Operating Systems

File System and More

(Chapter 41 ~ 42)

Dr. Young-Woo Kwon

Announcements

- 기말시험
  - 6.19 (IT5 245), 10:30AM ~ 11:45AM
  - 시험범위
    - Memory paging ~ Filesystem

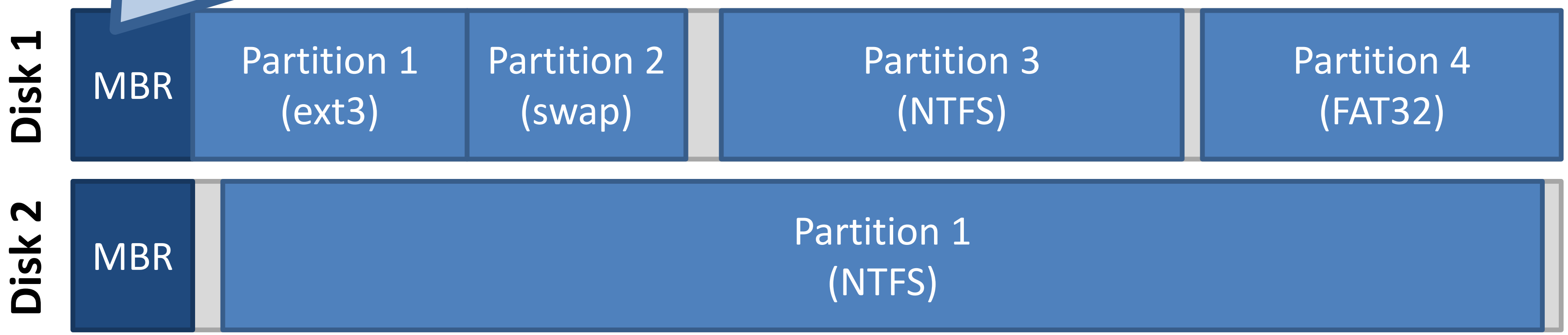
Building the Root File System

- One of the first tasks of an OS during bootup is to build the root file system
- Locate all bootable media
  - Internal and external hard disks
  - SSDs
  - Floppy disks, CDs, DVDs, USB sticks
- Locate all the partitions on each media
  - Read MBR(s), extended partition tables, etc.
- Mount one or more partitions
  - Makes the file system(s) available for access

The Master Boot Record

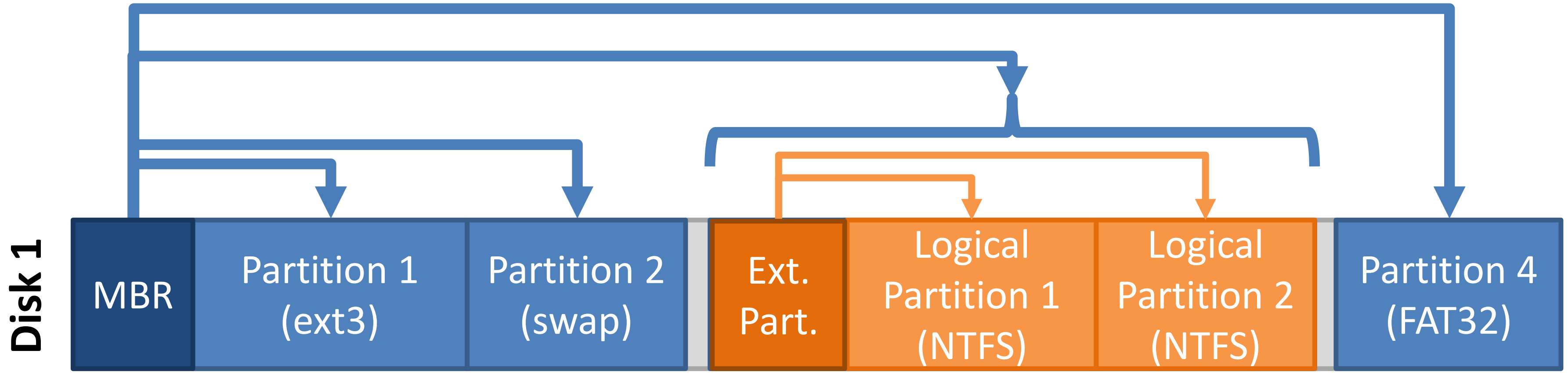
Address		Description	Size (Bytes)
Hex	**Dec.**		
0x000	0	Bootstrap code area	446
0x1BE	446	Partition Entry #1	16
0x1CE	462	Partition Entry #2	16
0x1DE	478	Partition Entry #3	16
0x1EE	494	Partition Entry #4	16
0x1FE	510	Magic Number	2
		Total:	**512**

Includes the starting LBA (logical block addressing) and length of the partition



Extended Partitions

- In some cases, you may want >4 partitions
- Modern OSes support extended partitions



- Extended partitions may use OS-specific partition table formats (meta-data)
  – Thus, other OSes may not be able to read the logical partitions

Types of Root File Systems

```
[cbw@ativ9 ~] df -h
Filesystem      Size  Used   Avail  Use%  Mounted on
/dev/sda7        39G   14G    23G   38%   /
/dev/sda2      296M   48M  249M    16%   /boot/efi
/dev/sda5      127G   86G    42G   68%   /media/cbw/Data
/dev/sda4       61G   34G    27G   57%   /media/cbw/Windows
/dev/sdb1      1.9G  352K   1.9G    1%   /media/cbw/NDSS-2013
```

1 drive, 4 partitions

1drive, 1 partition

- Linux has a single root
  - One partition is mounted as /
  - All other partitions are mounted somewhere under /
- Typically, the partition containing the kernel is mounted as / or C:

Mounting a File System

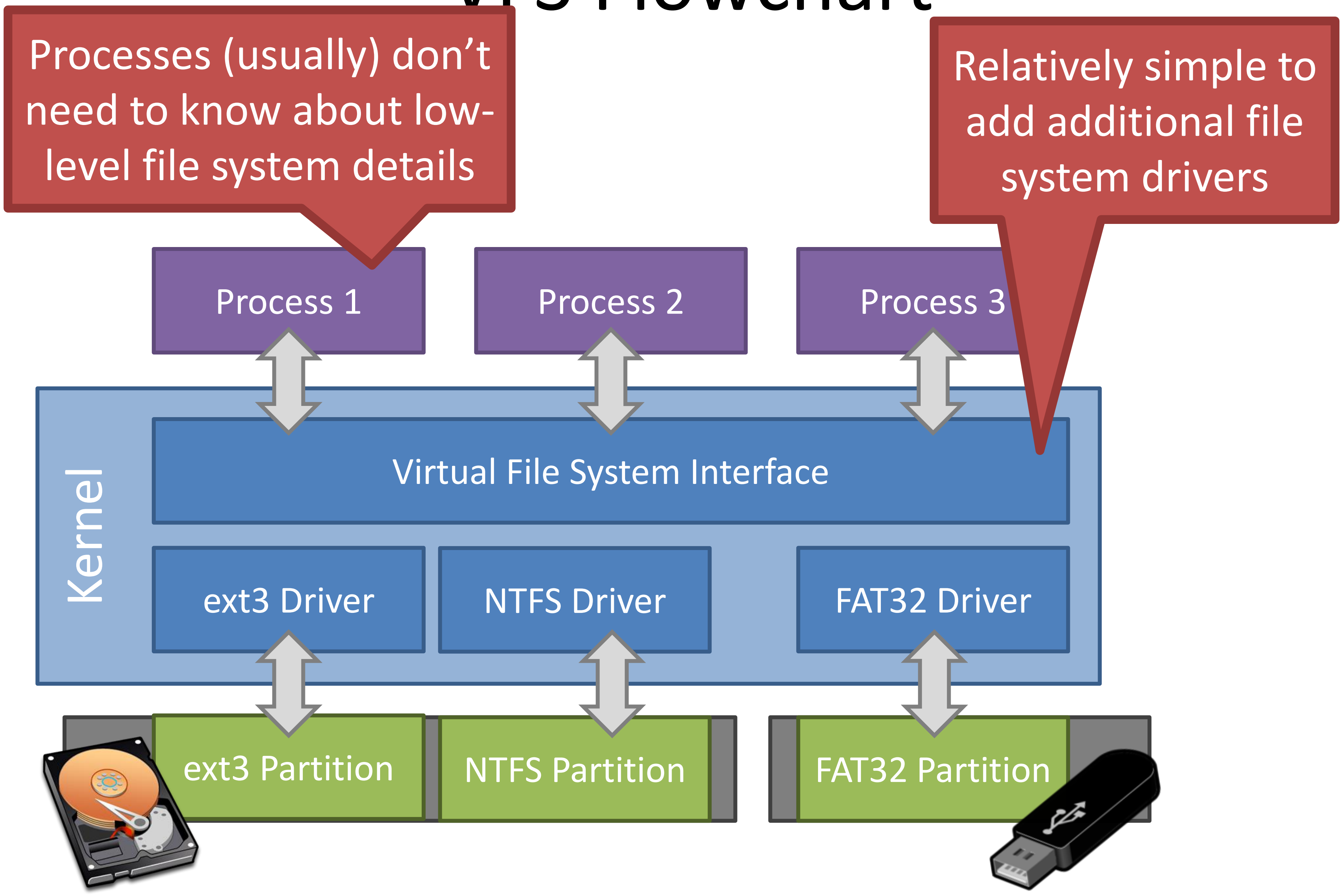
- Read the **super block** for the target file system
  - Contains meta-data about the file system
  - Version, size, locations of key structures on disk, etc.
- Determine the **mount point**
  - On Windows: pick a drive letter
  - On Linux: mount the new file system under a specific directory

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sda5	127G	86G	42G	68%	/media/cbw/Data
/dev/sda4	61G	34G	27G	57%	/media/cbw/Windows
/dev/sdb1	1.9G	352K	1.9G	1%	/media/cbw/NDSS-2013

Virtual File System Interface

- Problem: the OS may mount **several partitions** containing different underlying file systems
  – It would be bad if processes had to use different APIs for different file systems
- Linux uses a **Virtual File System** interface (VFS)
  – Exposes POSIX APIs to processes
  – Forwards requests to lower-level file system specific drivers
- Windows uses a similar system

VFS Flowchart



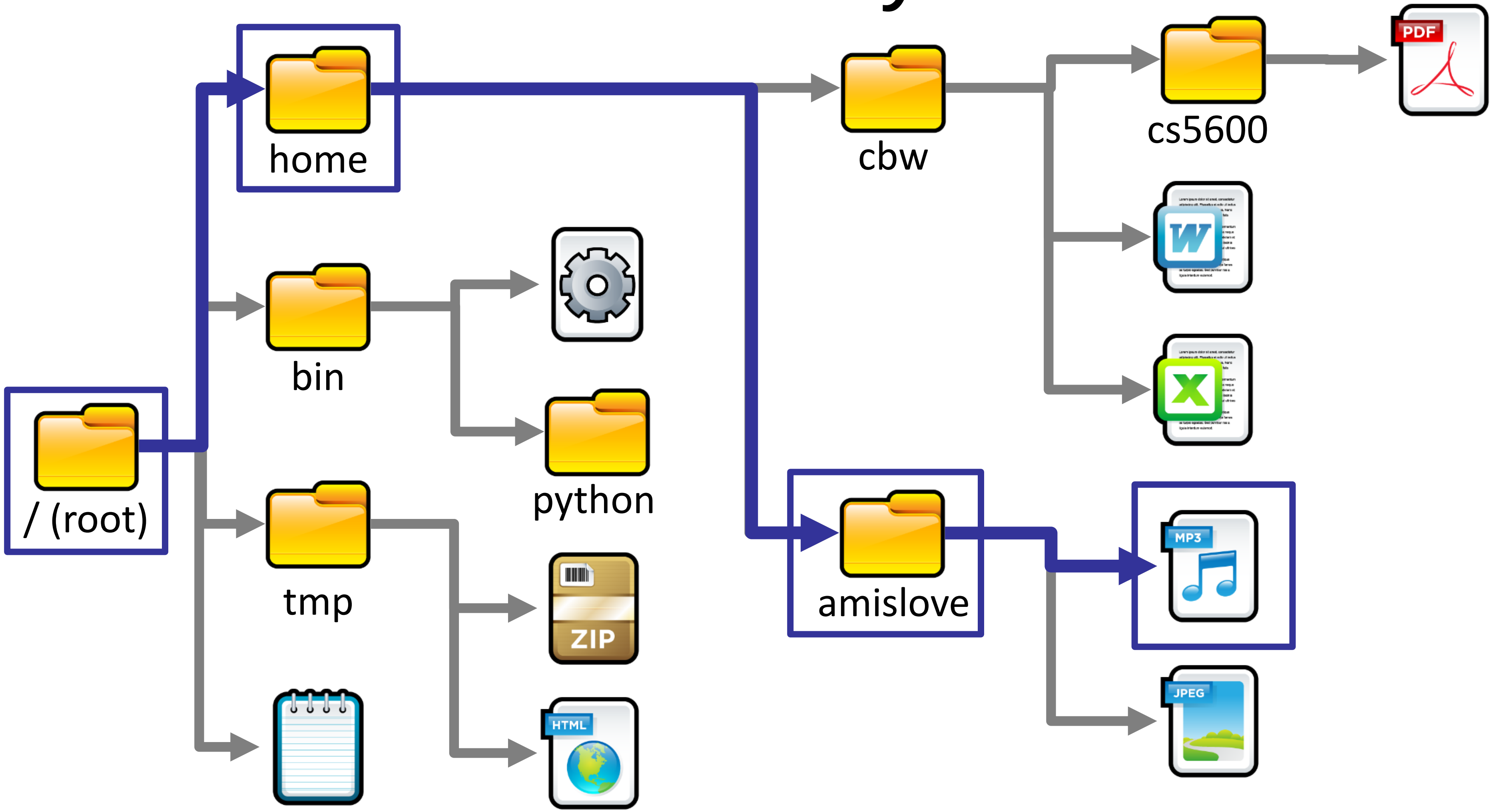
Mount isn’t Just for Bootup

- When you plug storage devices into your running system, mount is executed in the background
  - Example: plugging in a USB stick
- What does it mean to “*safely eject*” a device?
  - **Flush cached writes** to that device
  - Cleanly unmount the file system on that device

Status Check

- At this point, the OS can locate and mount partitions
- Next step: what is the on-disk layout of the file system?
  - We expect certain features from a file system
    - Named files
    - Nested hierarchy of directories
    - Meta-data like creation time, file permissions, etc.
  - How do we design on-disk structures that support these features?

The Directory Tree



- Navigated using a path
  - E.g. /home/amislove/music.mp3

Absolute and Relative Paths

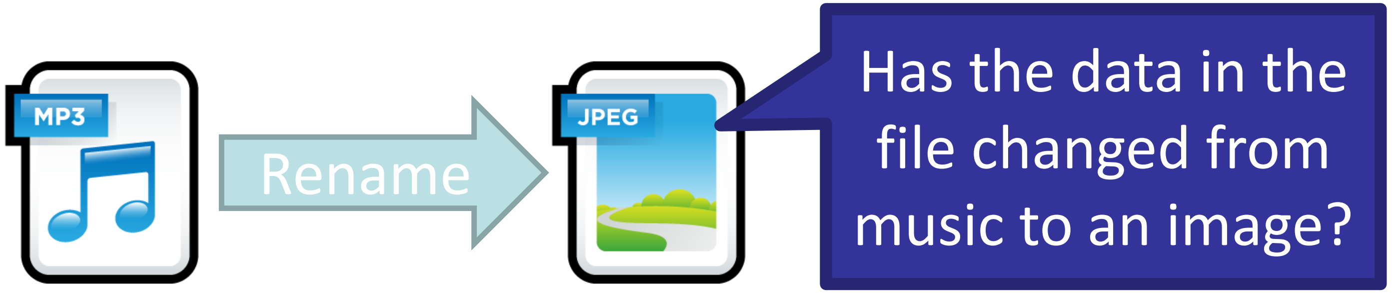
- Two types of file system paths
  - Absolute
    - Full path from the root to the object
      - Example: /home/cbw/cs5600/hw4.pdf
      - Example: C:\Users\cbw\Documents\
  - Relative
    - OS keeps track of the working directory for each process
    - Path relative to the current working directory
    - Examples [working directory = /home/cbw]:
      - syllabus.docx [→ /home/cbw/syllabus.docx]
      - cs5600/hw4.pdf [→ /home/cbw/cs5600/hw4.pdf]
      - ./cs5600/hw4.pdf [→ /home/cbw/cs5600/hw4.pdf]
      - ../amislove/music.mp3 [→ /home/amislove/music.mp3]

Files

- A file is a composed of two components
  - The file data itself
    - One or **more blocks (sectors)** of binary data
    - A file can **contain anything**
  - Meta-data about the file
    - Name, total size
    - What directory is it in?
    - Created time, modified time, access time
    - Hidden or system file?
    - Owner and owner's group
    - Permissions: read/write/execute

File Extensions

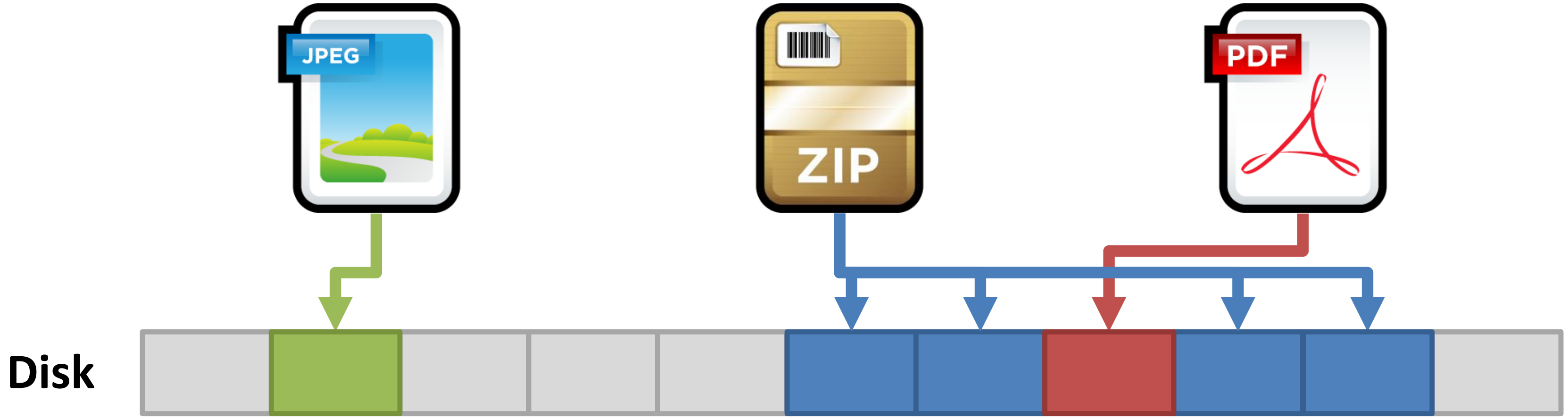
- File name are often written in dotted notation
  - E.g. program.exe, image.jpg, music.mp3
- A file's **extension does not mean anything**
  - Any file (regardless of its contents) can be given any name or extension



- Graphical shells (like Windows explorer) use extensions to **match files → programs**
  - This **mapping may fail** for a variety of reasons

More File Meta-Data

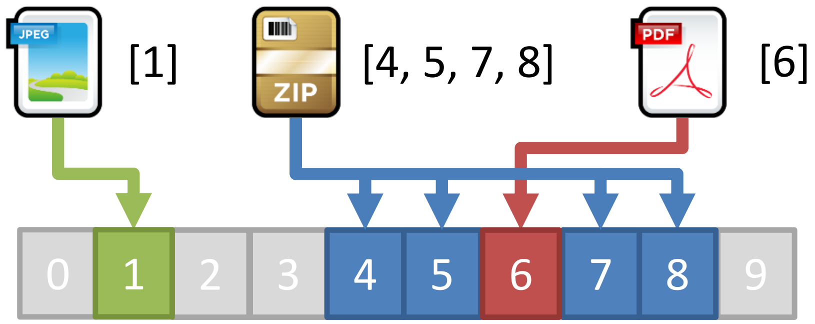
- Files have additional meta-data that is not typically shown to users
  - **Unique identifier** (filename? Inode?)
  - **Structure** that maps the file to blocks on the disk
- Managing the mapping from files to blocks is one of the key jobs of the file system



Mapping Files to Blocks

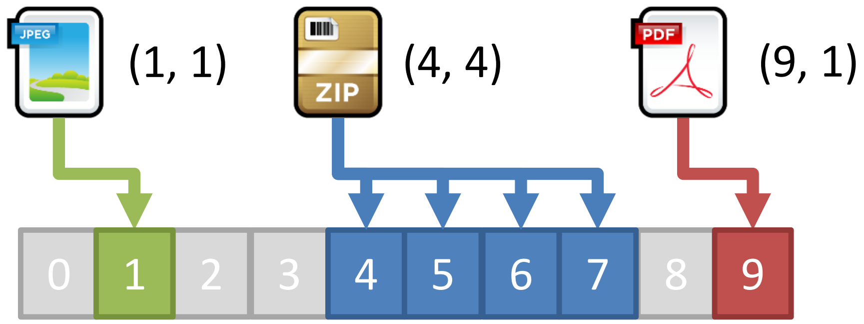
- Every file is composed of >=1 blocks
- Key question: how do we map a file to its blocks?

List of blocks



- Problem?
  - Really large files

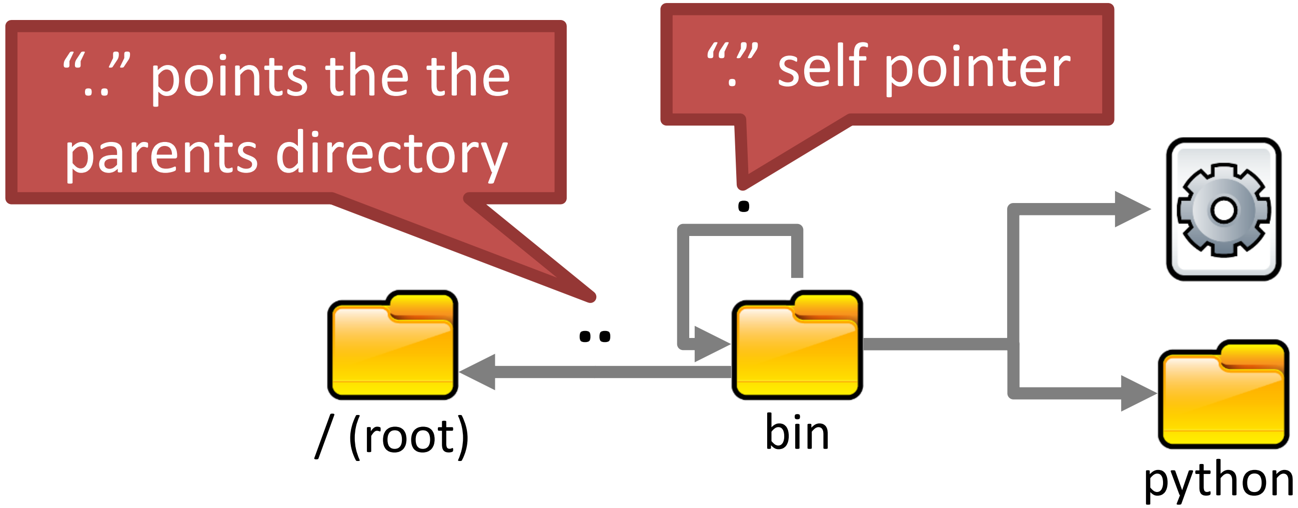
As (start, length) pairs



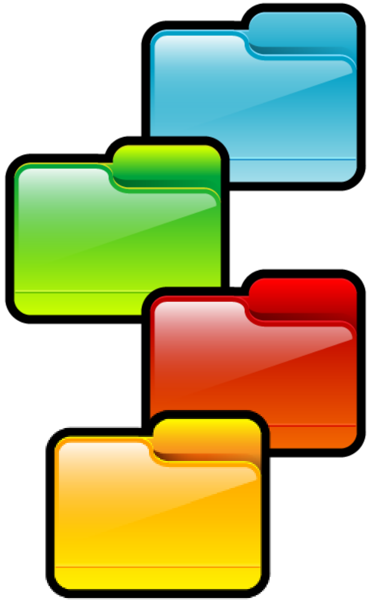
- Problem?
  - Fragmentation
  - E.g. try to add a new file with 3 blocks

Directories

- Traditionally, file systems have used a hierarchical, tree-structured namespace
  - Directories are objects that contain other objects
    - i.e. a directory may (or may not) have children
  - Files are leaves in the tree
- By default, directories contain at least two entries

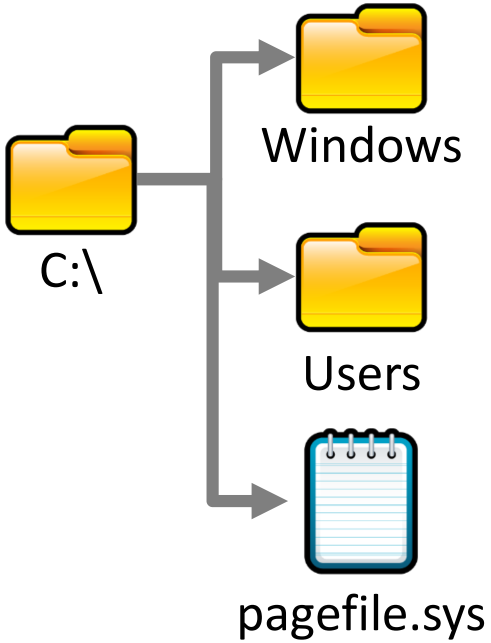


More on Directories

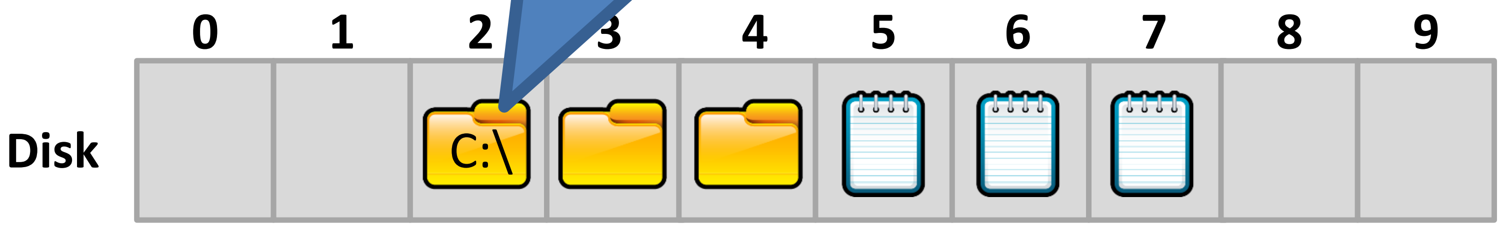


- Directories have associated meta-data
  - Name, number of entries
  - Created time, modified time, access time
  - Permissions (read/write), owner, and group
- The file system must encode directories and store them on the disk
  - Typically, directories are stored as a special type of file
  - File contains a list of entries inside the directory, plus some meta-data for each entry

Example Directory File



Name	Index	Dir?	Perms
.	2	Y	rwx
Windows	3	Y	rwx
Users	4	Y	rwx
pagefile.sys	5	N	r



Directory File Implementation

- Each directory file stores many entries
- Key Question: how do you encode the entries?

- Other alternatives: hash tables, B-trees
  - More on B-trees later…
- In practice, implementing directory files is complicated
  - Example: do filenames have a fixed, maximum length or variable length?

- Good: O(1) to add new entries
  - Just append to the file
- Bad: O(n) to search for an entry

- Good: O(log n) to search for an entry
- Bad: O(n) to add new entries
  - Entire file has to be rewritten

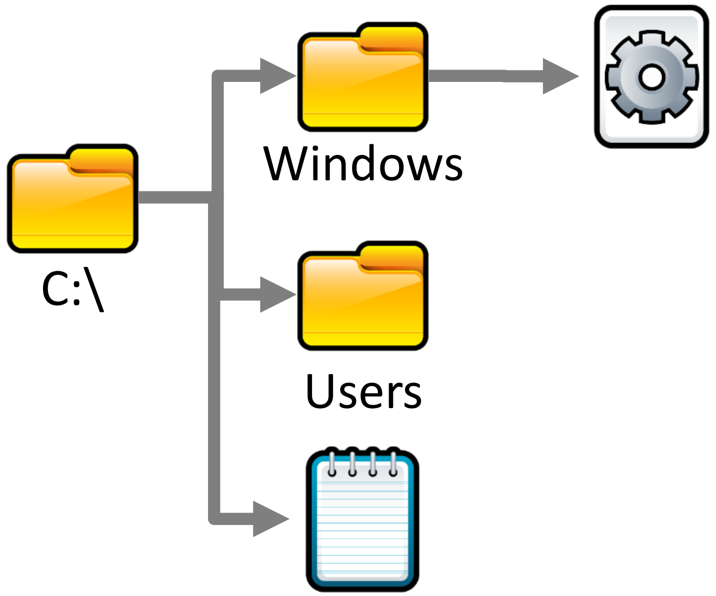
File Allocation Tables (FAT)

- Simple file system popularized by MS-DOS
  - First introduced in 1977
  - Most devices today use the FAT32 spec from 1996
  - FAT12, FAT16, VFAT, FAT32, etc.
- Still quite popular today
  - Default format for USB sticks and memory cards
  - Used for EFI boot partitions
- Name comes from the **index table** used to track directories and files

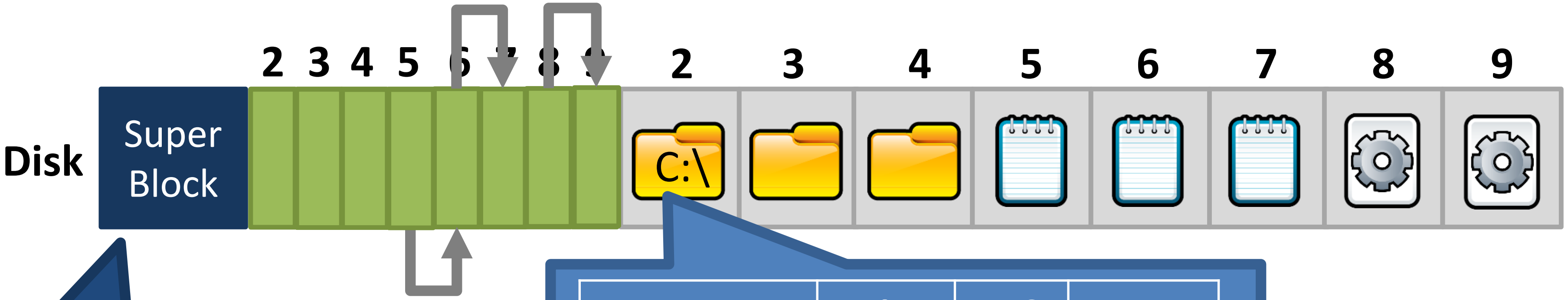
- Stores basic info about the file system
- FAT version, location of boot files
- Total number of blocks
- Index of the root directory in the FAT

- File allocation table (FAT)
- Marks which blocks are free or in-use
- **Linked-list structure** to manage large files

- Store file and directory data
- Each block is a fixed size (4KB – 64KB)
- Files may span multiple blocks



- Directories are special files
  - File contains a list of entries inside the directory
- Possible values for FAT entries:
  - 0 – entry is empty
  - 1 – reserved by the OS
  - 1 < N < 0xFFFF – next block in a chain
  - 0xFFFF – end of a chain



Root directory index = 2

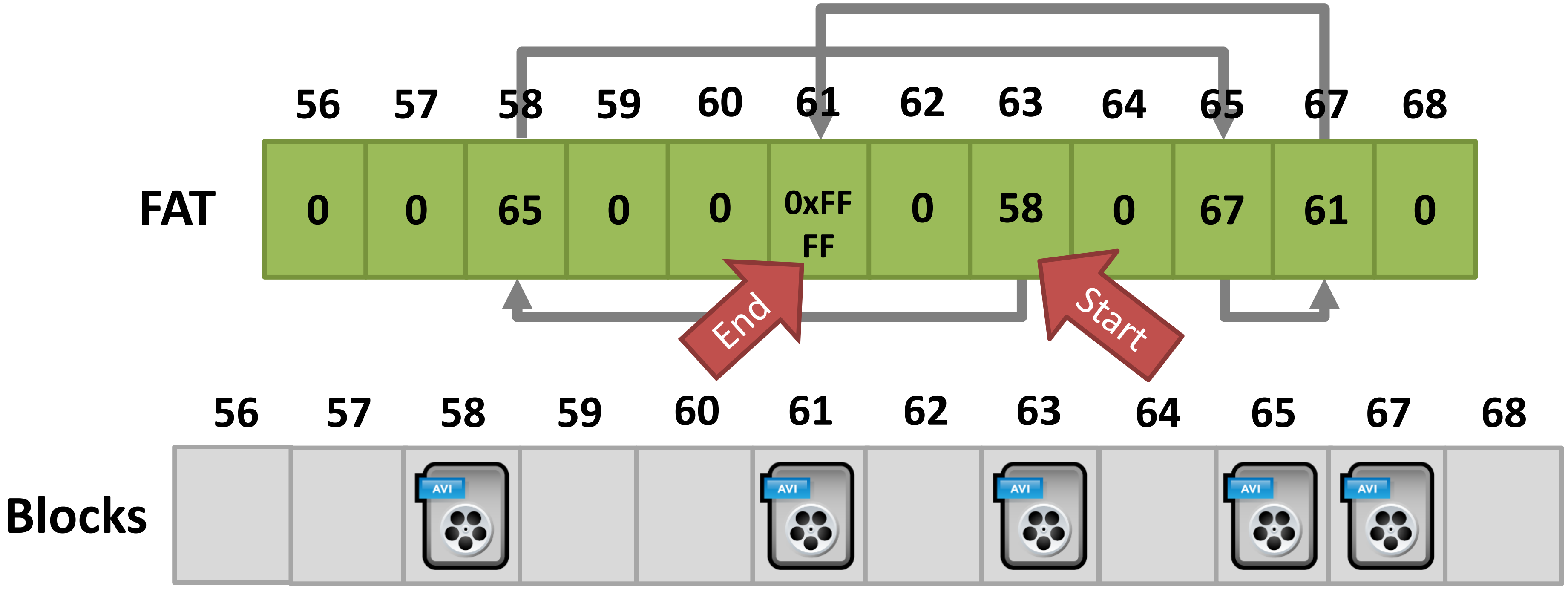
Name	Index	Dir?	Perms
.	2	Y	rwx

Fat Table Entries

- len(FAT) == Number of clusters on the disk
  - Max number of files/directories is bounded
  - Decided when you format the partition
- The FAT version roughly corresponds to the size in bits of each FAT entry
  - E.g. FAT16 → each FAT entry is 16 bits
  - More bits → larger disks are supported

Fragmentation

- Blocks for a file need not to be contiguous



Possible values for FAT entries:

- 0 – entry is empty
- 1 < N < 0xFFFF – next block in a chain
- 0xFFFF – end of a chain

FAT: The Good and the Bad

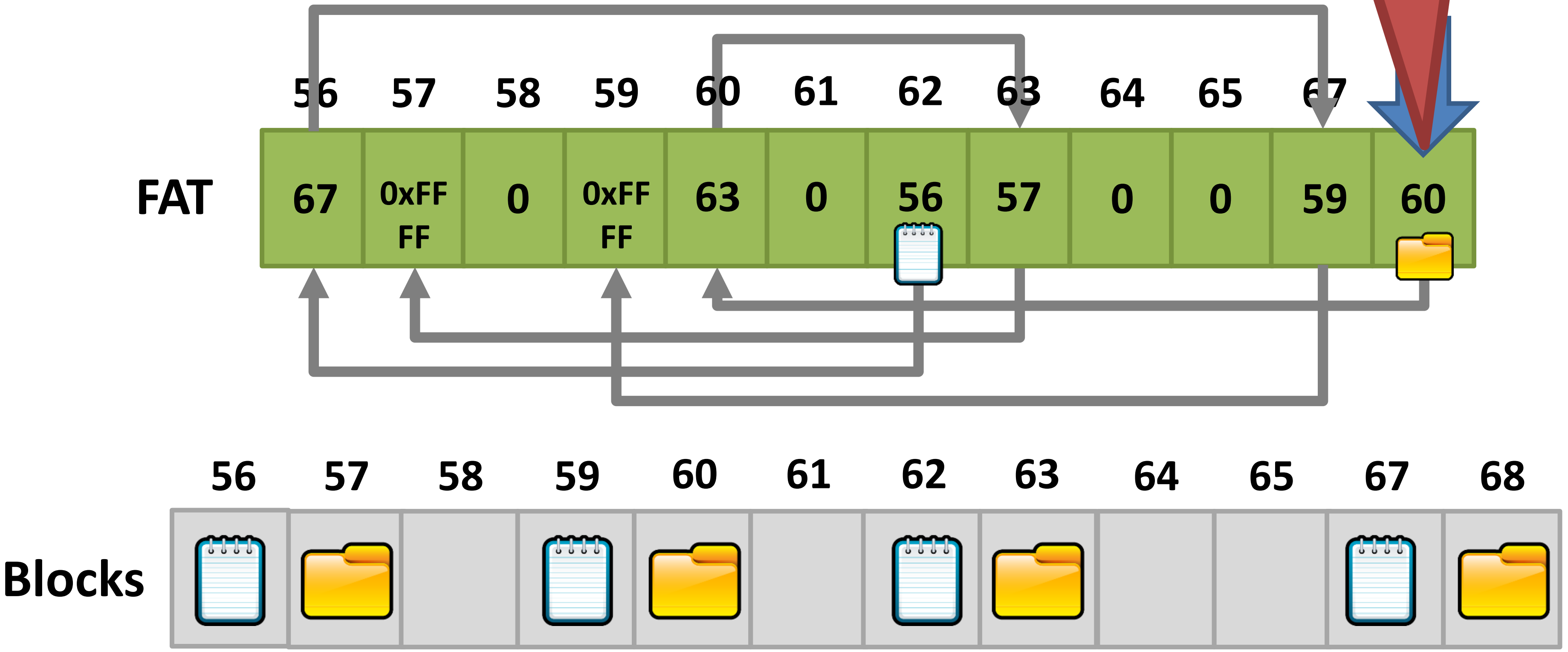
- The Good – FAT supports:
  - Hierarchical tree of directories and files
  - Variable length files
  - Basic file and directory meta-data
- The Bad
  - At most, **FAT32 supports 2TB disks**
  - Locating free chunks requires **scanning the entire FAT**
  - Prone to internal and external **fragmentation**
    - Large blocks → internal fragmentation
  - Reads require a lot of **random seeking**

Lots of Seeki

- Consider the following code:

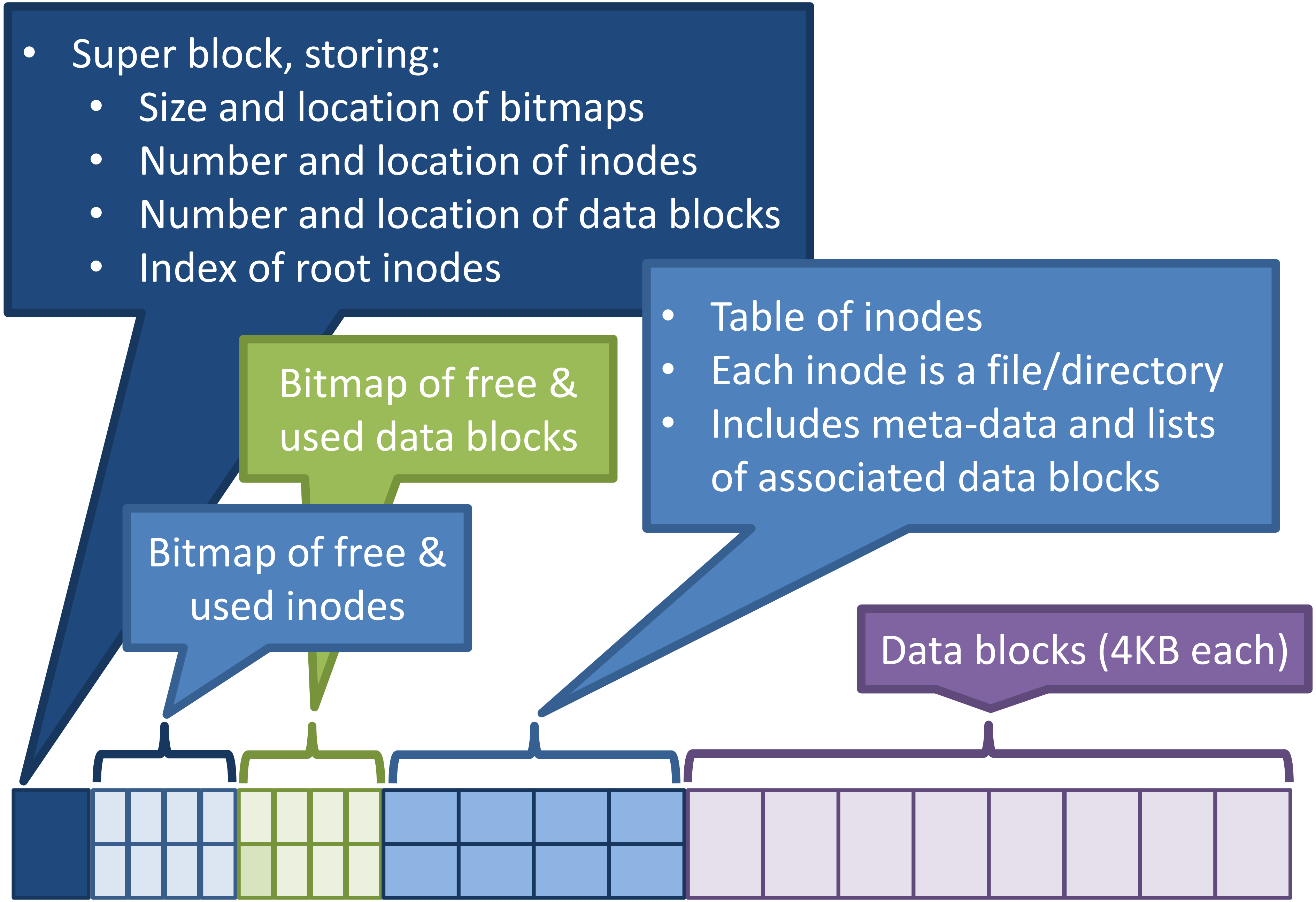
```
int fd = open("my_file.txt", "r");
int r = read(fd, buffer, 1024 * 4 * 4); // 4 4KB blo
```

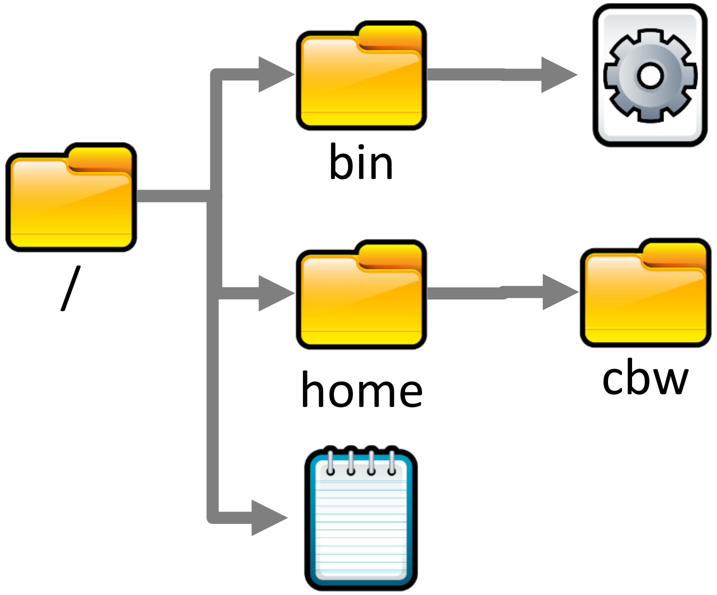
FAT may have very low spatial locality, thus a lot of random seeking



Size Distribution of Files

- FAT uses a linked list for all files
  - Simple and uniform mechanism
  - … but, it is **not optimized for short or long files**
- Question: are short or long files more common?
  - Studies over the last 30 years show that **short files are much more common**
  - 2KB is the most common file size
  - Average file size is 200KB (biased upward by a few very large files)
- Key idea: **optimize the file system for many small files!**

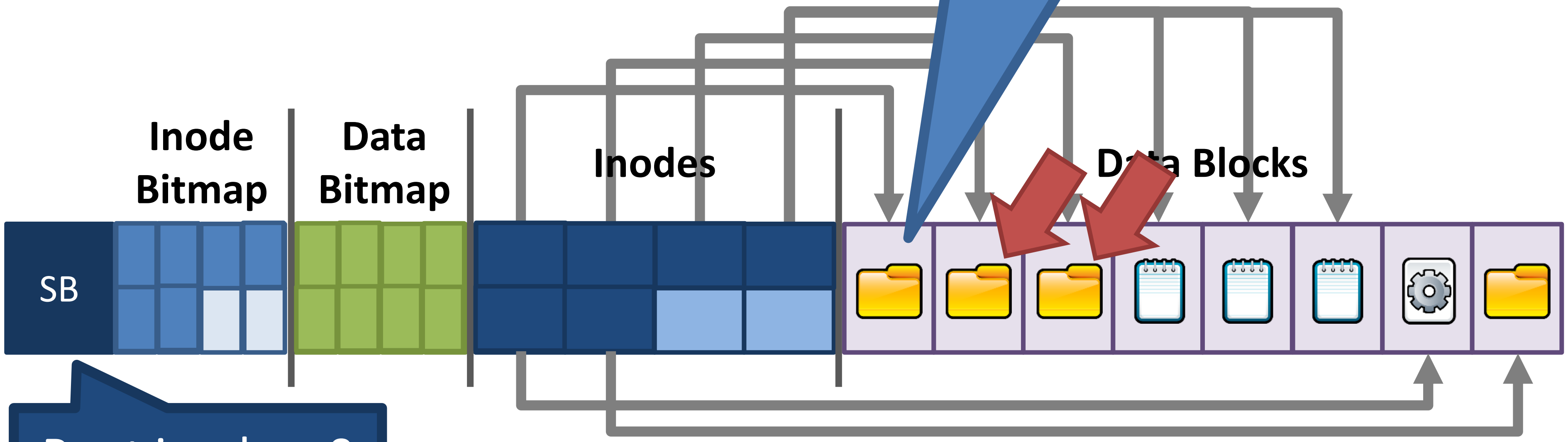

• Super block, storing:
• Size and location of bitmaps
• Number and location of inodes
• Number and location of data blocks
• Index of root inodes
Bitmap of free & used data blocks
• Table of inodes
• Each inode is a file/directory
• Includes meta-data and lists of associated data blocks
Bitmap of free & used inodes
Data blocks (4KB each)



- Directories are files
- Contains the list of entries in the directory

Name	inode

- Each inode can directly point to 12 blocks
- Can also indirectly point to blocks at 1, 2, and 3 levels of depth



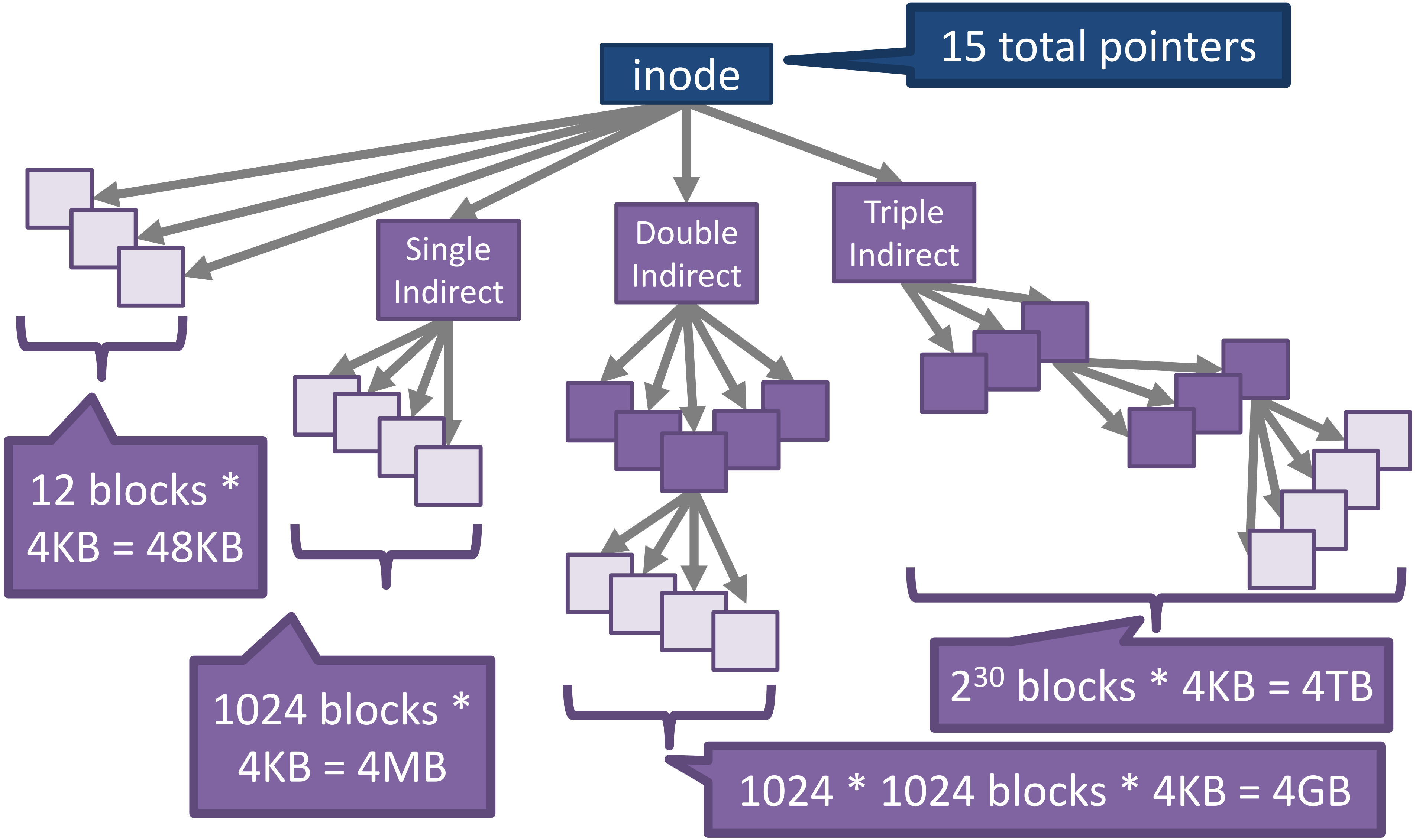
Root inode = 0

ext2 inodes

Size (bytes)	Name	What is this field for?
2	mode	Read/write/execute?
2	uid	User ID of the file owner
4	size	Size of the file in bytes
4	time	Last access time
4	ctime	Creation time
4	mtime	Last modification time
4	dtime	Deletion time
2	gid	Group ID of the file
2	links_count	How many hard links point to this file?
4	blocks	How many data blocks are allocated to this file?
4	flags	File or directory? Plus, other simple flags
60	block	15 direct and indirect pointers to data blocks

inode Block Pointers

- Each inode is the root of an unbalanced tree of data blocks



Advantages of inodes

- Optimized for file systems with many small files
  - Each inode can **directly point to 48KB of data**
  - Only one layer of indirection needed for 4MB files
- Faster file access
  - Greater meta-data **locality** → less random seeking
  - No need to traverse long, chained FAT entries
- Easier free space management
  - **Bitmaps can be cached in memory for fast access**
  - inode and data space handled **independently**

File Reading Example

	Bitmaps		inodes			Data Blocks				
	data	inode	root	tmp	file	root	tmp	file[0]	file[1]	file[3]
open("/tmp/file")			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Time

Update the last accessed time of the file

File Create and Write Example

	Bitmaps		inodes			Data Blocks		
	data	inode	root	tmp	file	root	tmp	file[0]
open("/tmp/file")			read					
						read		
				read				
							read	
		read						
		write						
					write			
							write	
				write				
write()					read			
	read							
	write							
								write
					write			

Time

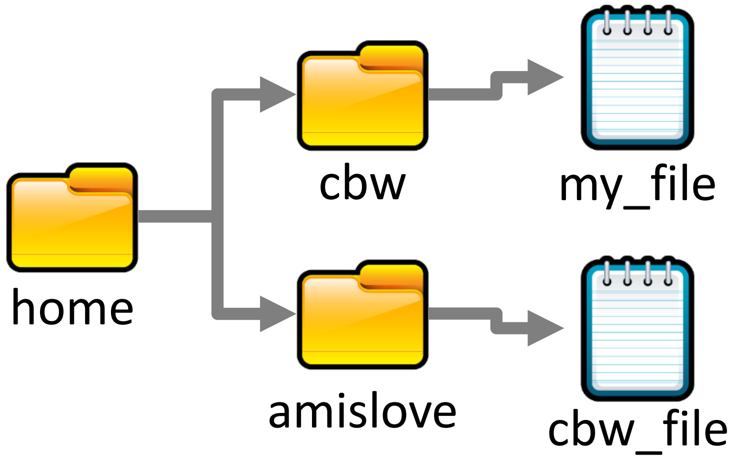
Update the modified time of the directory

ext2 inodes, Again

Size (bytes)	Name	What is this field for?
2	mode	Read/write/execute?
2	uid	User ID of the file owner
4	size	Size of the file in bytes
4	time	Last access time
4	ctime	Creation time
4	mtime	Last modification time
4	dtime	Deletion time
2	gid	Group ID of the file
2	links_count	How many hard links point to this file?
4	blocks	How many data blocks are allocated to this file?
4	flags	File or directory? Plus, other simple flags
60	block	15 direct and indirect pointers to data blocks

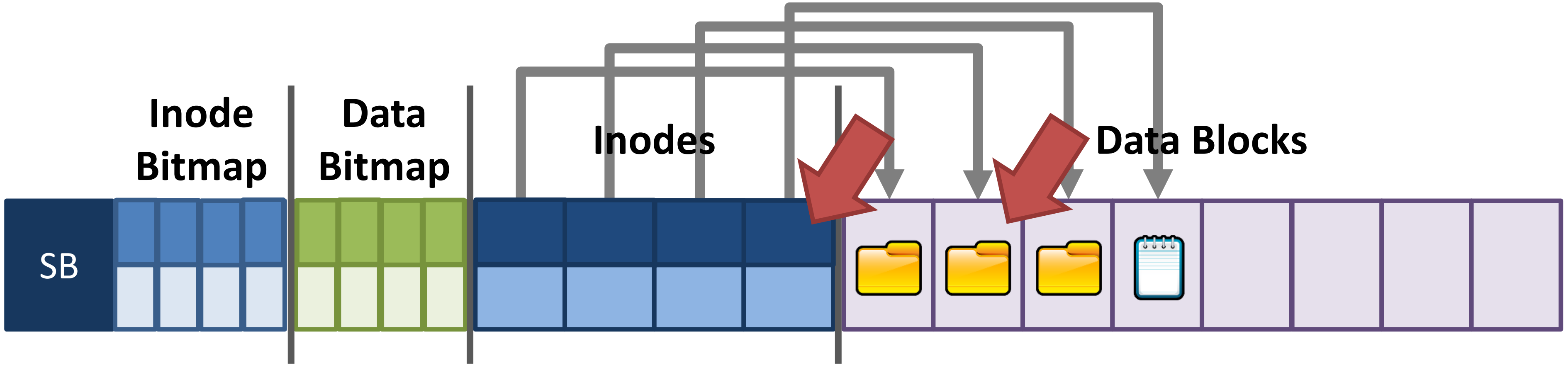
Hard Link Example

- Multiple directory entries may point to the same inode




```
[amislove@ativ9 ~] ln –T ../cbw/my_file cbw_file
```

1. Add an entry to the "amislove" directory
2. Increase the link_count of the "my_file" inode



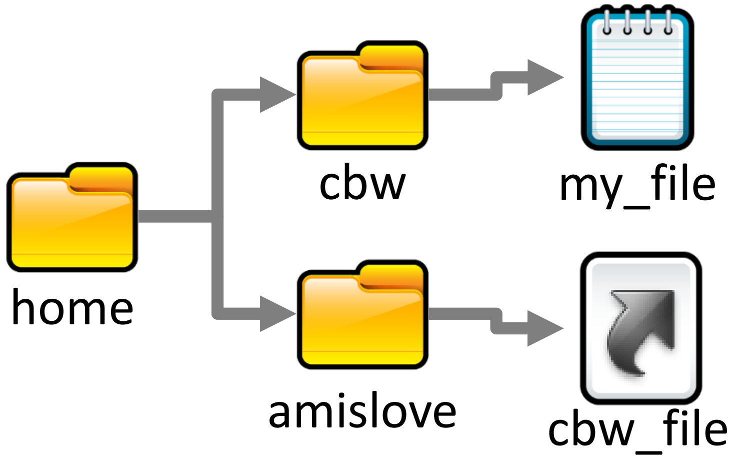
Hard Link Details

- Hard links give you the ability to create many **aliases** of the same underlying file
  - Can be in different directories
- Target file will not be marked invalid (deleted) until link_count == 0
  - This is why POSIX “delete” is called *unlink()*
- Disadvantage of hard links
  - **Inodes are only unique within a single file system**
  - Thus, can only point to files **in the same partition**

Soft(Symbolic) Links

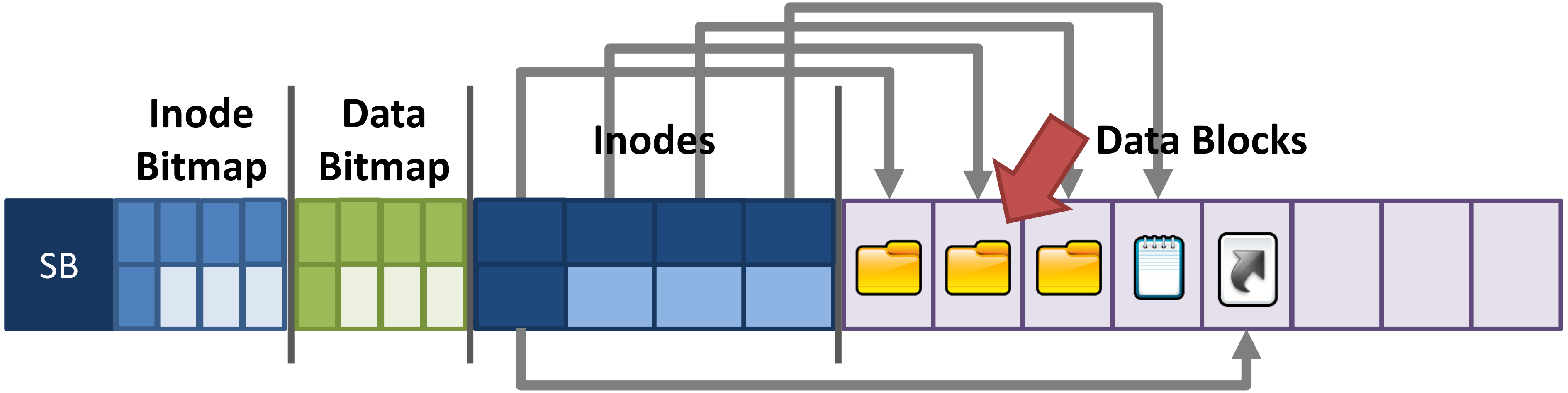
- **Soft links** are special files that include the path to another file
  - Also known as **symbolic** links
  - On Windows, known as **shortcuts**
  - File may be on **another partition or device**

Soft Link Example



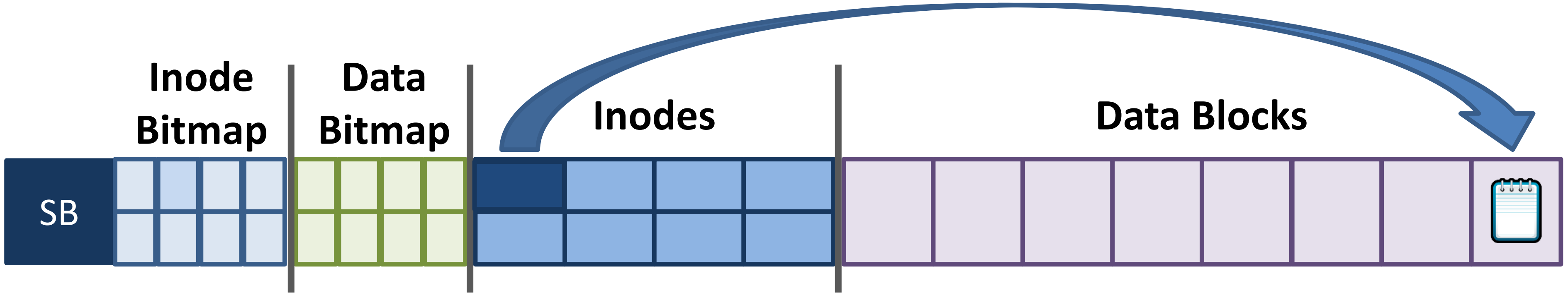

```
[amislove@ativ9 ~] ln –s ../cbw/my_file cbw_file
```

1. Create a soft link file
2. Add it to the current directory



ext: The Good and the Bad

- The Good – ext file system (inodes) support:
  - All the typical file/directory features
  - Hard and soft links
  - More performant (less seeking) than FAT
- The Bad: **poor locality**
  - ext is optimized for a particular file size distribution
  - However, it is not optimized for **spinning disks**
  - **inodes and associated data are far apart on the disk!**

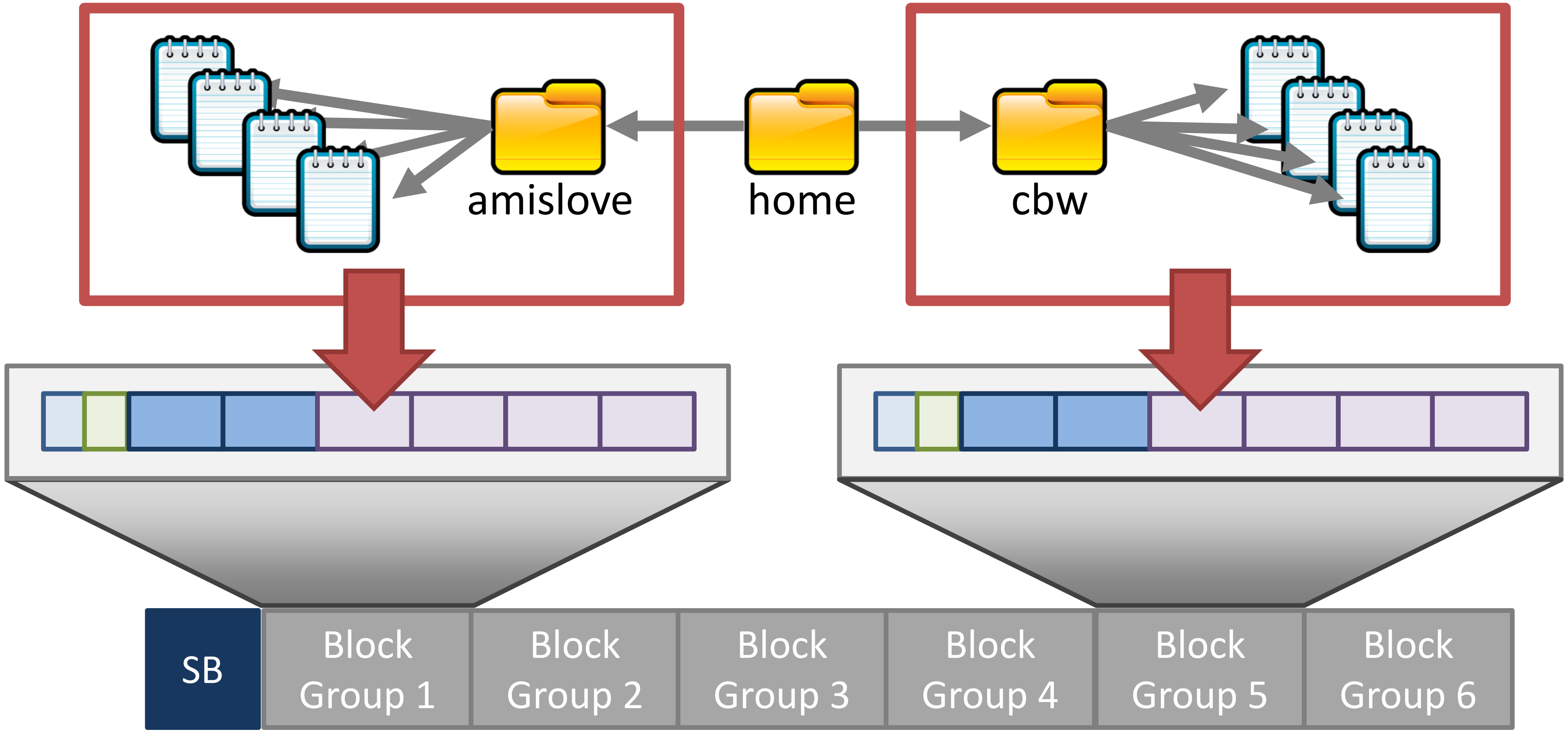


Fast File System (FFS)

- FFS developed at Berkeley in 1984
  - First attempt at a **disk aware** file system
  - i.e. optimized for performance on spinning disks
- Observation: processes tend to access files that are in the same (or close) directories
  - Spatial locality
- Key idea: place groups of directories and their files into **cylinder groups**
  - Introduced into ext2, called **block groups**

Allocation Policy

- ext2 attempts to keep **related files and directories within the same block group**



ext2: The Good and the Bad

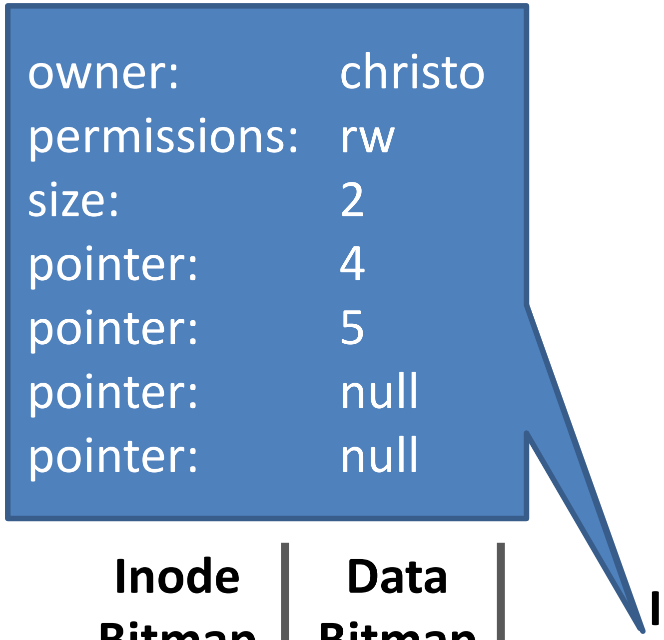
- The good – ext2 supports:
  - All the features of ext…
  - … with even better performance (because of increased spatial locality)
- The bad
  - Large files must **cross block groups**
  - As the file system becomes more complex, the chance of **file system corruption** grows
    - E.g. invalid inodes, incorrect directory entries, etc.

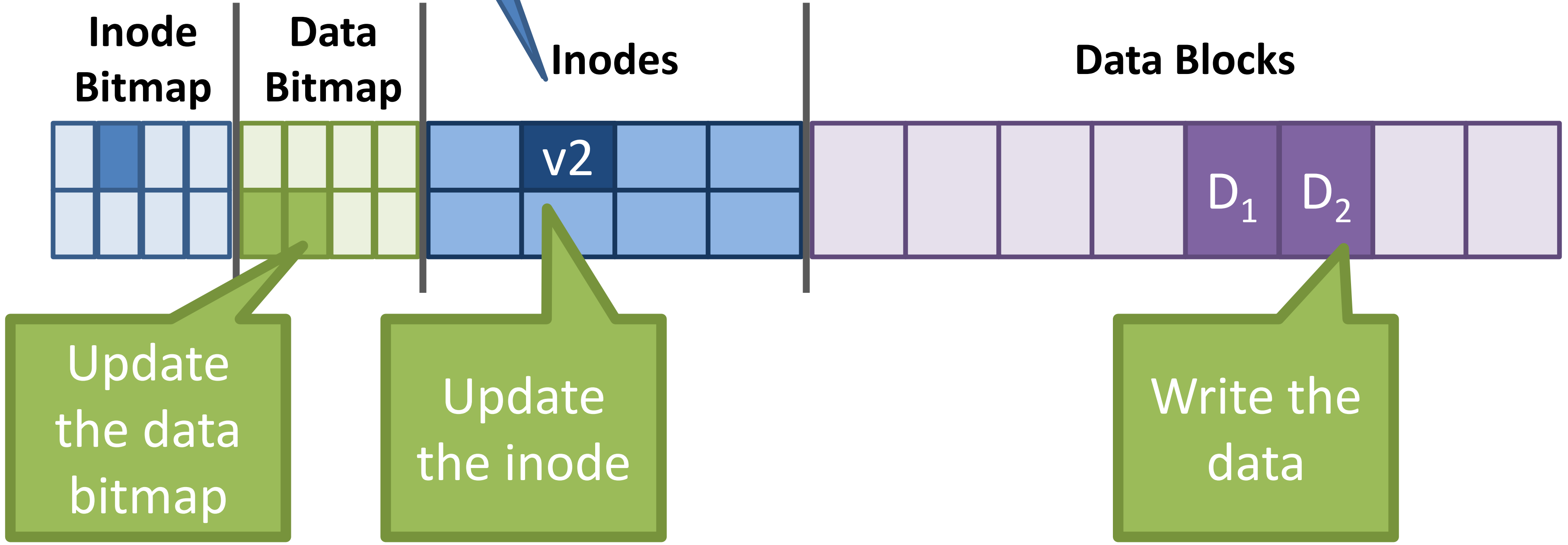
Maintaining Consistency

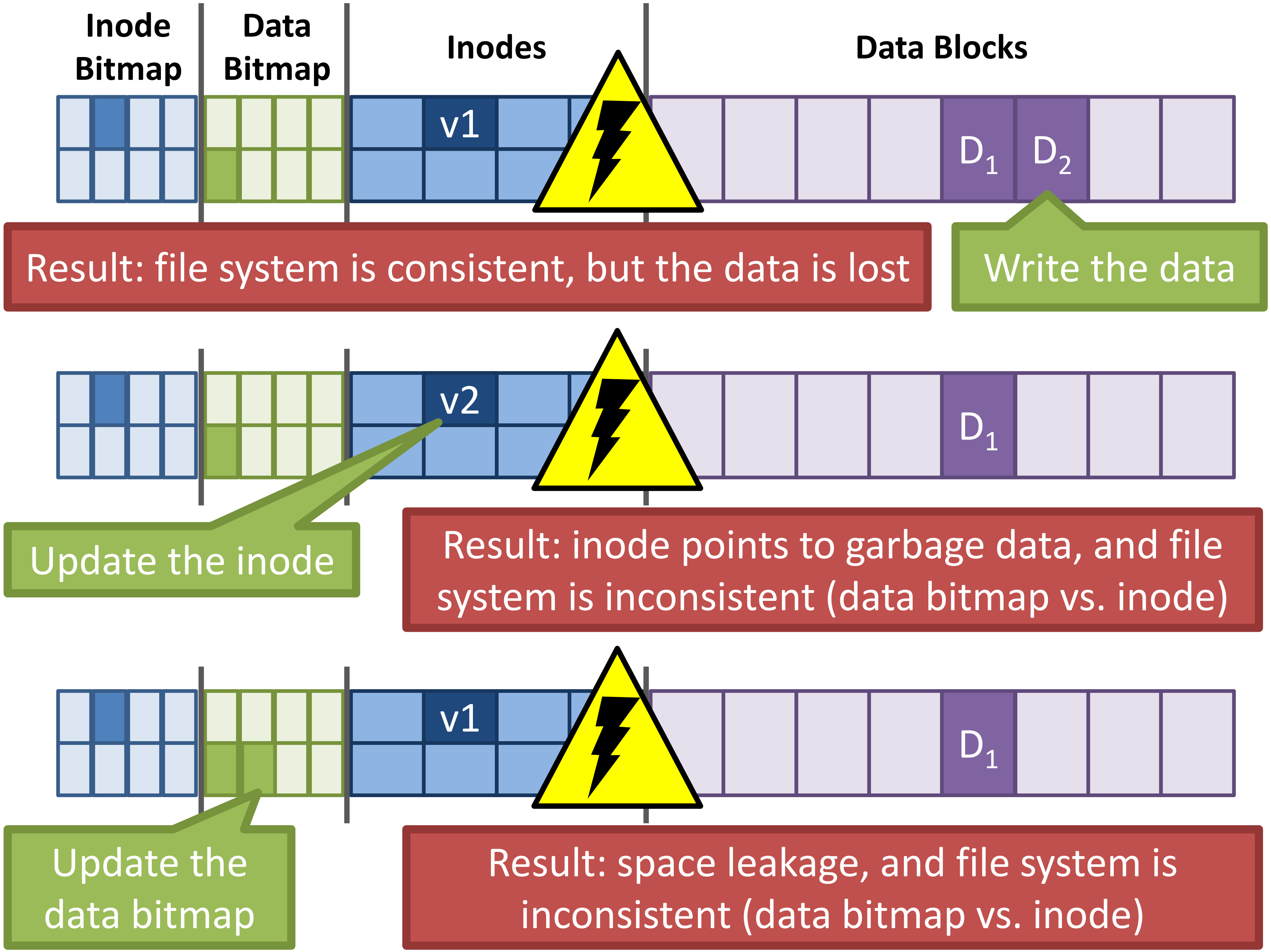
- Many operations results in **multiple, independent writes to the file system**
  - Example: append a block to an existing file
    - Update the free data bitmap
    - Update the inode
    - Write the user data
- What happens if the computer **crashes in the middle of this process**?

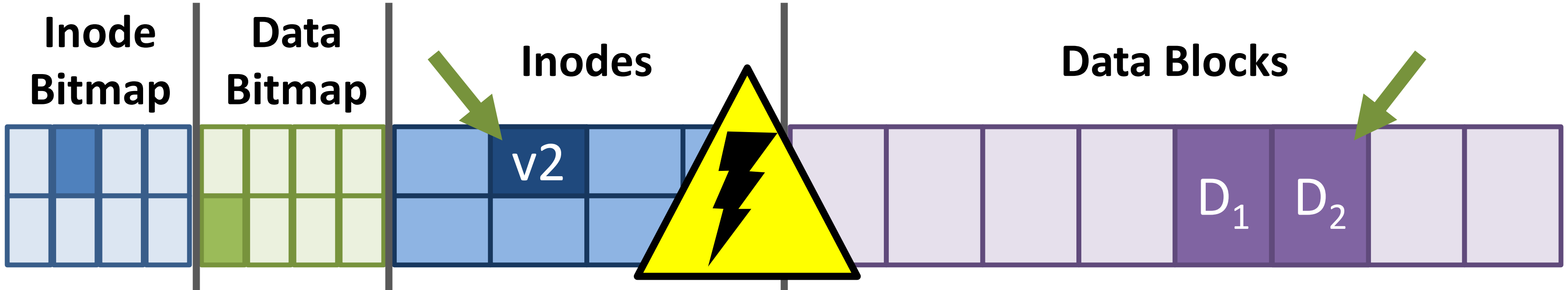
File Append Example

- These three operations can potentially be done in any order
- … but the system can crash at any time

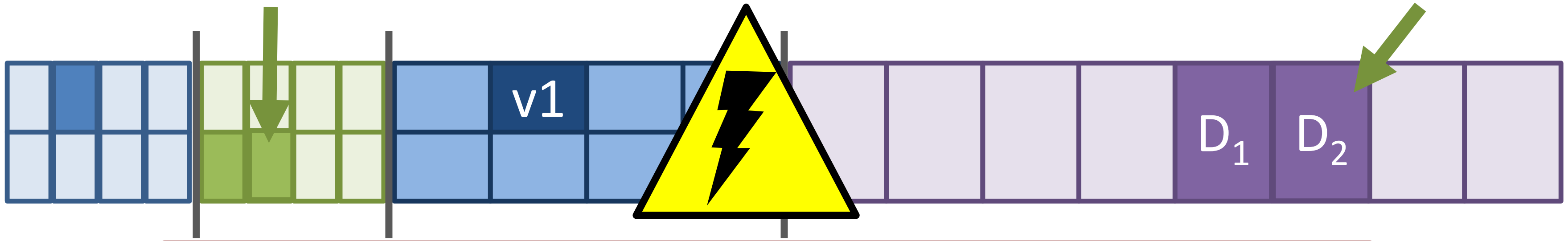




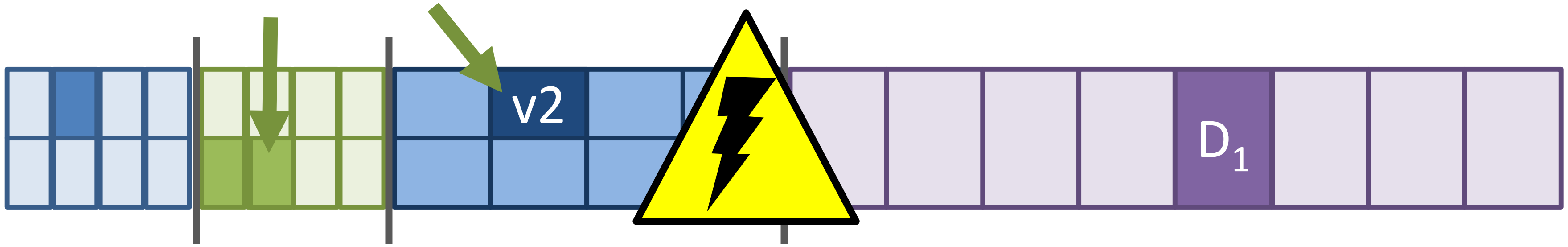

Inode Bitmap
Data Bitmap
Inodes
Data Blocks
v1
D1 D2
Result: file system is consistent, but the data is lost
Write the data
v2
D1
Update the inode
Result: inode points to garbage data, and file system is inconsistent (data bitmap vs. inode)
v1
D1
Update the data bitmap
Result: space leakage, and file system is inconsistent (data bitmap vs. inode)



Result: inode points to data, but file system is inconsistent



Result: file system is inconsistent, and the data is useless since it's not associated with an inode



Result: file system is consistent, but the inode points to garbage data

The Crash Consistency Problem

- The disk guarantees that **sector writes are atomic**
  - No way to make multi-sector writes atomic
- How to ensure consistency after a crash?
  - Don't bother to ensure consistency
    - **Accept that the file system may be inconsistent** after a crash
    - **Run a program that fixes** the file system during bootup
    - **File system checker (fsck)**
  - Use a transaction log to make multi-writes atomic
    - Log **stores a history** of all writes to the disk
    - After a crash the **log can be "replayed"** to finish updates
    - **Journaling file system**

Approach 1: File System Checker

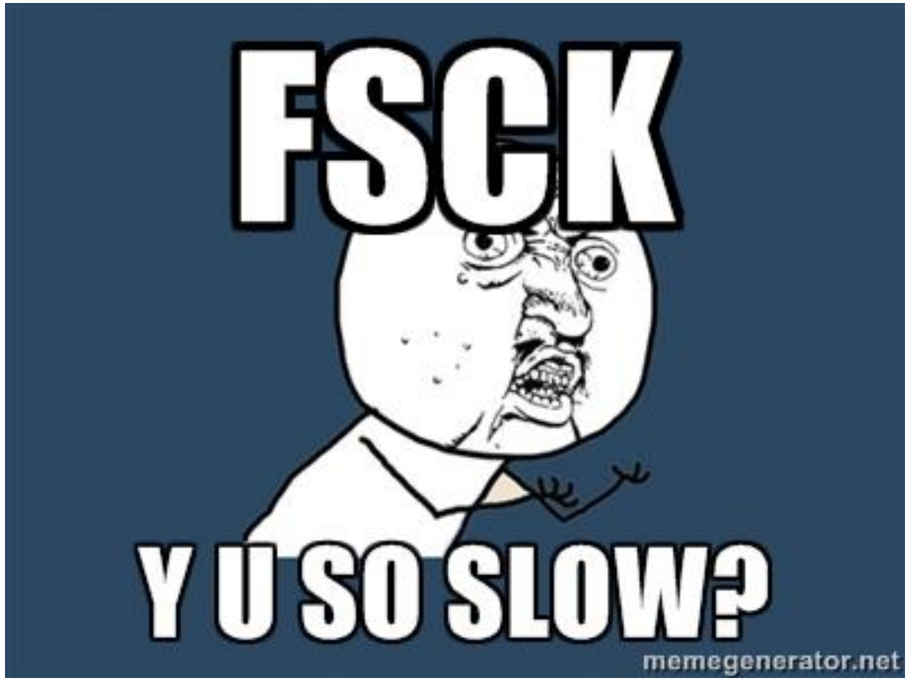
- Key idea: **fix inconsistent file** systems during bootup
  - Unix utility called **fsck** (**chkdsk** on Windows)
  - **Scans the entire file system** multiple times, identifying and correcting inconsistencies
- Why during bootup?
  - No other file system activity can be going on
  - After fsck runs, bootup/mounting can continue

fsck Tasks

- **Superblock:** validate the superblock, replace it with a backup if it is corrupted
- **Free blocks and inodes:** rebuild the bitmaps by scanning all inodes
- **Reachability:** make sure all inodes are reachable from the root of the file system
- **inodes:** delete all corrupted inodes, and rebuild their link counts by walking the directory tree
- **directories:** verify the integrity of all directories
- … and many other minor consistency checks

fsck: the Good and the Bad

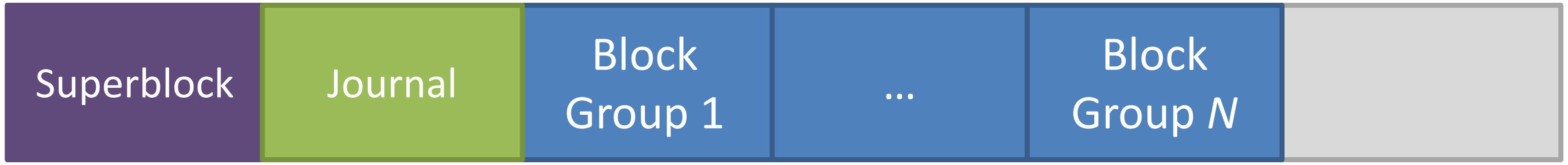
- Advantages of *fsck*
  - Doesn't require the file system to do any work to ensure consistency
  - Makes **the file system implementation simpler**
- Disadvantages of *fsck*
  - **Very complicated to implement** the *fsck* program
    - Many possible inconsistencies that must be identified
    - Many difficult corner cases to consider and handle
  - *fsck* is **super slow**
    - Scans the entire file system multiple times
    - Imagine how long it would take to fsck a 40 TB RAID array


FSCK
Y U SO SLOW?
memegenerator.net


FSCK
YOU!

Approach 2: Journaling

- Problem: *fsck* is slow because it checks the entire file system after a crash
  - What if we knew where the last writes were before the crash, and just checked those?
- Key idea: make writes transactional by using a write-ahead log
  - Commonly referred to as a journal
- **Ext3 and NTFS use journaling**



Write-Ahead Log

- Key idea: **writes to disk are first written into a log**
  - After the log is written, the writes execute normally
  - In essence, **the log records transactions**
- What happens after a crash…
  - **If the writes to the log are interrupted?**
    - The transaction is incomplete
    - The user's data is lost, but the file system is consistent
  - **If the writes to the log succeed, but the normal writes are interrupted?**
    - The file system may be inconsistent, but…
    - The log has exactly **the right information to fix the problem**

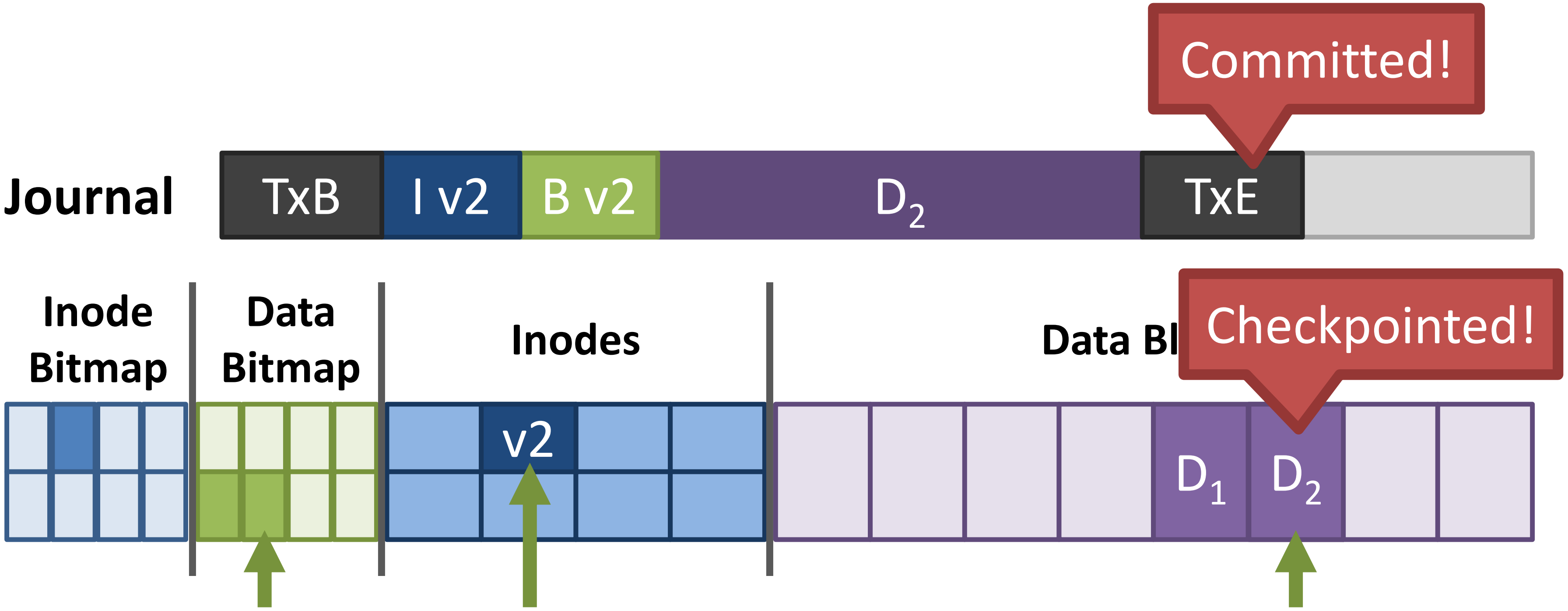
Data Journaling Example

- Assume we are appending to a file
  - Three writes: inode v2, data bitmap v2, data D_2
- Before executing these writes, first log them

1. Begin a new transaction with a unique ID=*k*
2. Write the updated meta-data block(s)
3. Write the file data block(s)
4. Write an end-of-transaction with ID=*k*

Commits and Checkpoints

- A transaction is committed after all writes to the log are complete
- After a transaction is committed, the OS checkpoints the update



- Final step: free the checkpointed transaction

Journal Implementation

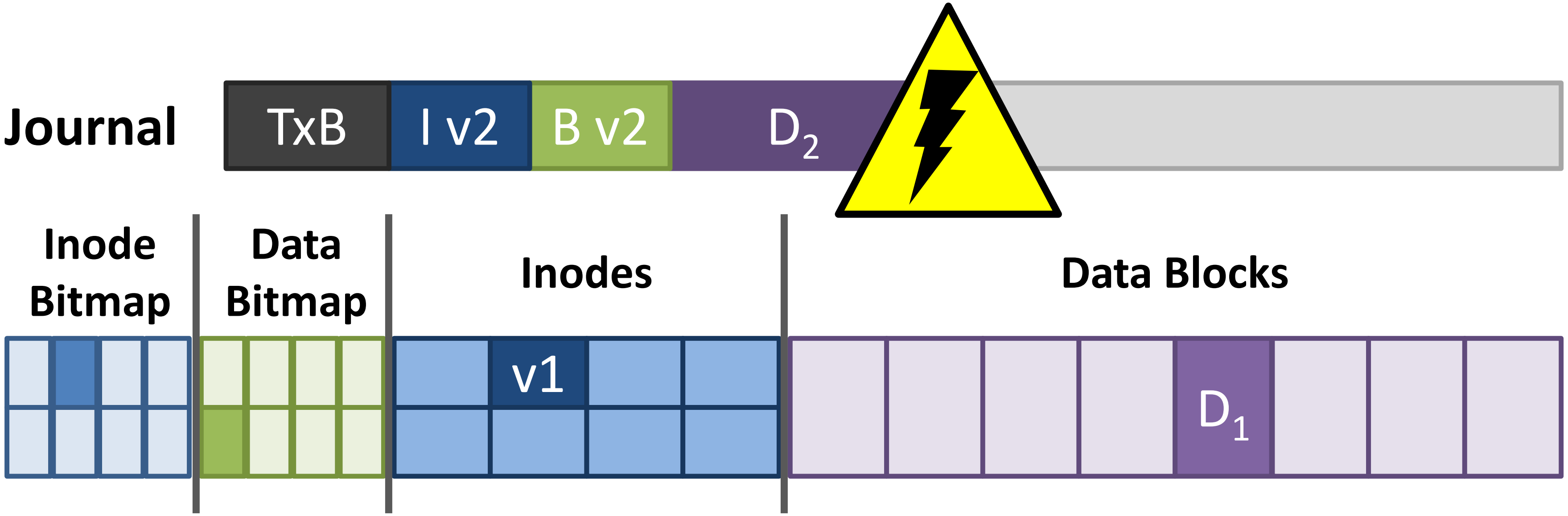
- Journals are typically implemented as a circular buffer
  - Journal is **append-only**
- OS maintains pointers to the front and back of the transactions in the buffer
  - As transactions are freed, the back is moved up
- Thus, the contents of the journal are never deleted, they are just overwritten over time

Data Journaling Timeline

	Journal				File System	
	TxB	**Meta-data**	**Data**	**TxE**	**Meta-data**	**Data**
Time ↓	Issue	Issue	Issue	Issue		
		Complete				
	Complete					
				Complete		
			Complete			
					Issue	Issue
						Complete
					Complete	

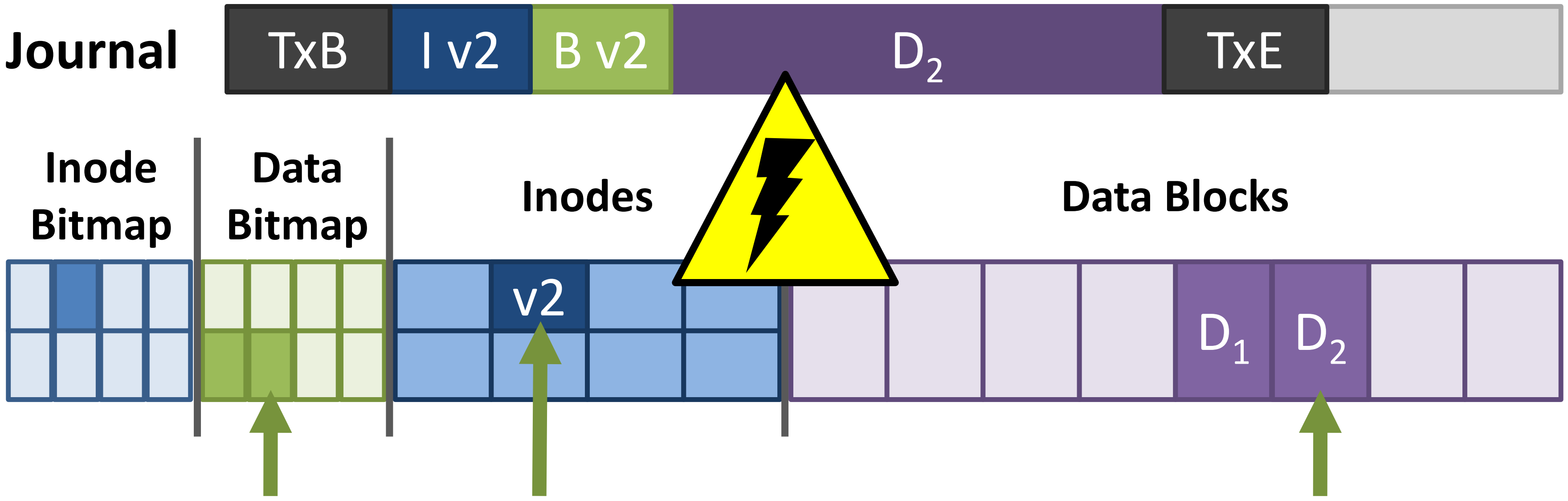
Crash Recovery (1)

- What if the system crashes during logging?
  - If the transaction is not committed, data is lost
  - But, the file system remains consistent



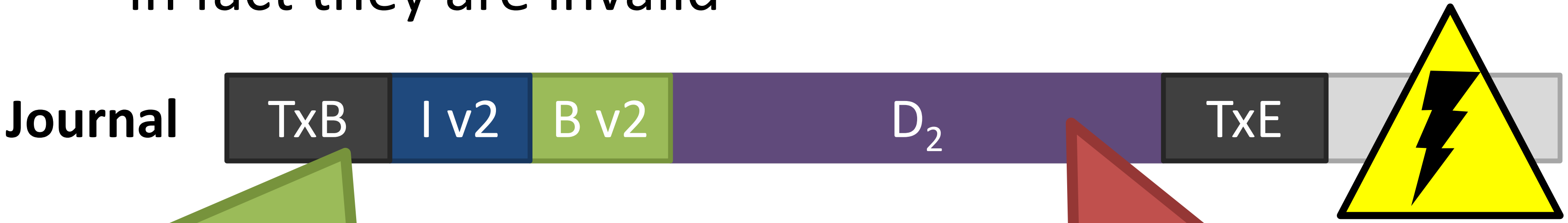
Crash Recovery (2)

- What if the system crashes during the checkpoint?
  - File system may be inconsistent
  - During reboot, transactions that are committed but not free are replayed in order
  - Thus, no data is lost and consistency is restored



Corrupted Transactions

- Problem: the disk scheduler may not execute writes in-order
  - Transactions in the log may appear committed, when in fact they are invalid



- Solution: add a checksum to TxB
- During recovery, reject transactions with invalid checksums
- Implemented on Linux in ext4

- Transaction looks valid, but the data is missing!
- During replay, garbage data is written to the file system

Journaling: The Good and the Bad

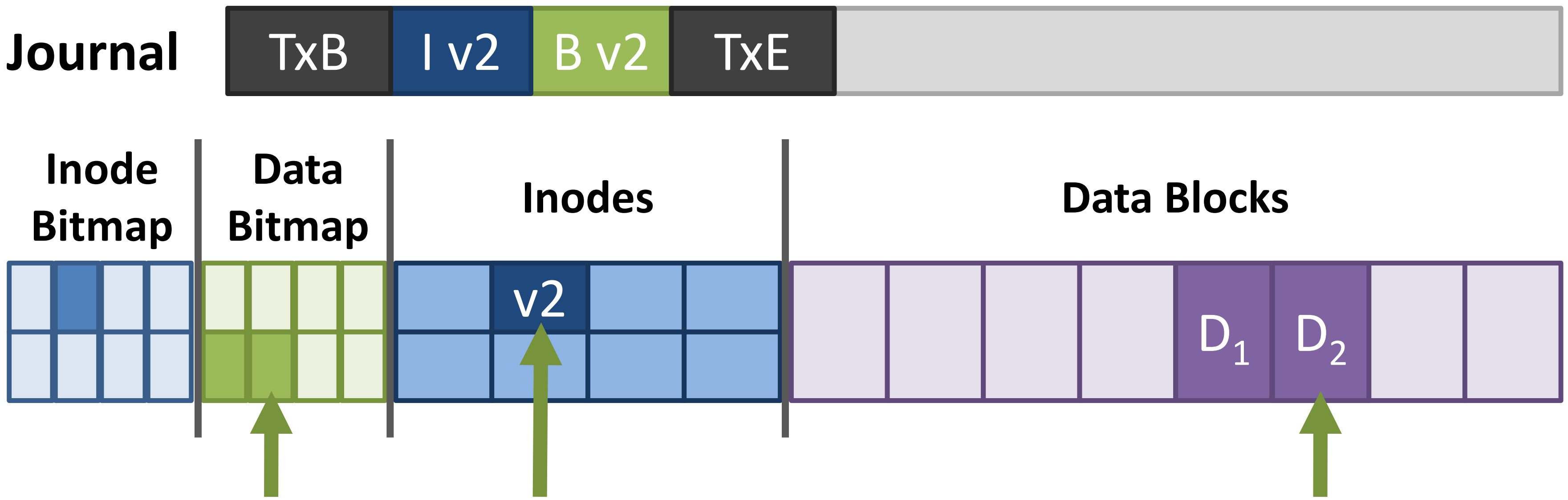
- Advantages of journaling
  - Robust, fast file system recovery
    - No need to scan the entire journal or file system
  - Relatively straight forward to implement
- Disadvantages of journaling
  - Write traffic to the disk is doubled
    - Especially the file data, which is probably large
  - Deletes are very hard to correctly log
    - Example in a few slides…

Making Journaling Faster

- Journaling adds a lot of write overhead
- OSes typically **batch updates to the journal**
  - Buffer sequential writes in memory, then issue one large write to the log
  - Example: ext3 batches updates for 5 seconds
- **Tradeoff between performance and persistence**
  - Long batch interval = fewer, larger writes to the log
    - Improved performance due to large sequential writes
  - But, if there is a crash, everything in the buffer will be lost

Meta-Data Journaling

- The most expensive part of data journaling is writing the file data twice
  - Meta-data is small (~1 sector), file data is large
- ext3 implements meta-data journaling



Meta-Journaling Timeline

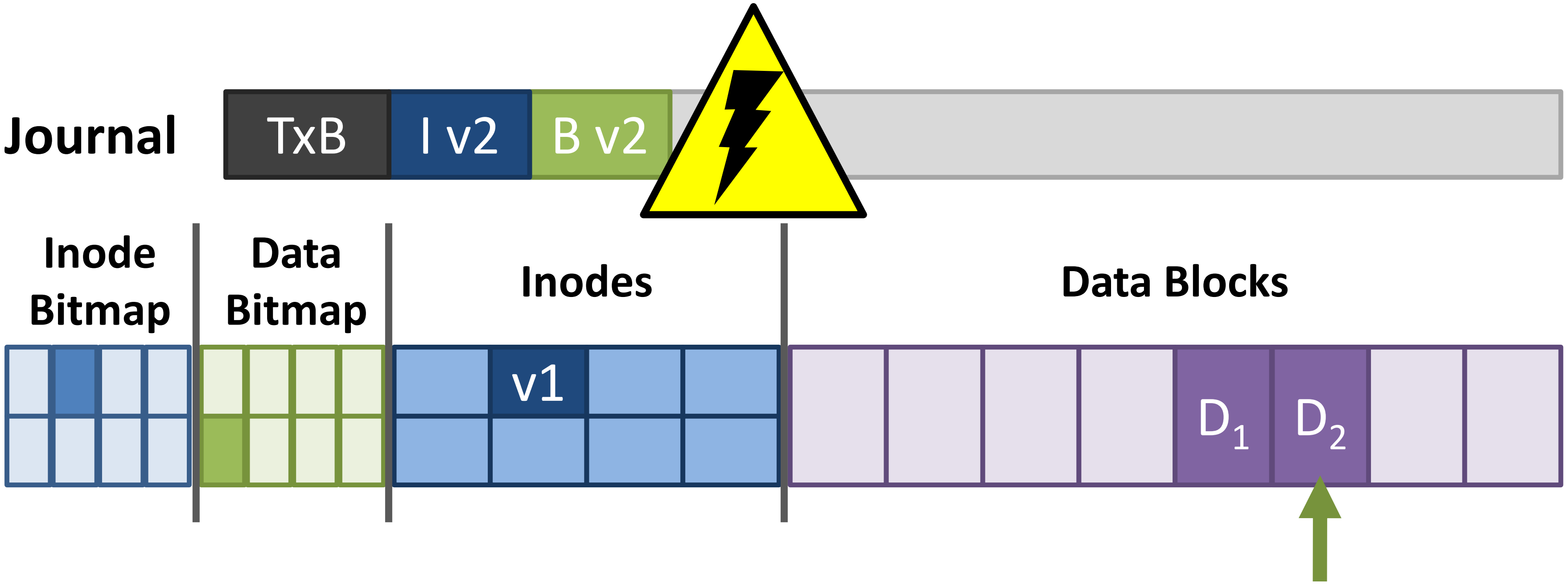
Journal			File System	
TxB	**Meta-data**	**TxE**	**Meta-data**	**Data**
Issue	Issue			Issue
Complete				
				Complete
	Complete			
		Issue		
		Complete		
			Issue	
			Complete	

Time

Transaction can only be committed after the meta-data and data are written

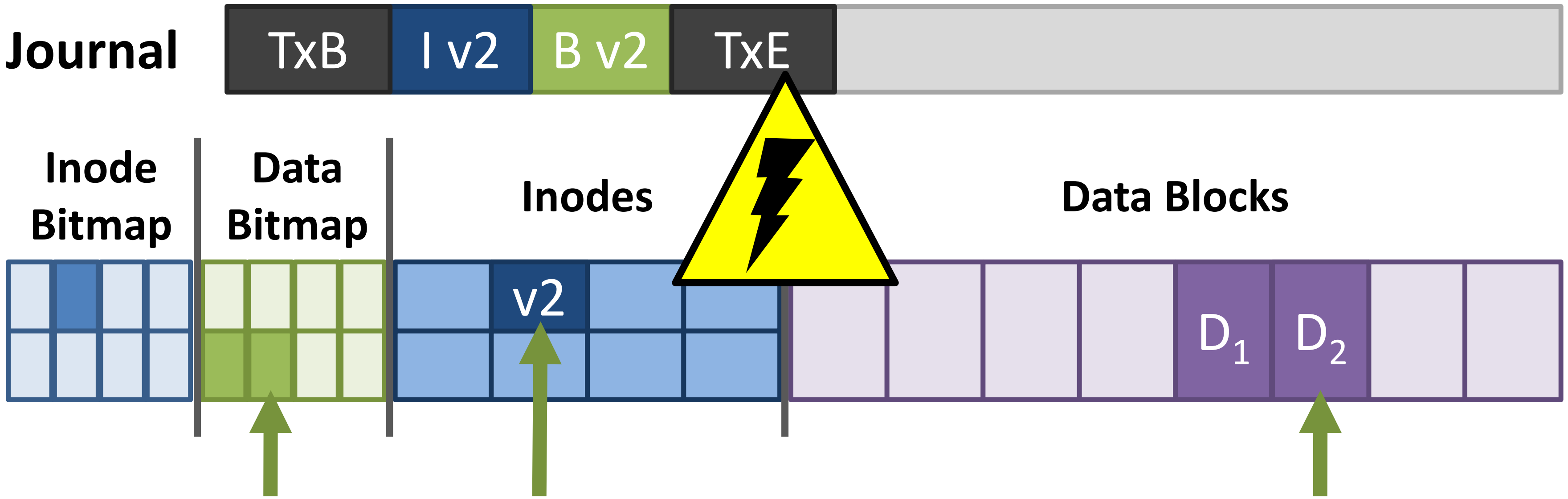
Crash Recovery (1)

- What if the system crashes during logging?
  - **If the transaction is not committed, data is lost**
  - D_2 will eventually be overwritten
  - The file system remains consistent

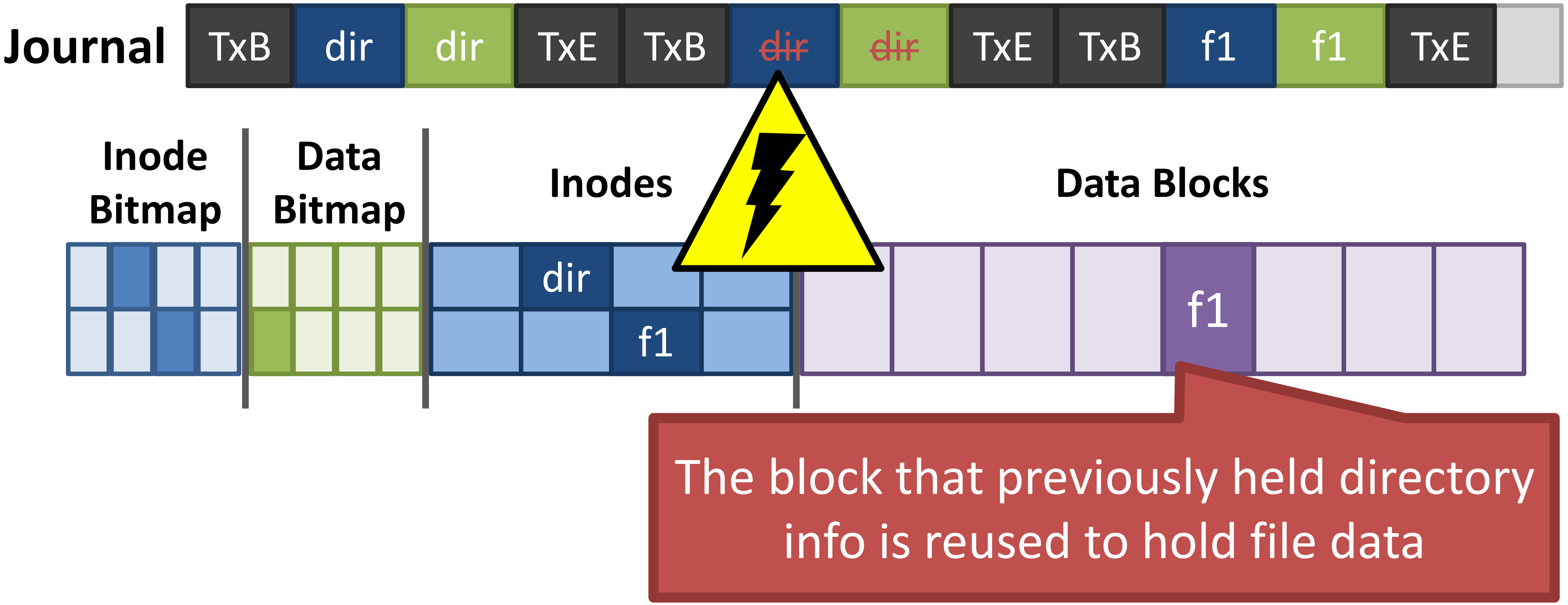


Crash Recovery (2)

- What if the system crashes during the checkpoint?
  - File system may be inconsistent
  - **During reboot, transactions that are committed** but not free are replayed in order
  - Thus, no data is lost and consistency is restored



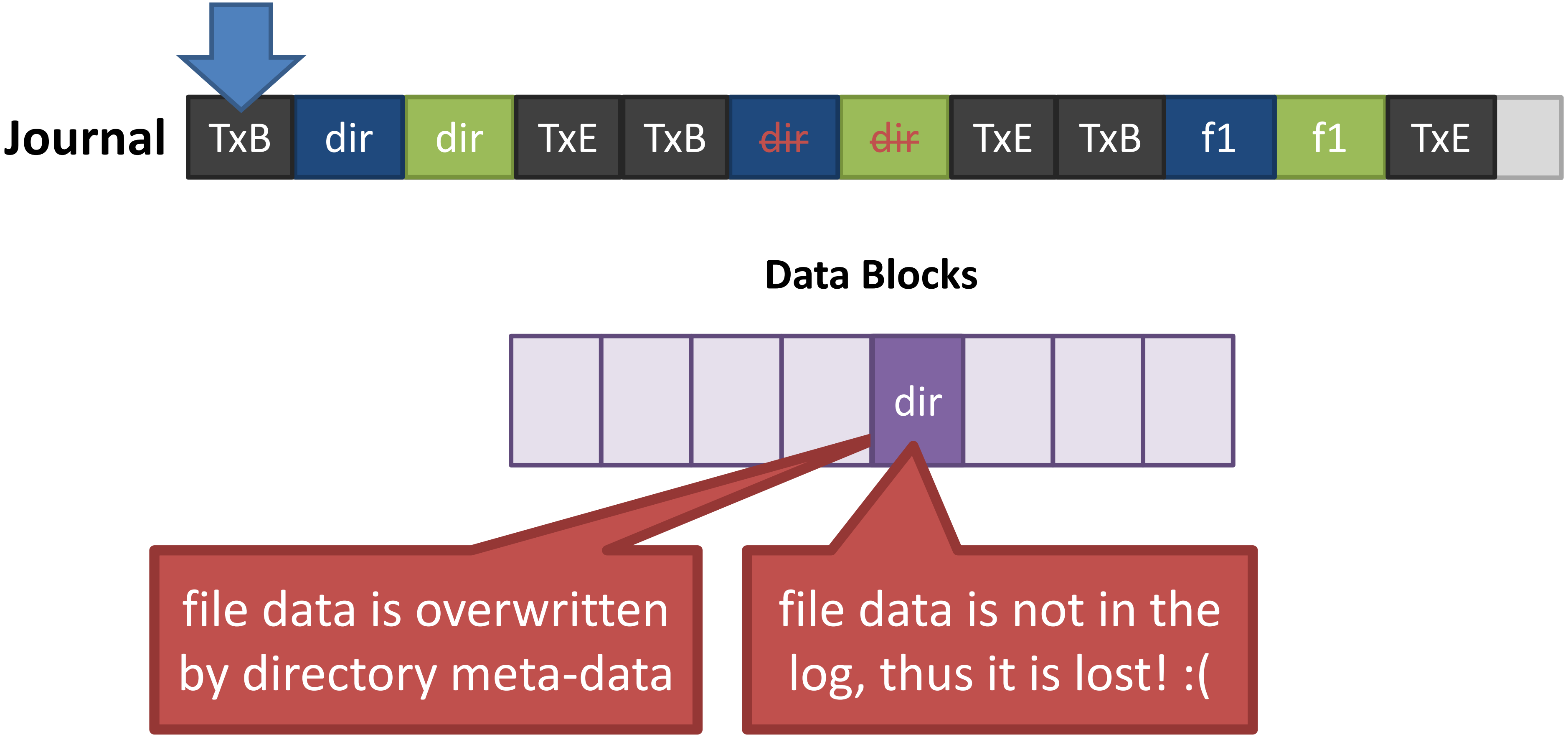
Delete and Block Reuse



1. Create a directory: inode and data are written
2. Delete the directory: inode is removed
3. Create a file: inode and data are written

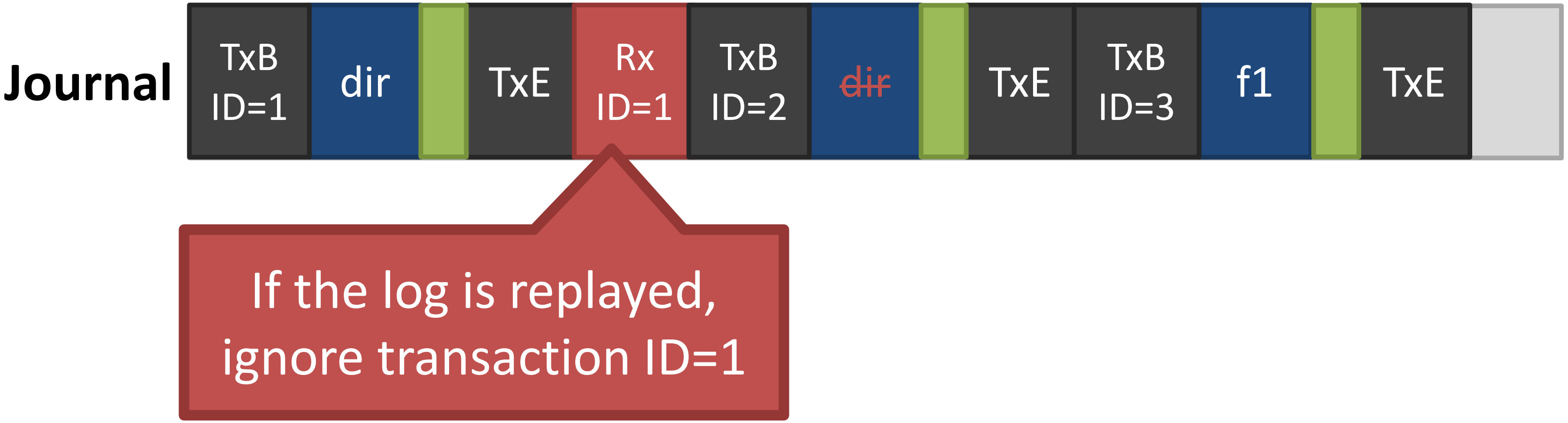
The Trouble With Delete

- What happens when the log is replayed?



Handling Delete

- Strategy 1: don't reuse blocks until the delete is checkpointed and freed
- Strategy 2: add a revoke record to the log
  - **ext3 used revoke records**



Journaling Wrap-Up

- Today, most OSes use journaling file systems
  - ext3/ext4 on Linux
  - NTFS on Windows
- Provides excellent crash recovery with relatively low space and performance overhead
- Next-gen OSes will likely move to file systems with copy-on-write semantics
  - btrfs and zfs on Linux