CS 60038: Advances in Operating Systems Design

File Systems

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

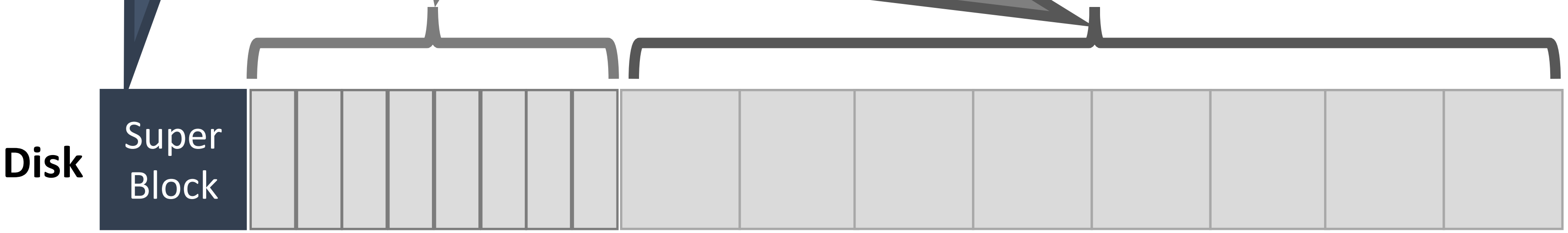
File Allocation Tables (FAT)

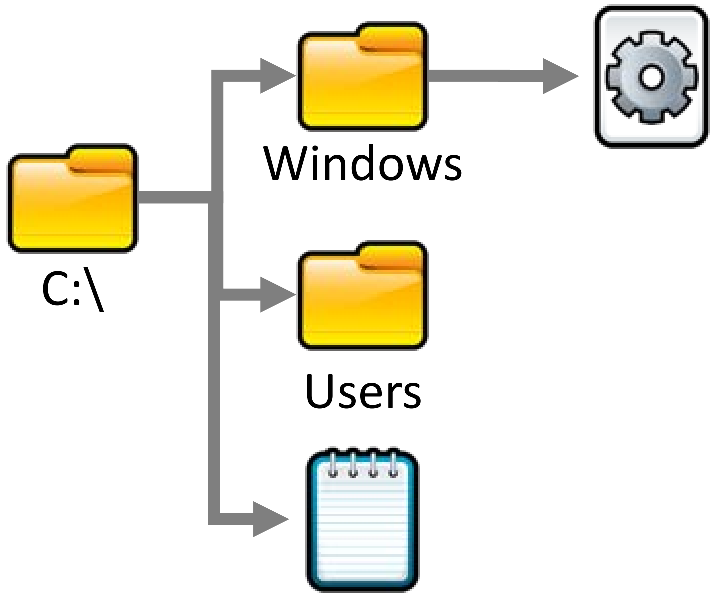
- Simple file system popularized by MS-DOS
  - First introduced in 1977
  - Most devices today use the FAT32 spec from 1996
  - FAT12, FAT16, VFAT, FAT32, etc.
- Still quite popular today
  - Default format for USB sticks and memory cards

- 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

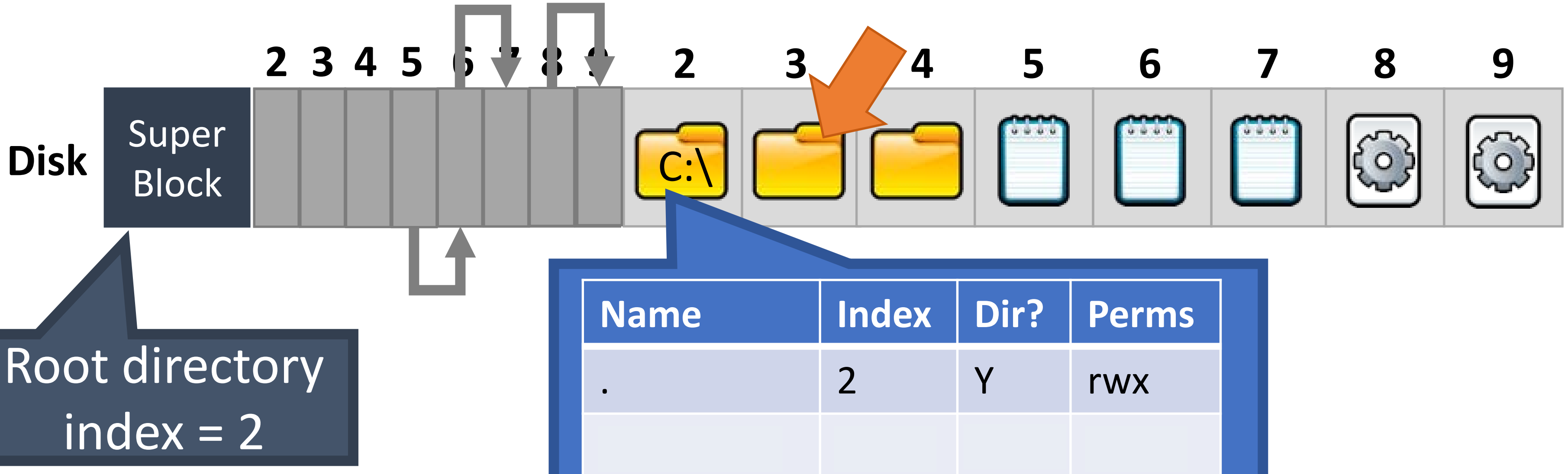
- File allocation table (FAT)
- Marks which blocks are free or in-use
- **Linked-list structure** to manage large files

- 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



Name	Index	Dir?	Perms
.	2	Y	rwx

ext2 inodes

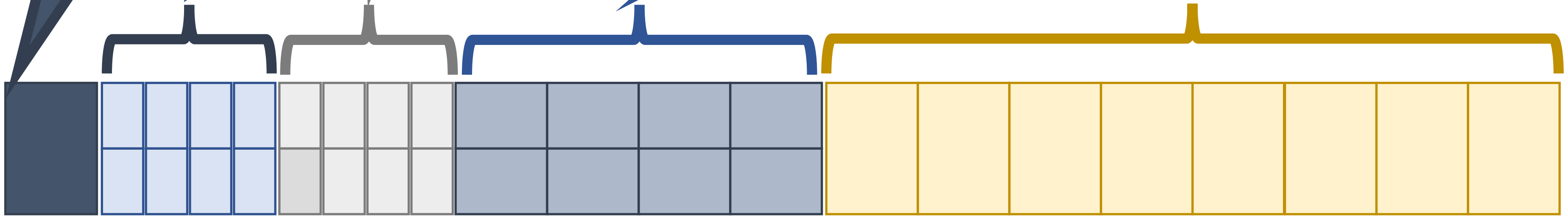
- Super block, storing:
  - Size and location of bitmaps
  - Number and location of inodes
  - Number and location of data blocks
  - Index of root inodes

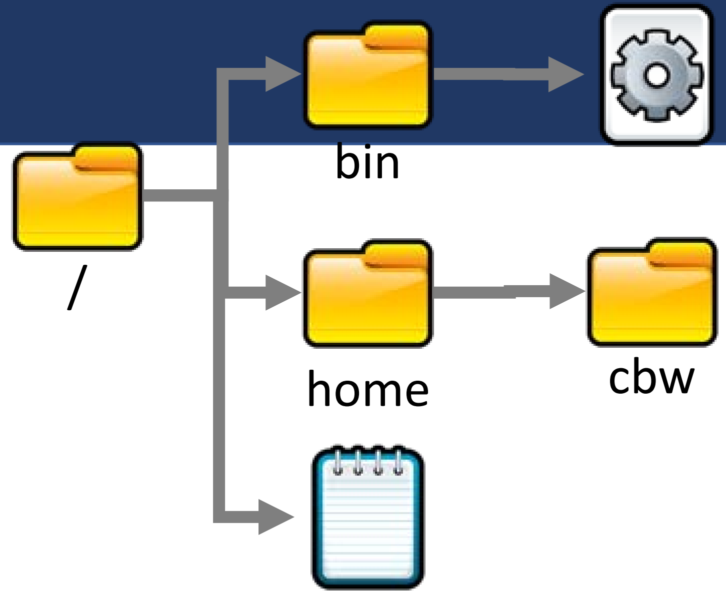
Bitmap of free & used data blocks

- Table of inodes
- Each inode is a file/directory
- Includes meta-data and lists of associated data blocks

Bitmap of free & used inodes

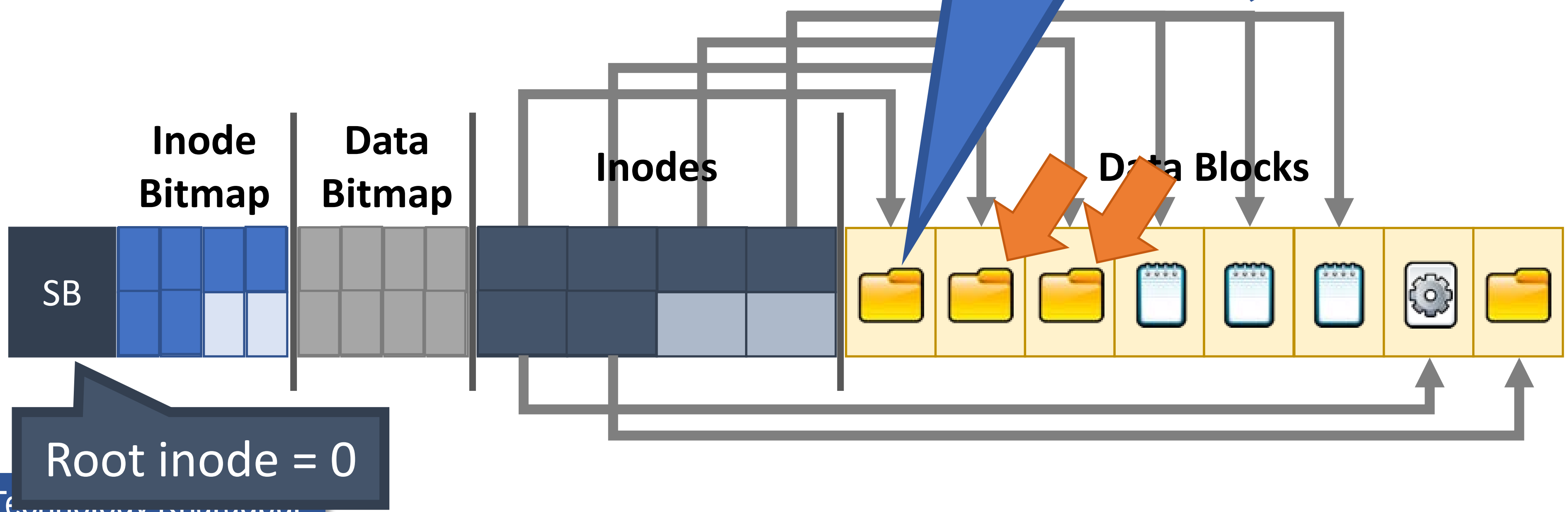
Data blocks (4KB each)





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

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

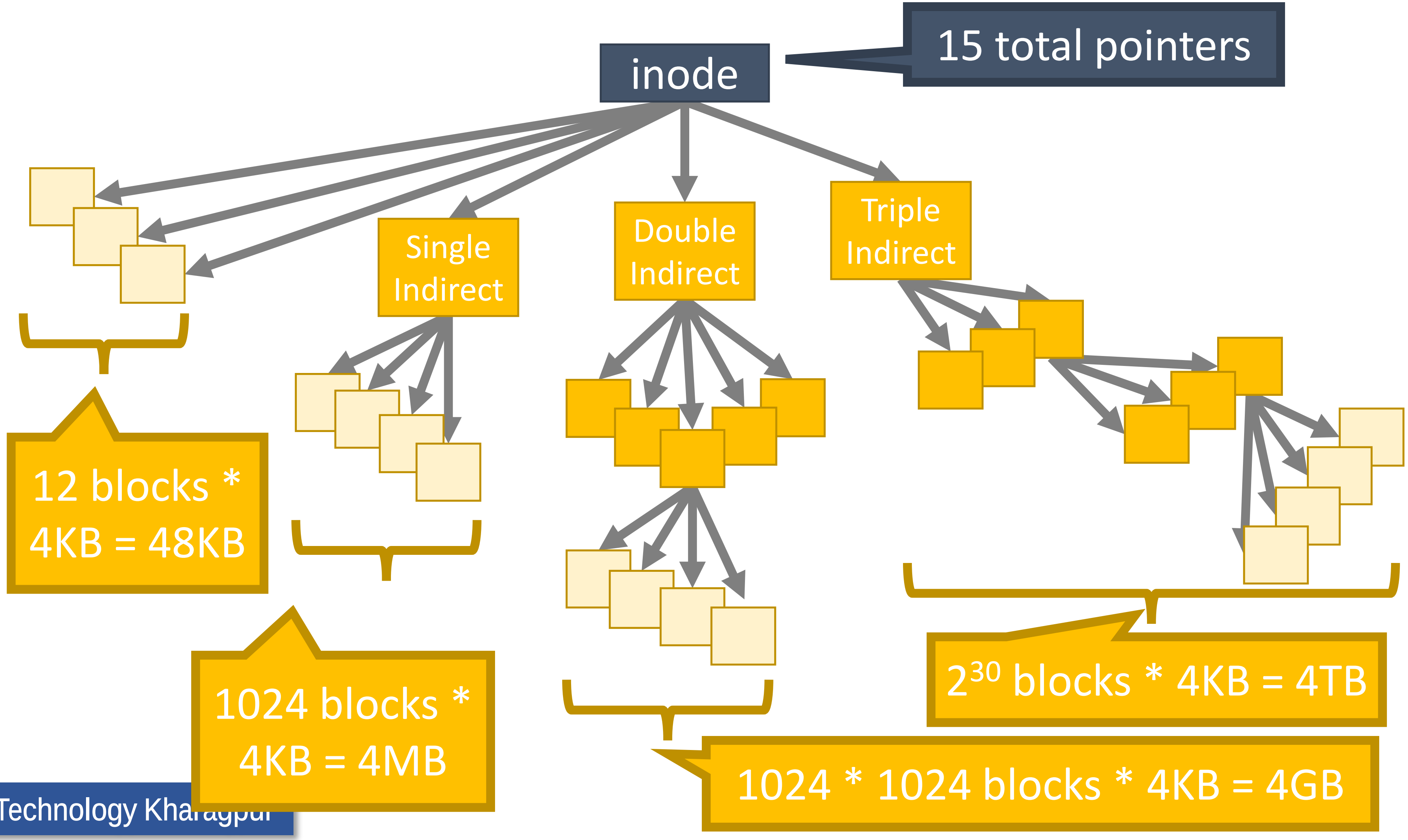


ext2 inodes

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

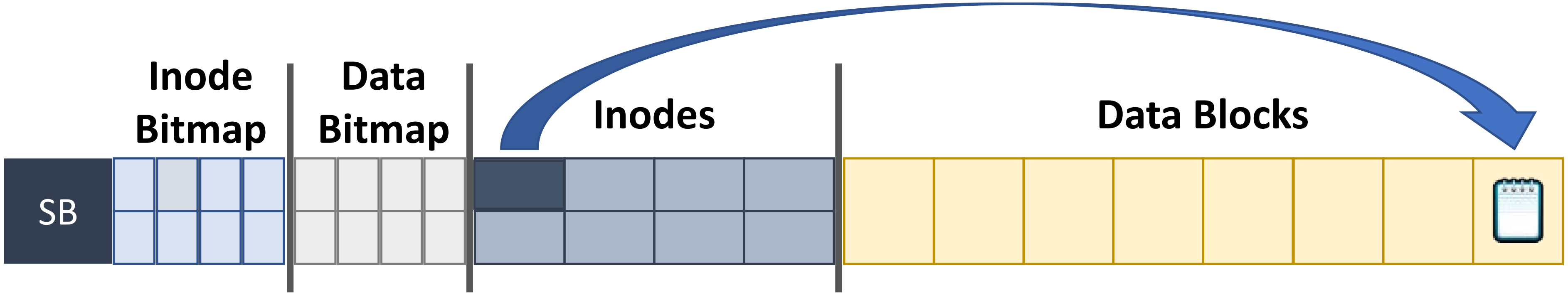
	Bitmaps		inodes			Data Blocks		
	data	**inode**	**root**	**tmp**	**file**	**root**	**tmp**	**file[0]**
open("/tmp/file")			read					
						read		
				read				
							read	
		read						
		write						
					write			
							write	
				write				
write()					read			
	read							
	write							
								write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

Time

Update the modified time of the directory

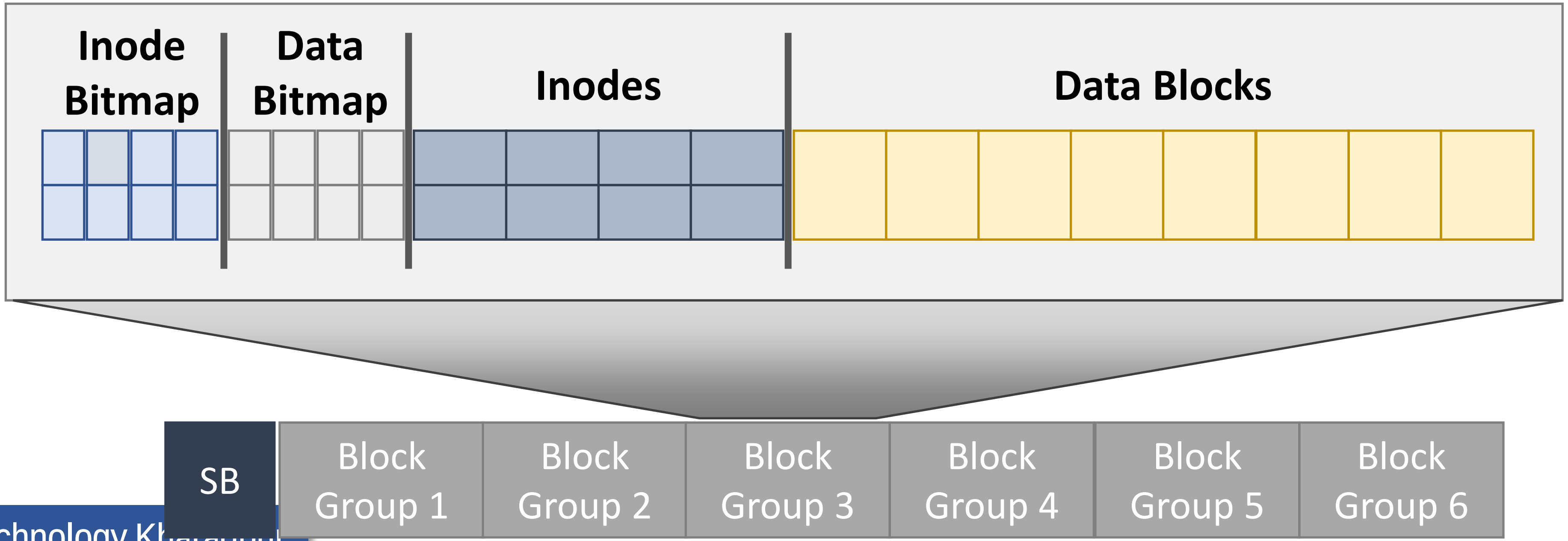
ext: The Good and the Bad

- 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!



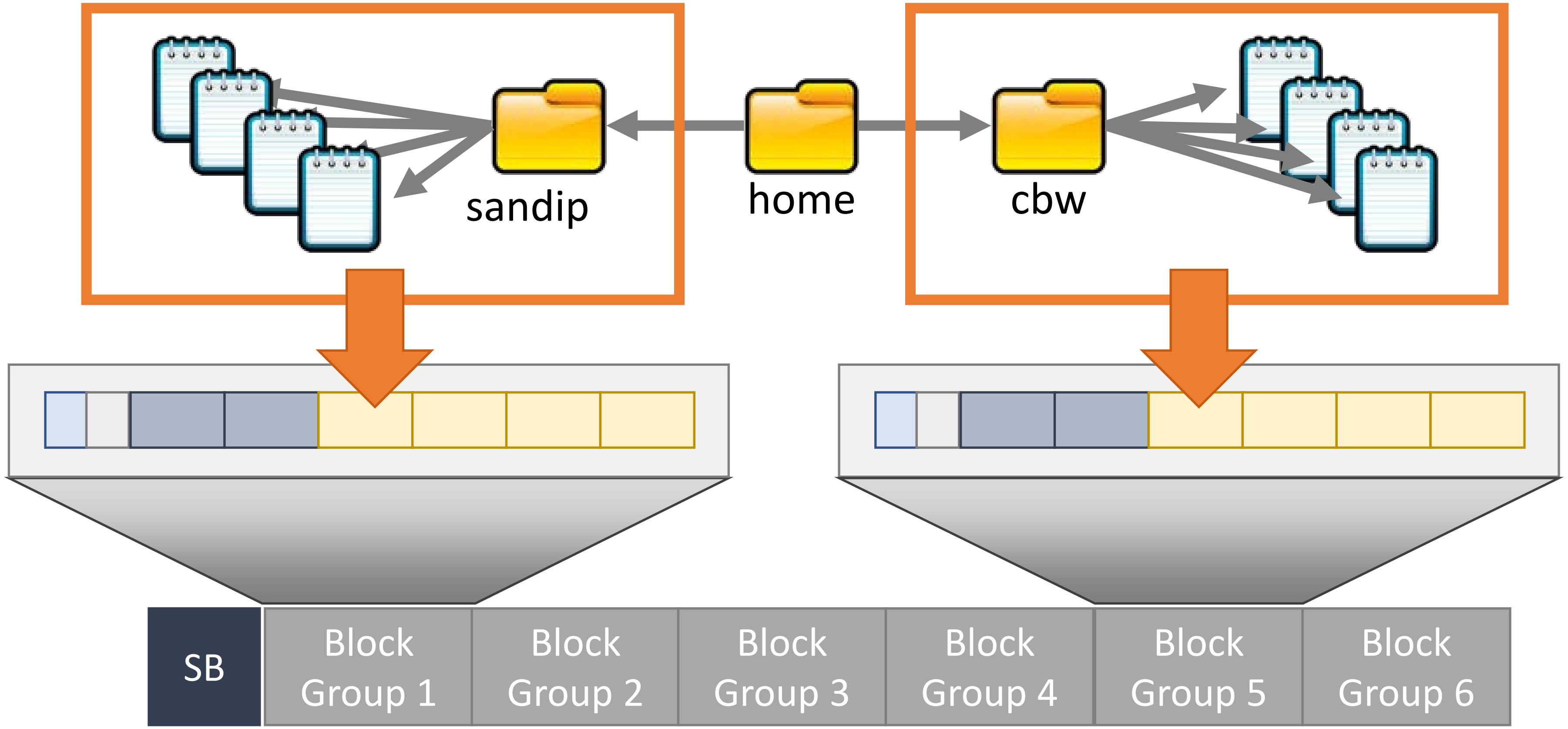
Block Groups

- In ext, there is a single set of key data structures
  - One data bitmap, one inode bitmap
  - One inode table, one array of data blocks
- In ext2, each block group contains its own key data structures



Allocation Policy

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

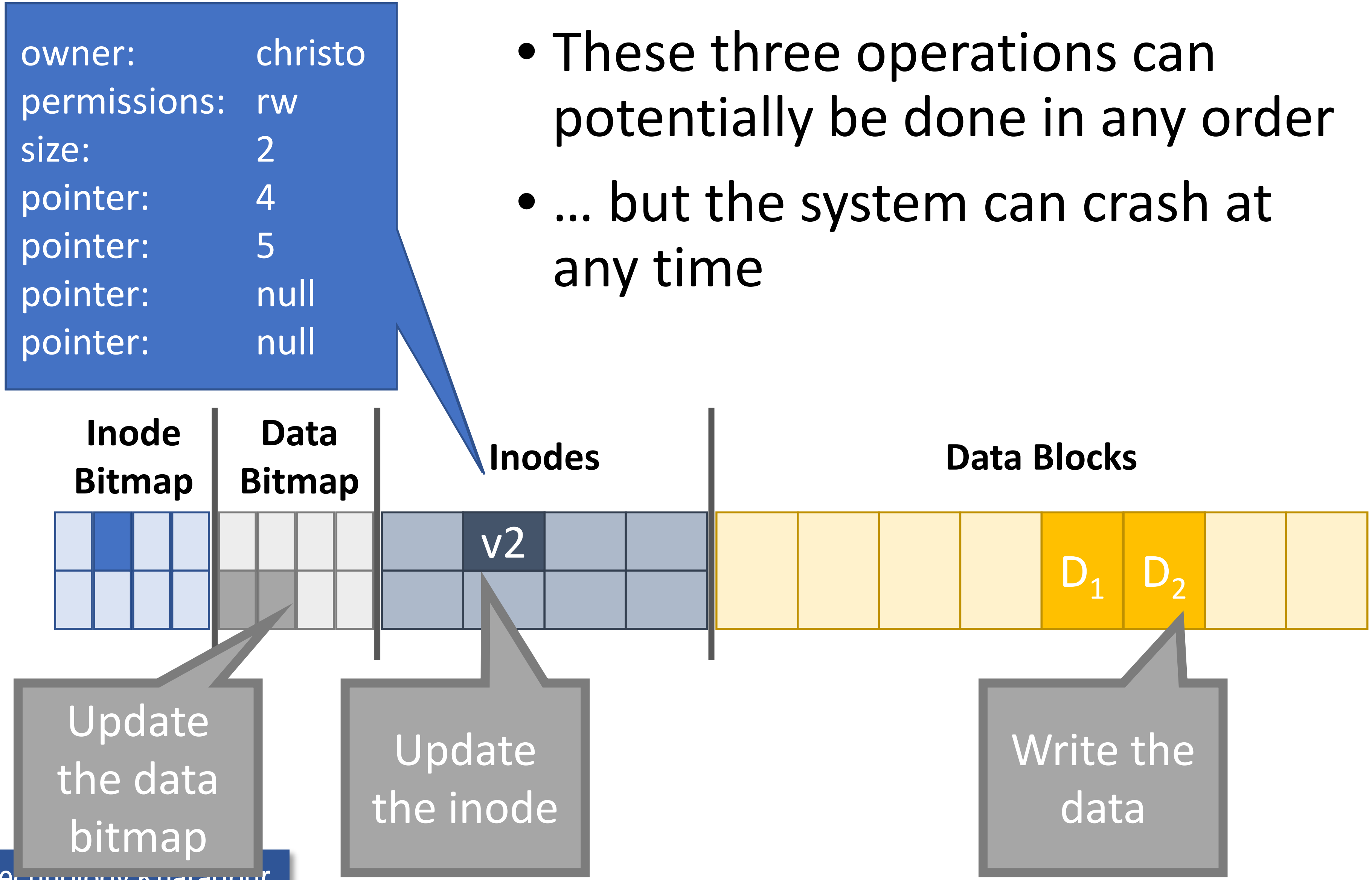


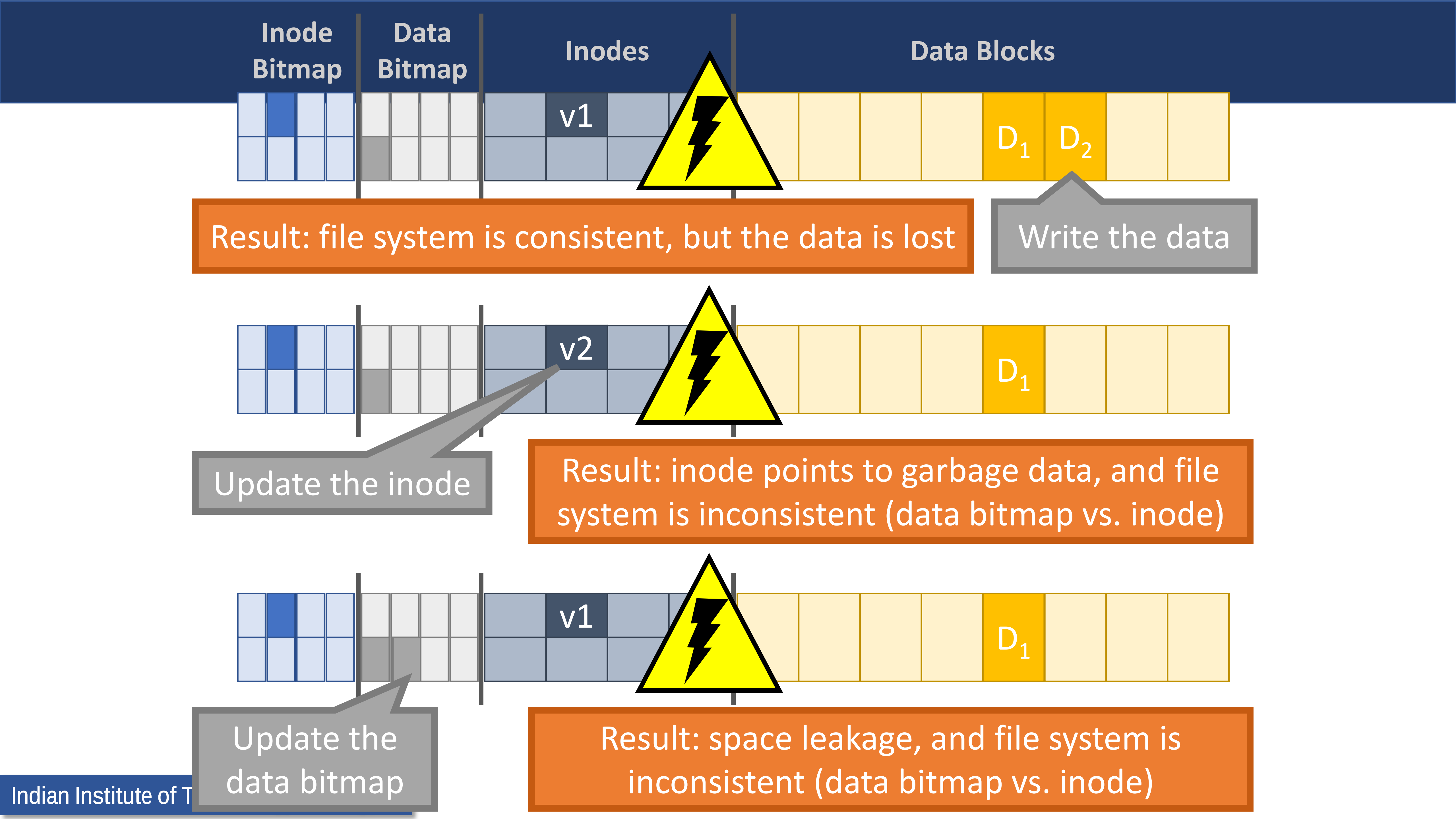
Maintaining Consistency

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

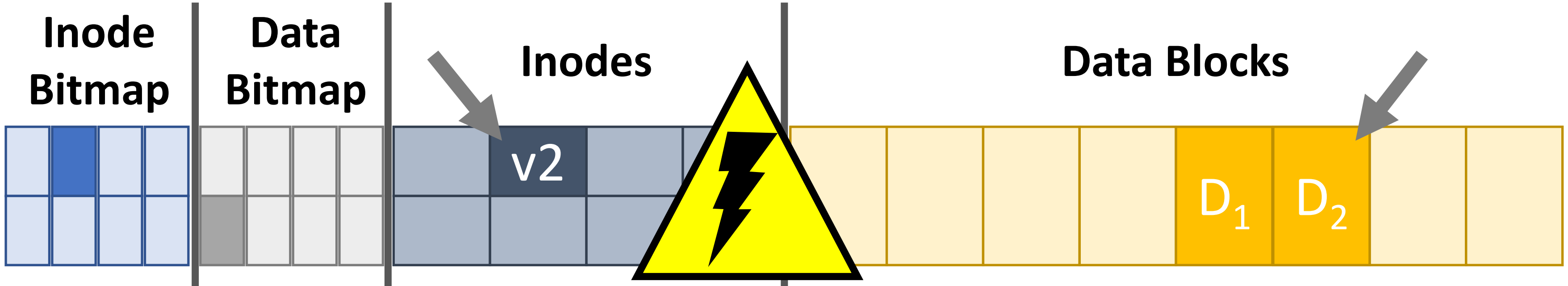
File Append Example

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

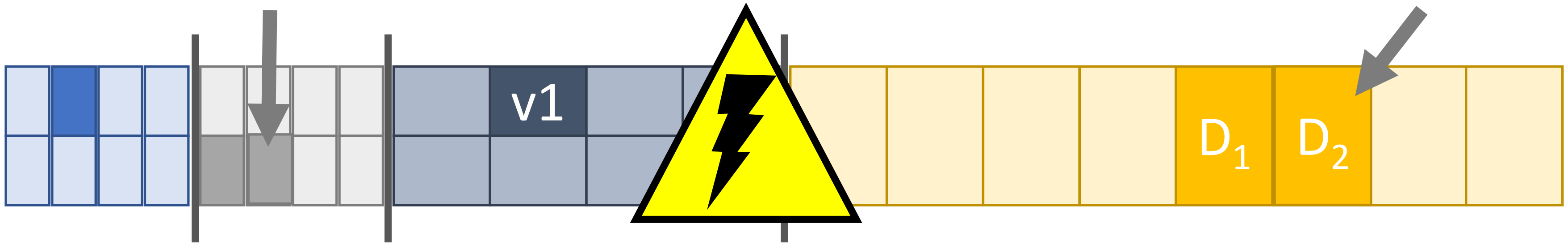




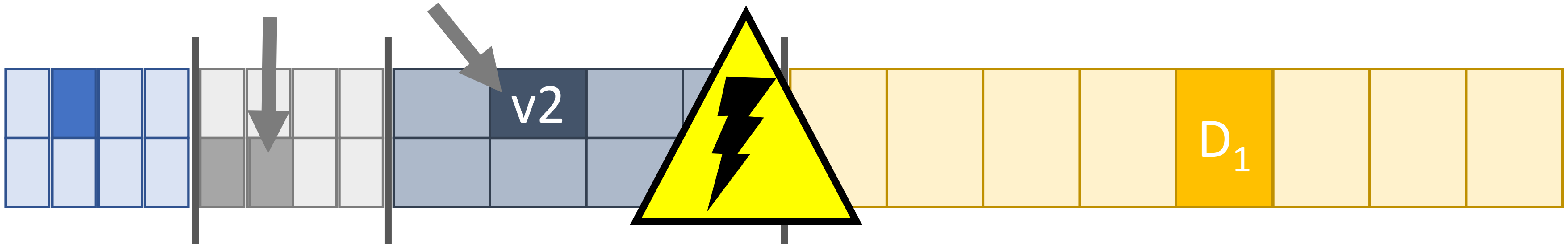
Inode Bitmap
Data Bitmap
Inodes
Data Blocks
v1
D1
D2
Result: file system is consistent, but the data is lost
Write the data
v2
D1
Update the inode
Result: inode points to garbage data, and file system is inconsistent (data bitmap vs. inode)
v1
D1
Update the data bitmap
Result: space leakage, and file 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?
  1. 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*)
  2. 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

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

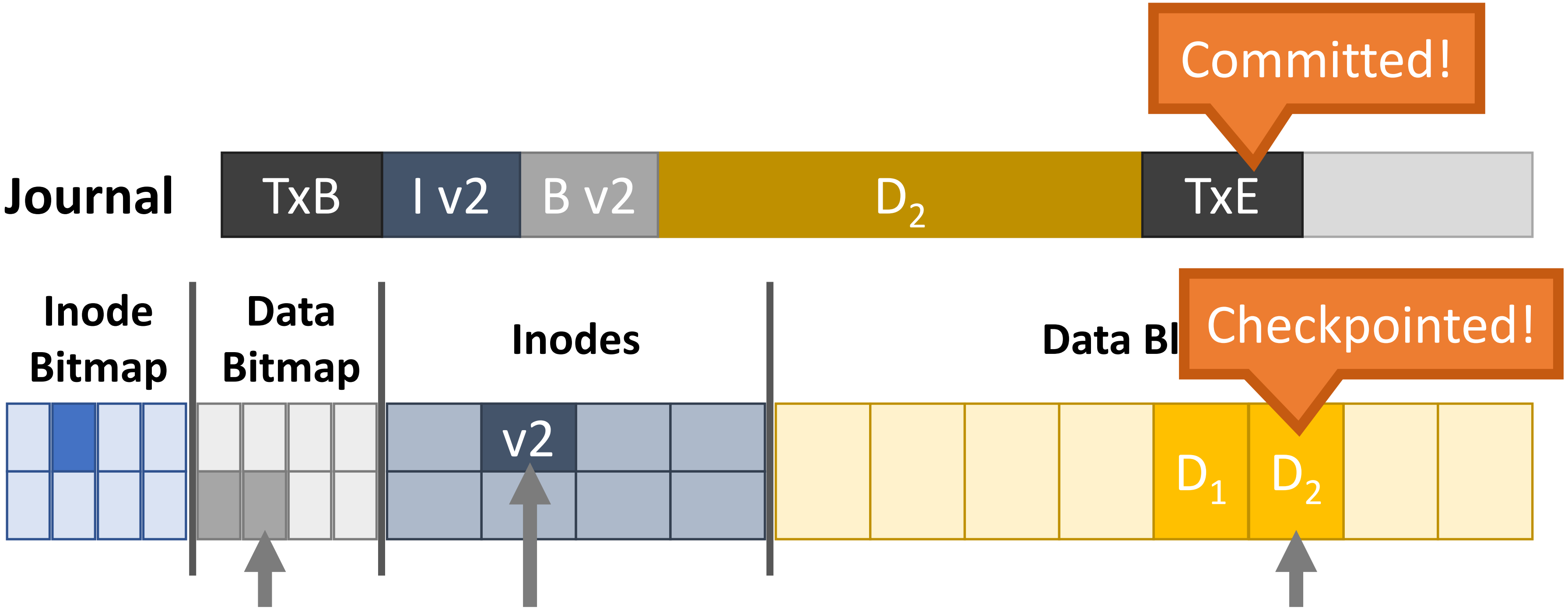
Data Journaling Example

- 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

- We say 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

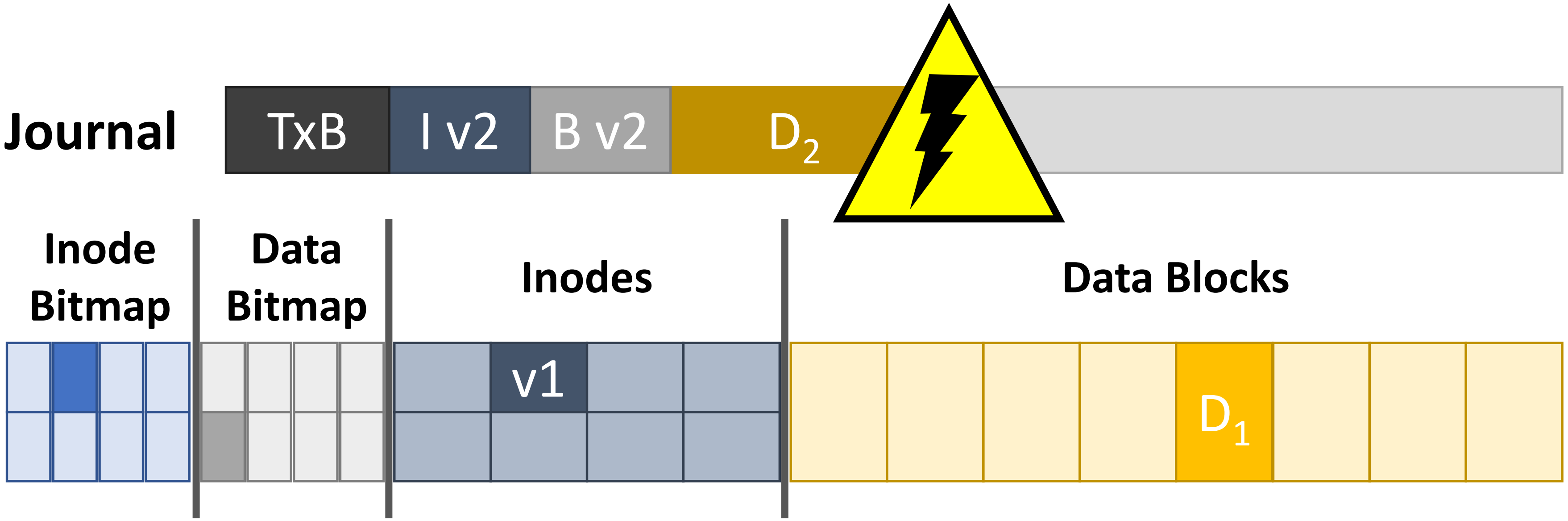
- Journals are typically implemented as a circular buffer
  - Journal is **append-only**
- OS maintains pointers to the front and back of the transactions in the buffer
  - As transactions are freed, the back is moved up
- Thus, the contents of the journal are never deleted, they are just overwritten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

Data Journaling Timeline

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

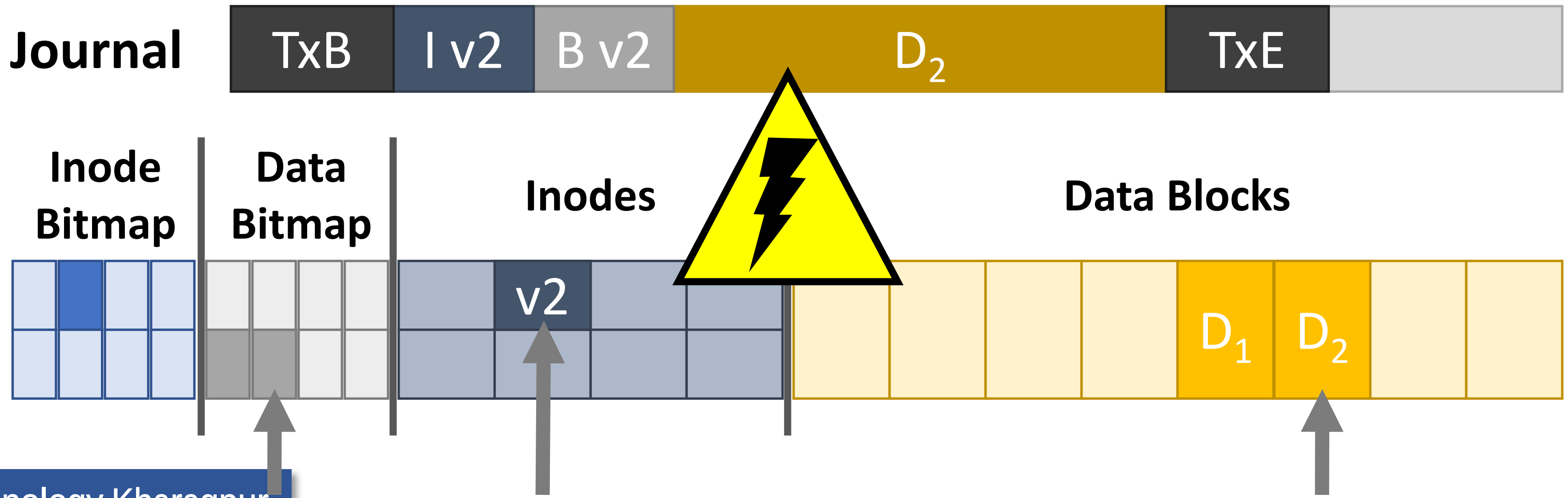
Crash Recovery (1)

- 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

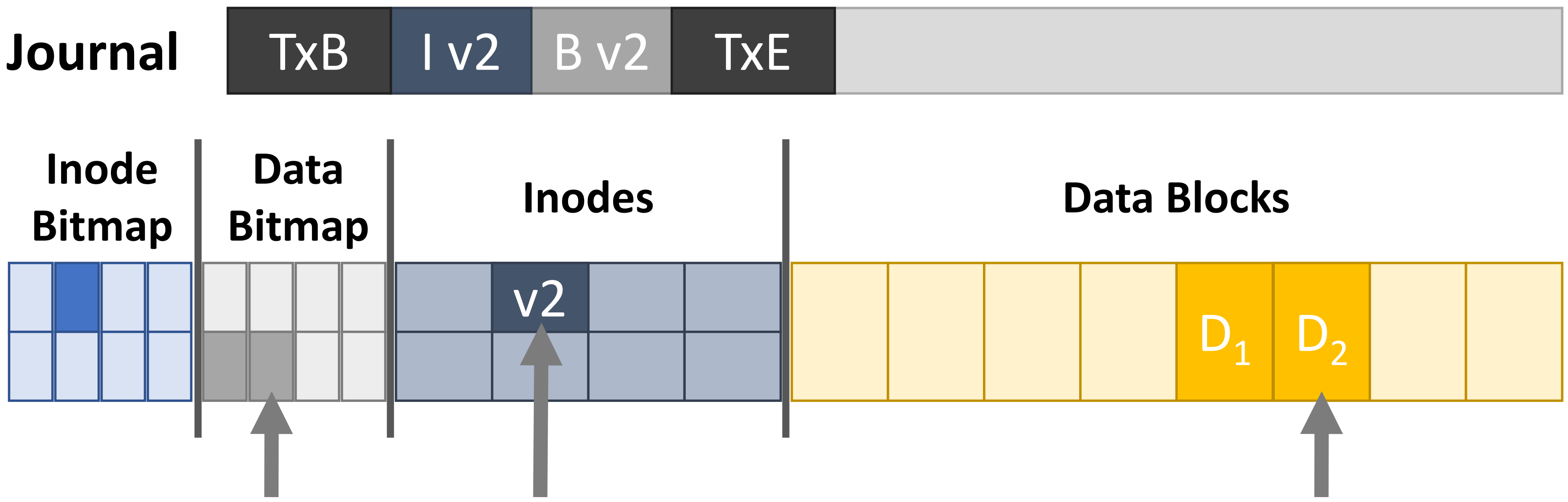
- Advantages of journaling
  - Robust, fast file system recovery
    - No need to scan the entire journal or file system
  - Relatively straightforward to implement
- Disadvantages of journaling
  - Write traffic to the disk is doubled
    - Especially the file data, which is probably large

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

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