Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 19: Transactions and LFS

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Ordering updates to a filesystem.
2. Shadowing.
3. Logging.
4. Log-structured filesystems (LFS).
5. Project 4 preview.

Agenda

1. Ordering updates to a filesystem.
2. Shadowing.
3. Logging.
4. Log-structured filesystems (LFS).
5. Project 4 preview.

Multiple updates and reliability

Data must survive crashes and power outages.

Only the update of a single block is atomic and durable.

Challenge: Crashes in midst of multi-step updates.

Move file from directory a to directory b.

1. Delete file from a.
2. Add file to b.

Create new (empty) file.

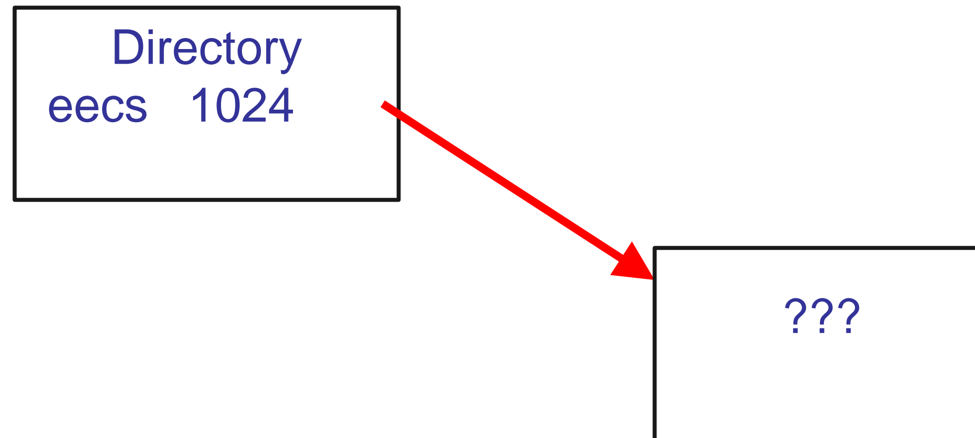
1. Update directory to point to new file header.
2. Write new file header to disk.

Ordering of updates

Careful ordering can fix some problems:

For example, creating file `482.txt` in directory `eecs`

Updating directory first could leave the FS corrupted.

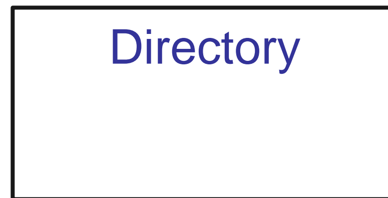


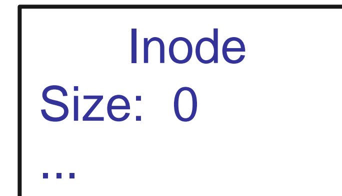
Never have a pointer from valid block to invalid one!

Ordering of updates

Careful ordering can fix some problems:

- For example, creating file `482.txt` in directory `eecs`
- Creating the inode first solves the problem.





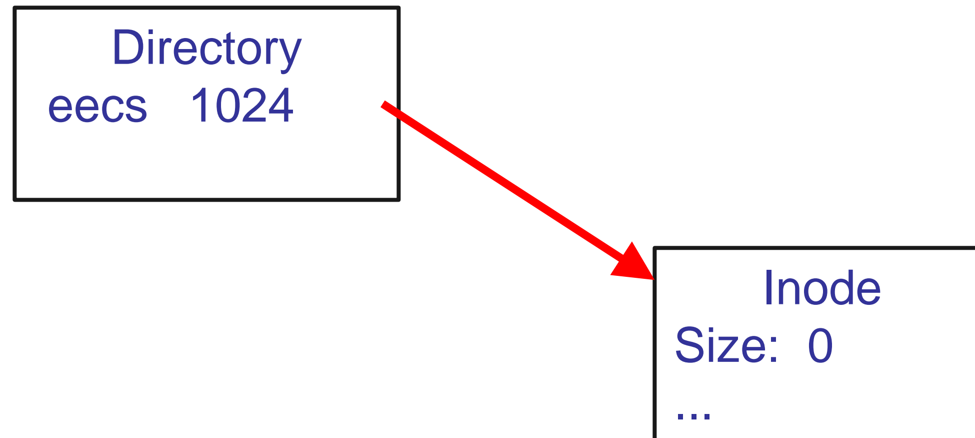
OK to modify unreachable blocks on disk.

Ordering of updates

Careful ordering can fix some problems:

For example, creating file `482.txt` in directory `eecs`

Creating the inode first solves the problem.



Careful ordering goes from one consistent state to another.

Ordering not always enough

Example: Create a file and update the free block list.

1. Write new file header to disk.
2. Update directory to point to new file header.
3. Write the new free map.

No ordering is safe.

ACID terminology

Database systems are commonly describing as offering ACID properties. For a filesystem, we mostly care about atomicity and durability.

Atomicity	All or nothing. The operation either succeeds or does nothing.
Consistency	Representation invariants observed before and after an operation.
Isolation	Any intermediate states are invisible to other transactions which only see the state before or after.
Durability	Once an operation succeeds, the changes persist and will not be undone, even in the event of a system failure.

Transactions

Need a way to create transactions with atomicity and durability. But only writes to a single sector to a disk are atomic.

How to make a sequence of updates atomic?

Two main methods:

1. Shadowing.
2. Logging

```
begin
   write disk
   write disk
   write disk
end // commit the transaction)
```

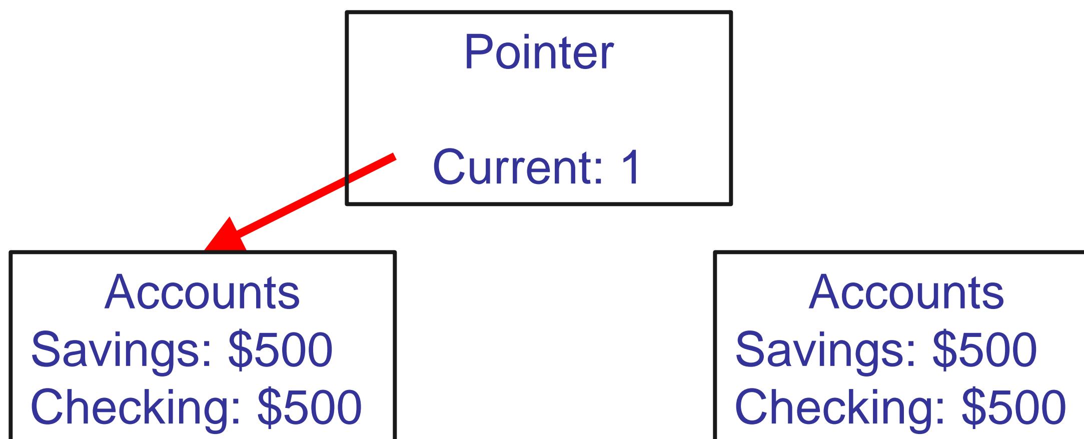
Agenda

1. Ordering updates to a filesystem.
2. Shadowing.
3. Logging.
4. Log-structured filesystems (LFS).
5. Project 4 preview.

Shadowing

Replicate the data across two stores:

- One is the current version, the other is backup.
- Current pointer points to the current version.

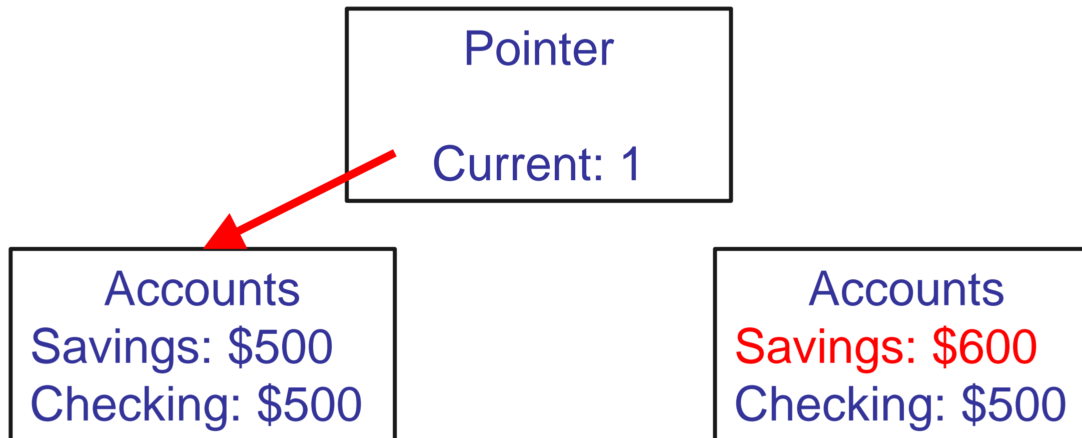


At beginning of transaction, both replicas are identical.

Shadowing

Transaction updates the backup (shadow)

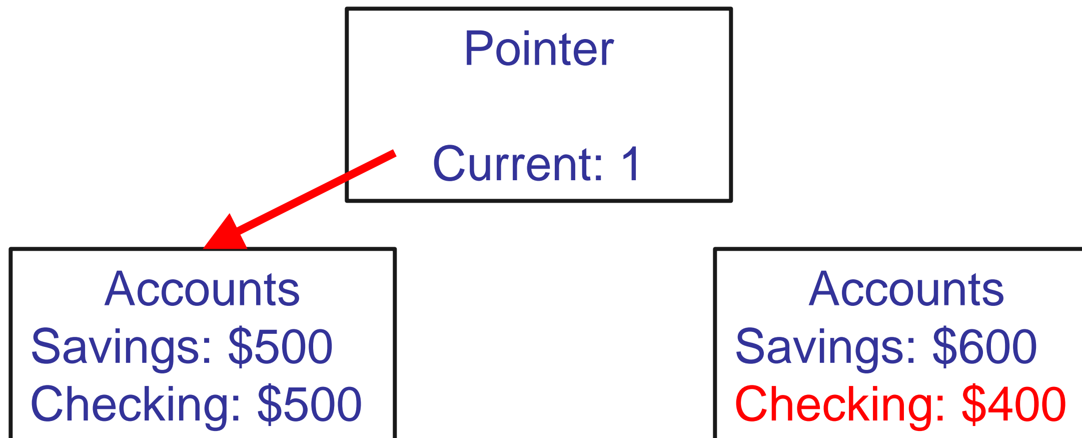
First add $100 to *savings*.



We are modifying “unreachable” block.

Shadowing

- Transaction updates the backup (shadow)
  - Next remove $100 from *checking*

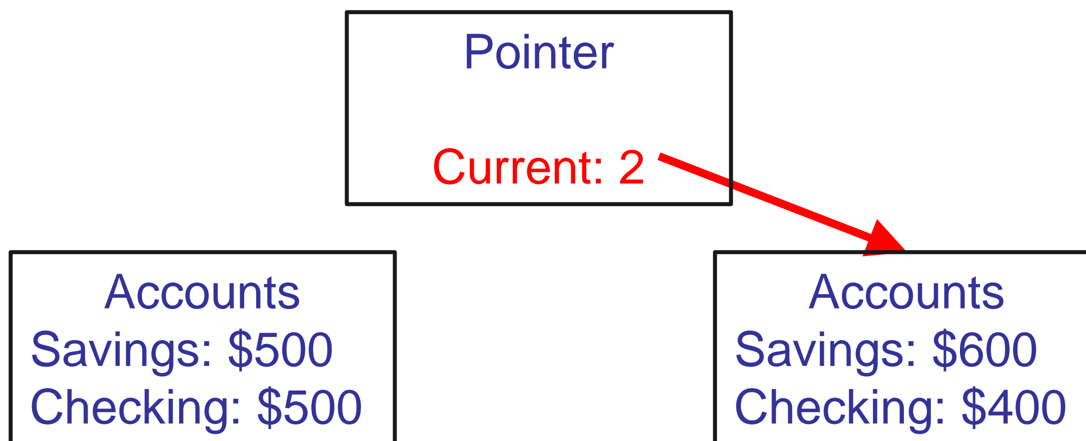


Again modifying an “unreachable” block.

Shadowing

Transaction commit switches the pointer.

This is the point when updates become *durable*.

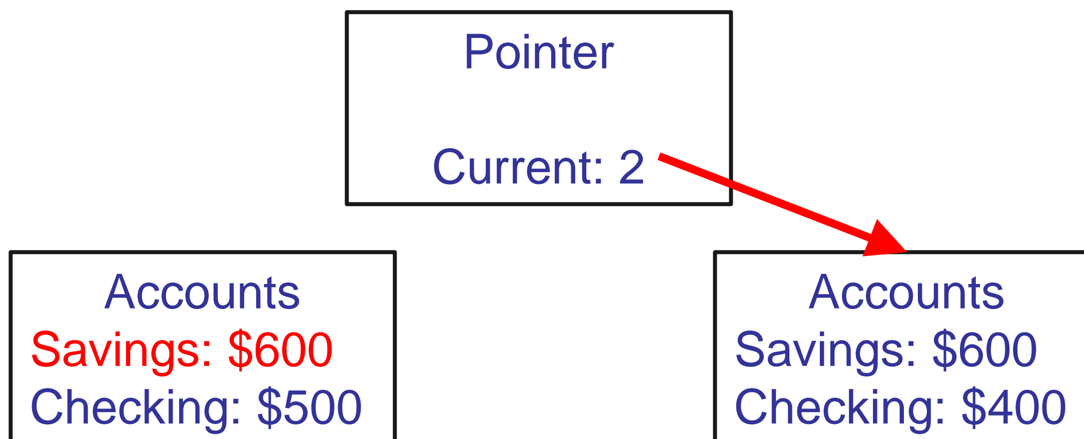


Updating a single block is atomic.

Shadowing

Finally, must update new shadow.

First, update savings.

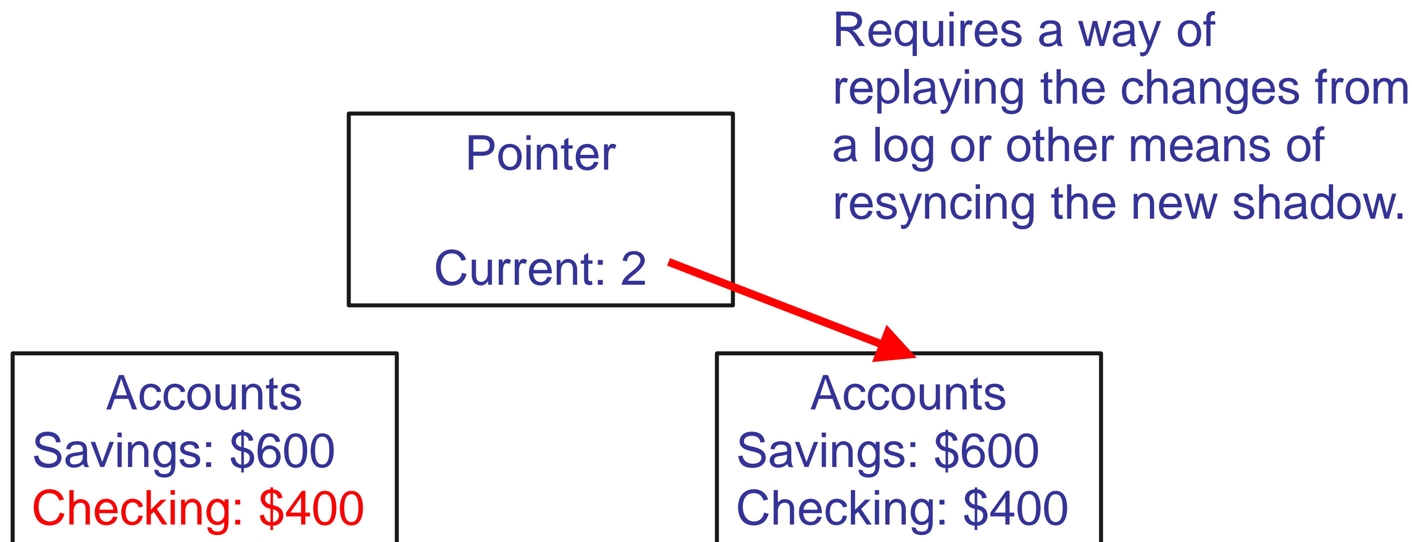


Again, updating an unreachable block.

Shadowing

Finally, must update new shadow.

Next, update checking.



Again, updating an unreachable block.

Shadowing summary

Can make arbitrary set of updates in transaction.

Pointer switch is always an atomic commit.

Downside?

Requires replicating the data store.

Requires means of resyncing the shadow.

Can reduce the cost by shadowing only when needed.

On demand. Sometimes called shadow paging.

Create the shadow only when you need to make a change.

Once the change has been committed, delete the old copy.

Used in modern file systems (WAFL, ZFS, ...).

Optimizing shadowing

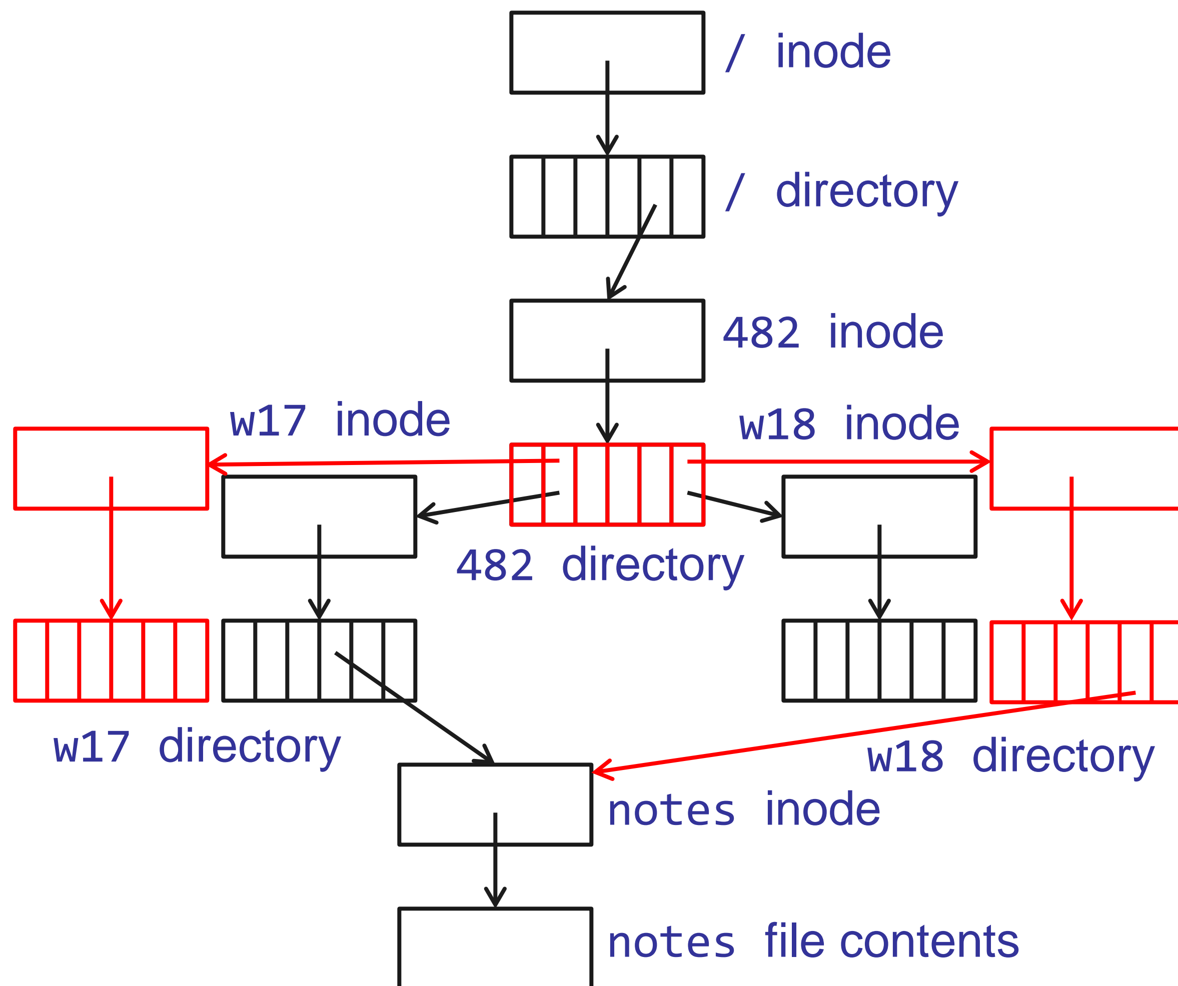
Block can store more than just a 1-bit pointer.

Example: move `notes` from `/482/w17/` to `/482/w18/`

Which blocks need to be updated?

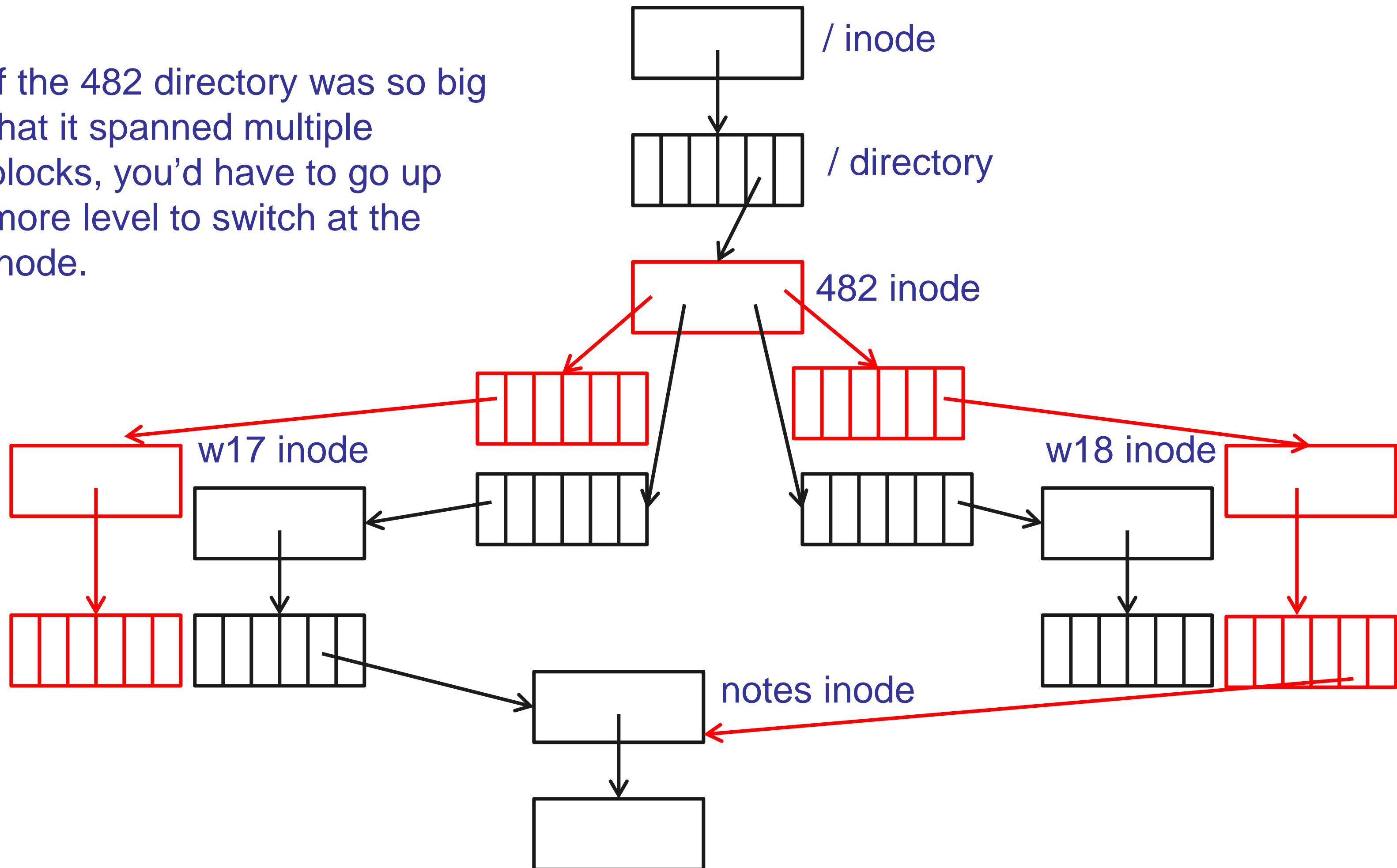
Which block can be updated in-place?

What if the 482 directory took more than one block?



Optimizing shadowing

If the 482 directory was so big that it spanned multiple blocks, you'd have to go up more level to switch at the inode.



Shadowing summary

Need to propagate shadowing up tree until you find a single block that can be changed.

- Find a common ancestor.
- May be the root of the file system.
  - For example, what if free block list persistent?
- Coalesce multiple transactions for efficiency.

Instead of deallocating, can keep old blocks.

- Snapshot (past version) of file system state.

Agenda

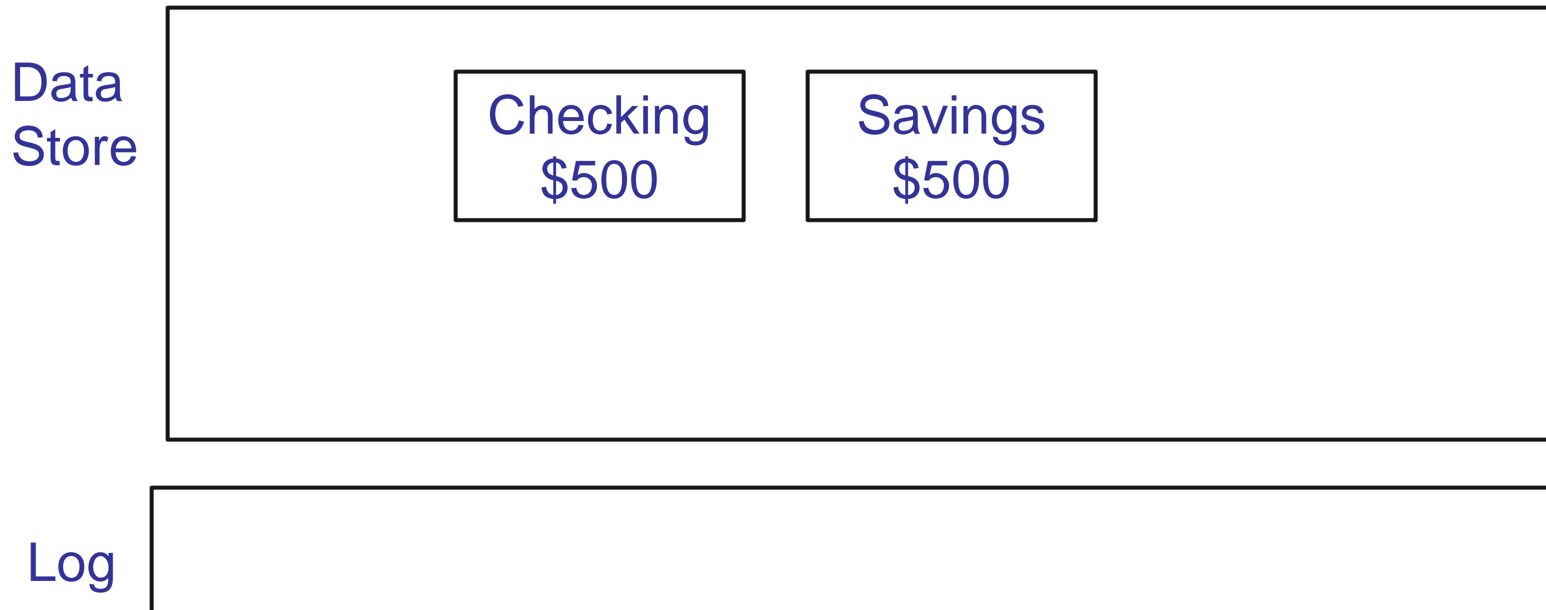
1. Ordering updates to a filesystem.
2. Shadowing.
3. Logging.
4. Log-structured filesystems (LFS).
5. Project 4 preview.

Transactions via Logging

Divide storage into:

Data store: Persistent copy of data.

Log: Sequential region that enables transaction updates.

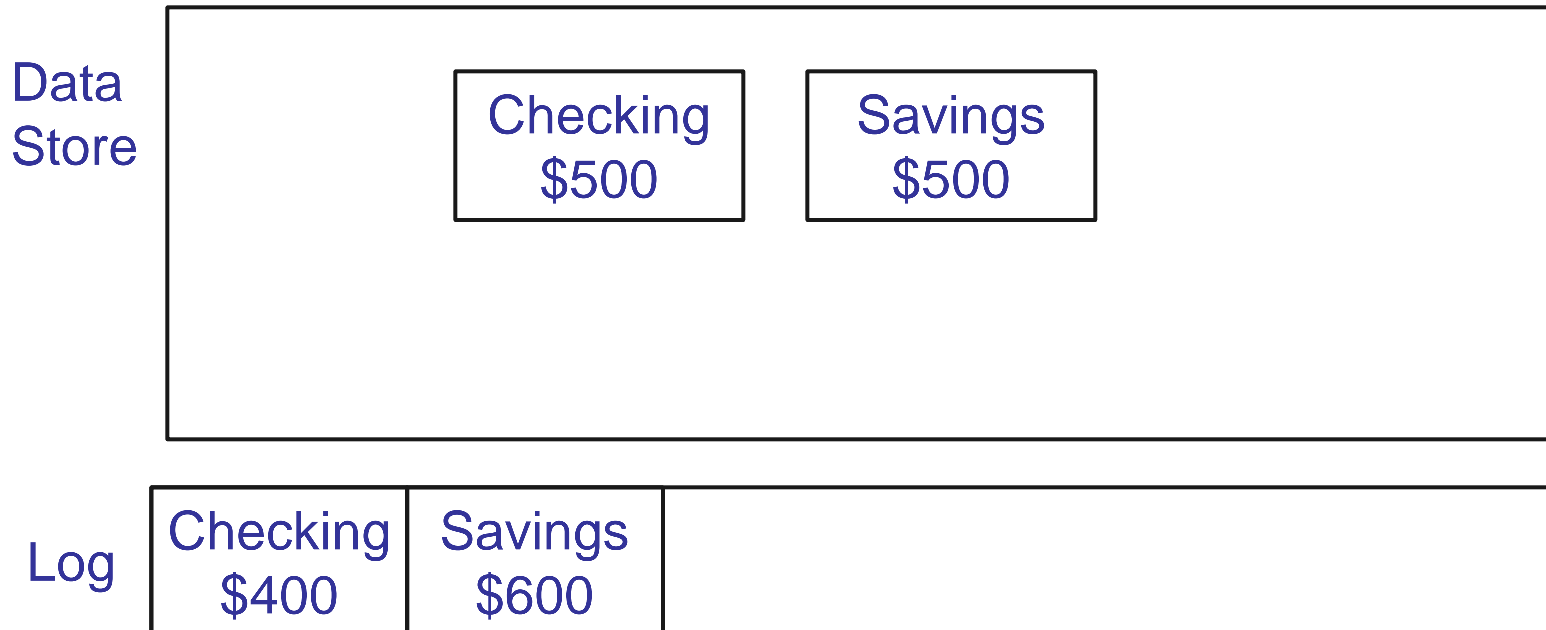


Logging example

Step 1: Append updates to log.

E.g., (LBN, data) tuples (value logging).

Data store not updated, so no changes if crash.

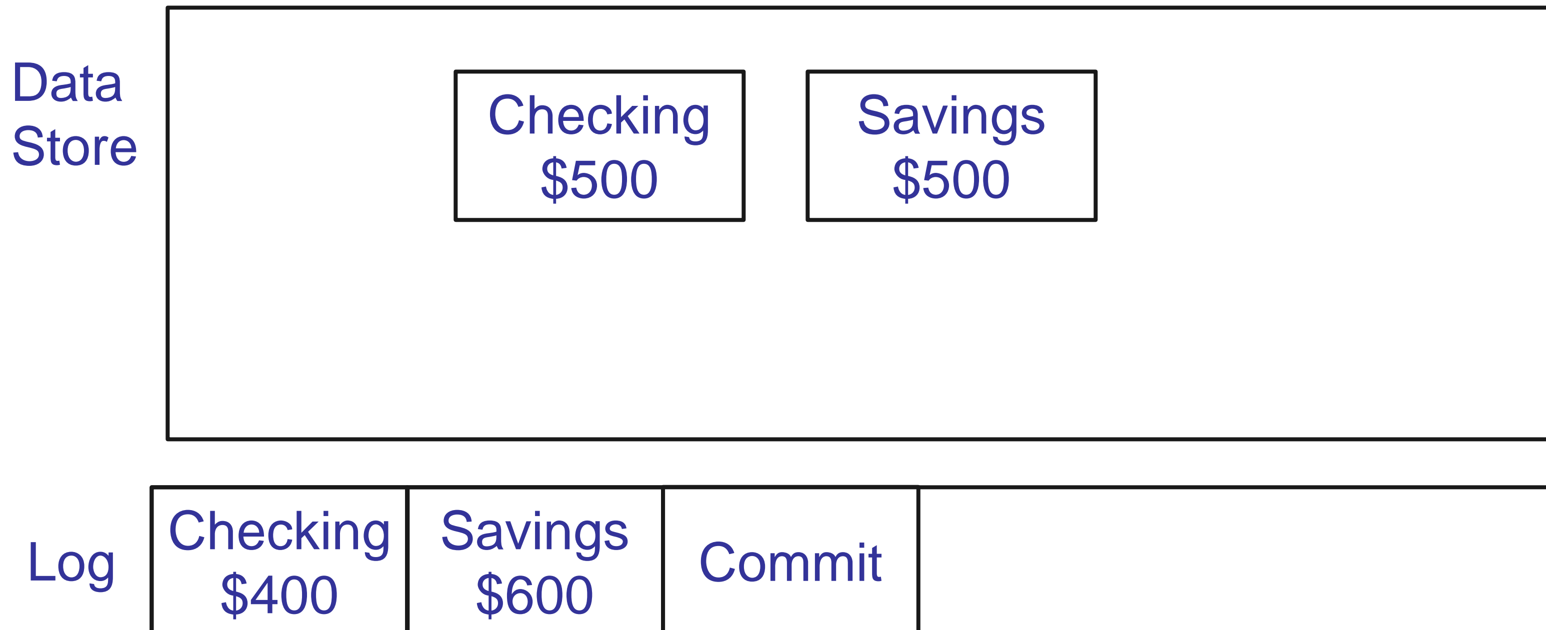


Logging example

Step 2: To commit transaction

Append "commit" record to log.

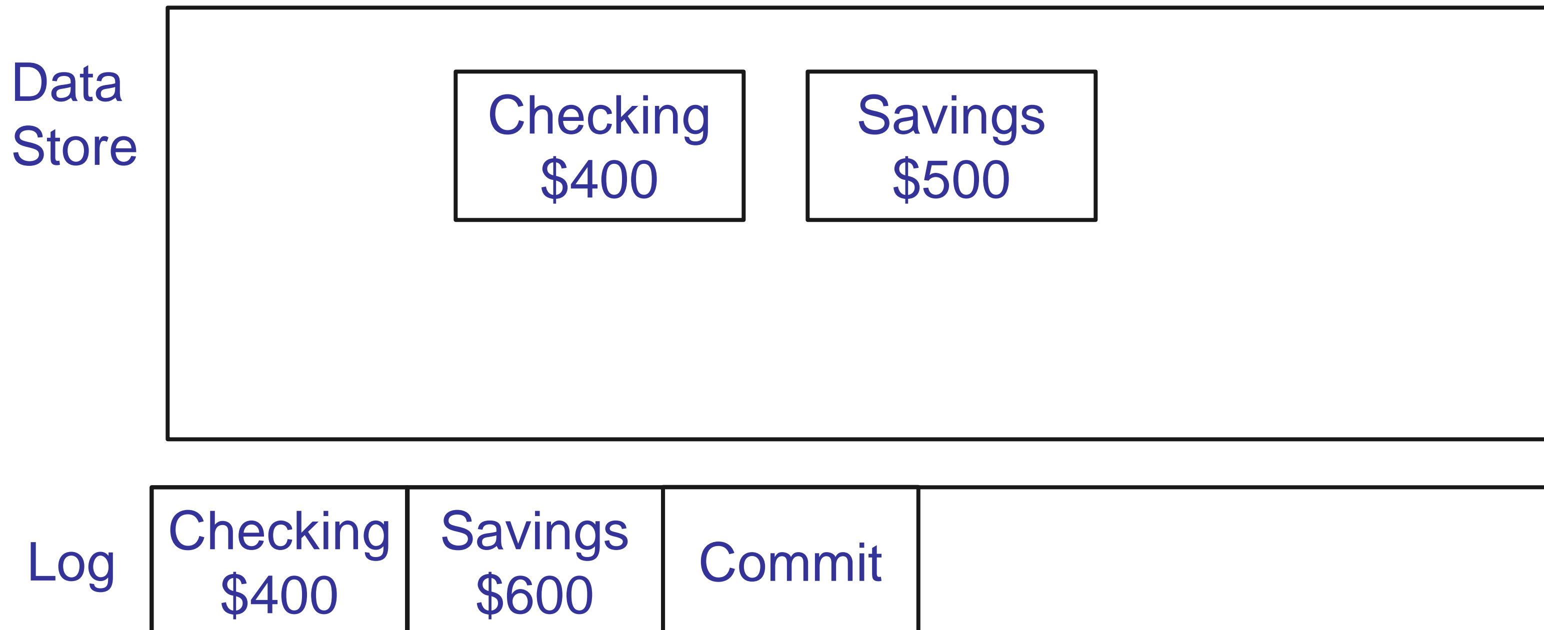
Step 3: Apply updates in log to data store.



Logging example

What if we crash before applying all updates?

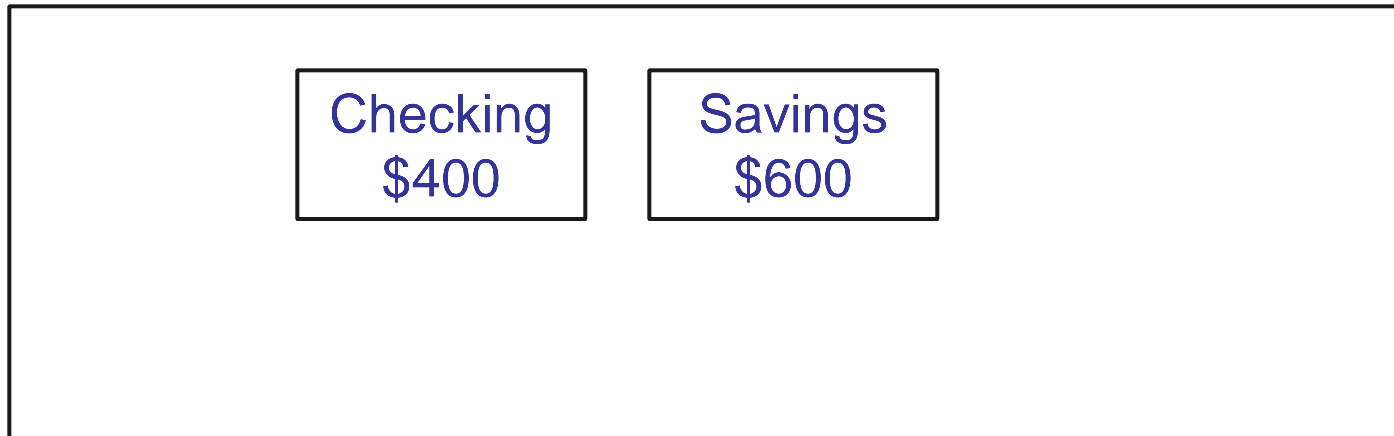
Upon restart, apply all updates in log until last commit record.



Logging example

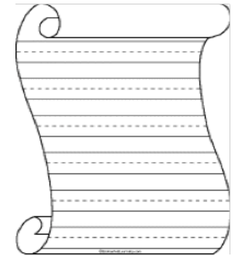
After applying updates:

Checkpoint log (remove records written to store).



Transactions with logging

Write updates to append-only log ***before*** updating file system



Write commit sector to end of log

Eventually, copy new data from log to file system

Transactions with logging

System crash before writing commit record?

Store unmodified, recovery ignores log records.

System crash after writing commit record, but before applying updates to data store?

Updates before commit record will be written to store during replay.

Transaction committed by single sector write.

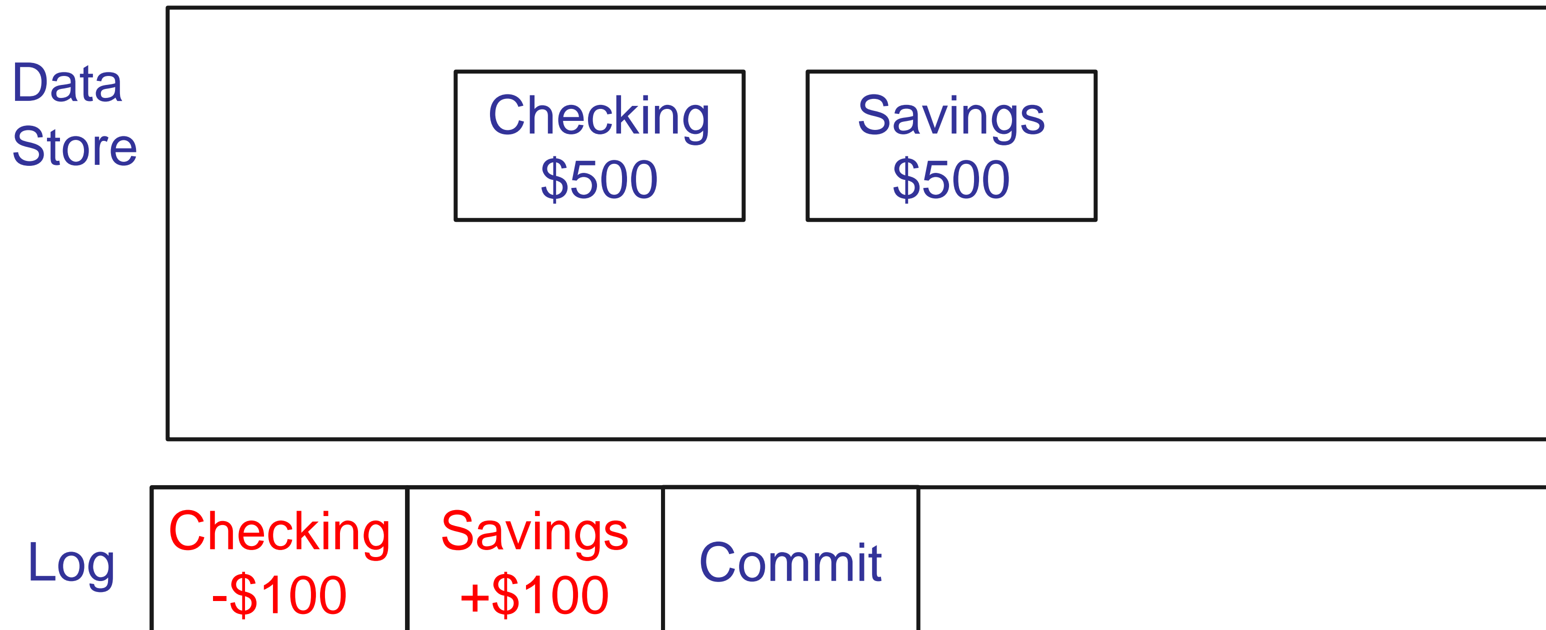
System crash while replaying log?

Format of log records

Why is logging in this form problematic?

Crash after updating checking = lose $100!

Log operations should be idempotent, meaning they can redone and produce the same result.



Journaling

- Many file systems implement transactions via logging.
  - Ext3, Ext4, NTFS, etc.
  - Often referred to as *journaling.*
- Journaling all updates felt to be too slow.
  - Why might this be?
    - Large file writes: 2x disk writes.
  - Ext4 has 3 modes:
    1. Journal all updates.
    2. Journal just the metadata (default).
    3. No journaling.

Agenda

1. Ordering updates to a filesystem.
2. Shadowing.
3. Logging.
4. Log-structured filesystems (LFS).
5. Project 4 preview.

Log-structured file system

Filesystem proposed by John K. Ousterhout and Fred Douglis in 1988.

First implemented in 1992 by Ousterhout and Mendel Rosenblum.

Log-structured file system

Goal: Try to do all the I/O to sequential blocks.

The application, not the OS, chooses what to read next, so we're forced to use caching for reads.

But we can write to any free block, meaning we can force writes into sequential blocks.

Basic idea: Treat the disk as an append-only log.

Append all writes to log, no data store.

What does it take to update the data in `/home/eecs/482/notes`?

LFS Write Example

What does it take to update the data in `/home/eecs/482/notes`?

1. Write data block for notes.

   But, now inode points to wrong block.

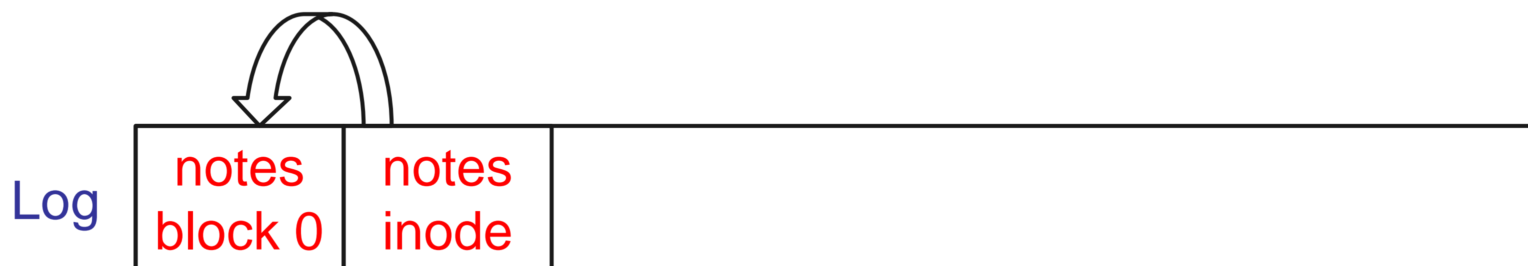
Log	notes block 0	

LFS Write Example

What does it take to update the data in `/home/eecs/482/notes`?

1. Write data block for notes.
2. Write inode for notes.

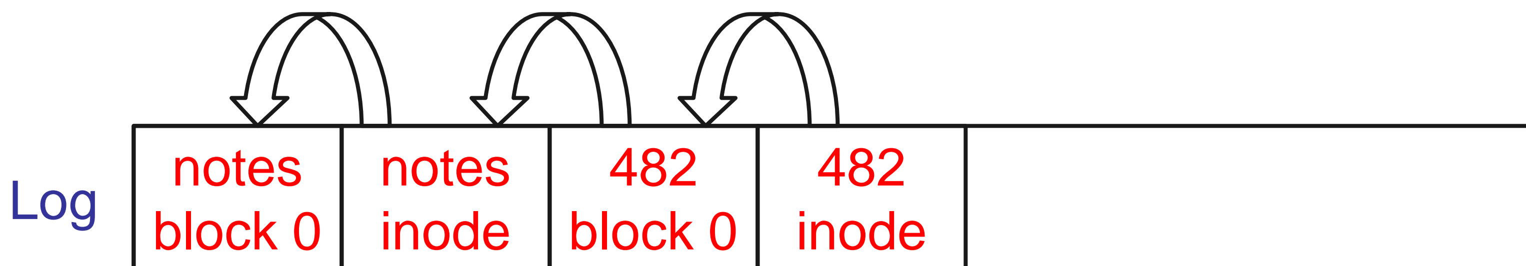
   But, now 482 directory contains wrong LBN.



LFS Write Example

What does it take to update the data in `/home/eecs/482/notes`?

1. Write data block for notes.
2. Write inode for notes.
3. Write data block, inode for 482.
4. Continue all the way up to root inode.



Finding data in LFS

New data structure: inode map (indirection!)

- Directory entries contain inode number.
- inode map translates inode number to disk block.

inode map is periodically checkpointed.

- Cached in memory for performance.

LFS: Garbage collection

LFS append-only quickly runs out of disk space.

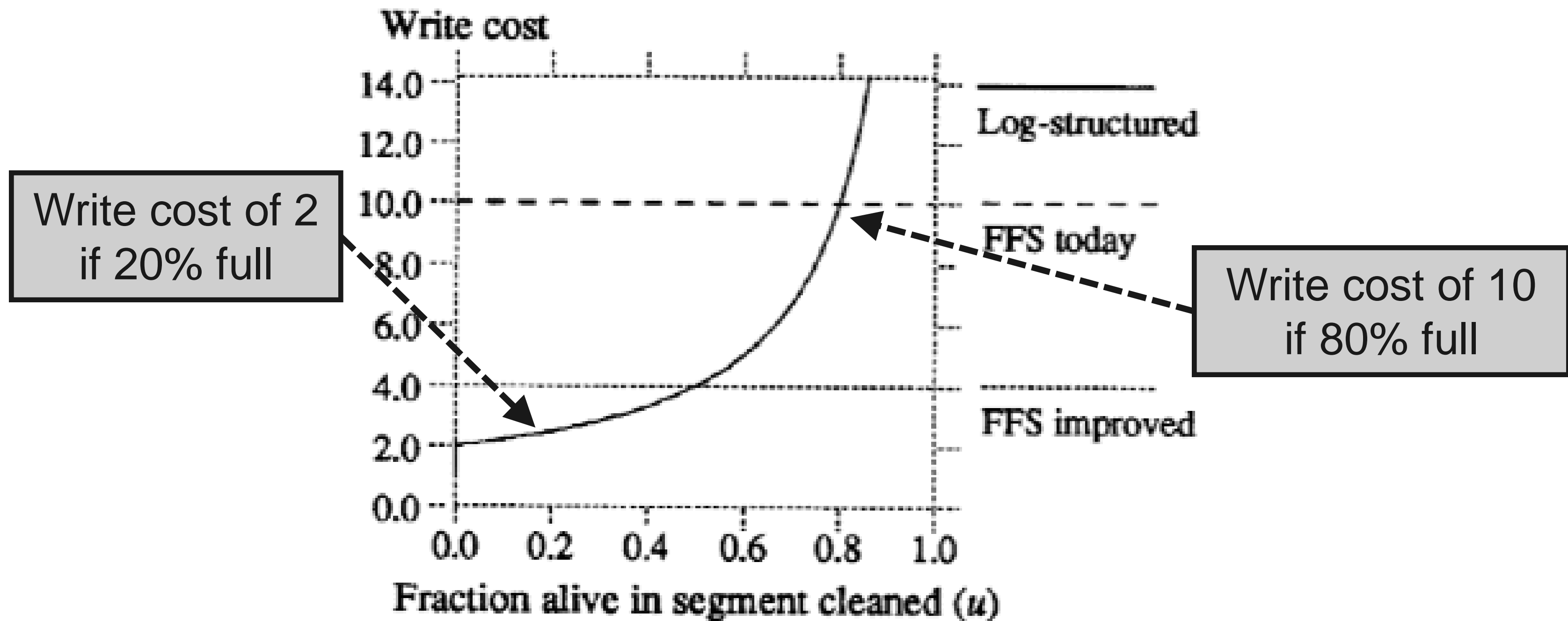
- Overwriting, deletion creates garbage.
- Need an efficient garbage collector (cleaner).

LFS divides log into large *segments*.

- Choose clean segment, write sequentially.
- Background cleaner creates new clean segments.
  - Read in full segments, Copy live data to end of log.

Cleaning is expensive for high utilization.

Write Cost Comparison



LFS on SSDs

LFS rarely used for hard drives.

But the characteristics of SSDs a perfect match for LFS.

1. Random reads very cheap, writes expensive.
   LFS optimizes for write performance.
2. Need to erase large chunks before overwrite.
   LFS log cleaning enables background erase.
3. SSDs have wearout after too many writes.
   Log structure does automatic wear leveling.

Flash Translation Layer essentially an LFS.

Agenda

1. Ordering updates to a filesystem.
2. Shadowing.
3. Logging.
4. Log-structured filesystems (LFS).
5. Project 4 preview.

Project 4: due August 17

Secure, multi-threaded network file server

Network programming, file systems, client-server, threads/concurrency, even a little security.

Experience writing significant concurrent program.

Good news: concepts simpler than projects 2 and 3.

Bad news: Perhaps 3x as much code as project 3.

Make sure to try out Friday’s lab questions.