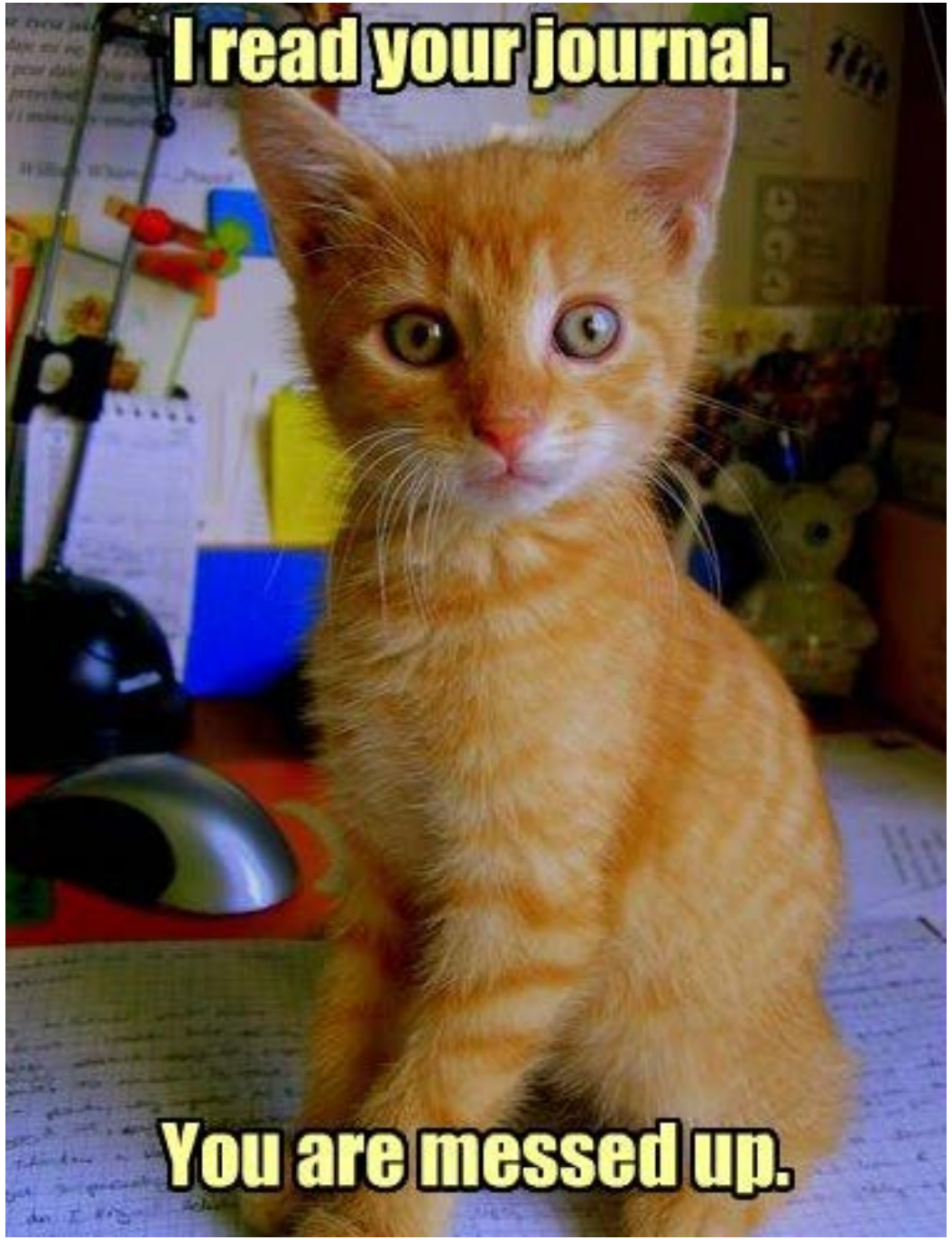
Journaling

CS 161: Lecture 14
4/4/17



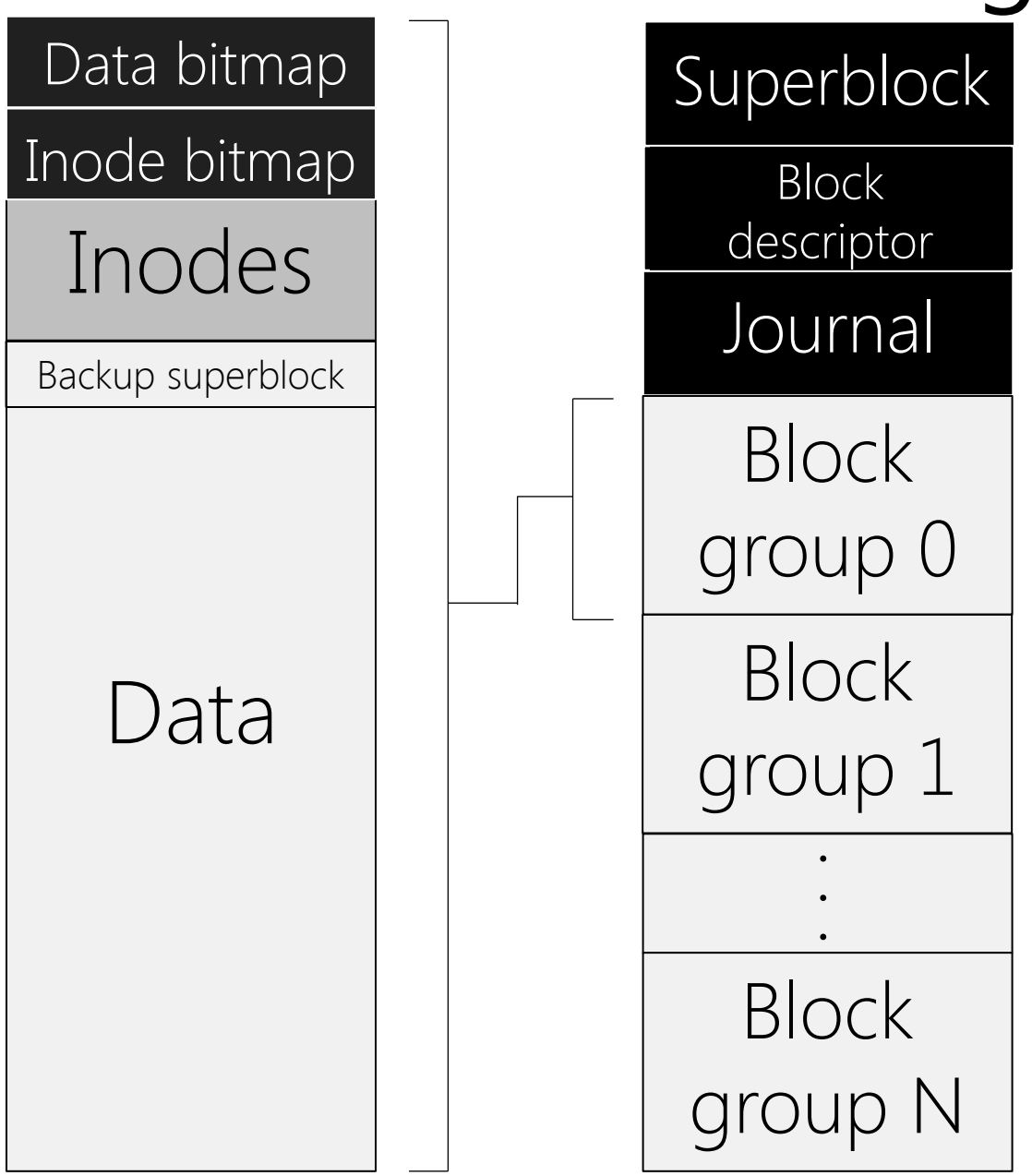
In The Last Episode . . .

- FFS uses fsck to ensure that the file system is usable after a crash
  - fsck makes a series of passes through the file system to ensure that metadata is consistent
  - fsck may result in lost data, but metadata will always be consistent
- fsck works, but has several unattractive features
  - fsck requires detailed knowledge of file system, making fsck difficult to write and maintain
  - fsck is extremely slow, because it requires multiple traversals through the entire file system
- Ideally, recovery time would be proportional to the number of recent writes that may or may not have made it to disk

File System Transactions

- A transaction is a sequence of operations that should be treated as a logical whole
- In the database world, transactions are described using A.C.I.D.
  - Atomic: Either all of the operations in the transaction succeed, or none of them do
  - Consistent: Each transaction moves the system from one consistent state to another consistent state
  - Isolation: Each transaction behaves as if it's the only one that is executing in the system
  - Durability: Once the system commits a transaction, that transaction must persist in the system, even if the system crashes or loses power
- Transactions provide an elegant abstraction for file systems to reason about consistency
  - Treat each file system operation (e.g., the creation of a new file) as a transaction
  - During failure recovery, ensure that:
    - Committed transactions are reflected in on-disk data structures
    - Uncommitted transactions (i.e., transactions that were unfinished at the time of the crash) are not visible in the post-crash disk state

Journaling In Action: ext3



- ext3 is a widely-used journaling file system on Linux
  - Preserves the same on-disk data structures as ext2, but adds journaling support
- The superblock contains file-system-wide info like the block size, the total number of blocks, etc.
- The block descriptor describes where the block groups start
- A block group is like an FFS cylinder group: a set of contiguous sectors on disk which are assumed to be fast to access in quick succession

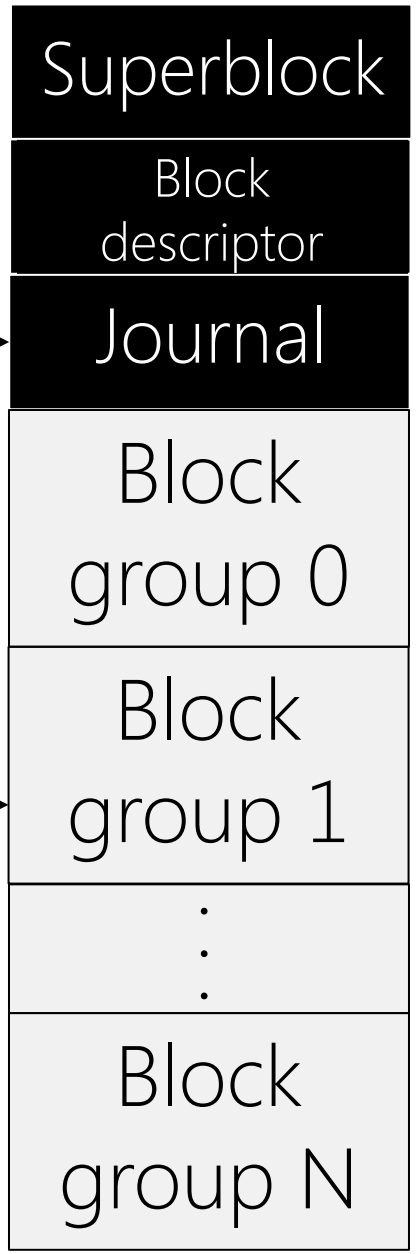
ext3: Redo Write-ahead Logging

Pre-crash

For each high-level file operation (e.g., write(), unlink()) that modifies the file system . . .

- Write the blocks that would be updated into the journal
- Once all blocks are in the journal, transaction is committed; now ext3 can issue the "in-place" writes to the actual data blocks and metadata blocks

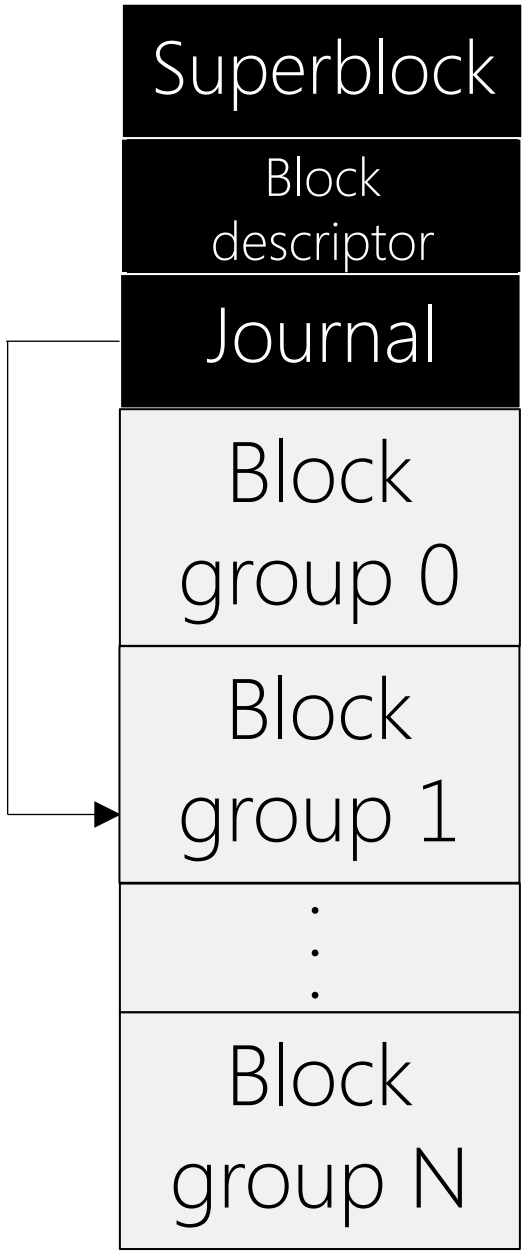
The journal is a circular buffer; asynchronously deallocate journal entries whose in-place updates are done



Post-crash

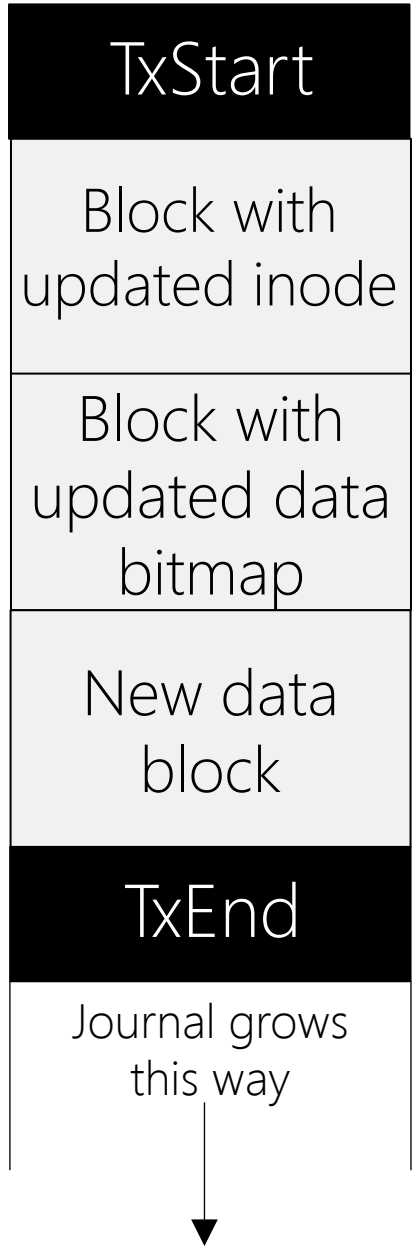
Iterate through the journal, reading the data blocks in each committed transaction, then writing them to the corresponding in-place region on disk

- If the system crashes during recovery, just restart the journal replay (this is safe because replaying entries is idempotent)
- Can deallocate journal entries once they've been replayed



ext3: Logging of Physical Blocks

- ext3 journals physical blocks
  - Even if only part of a physical block is updated, ext3 records the entire enclosing block in the journal
  - Ex: To journal an update to an inode (e.g., to update a data block pointer and file size), ext3 writes the inode's entire enclosing block to the journal (ext3 can use a block size of 1024, 2048, 4096, or 8192 bytes, but inodes are only 256 bytes large)
  - Ex: Even if only part of a data block is updated, ext3 logs the entire block
- Ex: Appending to a file requires three in-place writes
  (1) inode must be updated with a new file size and a new data block pointer
  (2) the data block bitmap must be updated to reflect a new block allocation
  (3) the data block itself must be written

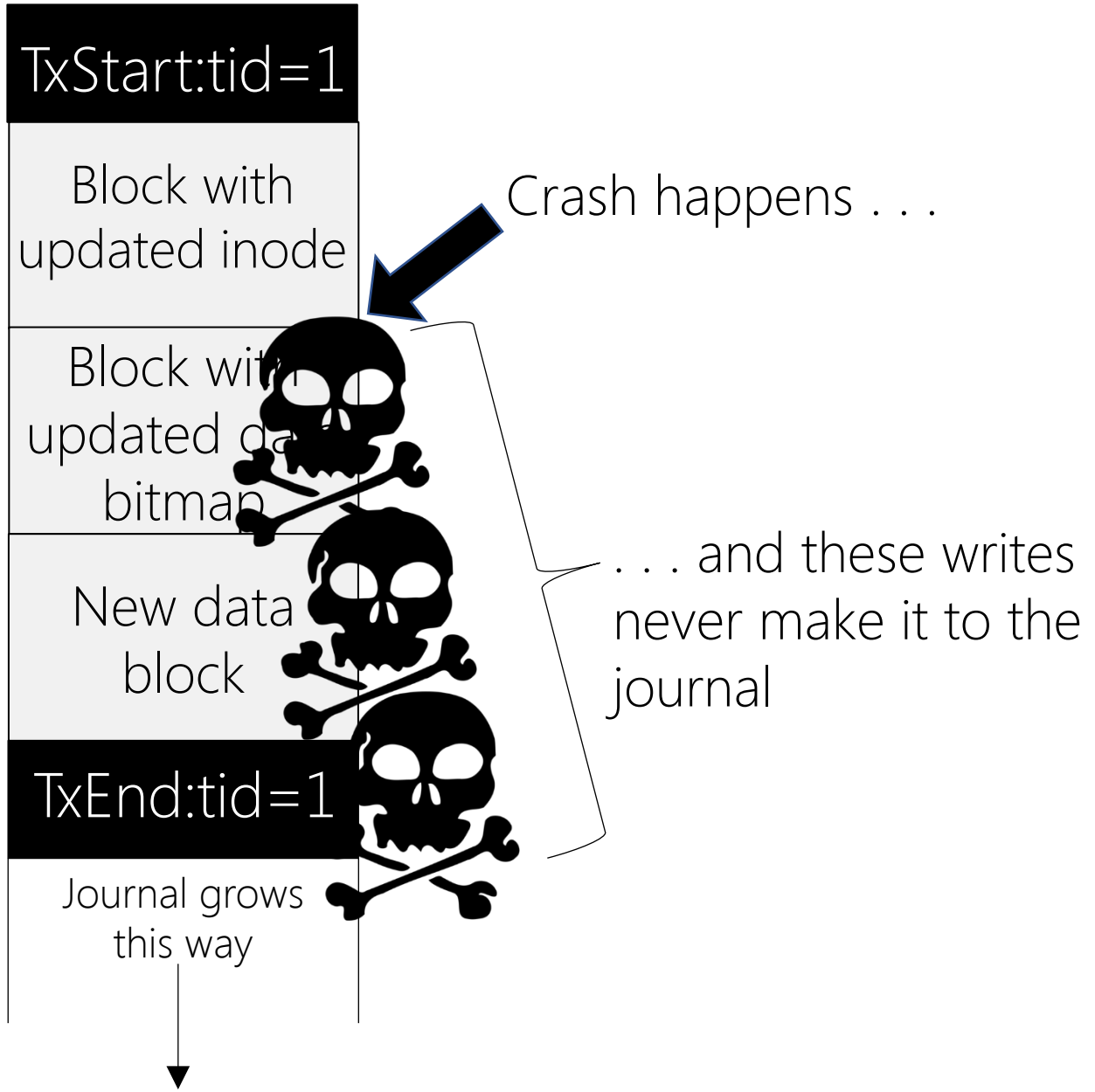


ext3: Logging of Physical Blocks

- How should ext3 issue the writes to the journal?
  - One possible strategy is to:

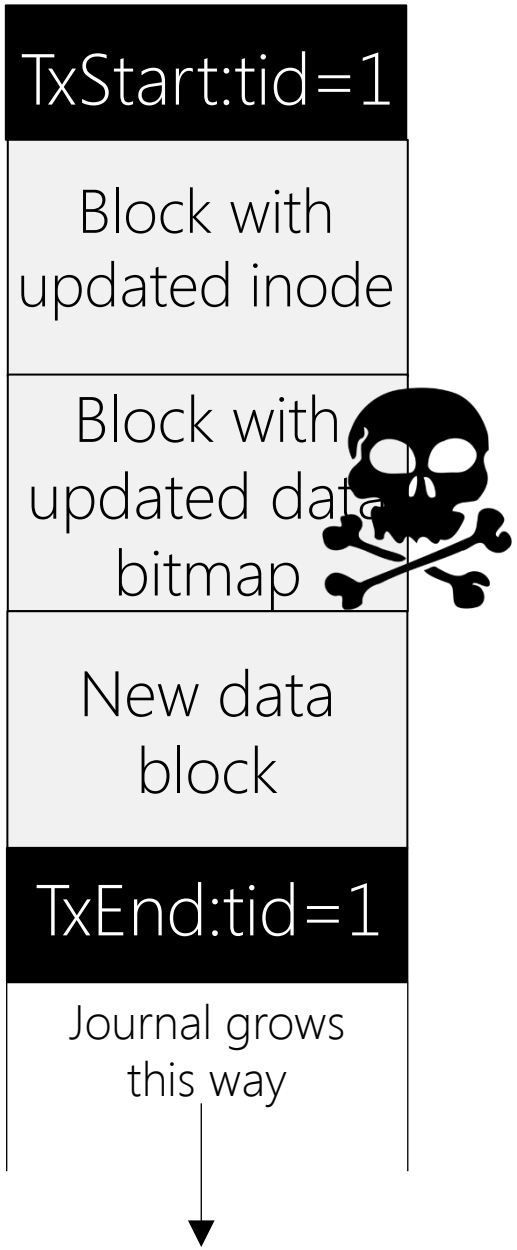
    (1) Issue the journal writes serially, waiting for write i to complete before issuing write i+1

    (2) After the last journal write finishes, issue the checkpoint (i.e., issue the in-place writes) at some future moment
  - If a crash happens in the midst of (1), we're fine
    - During crash recovery, we'll see a valid TxStart, but no valid TxEnd for the associated tid
    - If the data block made it to the journal, we'll have to discard it, but the file system will be consistent



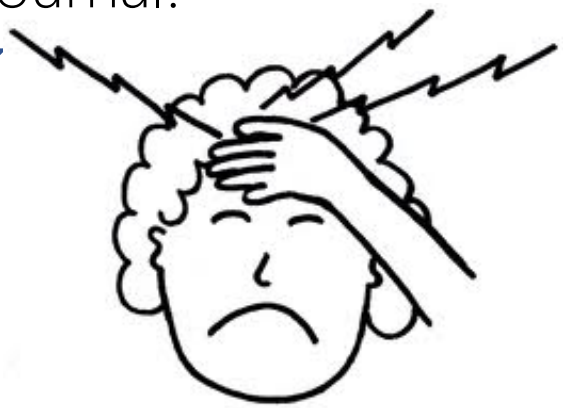
ext3: Logging of Physical Blocks

- The prior strategy works, but it's slow, since the writes are serially issued
- A faster strategy is to:
  (1) issue all of the journal writes at once
  (2) when they all complete, issue the checkpoint at some future moment
- Problem: the disk can reorder writes, which may cause havoc if a crash happens during (1)
  - Remember that only sector-sized writes are atomic!
  - For example, suppose that all writes except the middle one complete . . .



ext3 would find a matching TxStart and TxEnd, so the transaction will seem valid . . .

. . . so ext3 would update the data bitmap with whatever insanity was in the journal!



ext3: Logging of Physical Blocks

- The actual ext3 strategy is to:
  (1) Issue TxStart and everything up to (but not including) TxEnd
  (2) Once those writes have all completed, issue TxEnd
  (3) Once the TxEnd is persistent, the checkpoint can be issued at some future moment
- This protocol ensures that a valid-looking transaction is really composed of valid journal entries
  - Note that a TxEnd record is essentially just a tid, so it fits inside a single sector and will be written atomically
- Remember that the journal is finite-sized!
  - ext3 treats the journal like a circular buffer
  - In the background, ext3 deallocates journal transactions that have been checkpointed
    - The journal has its own superblock which records the start and end of the valid region
    - After a checkpoint occurs, ext3 asynchronously updates the superblock to indicate the new start of the log

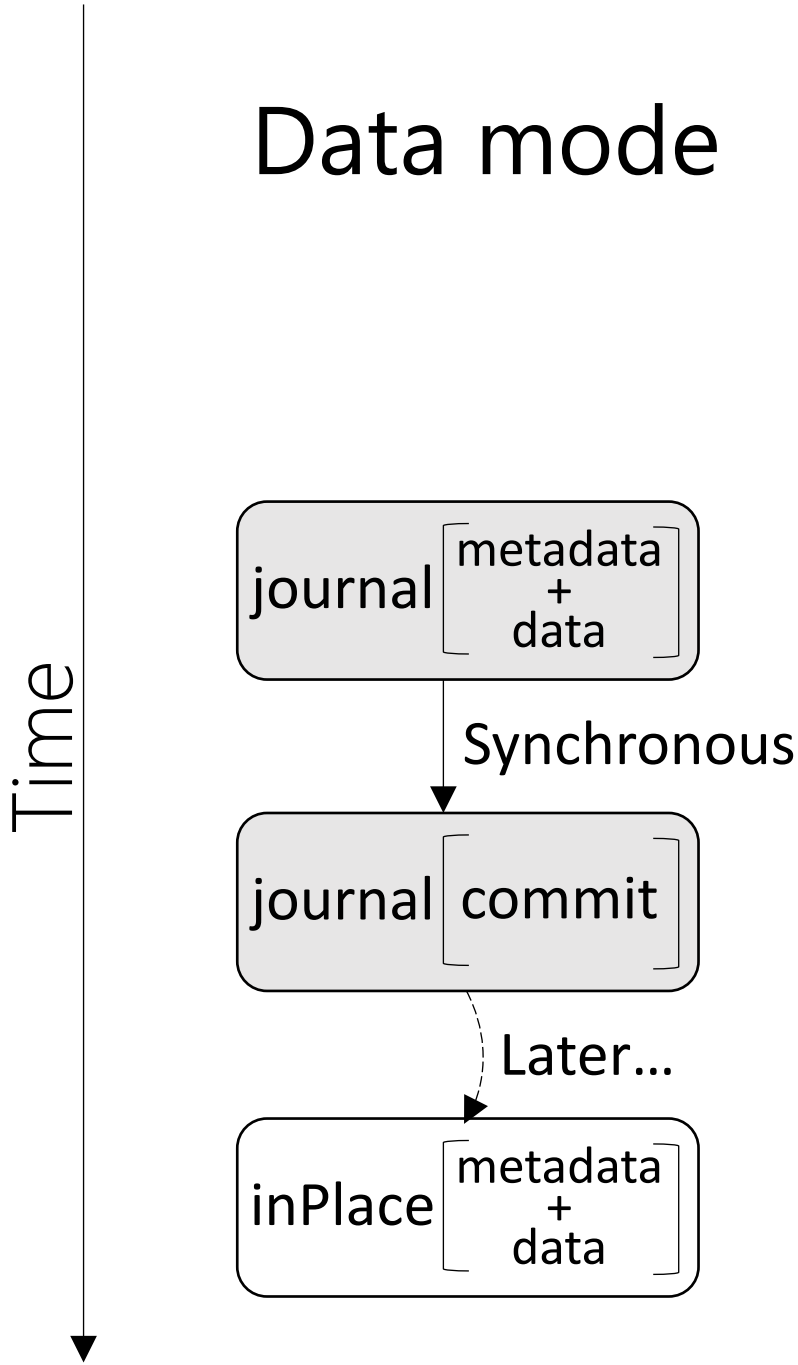
ext3: Controlling What Gets Journaled

- In the previous slides, we've assumed that ext3 journals both data and metadata
  - This policy provides the strongest consistency, but requires double-writes for all new data
  - However, many people are willing to accept data loss/corruption after a crash, as long as *metadata* is consistent
- So, ext3 defines three different journaling modes: data, ordered (the default), and writeback

ext3: Controlling What Gets Journaled

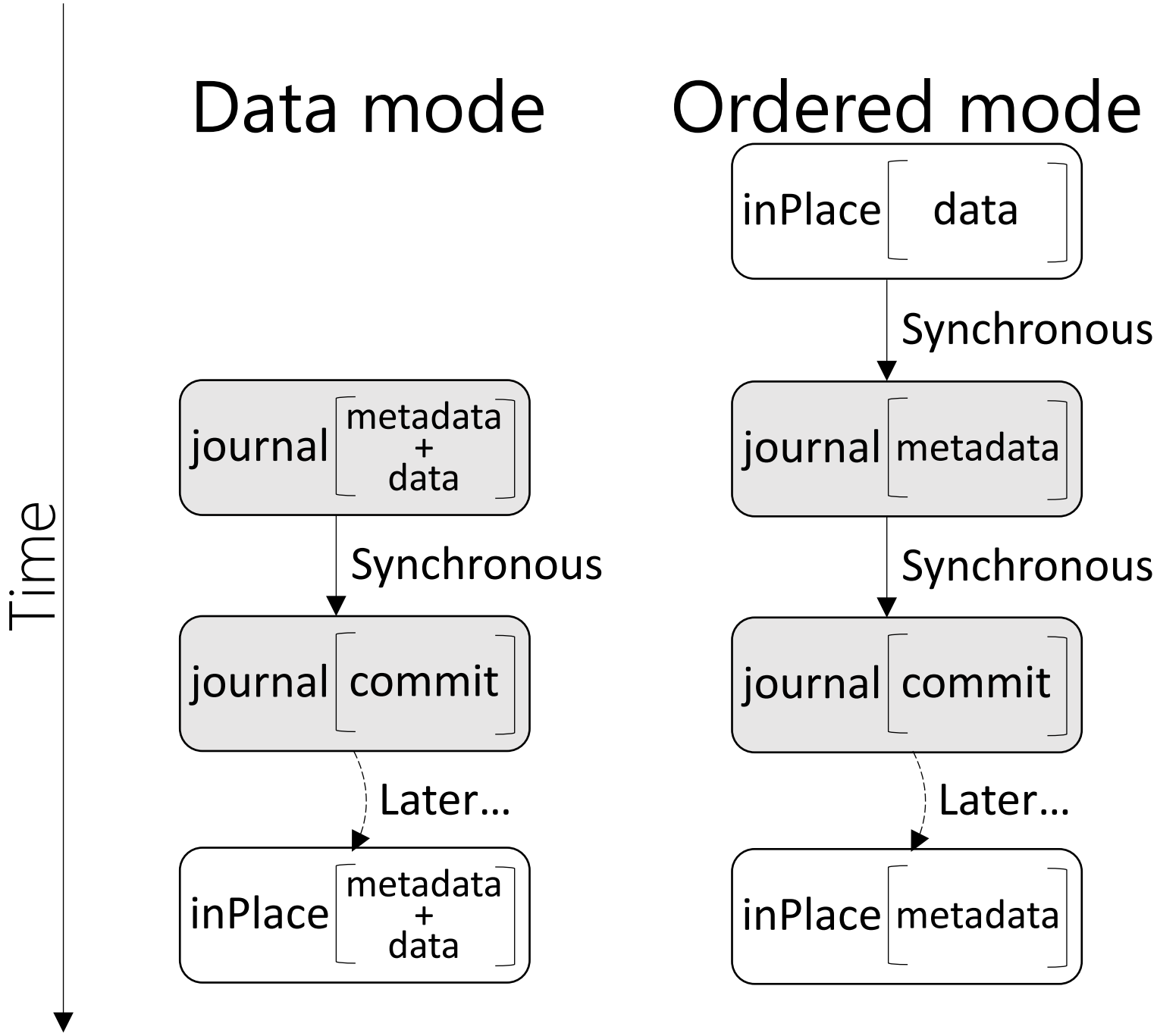
- Up to this point, we've looked at data mode
  - Both data and metadata are journaled
  - Post-crash, metadata is consistent, and files never contain junk (although writes may be lost)
  - Data mode incurs a double-write penalty for all data *and* metadata

Data mode



ext3: Controlling What Gets Journaled

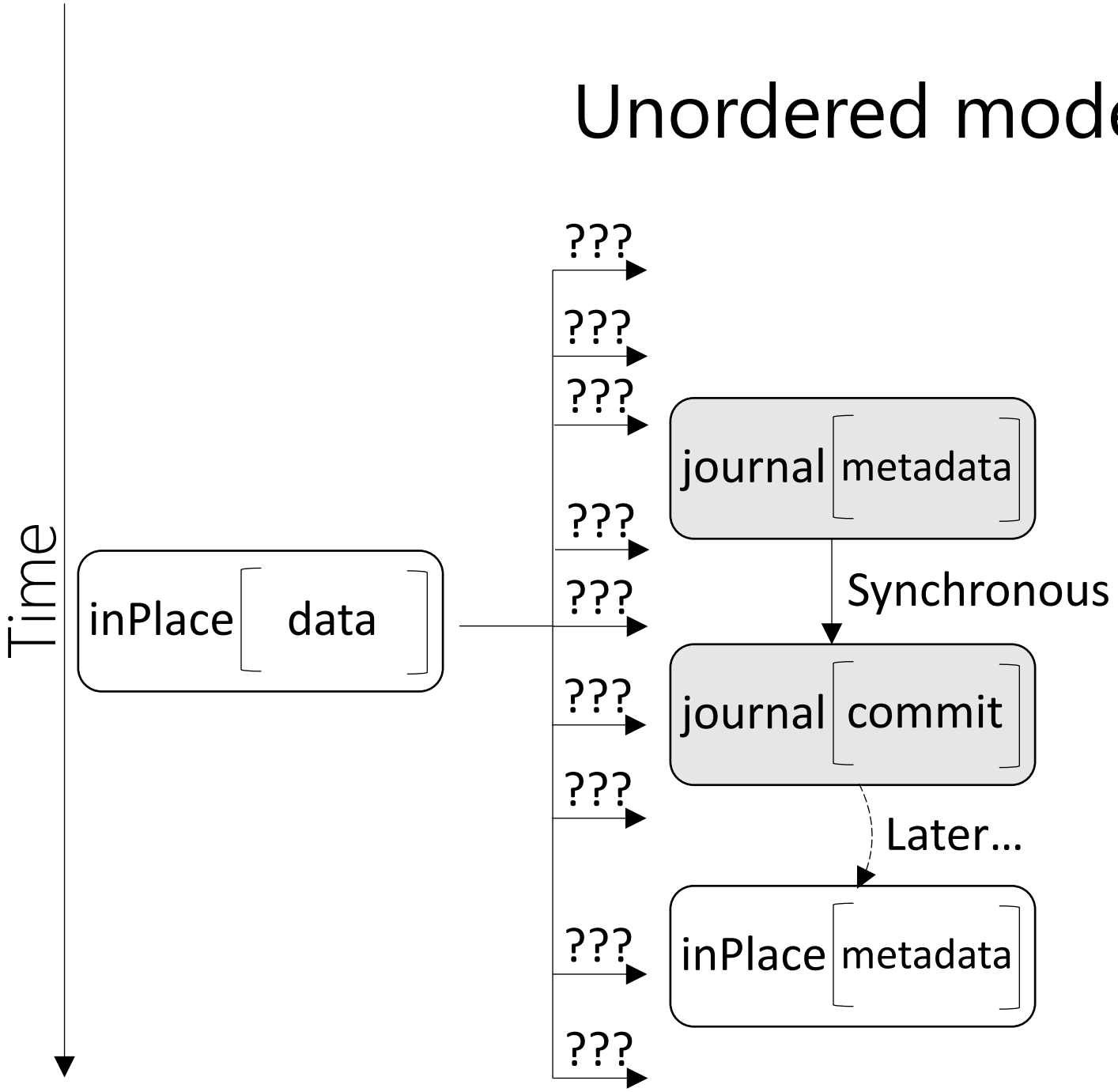
- Ordered mode does not journal data, but writes it in-place before issuing journal updates for metadata
  - Avoids double-write penalty for data, while ensuring that writes to preexisting regions of a file are always preserved post-crash if those writes make it to the disk
  - Still possible for appends to be lost post-crash
  - Forcing the journal update to wait for the data write can hurt performance



ext3: Controlling What Gets Journaled

- In unordered mode, the in-place data writes can be issued at any time w.r.t. journal updates and checkpoints for metadata
  - Allows the disk freedom to reorder those writes w.r.t. journal updates, improving performance
  - However, post-crash, files may contain junk data if the in-place data updates never hit the disk

Unordered mode



ext3: Batching Journal Updates

- Suppose that, in quick succession, a process creates three new files in the same directory (and thus the same block group)
  - ext3 will need to update the same directory, inode bitmap, and data block bitmap multiple times
  - To do so, ext3 could generate three separate transactions for each file create
  - However, this would force ext3 to repeatedly journal and in-place update the same set of physical blocks
- To minimize disk traffic, ext3 creates "global" transactions
  - ext3 defines a waiting period for collecting updates
  - During that period, ext3 uses an in-memory structure to record which blocks are dirty (i.e., have been updated during the period)
  - Once the period is over, ext3 issues a single transaction for all of the dirty blocks

Summary of ext3

- ext3 is a journaling file system that does physical redo logging
- To make a file system update in ordered mode, ext3 does the following:

  (1) Issue an in-place write for the data

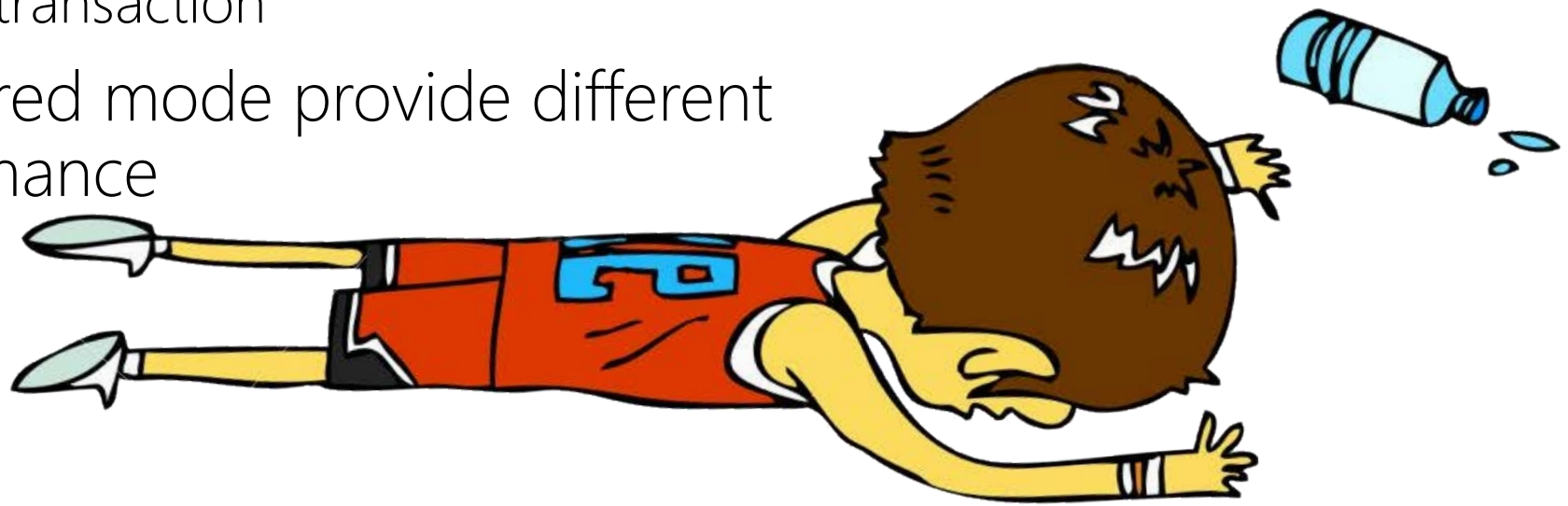
  (2) Once those writes complete, update the journal with TxBegin and journal entries for the metadata

  (3) Once those writes complete, issue a TxEnd

  (4) Once that write completes, asynchronously checkpoint the metadata (i.e., write the in-place metadata)

  (5) Once that write completes, asynchronously update the journal superblock to deallocate the associated transaction

- Data mode and unordered mode provide different consistency and performance



Journaling: Undo Logging vs. Redo Logging

- In redo logging, we make operation X persistent by:
  - Starting a transaction: update the journal with TxBegin and the new data that is associated with X
  - Once those writes complete, commit the transaction: update the journal with TxEnd
  - Once the transaction is committed, asynchronously perform the in-place updates
- During post-crash recovery, only replay committed transactions

- In undo logging, we make operation X persistent by:
  - Starting a transaction: update the journal with TxBegin and instructions for how to undo X's in-place updates (e.g., instructions might include the original on-disk values)
  - Once those writes complete, asynchronously perform the in-place updates that are associated with X
  - Once those writes complete, commit the transaction: update the journal with TxEnd
- During post-crash recovery, undo uncommitted transactions by rolling *backwards* through the log, applying undo instructions to in-place disk locations

Journaling: Undo Logging vs. Redo Logging

- Redo logging
  - Advantage: A transaction can commit without the in-place updates being complete (only the journal updates need to be complete)
    - In-place updates might be to random places on the disk, whereas journal writes are sequential
  - Disadvantage: A transaction's dirty blocks must be buffered in-memory until the transaction commits and all of the associated journal records have been flushed to disk
    - Buffering leads to increased memory pressure
    - Ideally, it would be safe to flush a dirty block after the associated journal record has been written to disk (even if the transaction has not committed yet)
- Undo logging
  - Advantage: A dirty buffer can be written in-place as soon as the corresponding journal entries have been written to disk
    - Useful if the file system is experiencing high memory pressure and needs to evict buffers
  - Disadvantage: A transaction cannot commit until all dirty blocks have been flushed to their in-place targets
    - Delaying a transaction's commit might delay other transactions who want to read or write the associated data
    - So, the file system has time pressure to issue those writes quickly, even if they would cause unfortunate seeking behavior

Journaling: Redo+Undo Logging

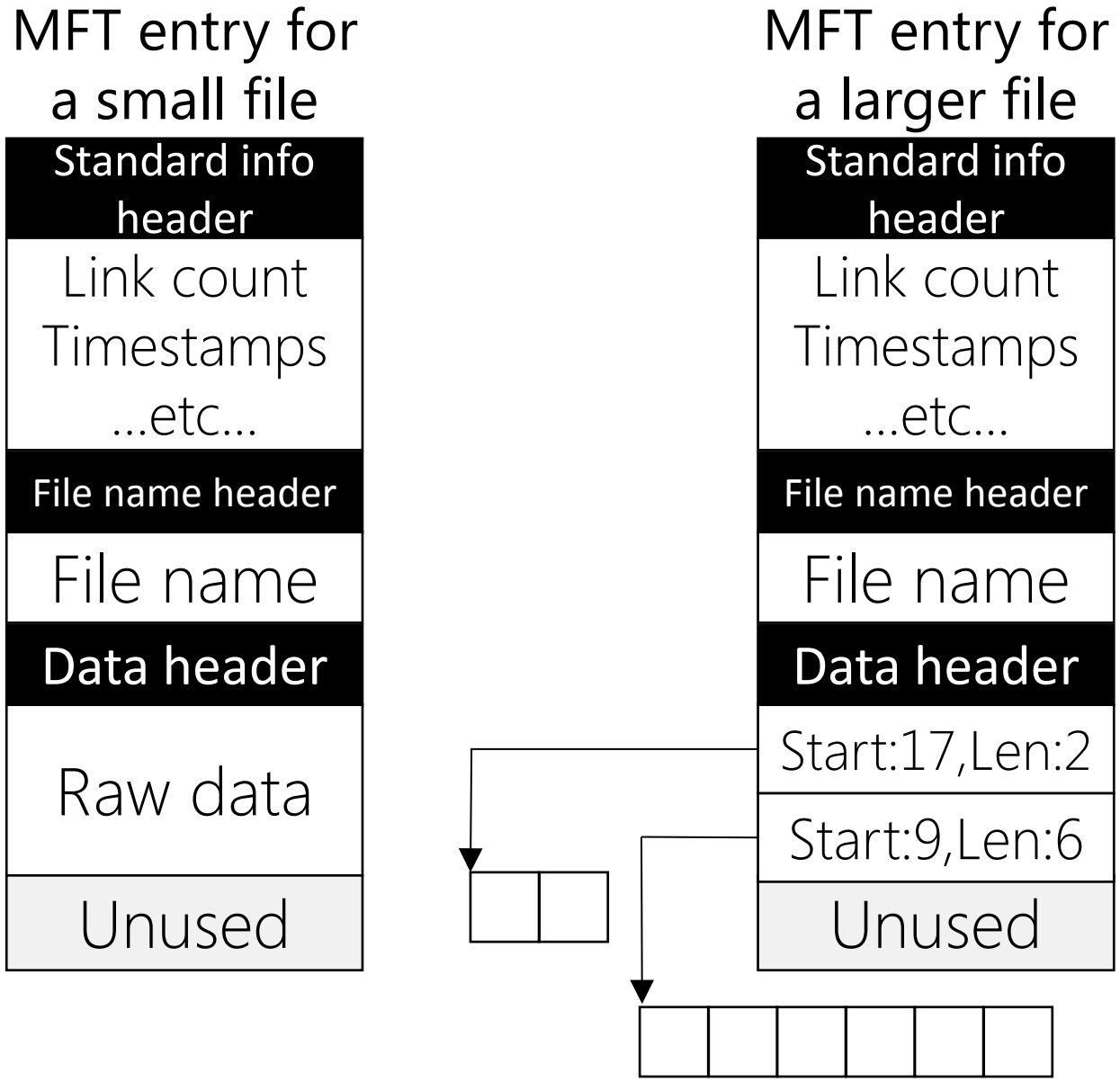
- The goal of redo+undo logging is to:
  - Allow dirty buffers to be flushed at any time after their associated journal entries are written (as in undo logging)
  - Allow a transaction to commit without its in-place updates being finished (as in redo logging)
- In redo+undo logging, we make a file system operation X persistent by:
  - Starting a transaction: Write TxBegin
  - For each component of the transaction, write a <redoInfo,undoInfo> record to the journal
    - Once the record has been written, issue an in-place update for the component at any time!
  - Once the journal operations finish, commit the transaction: Write TxEnd to journal
    - Note that some, all, or none of the in-place updates might be finished at this point
- Post-crash recovery now requires two phases
  - Roll forward through the log, redoing all committed transactions (potentially duplicating work if the transactions' in-place updates succeeded before the crash)
  - Roll backwards through the log, undoing all uncommitted transactions that might have issued in-place updates before the crash

Journaling: NTFS

- NTFS is the file system on Windows
  - NTFS performs redo+undo logging (but only journals metadata, not data)
  - Supports block sizes from 512 bytes up to 64KB, with a default of 4KB
  - Has more bells and whistles than stock ext3 (e.g., NTFS natively supports file compression and encryption)
- The root directory of an NTFS file system contains special files with reserved names that are used to implement key functionality
  - $MFT: the Master File Table, which contains metadata for all of the files and directories in the file system
  - $LogFile: the journal
  - $Bitmap: allocation information for blocks

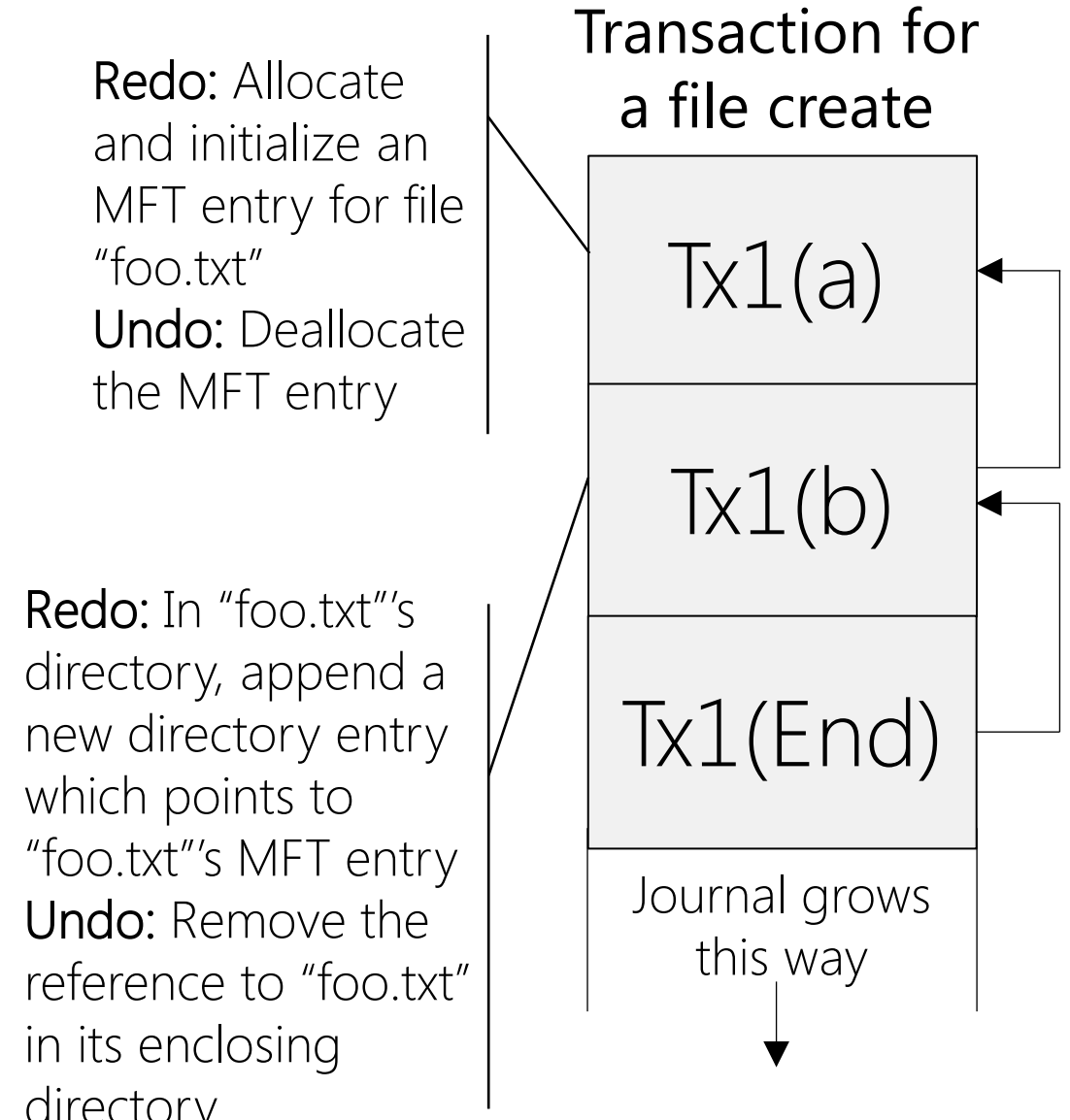
NTFS: Files and Directories

- The MFT contains an entry for each file and directory
  - Each entry is 1024 bytes long, and roughly corresponds to an inode
  - Each entry lists the attributes for the file/directory (e.g., name, link count, access timestamps, data characteristics like "compressed" or "encrypted")
  - Note that the file/directory data is just another attribute!
    - For small files/directories, all of the data is stored inside the MFT record
    - For larger files/directories, the MFT record has pointer(s) to the relevant on-disk extents
  - Programs can define new, custom attributes for their files!
- At file system initialization time, NTFS reserves a contiguous 12.5% region of the disk for the MFT

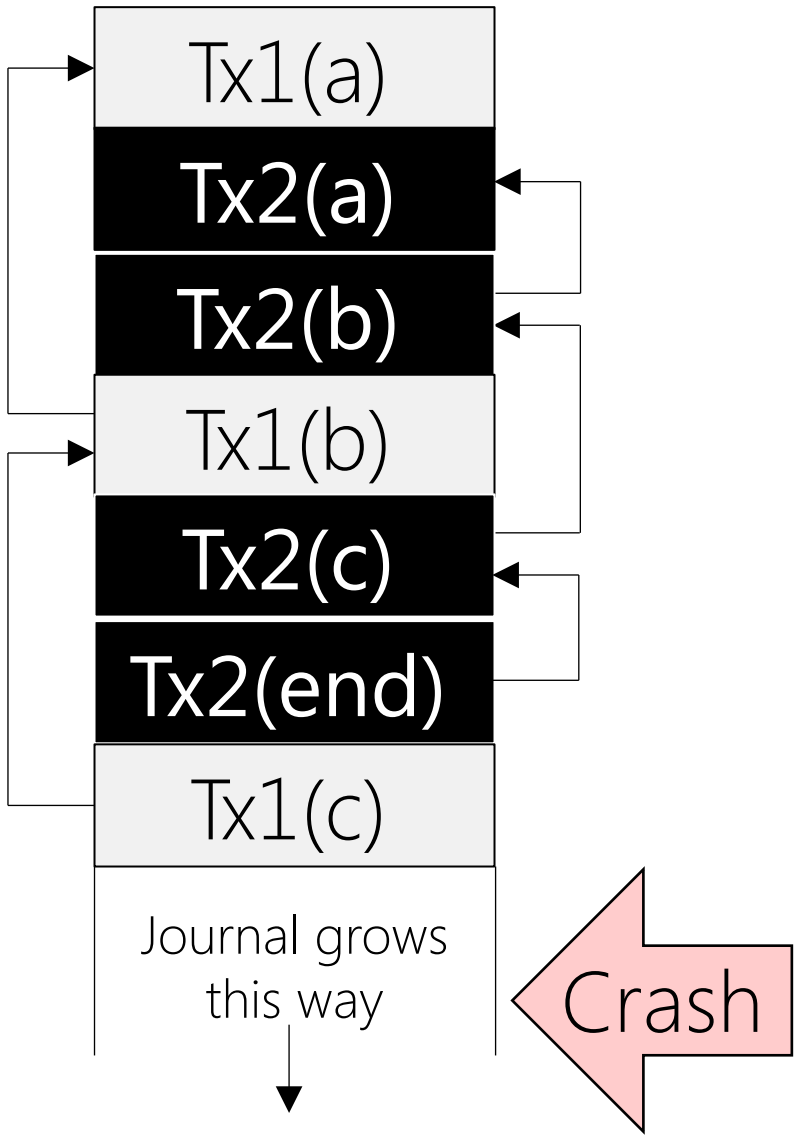


NTFS: Operation Logging

- Unlike ext3 (which uses physical logging), NTFS uses operation logging
  - An operation log describes modifications to file system data structures
  - Ex: "Set bit 4 in a bitmap" or "write the following values into an MFT entry"
  - Operation logging has smaller log entries than physical logging (which much store entire physical blocks), although replay logic is more complicated
  - ~~In A4, you must implement operation logging~~
- Unlike ext3 (which batches multiple file system operations into a single transaction), NTFS creates a separate transaction for each file system operation
  - Each NTFS transaction consists of sub-operations
  - Each sub-operation has:
    - a redo field
    - an undo field
    - a pointer to the previous sub-operation in the transaction



NTFS: Example of Crash Recovery



- First, NTFS rolls forward through the log, redoing *all* of the sub-operations in order
  - Tx1(a), Tx2(a), Tx2(b), Tx1(b), Tx2(c), and Tx1(c) are all redone, in that order
- Then, NTFS rolls backwards through the log, undoing the sub-operations from uncommitted transactions
  - Tx1(c), Tx1(b), and Tx1(a) are undone, in that order

Q: Why can't we eliminate the undo pass, and just have a forward pass in which we only redo sub-operations from committed transactions?

A: The presence of a log record means that the associated in-place writes may have hit the disk! If those writes belong to uncommitted transactions, they must be undone