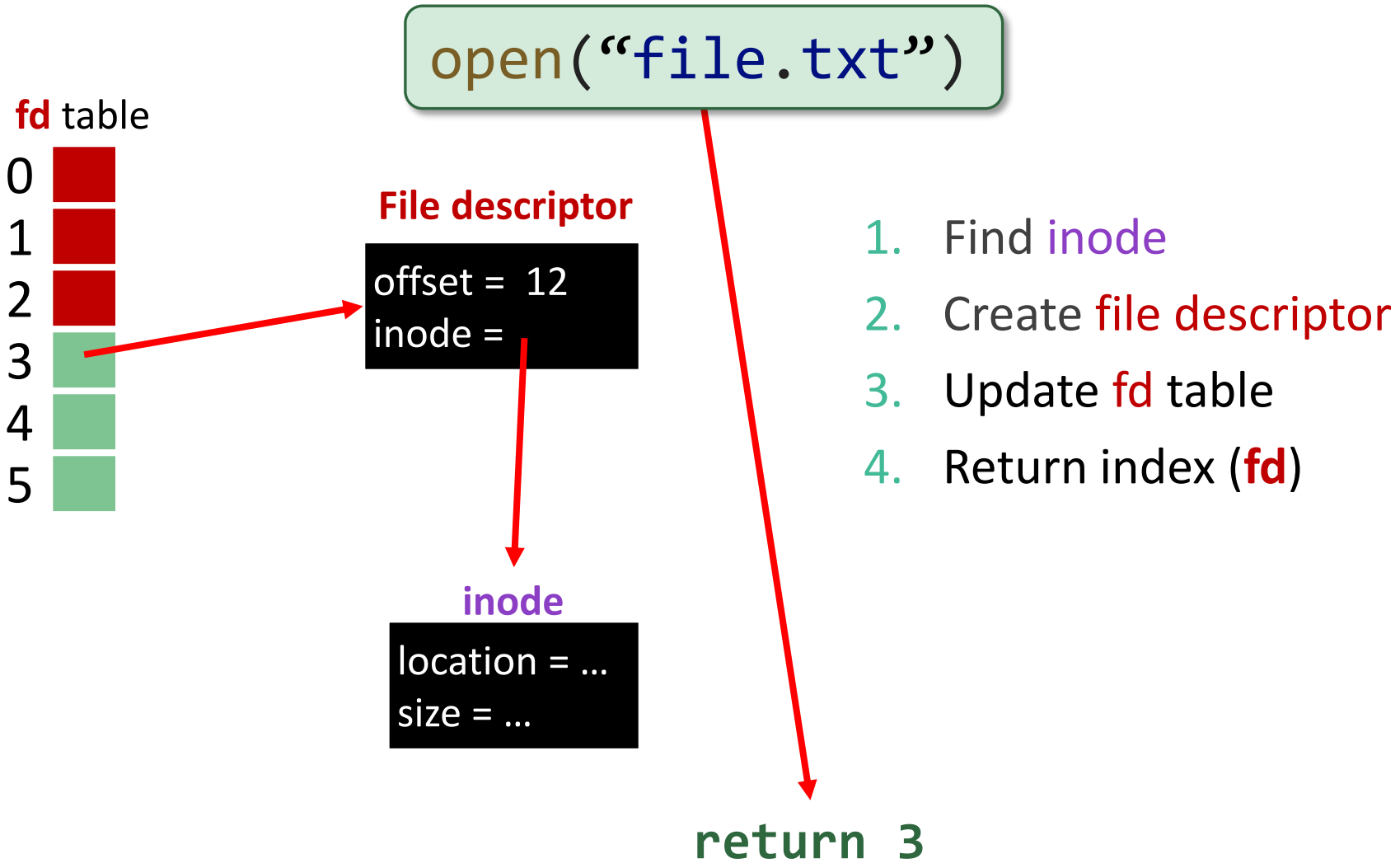
File Descriptors

- **File Descriptor** (**fd**)

  - Do expensive traversal once
  - Store inode in *descriptor* object
  - Do operations & track offset via descriptor

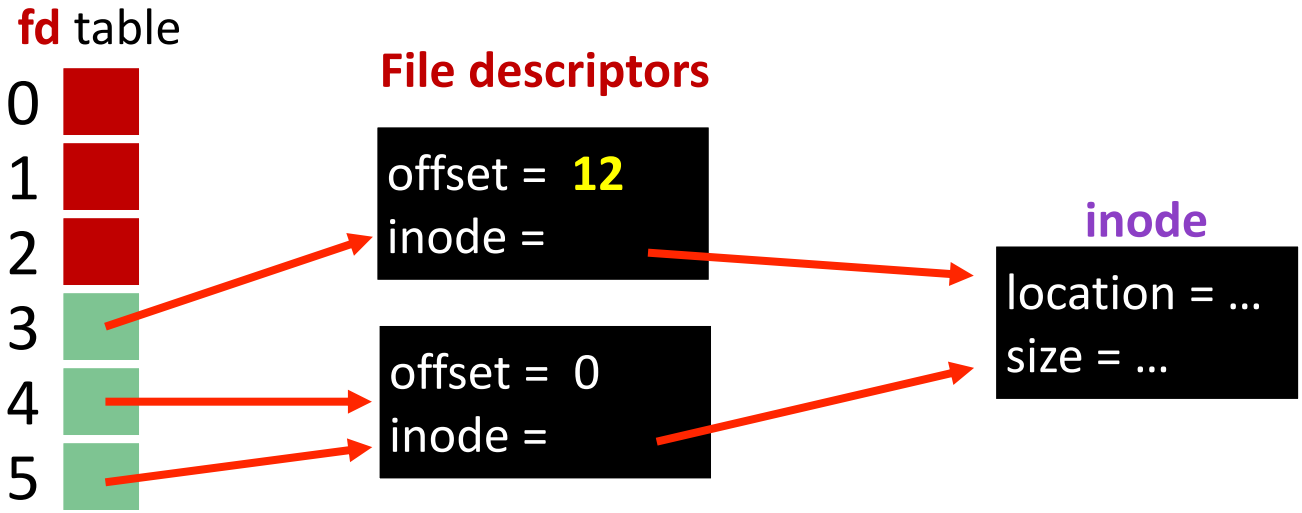
- *Each* process:
  - File descriptor table: pointers to open file descriptors
  - Integers for file I/O indexed into table
    - `stdin`: 0, `stdout`: 1, `stderr`: 2

File Descriptors



1. Find inode
2. Create file descriptor
3. Update fd table
4. Return index (**fd**)

File Descriptors




```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
int fd2 = open("file.txt"); // returns 4
int fd3 = dup(fd2);         // returns 5
```

File API

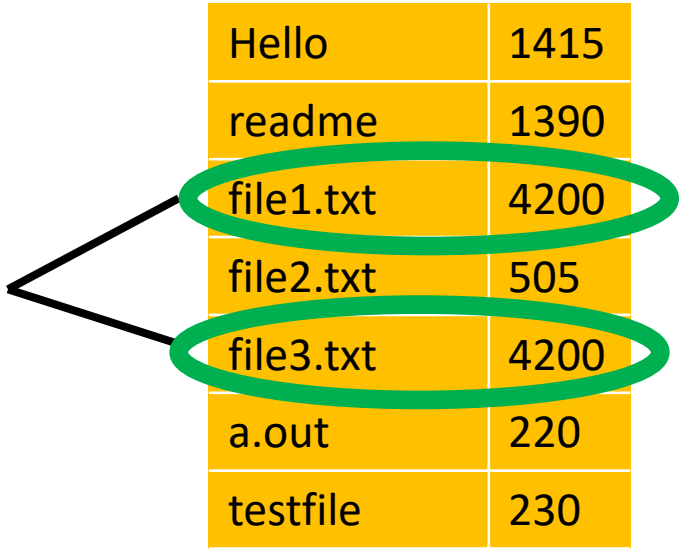
- 3rd attempt:
  - **int open**(**char** *path, **int** flag, **mode_t** mode)
  - **read**(**int** fd, **void** *buf, **size_t** nbyte)
  - **write**(**int** fd, **void** *buf, **size_t** nbyte)
  - **close**(**int** fd)

  - String names
  - Hierarchical
  - Traverse once
  - Different offsets precisely defined

Hard Links

- No system call to delete files!

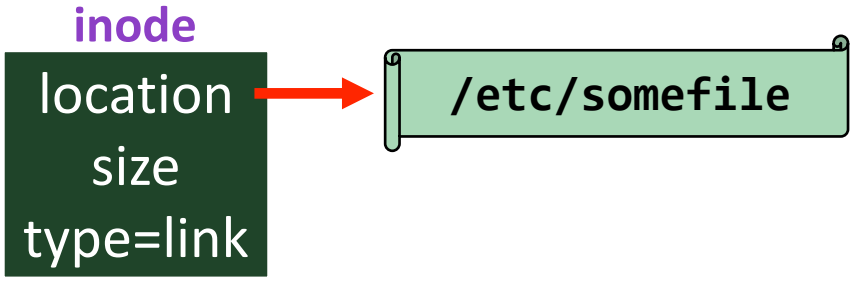
Two paths/names for the same file?



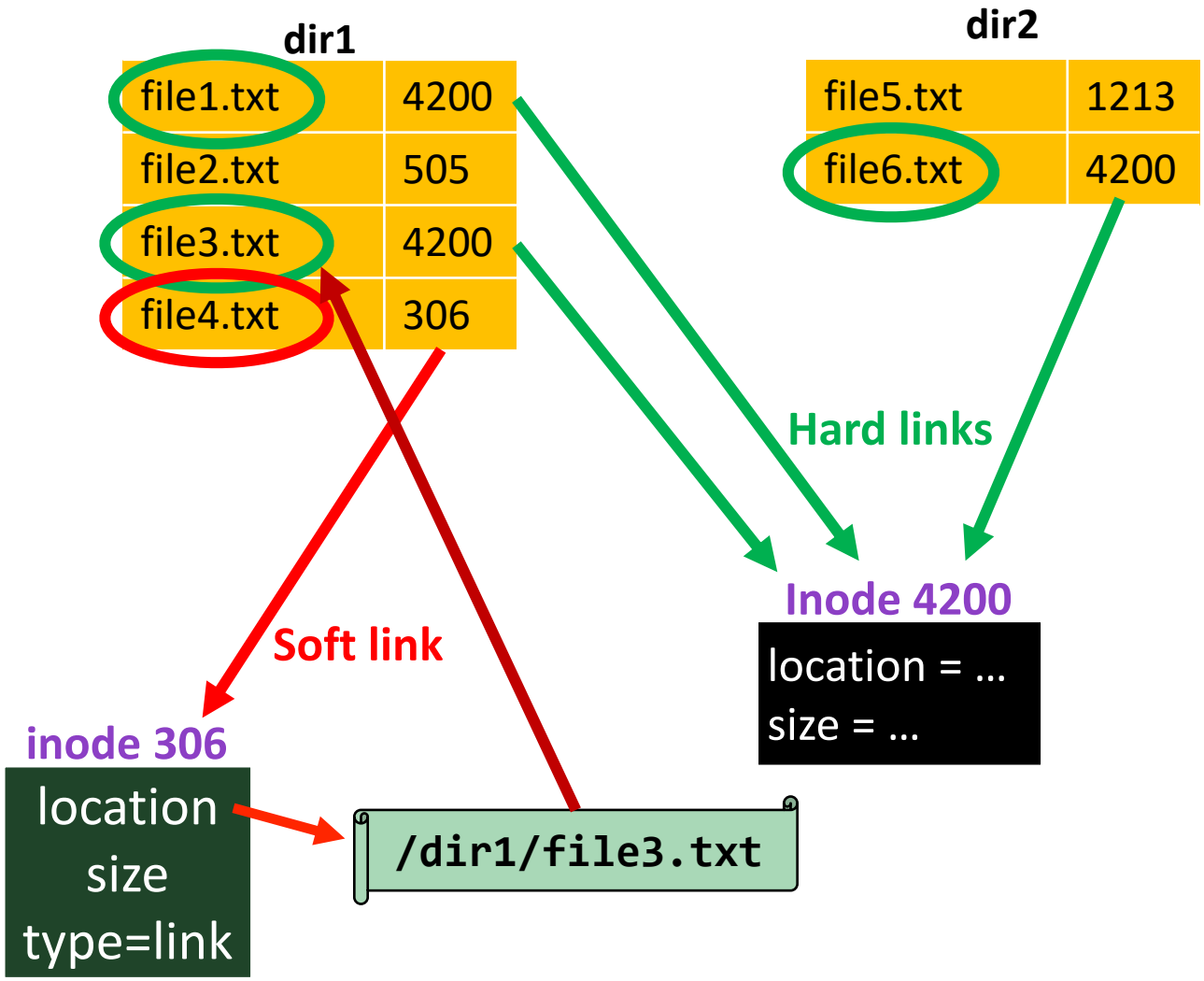
- File paths are called **hard links**
  - Across *same* or *different* directories
  - inode tracks # of hard links
  - File is **garbage collected** when there are no references

Symbolic Links

- **Symbolic links**
  - Special file, points to another ***path name***
  - Like Windows shortcuts



Links


dir1
file1.txt
4200
file2.txt
505
file3.txt
4200
file4.txt
306
dir2
file5.txt
1213
file6.txt
4200
Hard links
Soft link
Inode 4200
location = …
size = …
inode 306
location
size
type=link
/dir1/file3.txt

File System Implementation

- File system: pure software
  - Many different file systems exist

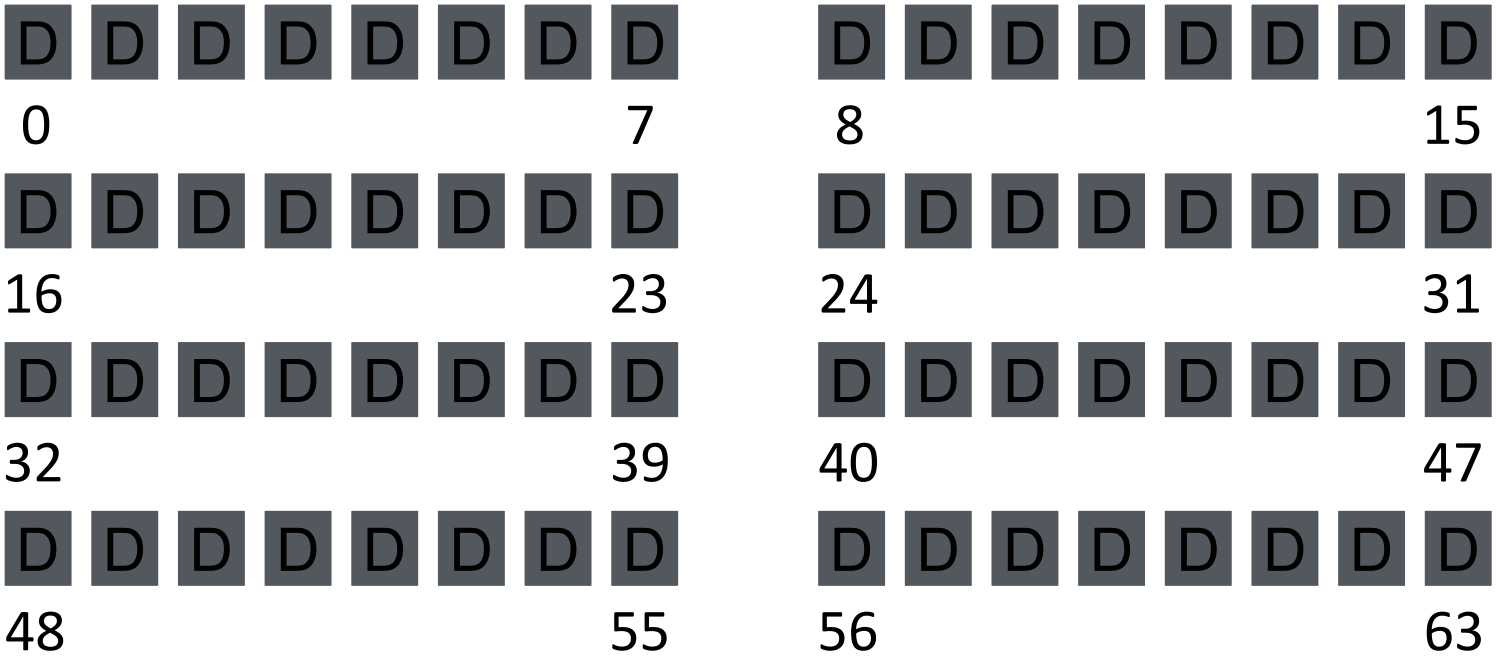
- Start with case study: **VSFS**
  - Simplified version of typical UNIX file system

Very Simple File System

- **Data structures**
  - What type of on-disk structures?
- **Access methods**
  - How to map calls (`open()`, `read()`, `write()`, etc.)?
  - Read which structures during which calls?

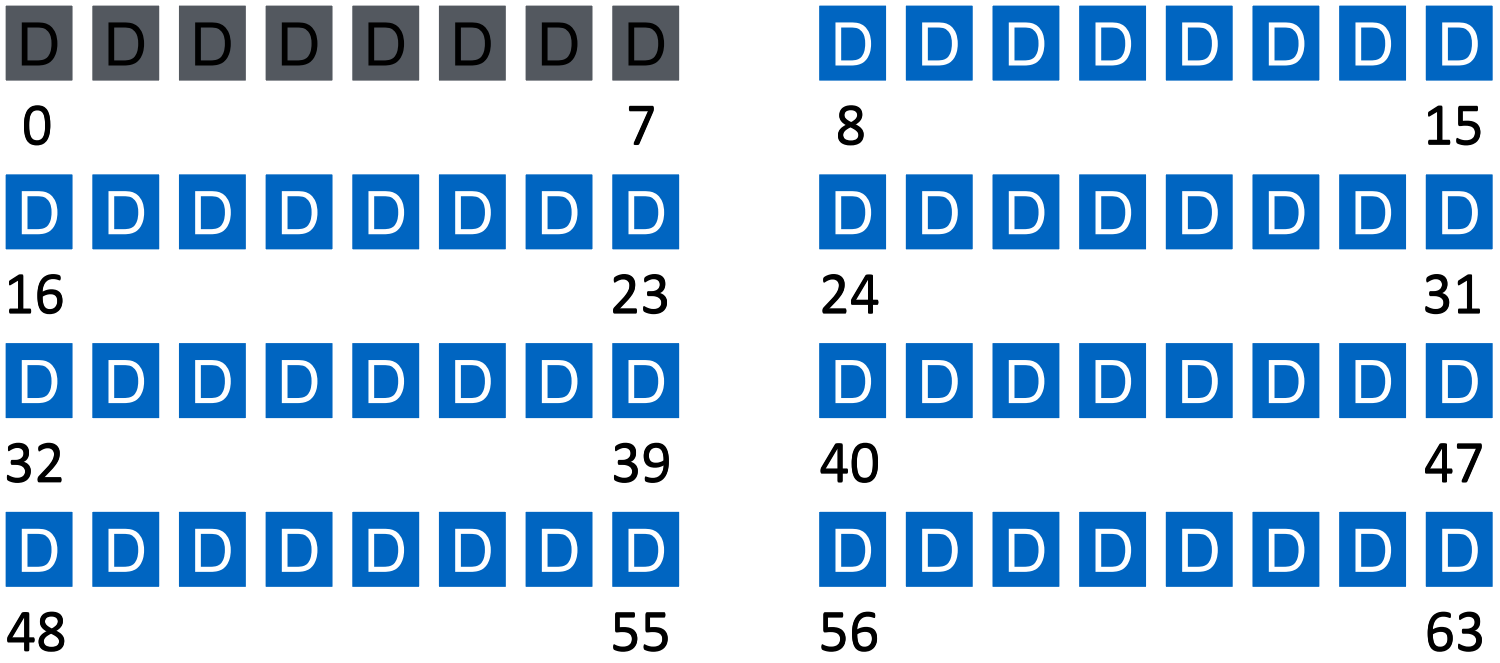
Overall Organization

- Divide disk into **blocks**
  - Addressed *0* to *N-1, c*ommonly-used size: 4KB



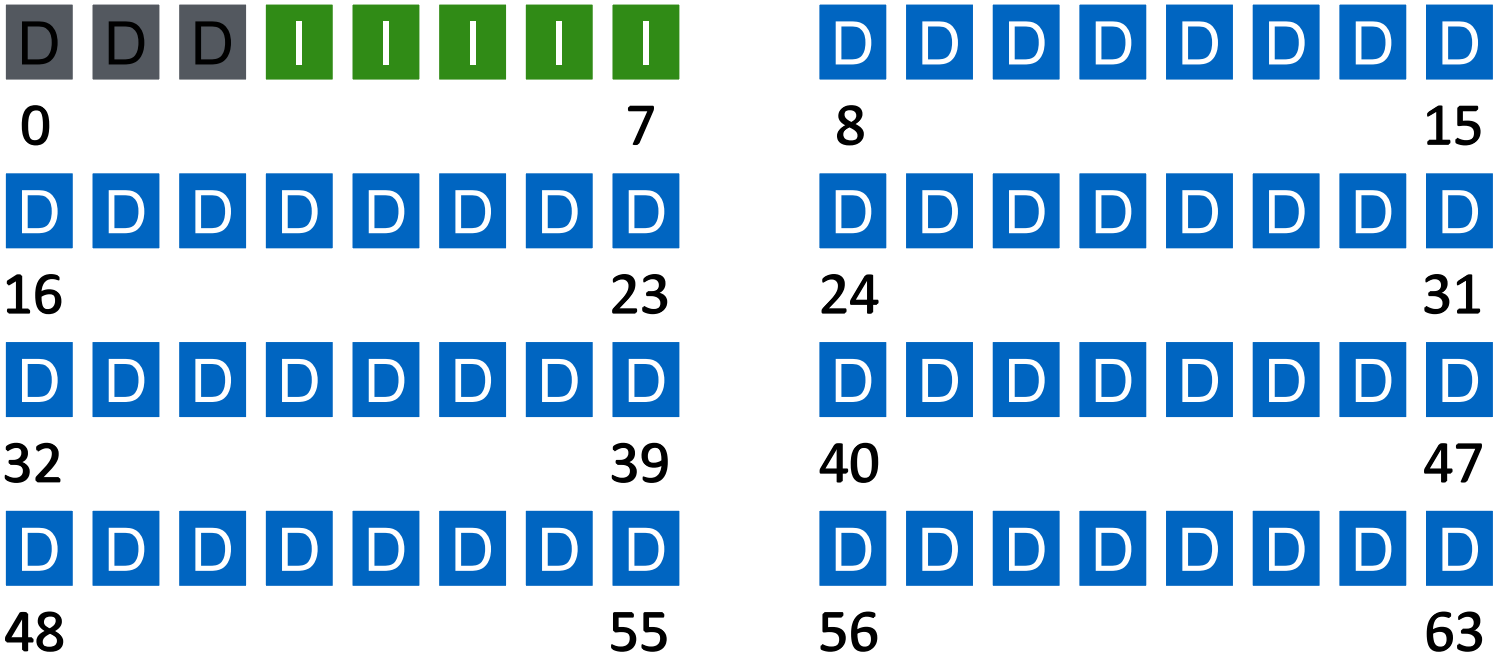
Overall Organization

- Reserve **data region** for user data
  - e.g., fixed portion: 54 of 63 blocks



Overall Organization

- Reserve blocks for file **metadata** (inode)
  - Typically 256 bytes per inode
  - 4KB disk block → 16 inodes per inode block



Overall Organization

- **Inode**
  - Type (file / directory)
  - UID (owner)
  - rwx (permissions)
  - Size (in bytes)
  - Data blocks
  - Times (access / create)
  - Links count (# paths)
  - …

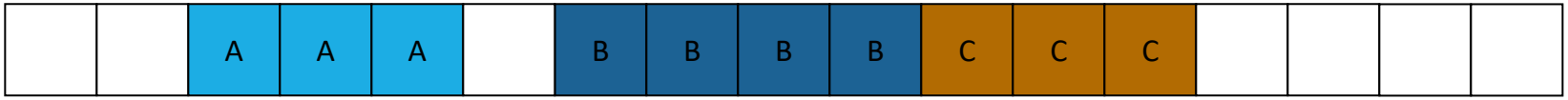
Allocation

- Where is file data located?

- Need a way to allocate data blocks to files
  - Store information in inode
    - Creates external fragmentation?
    - Can the file grow?
    - What is the performance?
    - Metadata overhead?

Allocation

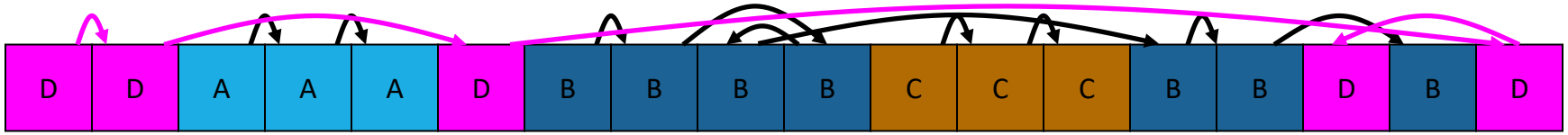
- Contiguous allocation
  - Metadata: starting block and file size (single **extent**)
  - OS allocates by finding sufficient free space
    - Must predict future size of file to reserve enough space



- Fragmentation: Bad external fragmentation
- Grow file: Not without moving
- Sequential access: Excellent performance
- Random access: Simple calculation
- Metadata: Very little overhead (“wasted space”)

Allocation

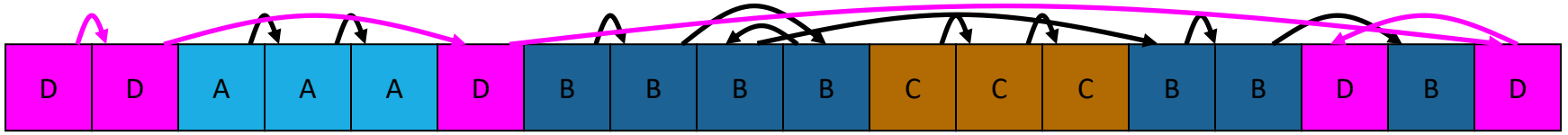
- **Linked-list**
  - Inode pointer to 1st block, end of data → another pointer



- Fragmentation: No external fragmentation
- Grow file: Can grow easily
- Sequential access: Depends on data layout
- Random access: Very poor (traverse all blocks)
- Metadata: Pointer per block wasted

Allocation

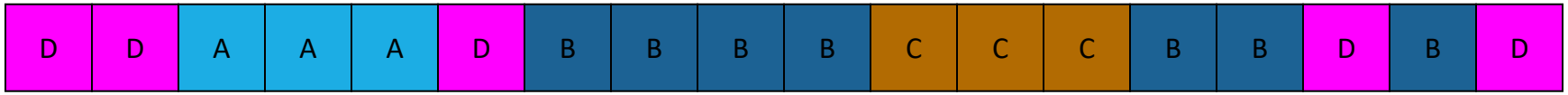
- File-Allocation Table (**FAT**)
  - Linked-list information in on-disk FAT table
  - Directory entries instead of inodes (hard links impossible)



  - Read from two disk locations for every data read
  - Optimize: cache FAT in main memory
    - Greatly improves random access
    - What to cache? Does not scale with file system

Allocation

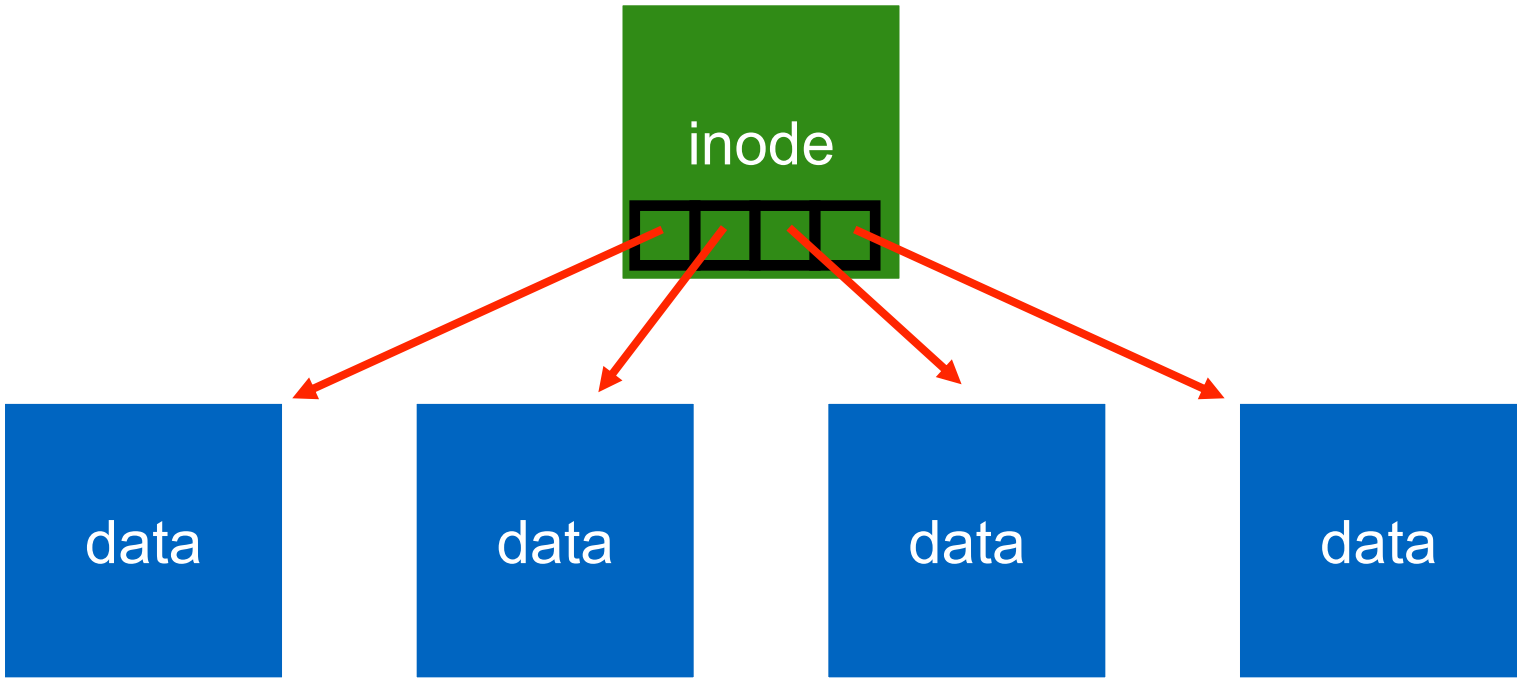
- Indexed allocation
  - Fixed-size blocks for each file, as described earlier



- Fragmentation: No external fragmentation
- Grow file: Can grow easily
- Sequential access: Depends on data layout
- Random access: Good
- Metadata: Wasted space for pointers (most files are small)

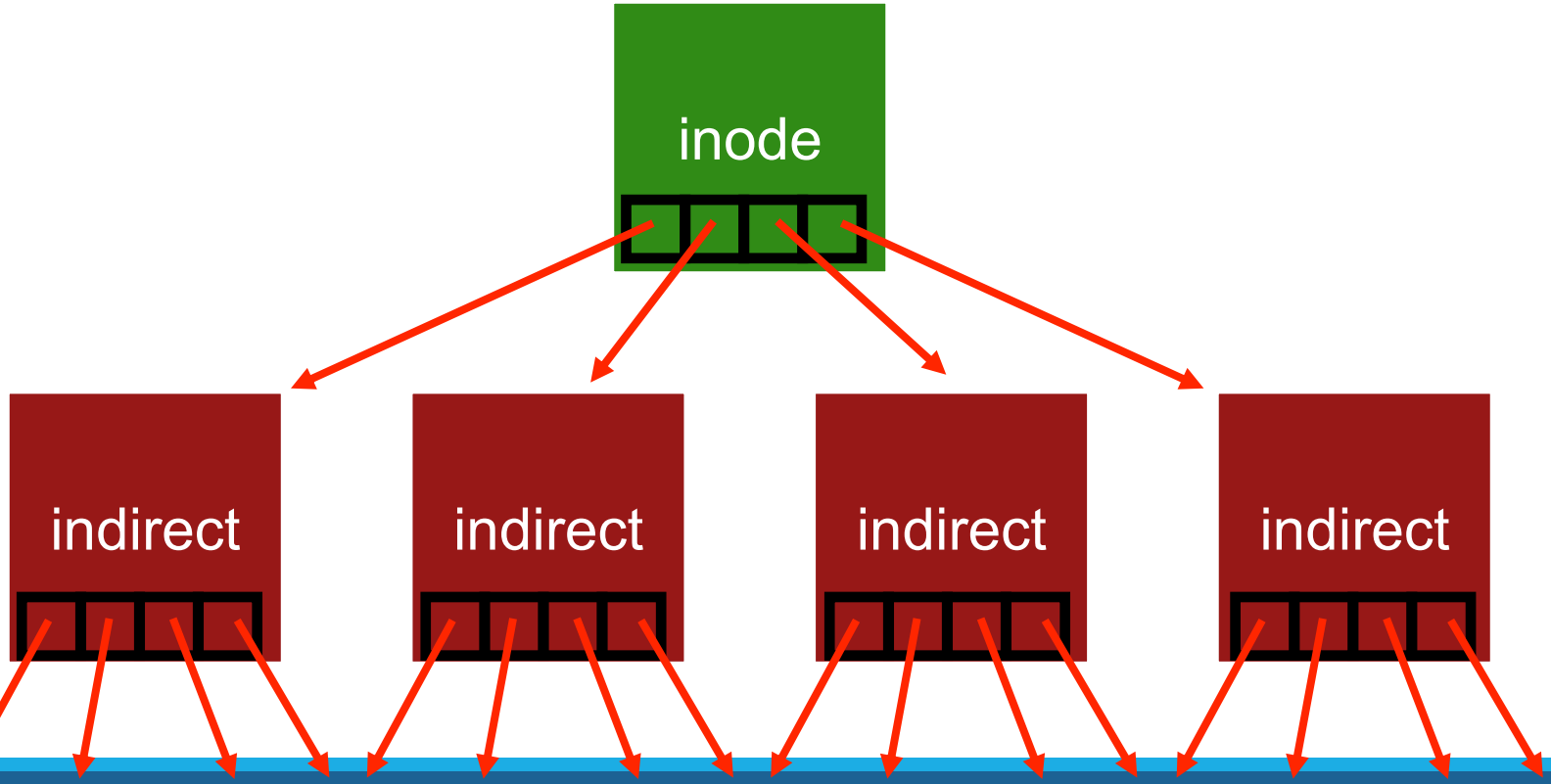
Allocation

- Where are the data blocks?
  - Inode contains one or more *direct* pointers
  - File size is limited



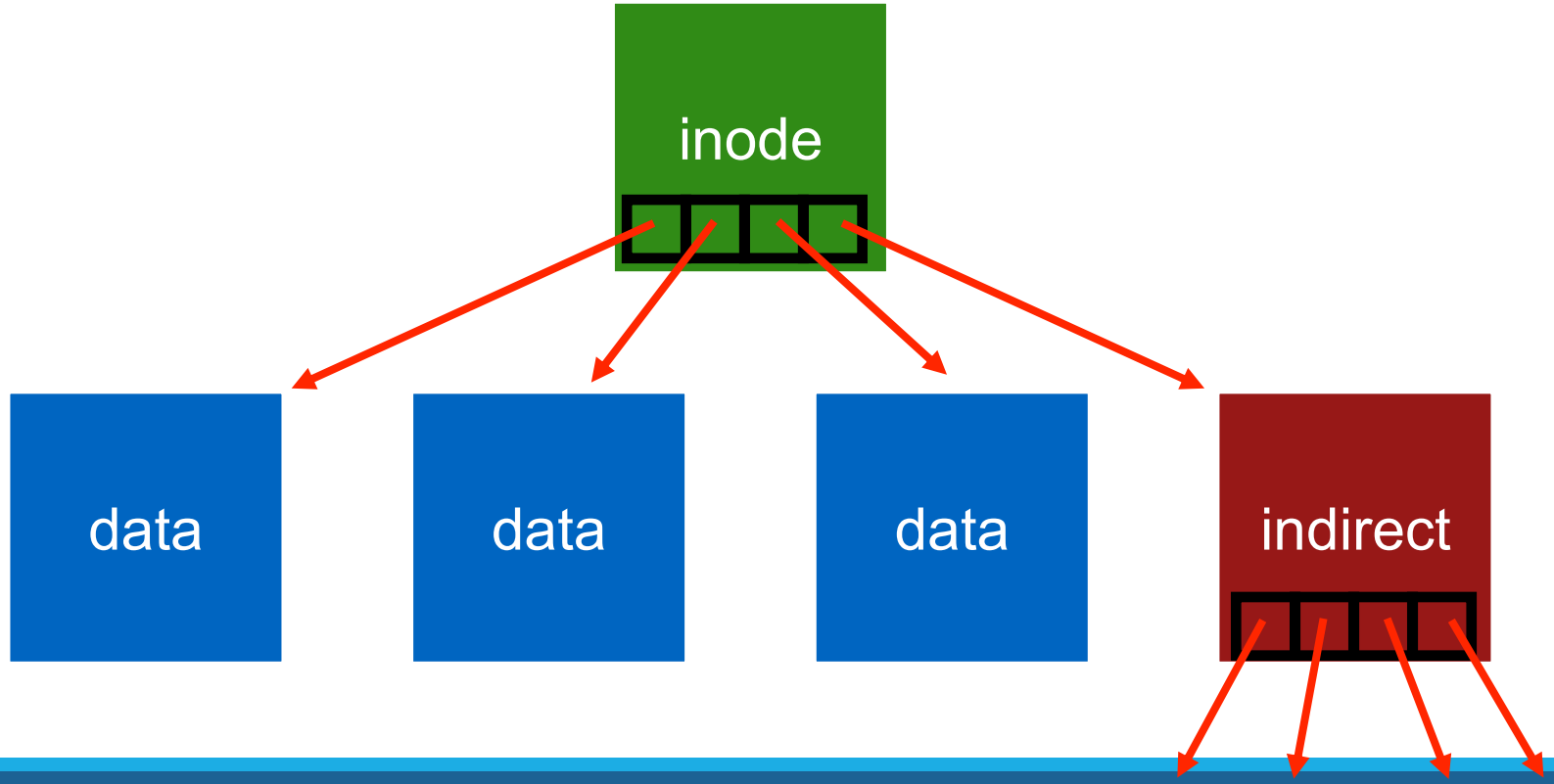
Allocation

- **Indirect pointers**
  - Point to a block that contains more pointers
  - Indirect blocks are stored in regular data blocks



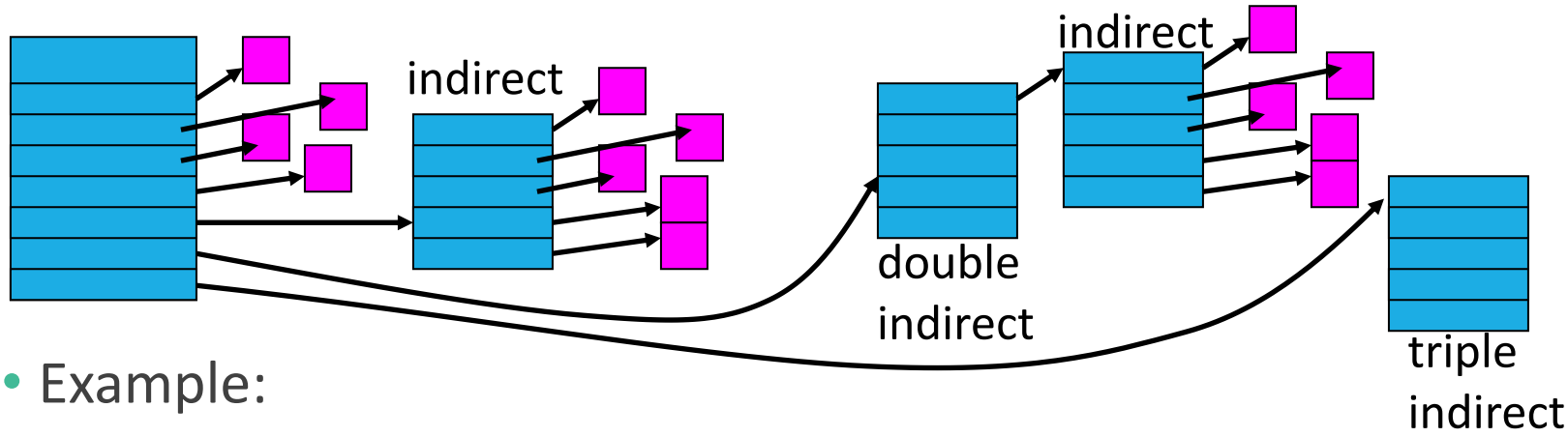
Allocation

- Optimize for small files
  - Fixed number of direct pointers, single indirect pointer
  - Also: double indirect, triple indirect, etc.



Multi-Level Index

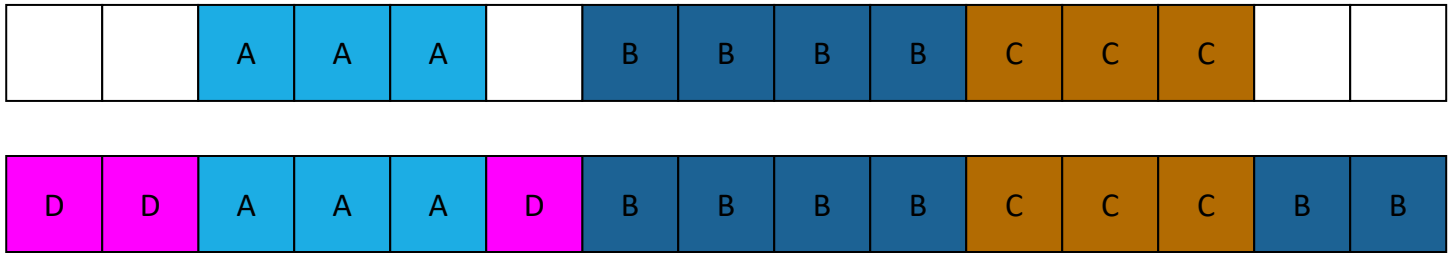
- Double indirect: points to block of indirect pointers
- Triple indirect: points to block of double indirect pointers



- Example:
  - Block size 4KB, 4-byte pointers
  - 12 direct pointers, 1 single and 1 double indirect pointers
  - Can accommodate 4GB file
  - $((12 + 1024 + 1024^2) \cdot 4KB)$

Allocation

- Extents
  - Metadata: small array (2-6) of extents



- Fragmentation: Helps external fragmentation
- Grow file: Can grow (until out of extents)
- Sequential access: Excellent
- Random access: Simple
- Metadata: Small overhead

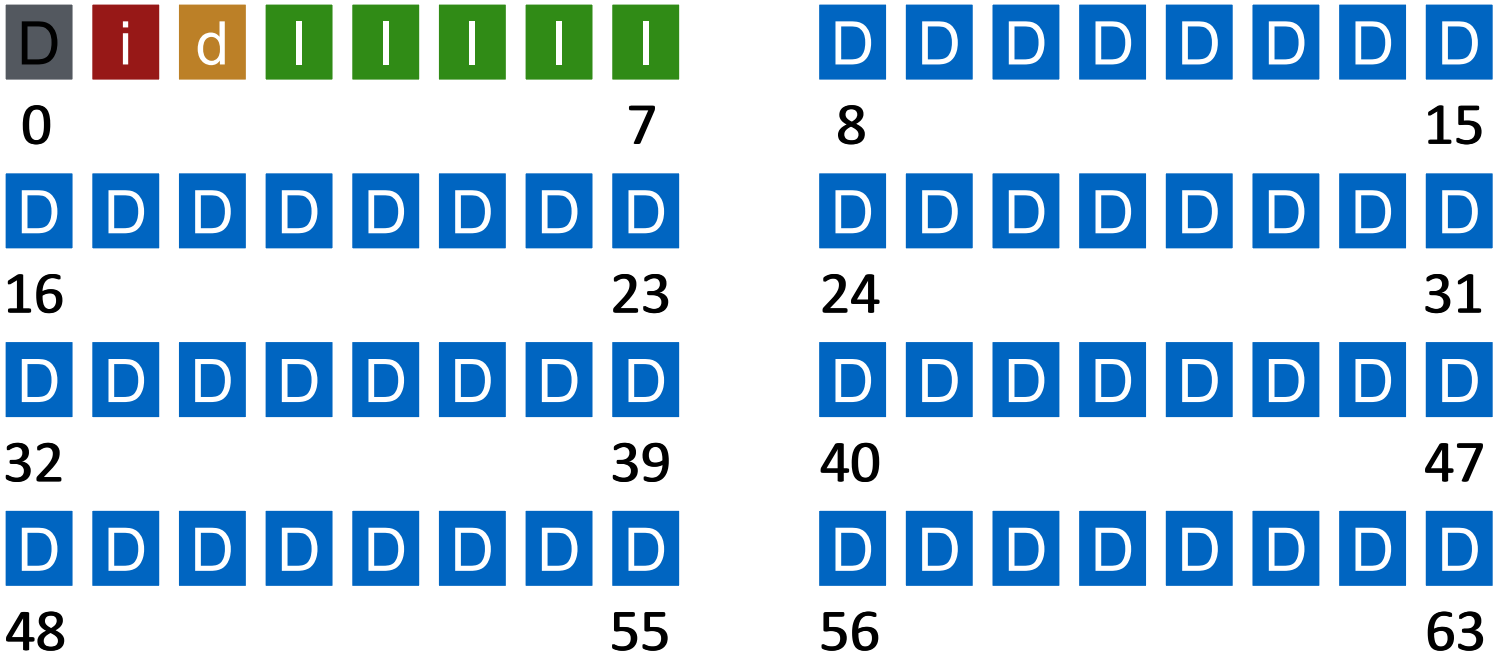
Multi-Level Index

- Many file systems use multi-level index
  - Linux ext2 and ext3, NetApp's WAFL, Unix file system
  - SGI XFS and Linux ext4 use **extents**

- **Extents**
  - Disk pointer plus length
  - Avoids large metadata per file

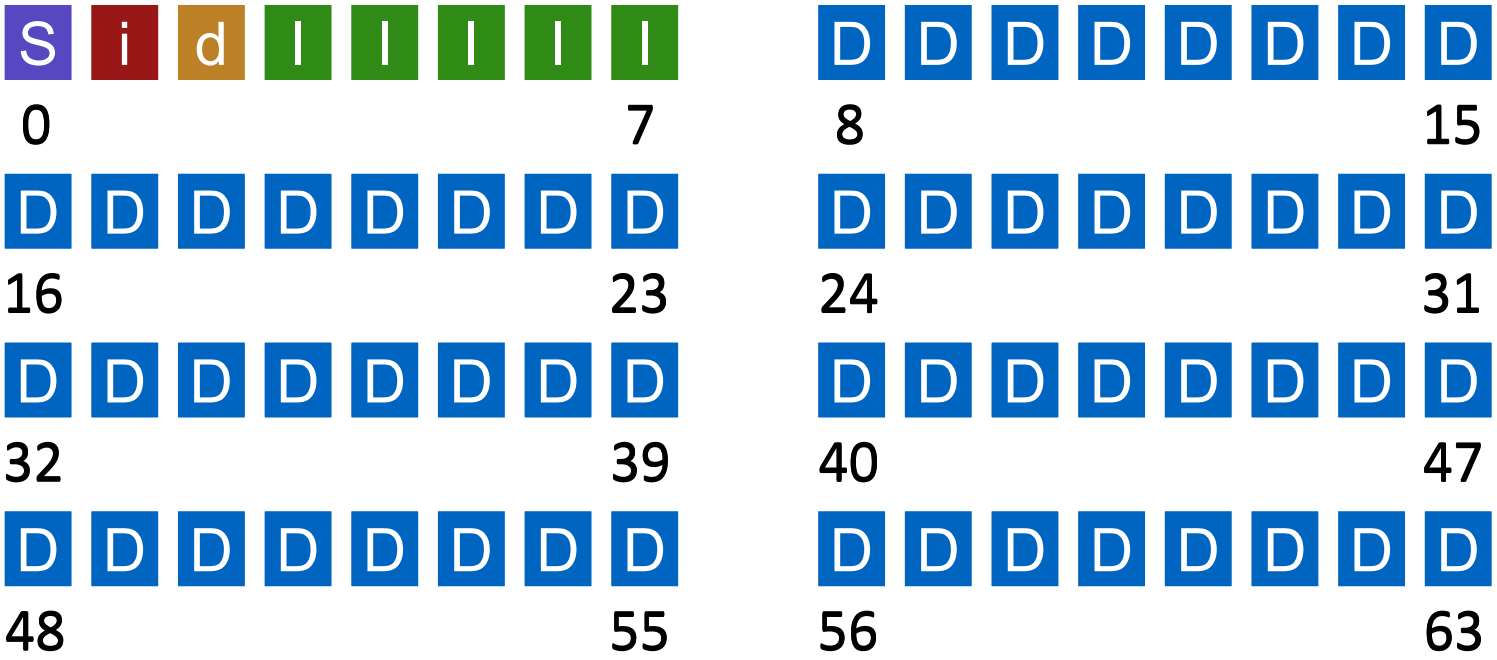
Allocation

- How do we find free data blocks or free inodes?
  - **Free list**
  - **Bitmap**: for data region and inode table



Superblock

- On mount, OS reads **superblock** to initialize
  - Information about file system



Access Paths

- Open file `/foo/bar` (assume it exists)
  - **Traverse** pathname to locate desired inode
  - Begin at root: well-known, usually inode 2
  - Read block that contains inode 2
  - Look inside It – read data block to find inode number of `foo`
  - Read inode of `foo` to find data block of `foo`
  - Read data blocks of `foo` to find inode of `bar`
  - Read inode of `bar`

- **5** reads just to open a file!

Access Paths

- Write to `/foo/bar` (assume it is open)
  - Read data bitmap
  - Write updated data bitmap (newly-allocated block to use)
  - Read the inode of `bar`
  - Write updated inode with new block location
  - Write the actual block itself

- Read `/foo/bar` (assume it is open)
  - Read `bar` inode to find data block
  - Read data block
  - Write `bar` inode – update access time

Access Paths

- Create file `/foo/bar` (doesn't exist)
  - Read inode bitmap, write updated bitmap with allocated inode
  - Read block of found inode and write inode itself for bar
  - Find data block of foo
    - Read block that contains inode 2 (root)
    - Read data block to find inode number of `foo`
    - Read inode of `foo` to find data block of `foo`
    - Read and write data block of `foo` to add `bar` entry to directory
    - Write update inode of foo (date change)
  - Directory `foo` needs to grow? Additional I/O

Caching and Buffering

- Simple operations, huge number of I/Os
  - Long pathnames can lead to hundreds of reads

- Solution? Use memory (RAM) to cache important blocks
  - **Write buffering**
    - **Batch** updates for less I/O (several updates to inode bitmap)
    - Buffer writes in memory, **schedule** subsequent I/Os
    - **Avoid** writes, e.g., file created and then deleted

Very Simple File System

- Described a very simple FS
  - Basic on-disk structures
  - Basic operations
  - Poor performance

- In practice:
  - Allocate **efficiently** to obtain good performance
  - Handle **crashes**

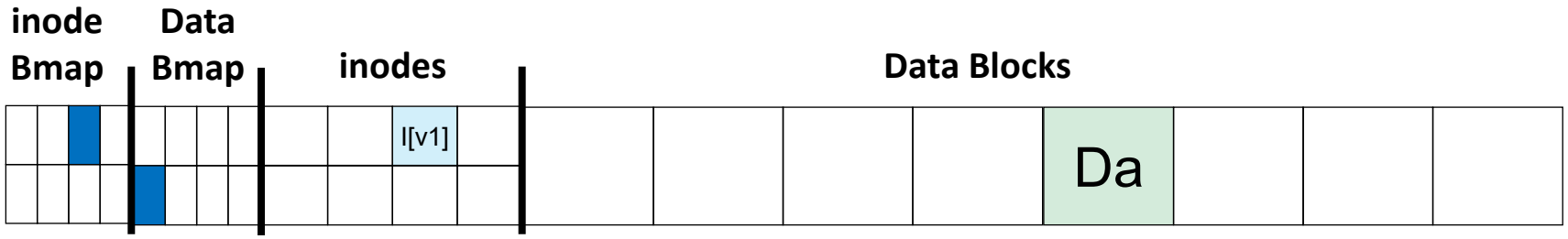
Crash Consistency

- File system data structures must **persist**
  - Major challenge: update structures despite **power loss** or **system crash**
- **Crash-consistency problem**
  - On-disk structures left in an **inconsistent** state

Crashes may occur at arbitrary times.
How do we ensure the file system remains valid?

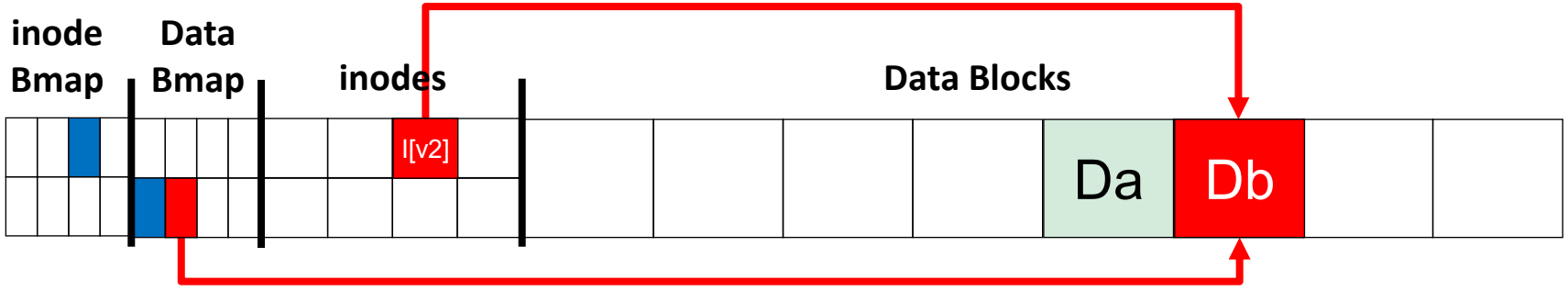
Example

- Append 4KB to an existing file
  - Open file → `lseek()` → issue 4KB write → close file
- Simple file system:
  - **Inode bitmap**, **data bitmap**, inodes, data blocks



Example

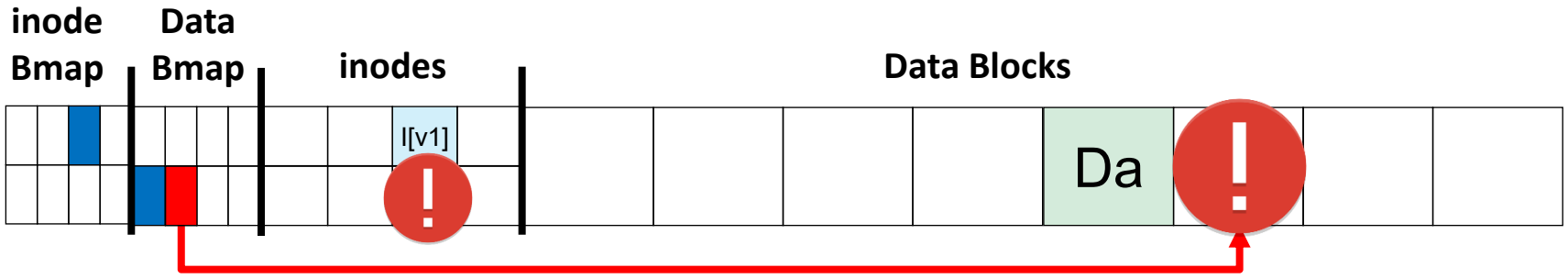
- Write three blocks:
  - Updated data bitmap
  - Updated inode
  - New data block



What if a crash happens?

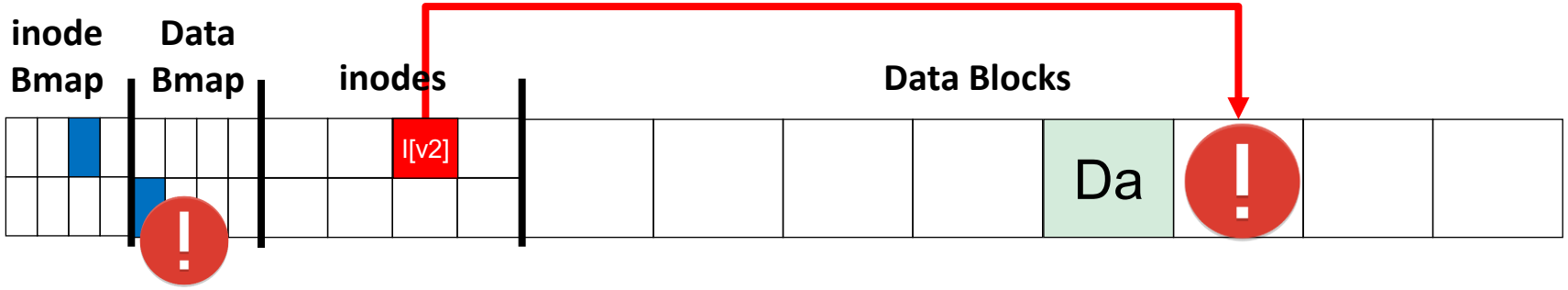
Crash Scenarios

- Crash after a single write:
  - **Updated data bitmap**



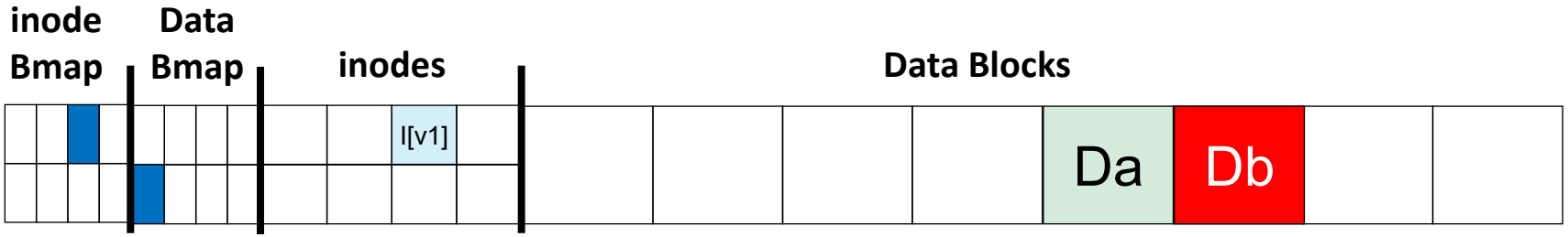
Crash Scenarios

- Crash after a single write:
  - **Updated inode**



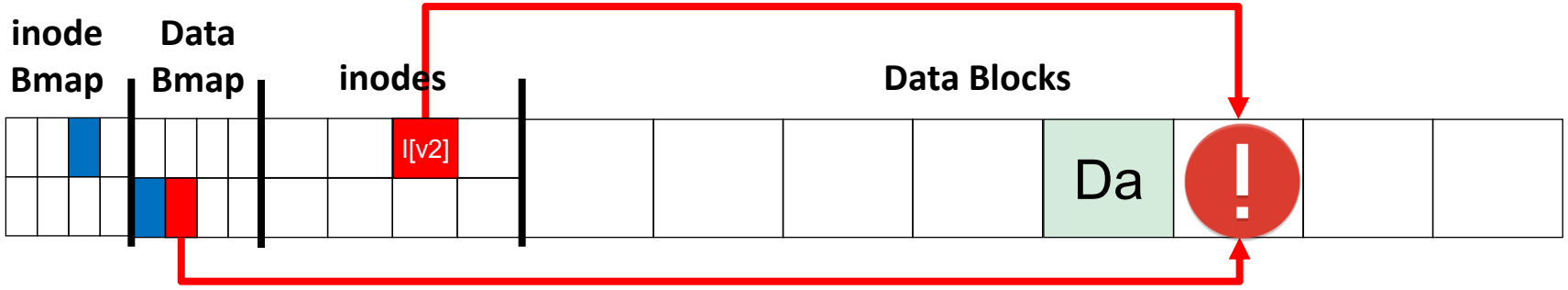
Crash Scenarios

- Crash after a single write:
  - **New data block**



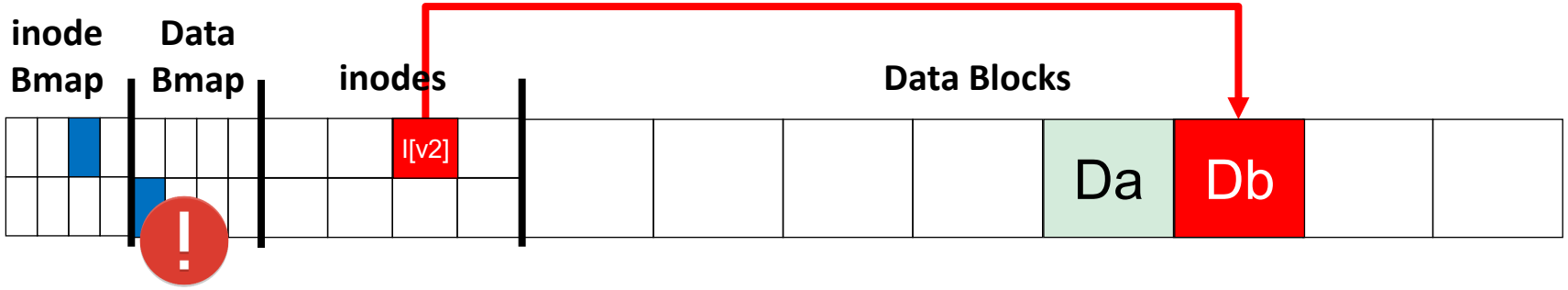
Crash Scenarios

- Crash after *two* writes:
  - **Updated inode and data bitmap**



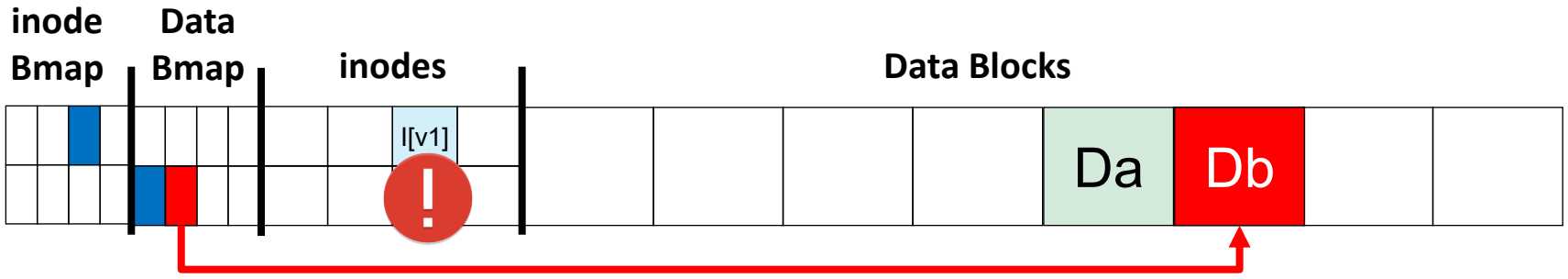
Crash Scenarios

- Crash after *two* writes:
  - **Updated inode and new data block**



Crash Scenarios

- Crash after *two* writes:
  - **Updated data bitmap and new data block**



Crash Scenarios

- Goal: move *fs* from consistent state to another **atomically**
  - Don't care about data loss, only consistency
  - But disk only commits ***one*** write at a time
  - Crash or power loss may occur

- **Crash-consistency problem**

Solution #1

- Let inconsistencies happen, fix them later

- File System Checker (`fsck`)
  - UNIX tool for finding and repairing inconsistencies
  - Can't fix all problems
    - e.g., file system looks consistent but inode points to garbage data

  - Only goal: keep file system *metadata* consistent

Solution #1

- **Free blocks**
  - Scan inodes to produce correct version of data bitmap
- **Superblock** and **Inode state**
  - Make sanity checks on file system & allocated blocks
  - Integrity checks on directory contents
  - Check each inode for corruption
  - Verify link count of each allocated inode
- **Duplicates**
  - Check for duplicate points to same data block
  - Clear bad inode or duplicate block

Solution #1

- Building `fsck` requires intricate knowledge of file system

- Too slow
  - Scan entire disk to find allocated blocks
  - Read entire directory tree
  - Many minutes or *hours*

  - Previous example:
    - Just three blocks are written to disk
    - Scan entire disk for problems

Solution #2

- **Journaling** (**write-ahead logging**)
  - Add work during updates → reduce work during recovery
  - First, write a *note* describing operation
  - If crashed, look at notes and try again
  - Know exactly what to fix instead of scanning entire disk

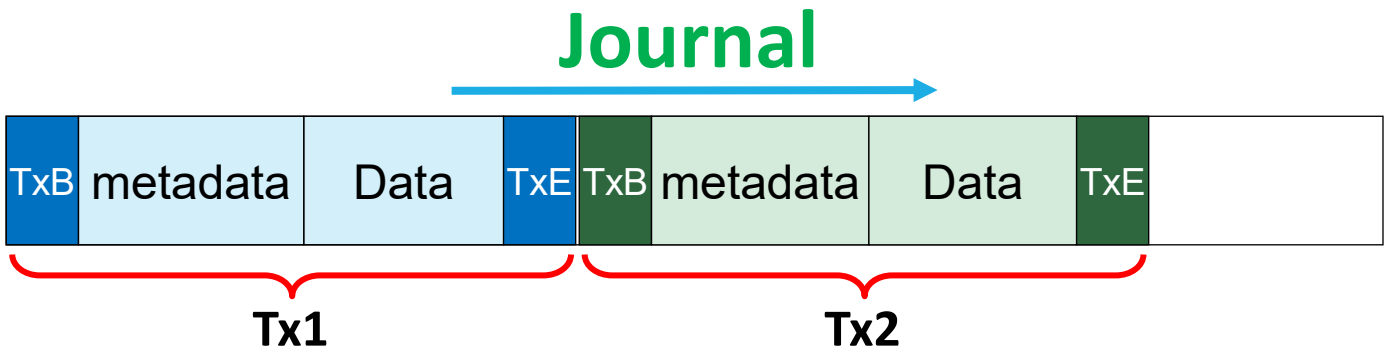
Fsync

- File systems keep newly written data in memory
  - **Write buffering** improves performance

- What if system crashes?
  - **`fsync`**`(`**`int`** *`fd`*`)`
    - Forces buffers to flush to disk
    - Usually tells disk to flush its write cache as well

Journaling

- New key structure: **journal** itself
  - Occupies small amount of space within partition
  - …or another device



Journaling

- **Checkpoint**: transaction safely in **journal**
  - Ready to overwrite structures in file system

- Initial sequence:
  1. **Journal write**
     - Write the transaction
     - Wait for writes to complete
  2. **Checkpoint**
     - Write data and metadata to final locations

Journaling

- Crash during **journal** write?

- Issue writes in two steps

Journaling

- Updated protocol:
  1. **Journal write**
     - Write transaction contents (*TxB*, *metadata*, *data*)
     - Wait for writes to complete
  2. **Journal commit**
     - Write transaction commit block (***TxE***)
     - Wait for write to complete → transaction is **committed**
  3. **Checkpoint**
     - Write data and metadata to final locations

Recovery

- Crash before transaction is written to log?
  - Pending update is skipped

- Crash after commit, before checkpoint completes?
  - Can **recover** the update
  - On boot, scan log for committed transactions to **replay**
  - Called **redo logging**

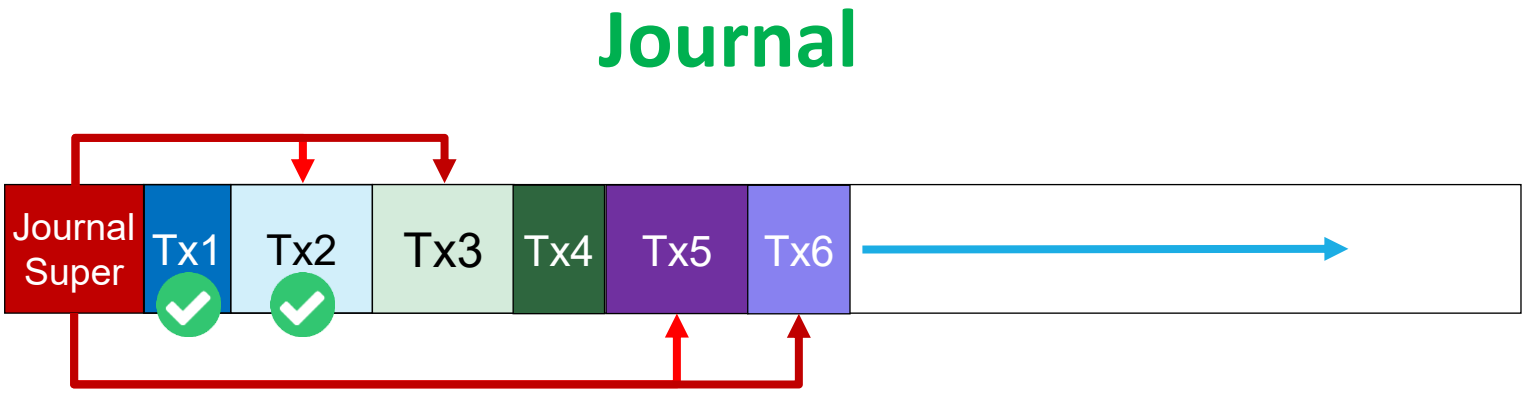
Finite Log

- Log of finite size:
  - Larger log → longer recovery
  - Log full → no more transactions

- Solution? **Circular log**
  - Free log space used by the transaction
  - Some time after a checkpoint

Finite Log

- **Journal superblock**
  - Marks oldest and newest non-checkpointed transactions

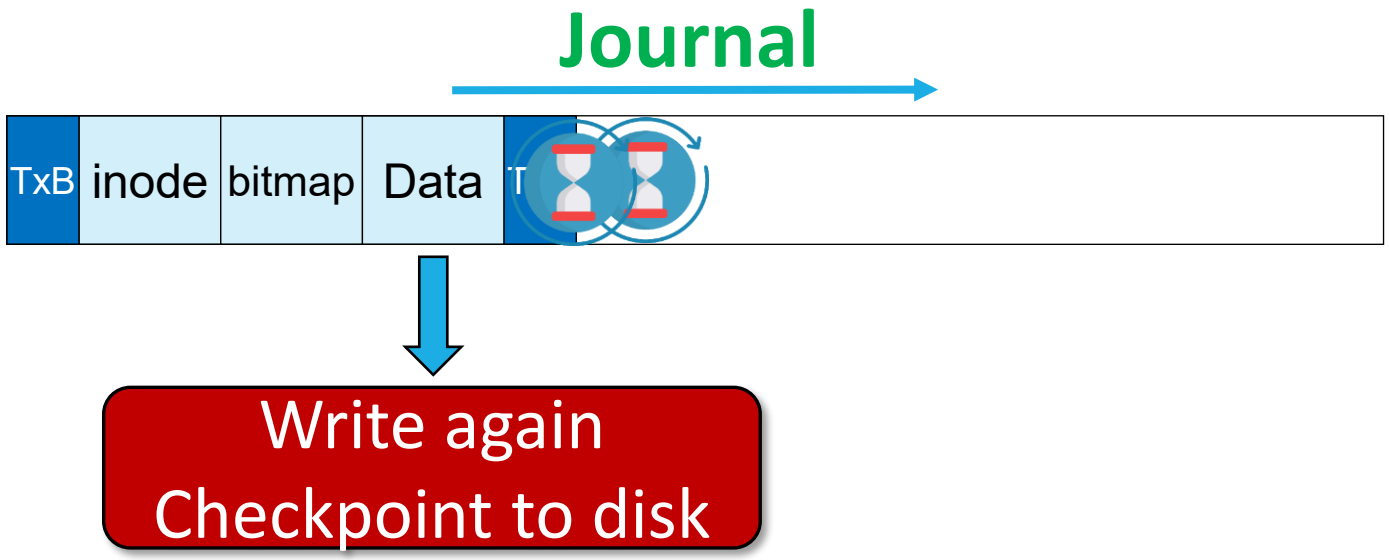


Finite Log

- Another step in our protocol
  1. **Journal write**
  2. **Journal commit**
  3. **Checkpoint**
  4. **Free**
     - Some time later, mark transaction as free in journal
     - By updating the journal superblock

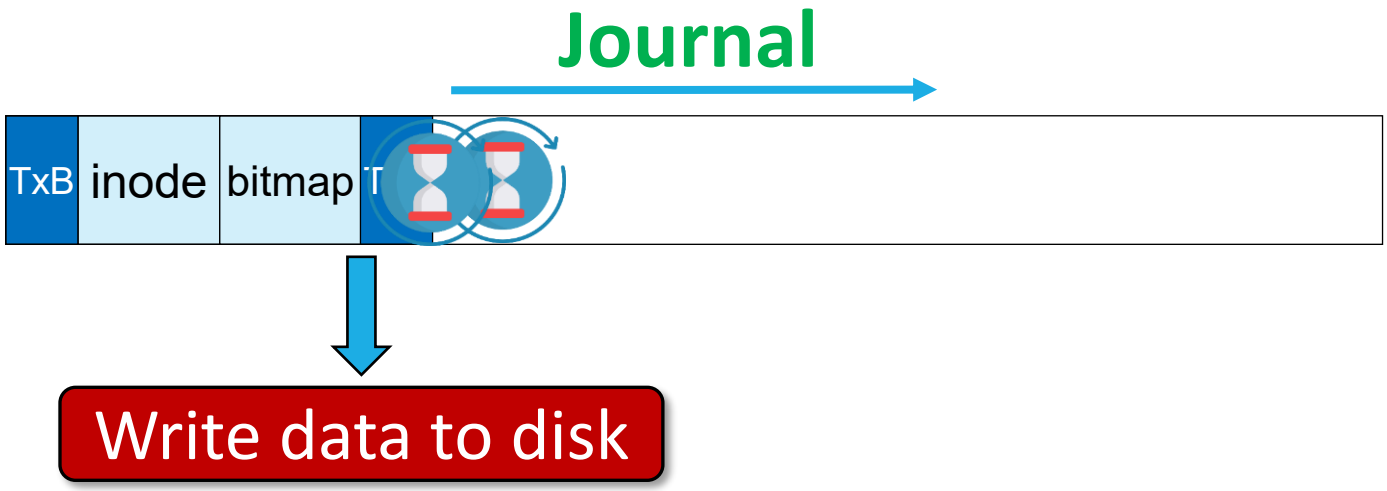
Metadata Journaling

- Problem: writing every data block twice
  - Commit to log
  - Checkpoint to disk



Metadata Journaling

- **Ordered journaling**: user data is not written to journal
  - Write data block to file system proper



  - Write data *after* transaction → may point to garbage data
  - Write data *before* transaction → avoids problem

Journaling

- Final protocol:
  1. **Data write**
     - Write data to final location, wait for completion
  2. **Journal write**
     - Write transaction contents (TxB and metadata), wait for completion
  3. **Journal commit**
     - Write transaction commit block (**TxE**)
     - Wait for write to complete → transaction is **committed**
  4. **Checkpoint**
     - Write metadata to final locations
  5. **Free**
     - Later, mark transaction as free in journal superblock