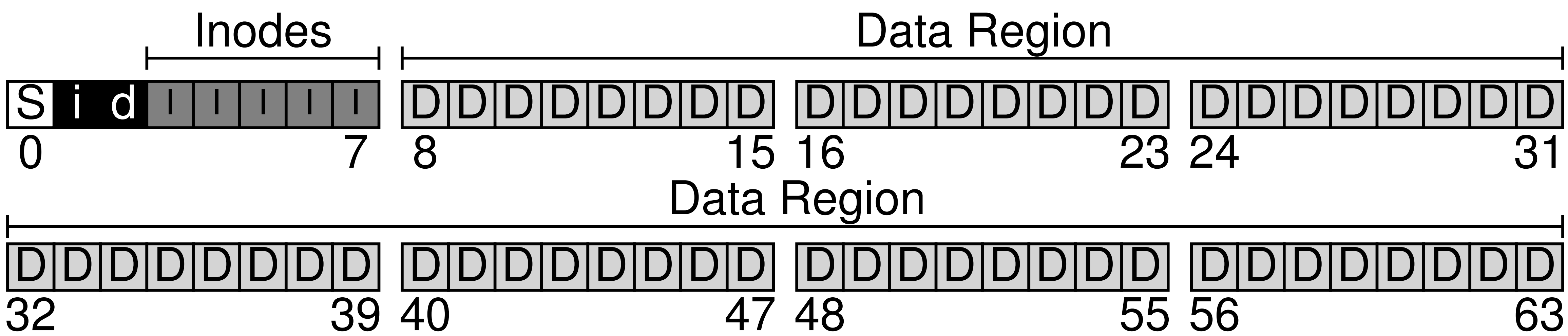
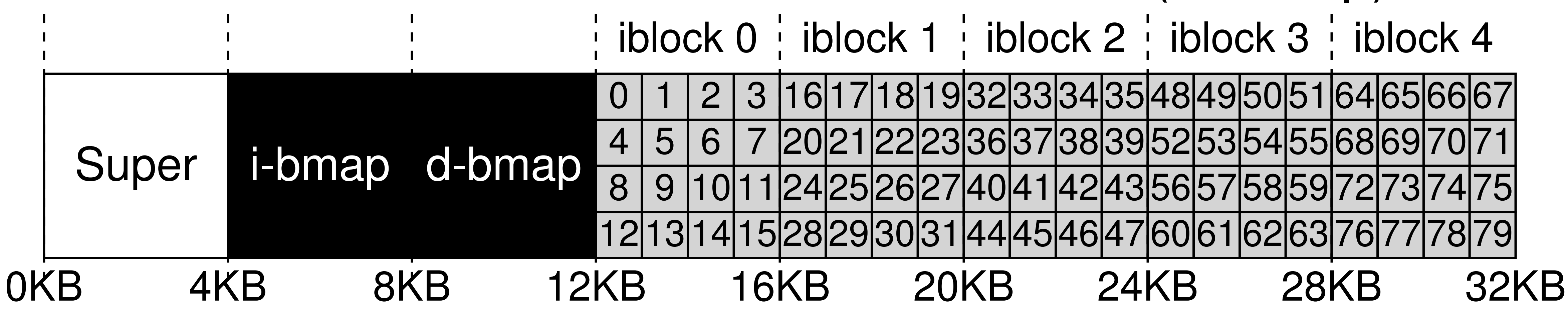
Journaling File Systems

Operating Systems
Baochun Li
University of Toronto

Review: File System Implementation



The Inode Table (Closeup)



File 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					

File Creation Timeline

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					
write()					read					
	read									
	write									
									write	
					write					
write()					read					
	read									
	write									
										write
					write					

The Crash Consistency Problem

What happens if power is lost when updating on-disk data structures?

Crash consistency

Imagine that you need to update two on-disk data structures, A and B, to complete an operation

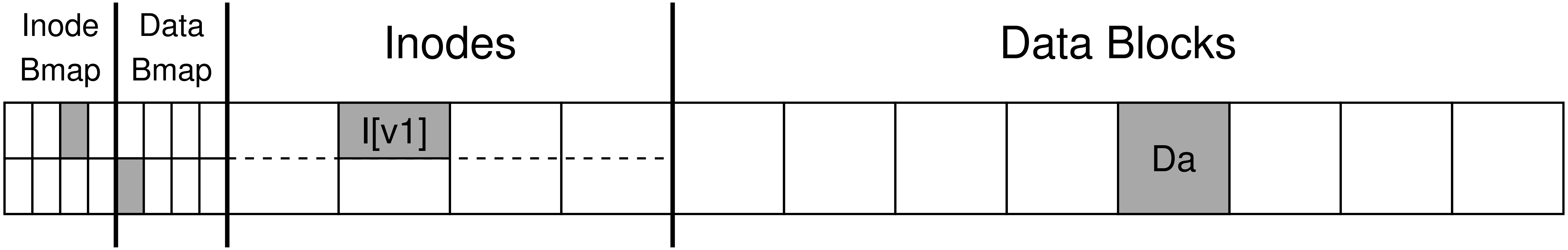
One of these will reach the disk first

If the system crashes after one write completes, the on-disk structure will be left in an inconsistent state

An example

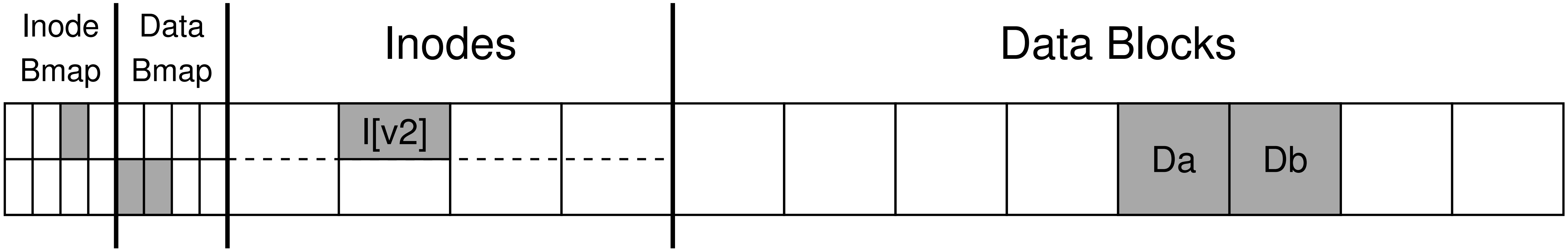
Append 4KB to the end of a file

Open the file, seek to the end, issue a single 4KB write




```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

Append needs three operations




```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

Crash scenarios: one write succeeded

Just the data block is written to the disk

Not a problem, the file system is still consistent

Just the updated inode is written to the disk

If we trust the inode, we will read garbage data from the data block

We also have file-system inconsistency, since the on-disk bitmap is saying that the block is not used, but the inode disagrees

Just the updated bitmap is written to the disk

File-system inconsistency — "space leak" in the file system

Crash scenarios: two writes succeeded

The inode and bitmap are written, not the data block

Consistent, but garbage data

The inode and the data block are written, not the bitmap

inode pointing to the correct data, but the bitmap is not consistent

The bitmap and the data block are written, not the inode

Inconsistency between the bitmap and the inode

No idea which file the data block belongs to

Objective of crash consistency

Move the file system from one consistent state to another, atomically

Solution #1: The File System Checker

Idea: let inconsistencies happen, and fix them later when rebooting

In UNIX: fsck

- Scans the superblock: sanity checks
- Scans the inodes to build a correct version of the data bitmap, and check if it is consistent with the one in the file system — trust the inodes
- Check the reference count in each inode, and see if it is consistent with the directory structure — if a file is not in any directory, add to **lost+found**

Major problem: too slow

Solution #2: Journaling

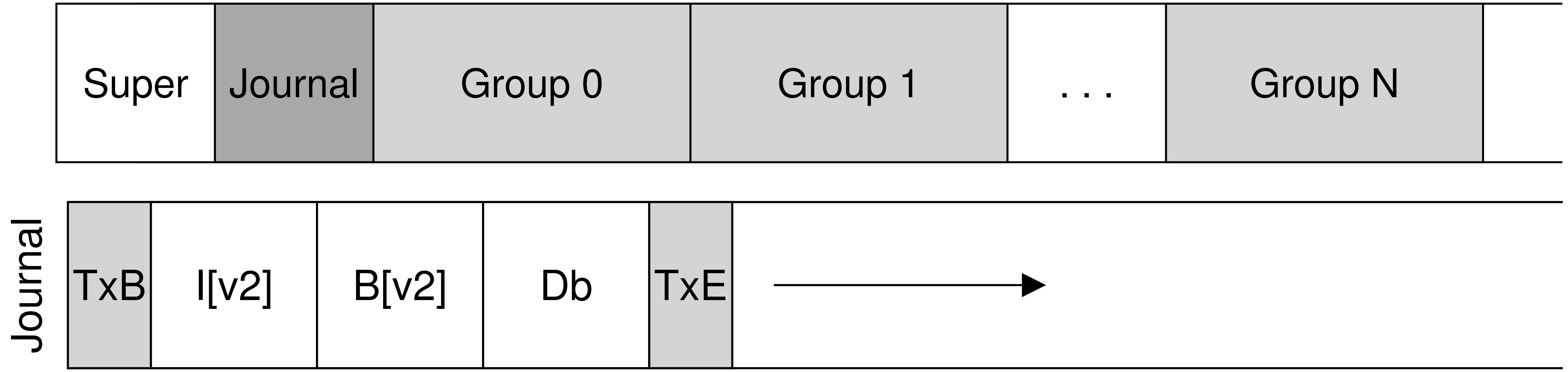
Idea: write-ahead logging, similar to databases

Famous examples: Linux ext3, ext4, ReiserFS, IBM's JFS, SGI's XFS (ported to Linux), Windows NTFS

Before overwriting the structures (bitmap, inode, data block), first write down a little note somewhere else in a well-known location

Write what you are about to do in a log

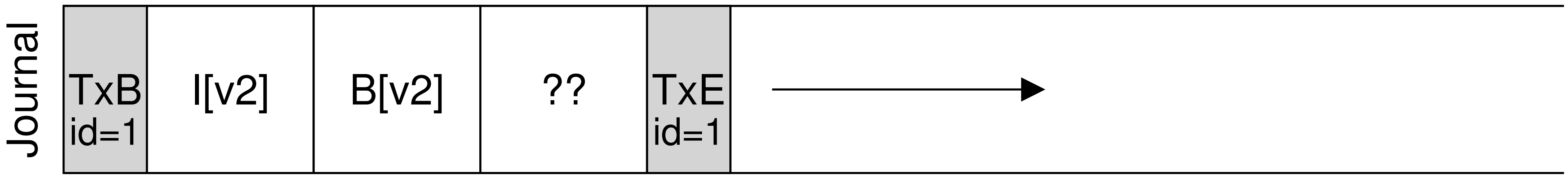
Example: Linux ext3



What if a crash happens when journaling?

Writes may occur out of order due to the extensive use of caches in the disk itself

If TxB, I[v2], B[v2], and TxE are written, but not the data block:

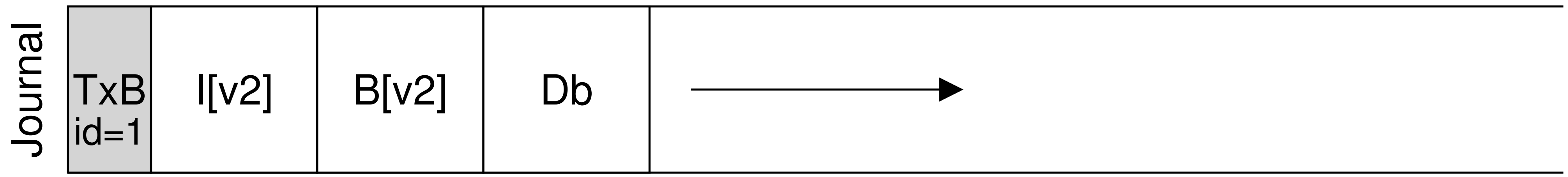


When replaying the journal, garbage data will be written

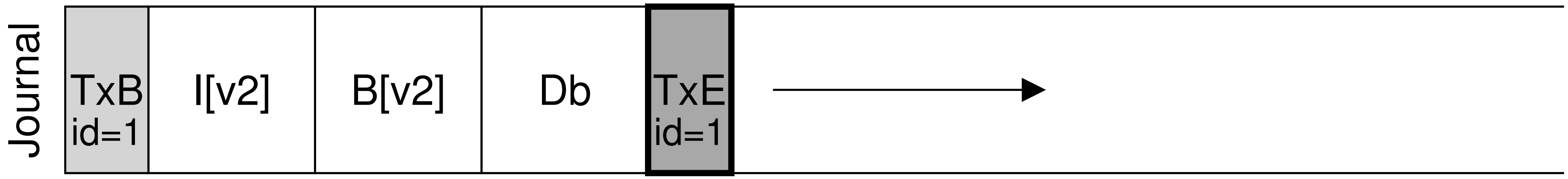
Bad for a data block, but if it is the superblock that is written, the file system may not be mountable!

Fixing the problem: Idea #1

First write all blocks except the TxE block to the journal:



When all these writes complete, write the TxE block (using the "write barrier" mechanism supported by the disk):



To make sure TxE is written atomically, make it 512 bytes.

Fixing the problem: Idea #2

When writing a transaction to a journal, include a checksum in the TxB and TxE blocks

- If there is a mismatch between the stored checksum and the computed one, a crash has occurred
- ACM SOSP 2005 paper, eventually used in Linux ext4

Recovery

If a crash happens before the transaction is logged, do nothing and skip the pending update

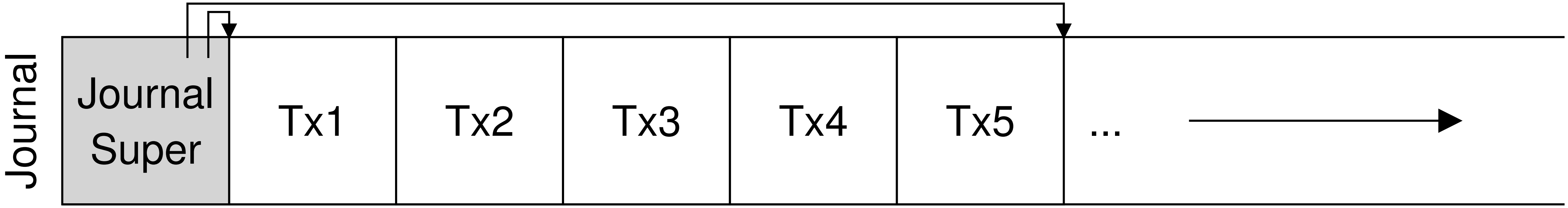
For those transactions that committed (TxE block written) successfully

- Redo the log by replaying all committed transactions

Performance and conserving space for logs

To improve performance, buffer all the updates in the memory cache as a global transaction, and avoid excessive writes to the disk

Some time after the on-disk structures are updated (called “checkpoint”), mark the transaction free in the journal by updating a journaling superblock



Data journaling: a timeline

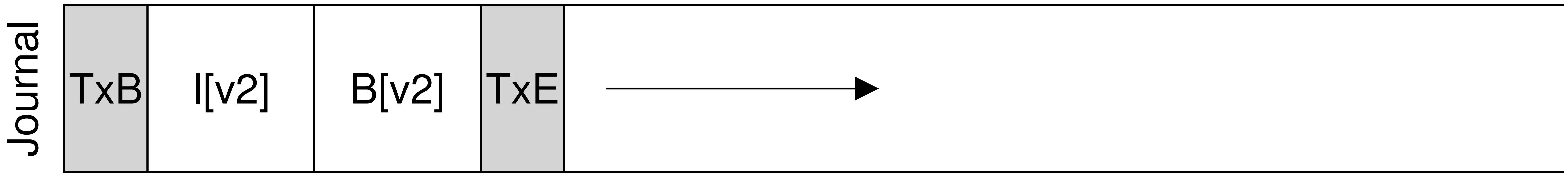
Journal: TxB	Journal: Contents (metadata)	Journal: Contents (data)	Journal: TxE	File System: Metadata	File System: Data
issue	**issue**	**issue**			
complete					
	complete				
		complete			
			issue		
			complete		
				issue	**issue**
					complete
				complete	

But we are still writing each data block to the disk **twice!**

Metadata journaling

Writing each data block to the disk twice is a heavy cost to pay for rare crashes!

Idea: the user data is not written to the journal at all



If we wish to make sure that the inode will not point to garbage data blocks, simply write data blocks first before writing the metadata to the journal

Both Windows NTFS and SGI's XFS (ported to Linux) use metadata journaling — my two favourites!

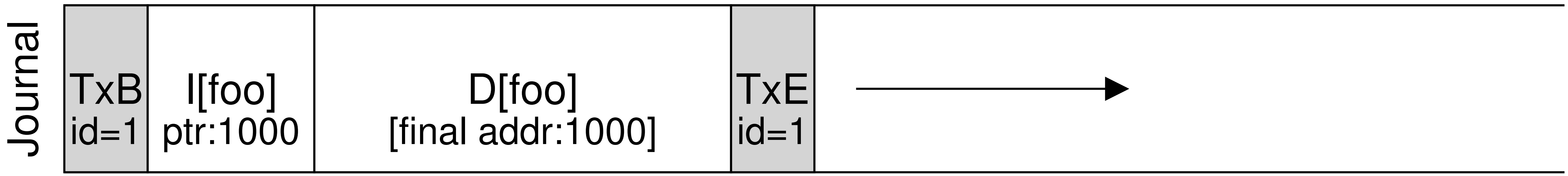
Metadata journaling: a timeline

Journal			File System	
TxB	Contents (metadata)	TxE	Metadata	Data
issue	**issue**			**issue**
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

Tricky case: block reuse with deletion

The user adds an entry to a directory, `foo`, by creating a file

The content of this directory (say, data block 1000) will be written to the log (metadata journaling)



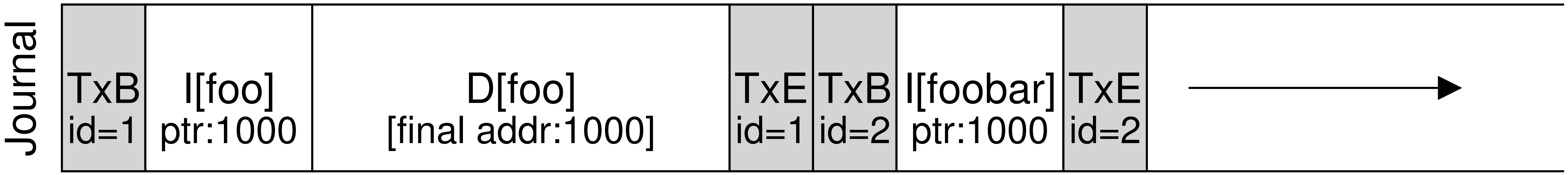
Tricky case: block reuse with deletion

The user then deletes everything in the directory as well as the directory itself, freeing up the data block 1000 for reuse

Then the user creates a new file, `foobar`, which reuses block 1000

`foobar`'s inode and data are committed to the disk, but only its inode is committed to the journal

During recovery, the replay overwrites `foobar`'s data with the old directory!



Potential solutions

Idea #1: Never reuse blocks until the deletion of these blocks is checkpointed out of the journal

Idea #2: (Linux ext3) add a new type of record to the journal, known as a revoke record

- Deleting a directory will cause a revoke record to be written to the journal
- When replaying the journal, the system first scans for such revoke records — any such revoked data is never replayed

Alternative approach to journaling: Copy-on-Write

Used by Sun ZFS (Jeff Bonwick, who also designed slab allocation) and Apple APFS (used since MacOS 10.11 High Sierra, iOS 11)

Idea: never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk

After a number of updates are completed, copy-on-write file systems flip the root structure of the file system to include pointers to the newly updated structures

What we've covered so far

Three Easy Pieces: Chapter 42 (Crash Consistency)