CS202 (003): Operating Systems
File System III

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last Time

Problem setup

A lot of data structures are involved in implementing a file system:
bitmap of free blocks, directories, inodes, indirect blocks, data blocks, etc.

We want these data structures to be ***consistent*** (i.e., certain invariants to hold)

We also want to ensure that data on the disk remains ***consistent***

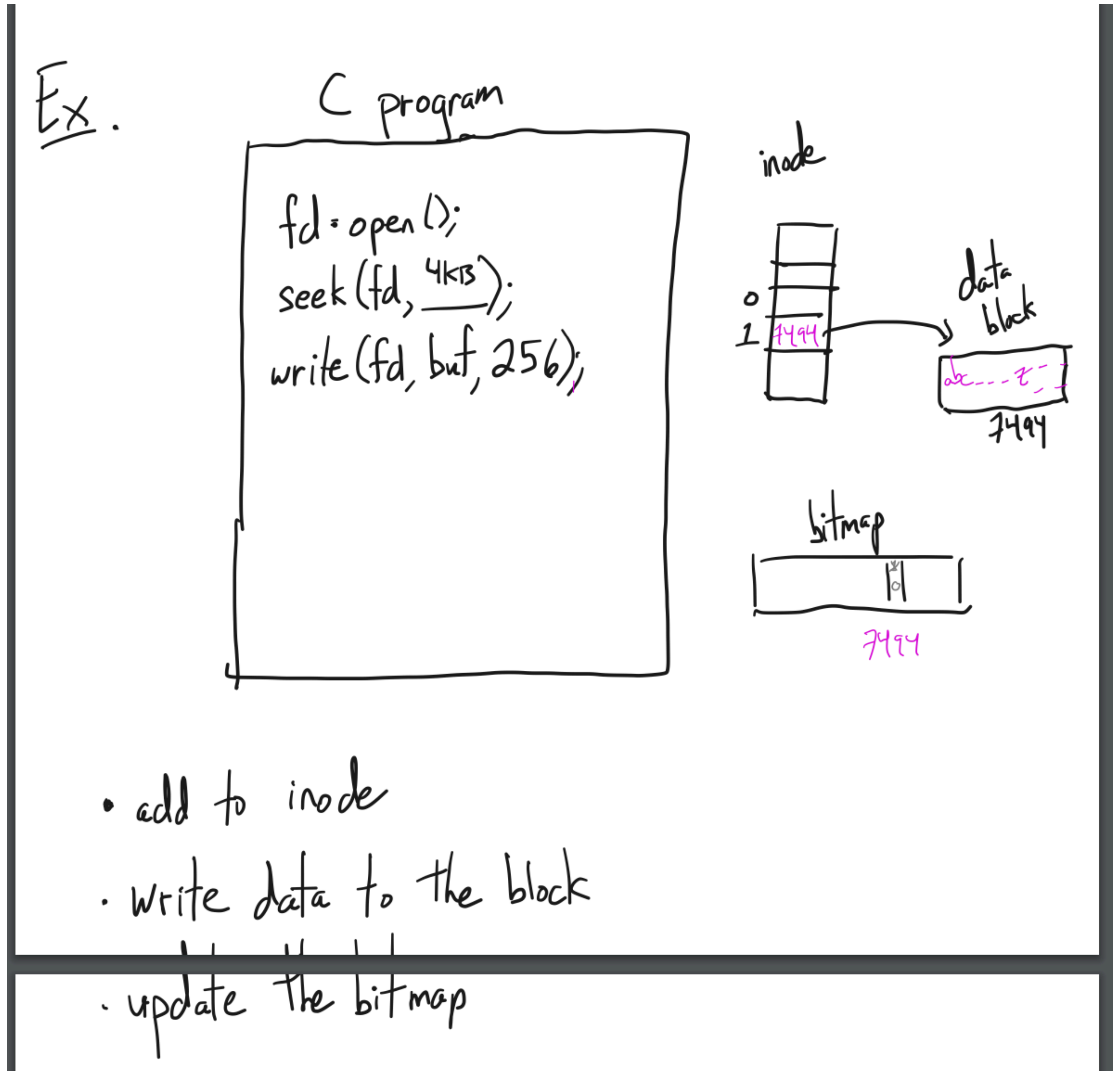
Key issue: crashes or power failures

Some more problematic optimizations

Remember write-back caching and non-ordered disk writes?

OS delays writing back modified disk blocks

modified disk blocks can write to disk in an unspecified order



What happen if something goes wrong in any of these operations?

The system requires some notion of atomicity

Imagine that a crash can happen at any time.
You want to arrange for the world to look sane, regardless of where a crash happens.

"Hmmm… Can we increase the atomic unit size?"

Challenge: metadata and data is spread across several disk blocks!

What a file system designer can leverage on:
Arrange for some disk writes to happen **synchronously**
(system won't do anything until these disk write complete)
Impose some ordering on the actual writes to the disk

The system requires some notion of atomicity

High-level operations

It's impossible to make it actually atomic

Key idea: make "adding data to file" to ***look*** atomic!
(an update either occurs or it doesn't)

Update from who's perspective?

Crash recovery

Ad-hoc (“fsck” in textbook)

Copy-on-write approaches

Journaling (i.e. write-ahead logging)

Ad-hoc

Goal: metadata consistency (not data consistency)

too expensive to provide data consistency

Arrange to send file system upstates to the disk in a way that, if there is a crash, **fsck** can clean up inconsistencies

for example

write data to file → update inode → mark inode "allocated" in bitmap → mark data block "allocated" in bitmap → update directory

Disadvantages

(a) need to get the reasoning exactly right

(b) poor performance: multiple updates to the same block require that they are issued separately

(c) slow recovery: need to scan the entire disk

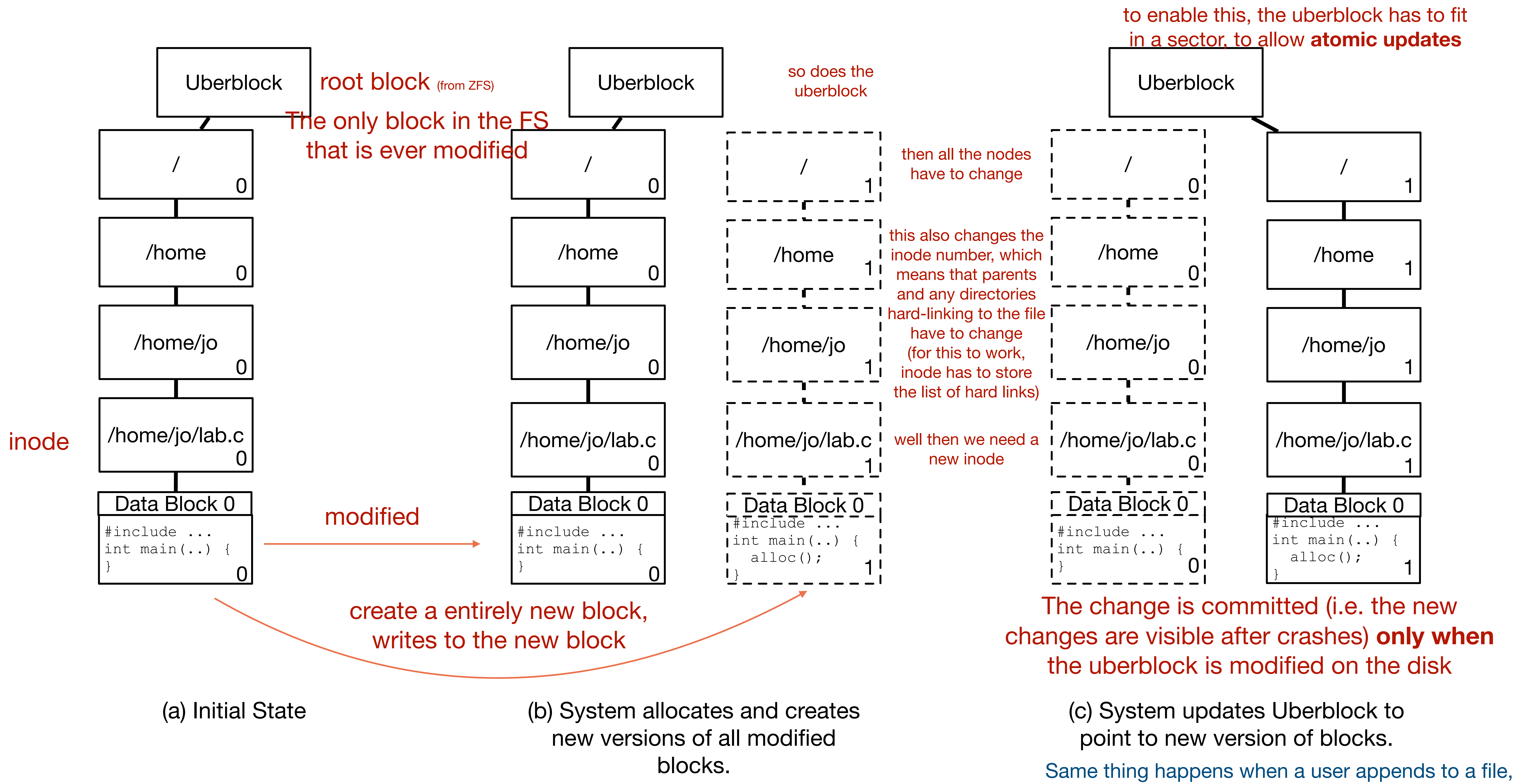
Copy-on-write

Goal: provide both metadata and data consistency, by using more space

disks have gotten larger, space is not at a premium

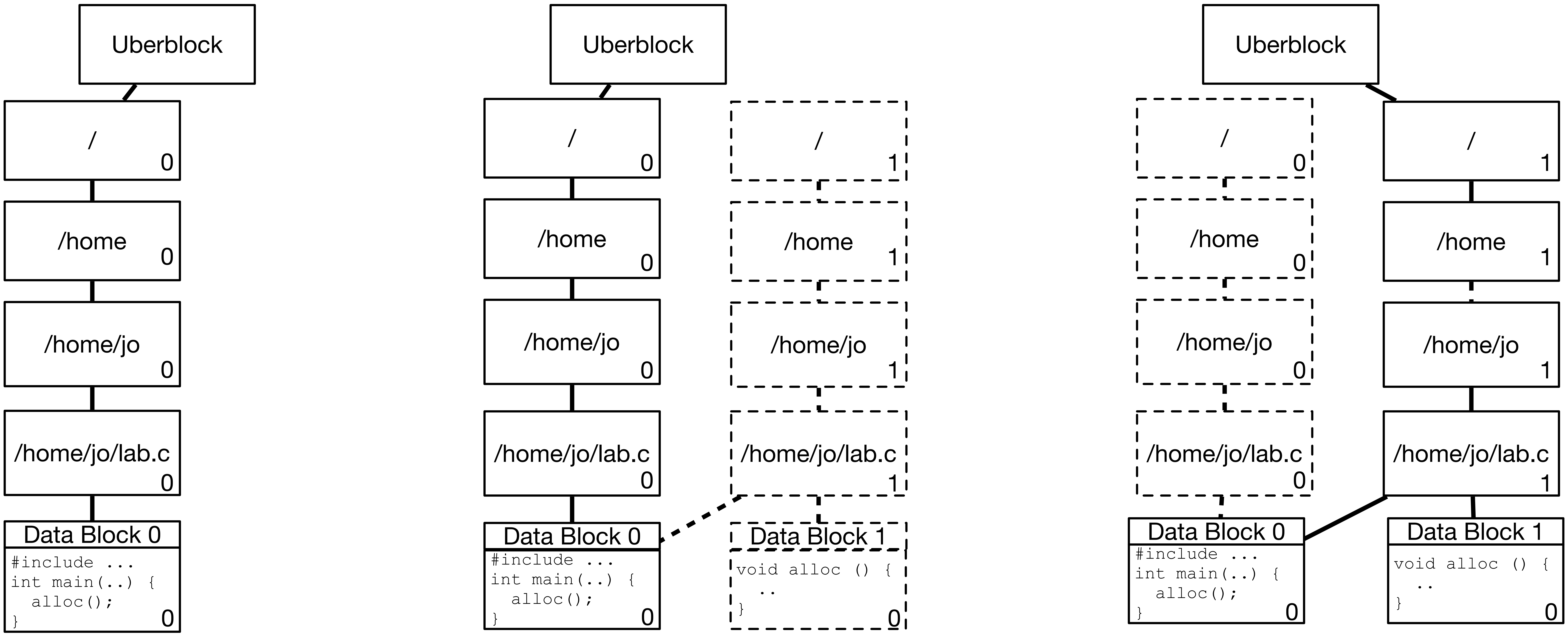
Never modify a block, instead always make a new copy

https://www.cs.hmc.edu/~rhodes/courses/cs134/sp19/readings/zfs.pdf



Same thing happens when a user appends to a file, creating another block, and when creating a file (since the directory inode has to change)

Figure 1: Copy-on-write filesystem: modifying a data block

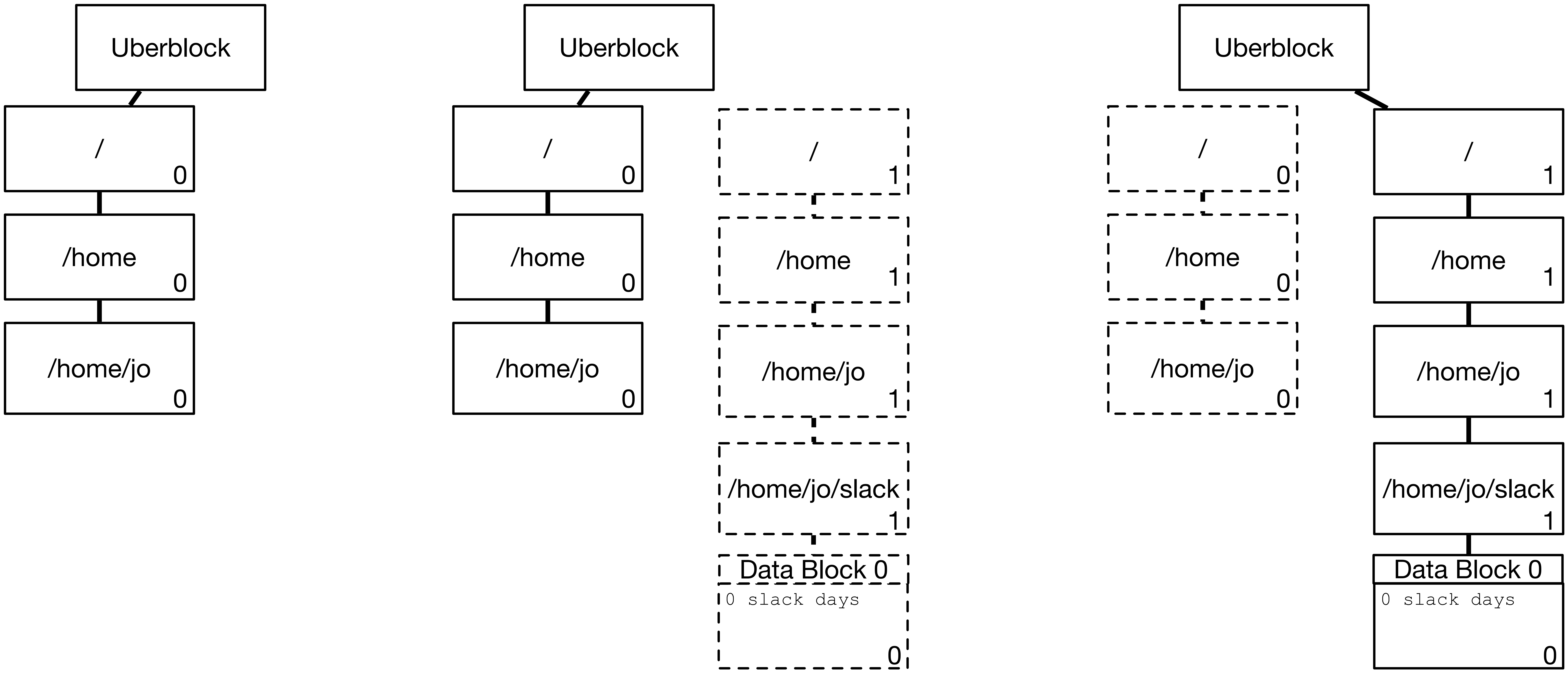


(a) Initial State

(b) System allocates and creates new versions of all modified blocks.

(c) System updates Uberblock to point to new version of blocks.

Figure 2: Copy-on-write filesystem: adding a data block



(a) Initial State

(b) System allocates and creates new versions of all modified blocks.

(c) System updates Uberblock to point to new version of blocks.

Figure 3: Copy-on-write filesystem: creating a file

Copy-on-write

Goal: provide both metadata and data consistency, by using more space

disks have gotten larger, space is not at a premium

Never modify a block, instead always make a new copy

Benefits

(a) most changes can be committed in **any order** (which brings performance benefits)

(b) on-disk structure and data is **always** consistent (no need for fsck, or run recovery)

(c) FS incorporates versioning similar to Git or other tools (requires not throwing away old blocks)

Disadvantages

(a) significant write amplification (any writes require changes to several disk blocks)

(b) significant space overhead: need enough space to code metadata blocks in order to make any changes

(c) need the use of a garbage collection daemon in order to reclaim blocks from old versions of the FS

Apparently, we can achieve data consistency when modifications do not modify the current copy

Journaling (borrowed from how transitions are implemented in databases)

Goal: Reduce write/space overhead without violating atomicity

Treat file system operations as transactions:
after a crash, failure recovery ensures

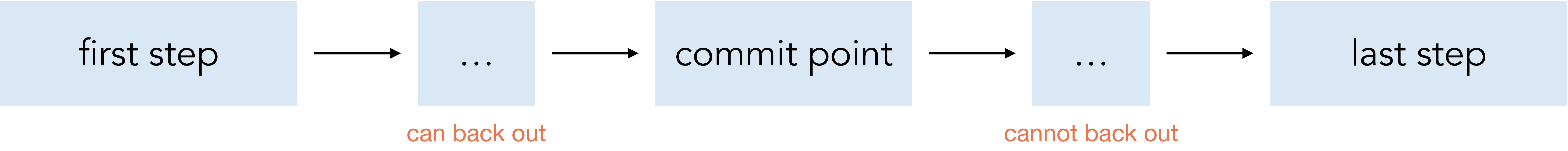
1. committed file system operations are reflected in on-disk data structures
2. uncommitted file system operations are not visible after crash recovery

Record enough information to finish applying committed operations *(redo operations)*
and/or roll-back uncommitted operations *(undo operations)*
This information is stored in a redo/undo log

Journaling

Commit point: the point at which there is no turning back

for example



What is the commit point in copy-on-write?

Journaling — redo logging (used by ext3 & ext4)

FS computes what would change due to a operation

↓

FS computes where in the log it can write this transaction, and writes a transactions begin record . Do not have to wait on this.

↓

FS writes a record(s) with all the changes it computed in step 1. FS **must wait** for changes and TxnBegin to finish written to the disk

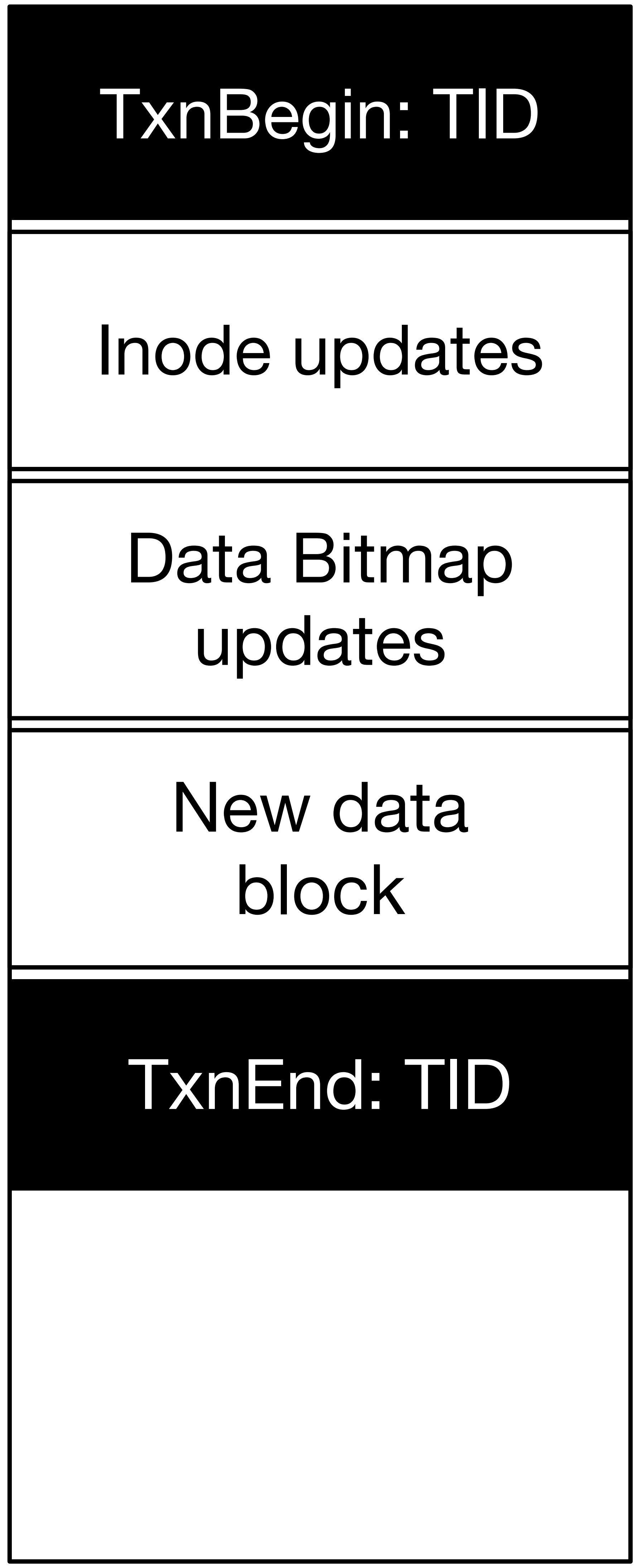
↓

Once step 3 finishes, the system writes a transaction end record

↓

Once the TxnEnd has been written, the FS asynchronously performs the actual FS changes

“checkpointing”



ext3 journal layout

Journaling — crash recovery of redo logging

High-level idea:
read through the logs, find **committed operations** and apply them

How to check whether ops are committed? Look at TxnBegin and TxnEnd!
It is safe to apply the same redo log multiple times

FS starts scanning from the **beginning of the log**

↓

Every time it finds a TxnBegin entry, it looks for the corresponding TxnEnd entry

↓

If matching (TxnBegin, TxnEnd) found, FS checkpoints the changes

↓

Recovery is completed once the entire log is scanned

What to log?

Logging can double the amount of data written to the disk
Ext3 and 4 allows user to choose what to log

Default: metadata only (assuming people are fine with data loss after crash)
Can change to force data to be logged w/ metadata

Journaling — undo logging

(Not used in isolation by any file system)

Write a TxBegin entry to the log

↓

For each op, write instructions for how to undo any updates.
Changes to the block can be made right after writes finishes

↓

Wait for in-place changes to finish for all blocks

↓

Write a TxnEnd entry into the block

all changes have been written to the actual FS data structures

Journaling — crash recovery from undo logging

Scan to find all uncommitted transactions from **the end of the log**

↓

For each such transaction, check whether undo entry is valid (checksum)

↓

Apply all valid undo entries found

disk back to a consistent state

Benefits	Changes can be checkpoints to disk as soon as the undo log has been updated — useful when the amount of buffer cache is low
Disadvantages	A transaction is not committed until all dirty blocks have been flushed to their in-place targets

Redo logging vs. Undo logging

Benefits

A transaction can commit without all in-place updates (writes to actual disk locations) being completed — useful when in-place updates might be scattered all over the disk

Disadvantages

A transaction's dirty blocks need to be kept in the buffer-cache until the transaction commits and all of the associated journal entries have been flushed to disk. This might increase memory pressure.

Benefits

Changes can be checkpoints to disk as soon as the undo log has been updated — useful when the amount of buffer cache is low

Disadvantages

A transaction is not committed until all dirty blocks have been flushed to their in-place targets

HW 10 is due tomorrow
HW 11 is released today