EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Multiple updates and reliability

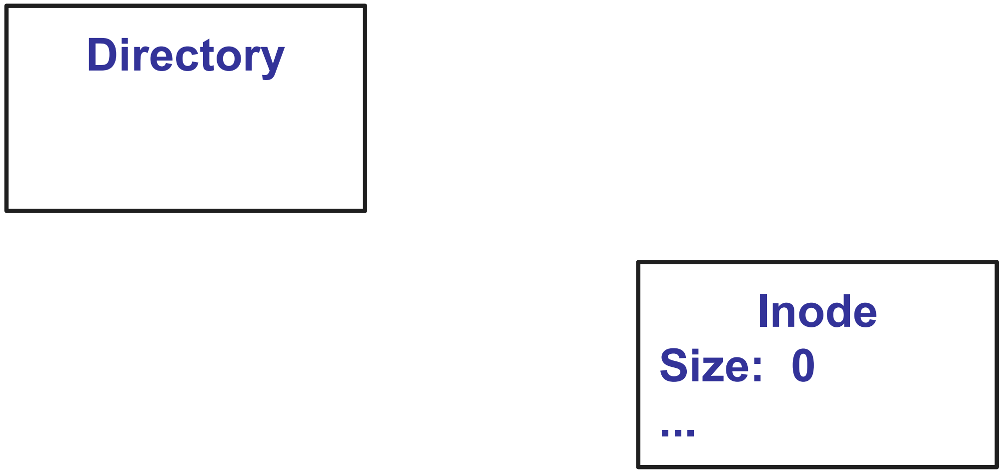
- Data must survive crashes and power outages
  - Assume: update of one block atomic and durable
  - Challenge: Crashes in midst of multi-step updates

- 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 dir harshavm
  - Create inode first



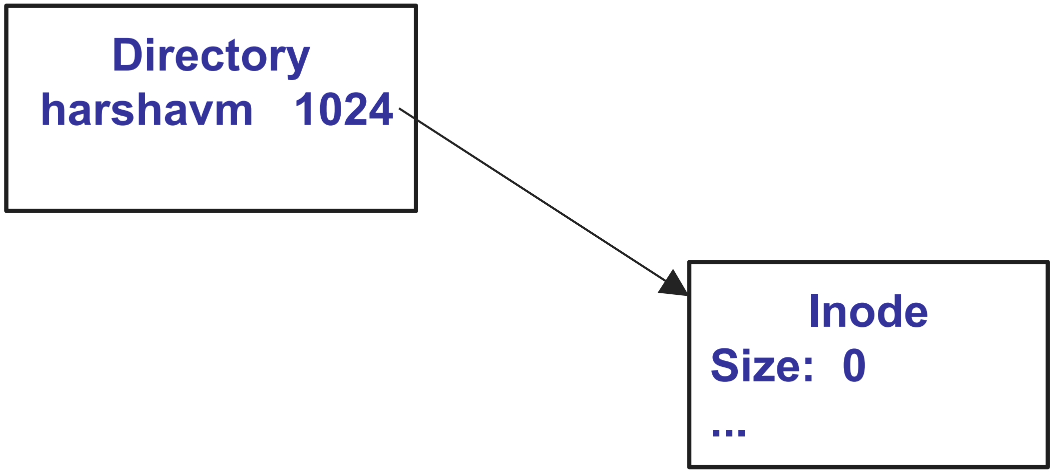
Directory

Inode
Size: 0
...

OK to modify unreachable blocks on disk

Ordering of updates

- Careful ordering can fix some problems:
  - For example, creating file 482.txt in dir harshavm
  - Create inode first, then link to it



Careful ordering goes from one consistent state to another

Ordering not always enough

- Example: Create file and update 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

Transactions

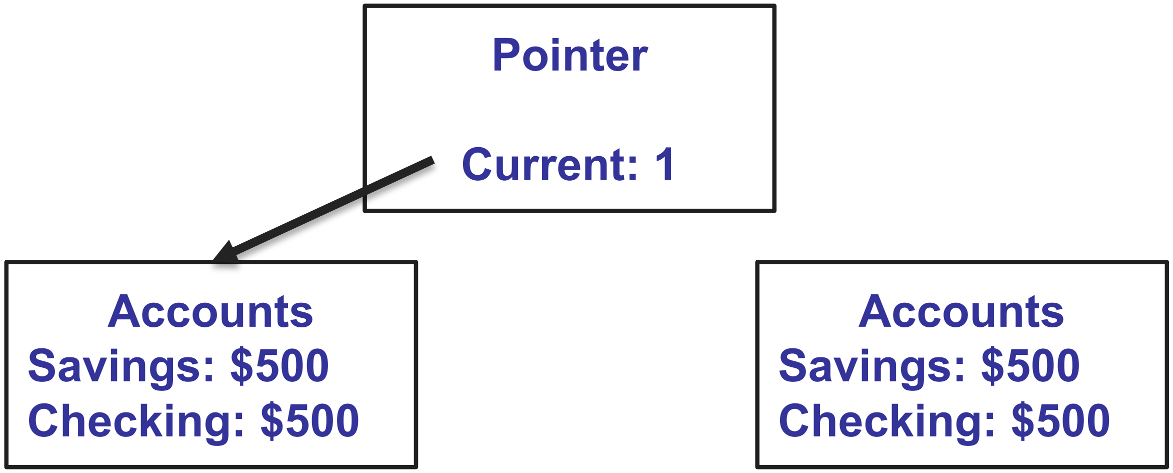
- Commonly used to mean ACID property
- Main aspect for file systems: atomicity and durability (all or nothing)

```
begin
    write disk
    write disk
    write disk
end (this "commits" the transaction)
```

- Writes to single sector to disk are atomic
  - How to make a sequence of updates atomic?
  - Two main methods: shadowing and logging

Shadowing

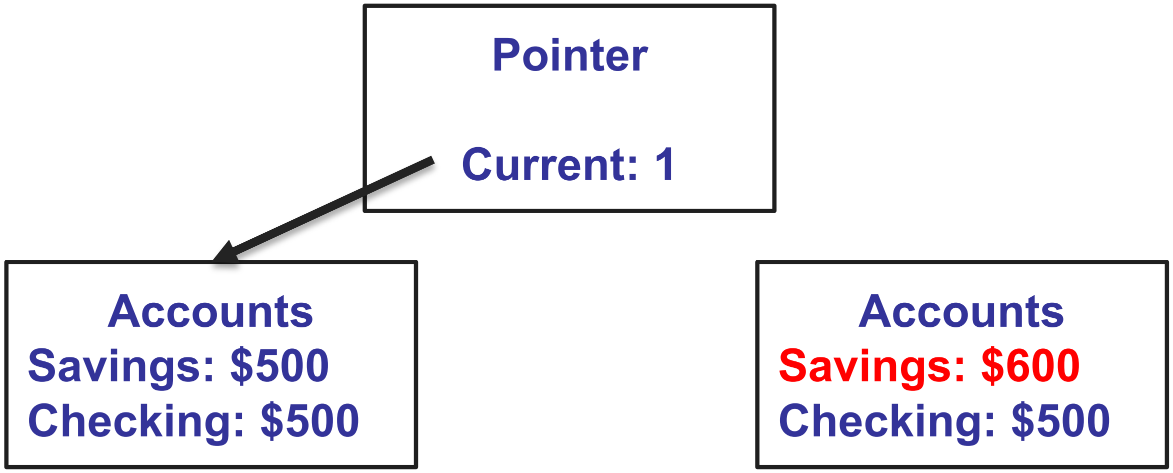
- Replicate the data across two stores:
  - One is current version, other is backup
  - Current pointer points to the current version



At beginning of transaction, both replicas are identical

Shadowing

- Transaction updates the backup (shadow)
  - First add $100 to savings



Pointer

Current: 1

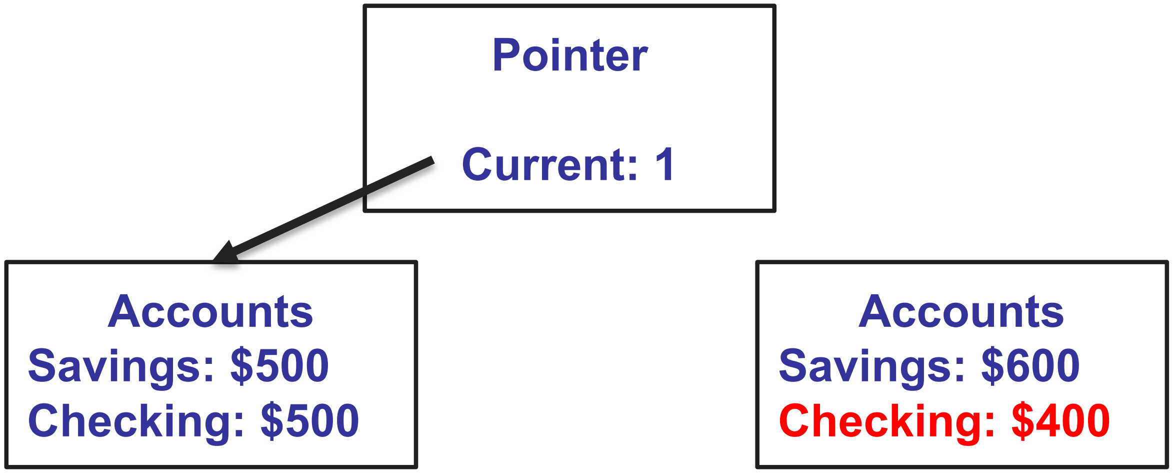
Accounts
Savings: $500
Checking: $500

Accounts
Savings: $600
Checking: $500

Note: modifying “unreachable” block

Shadowing

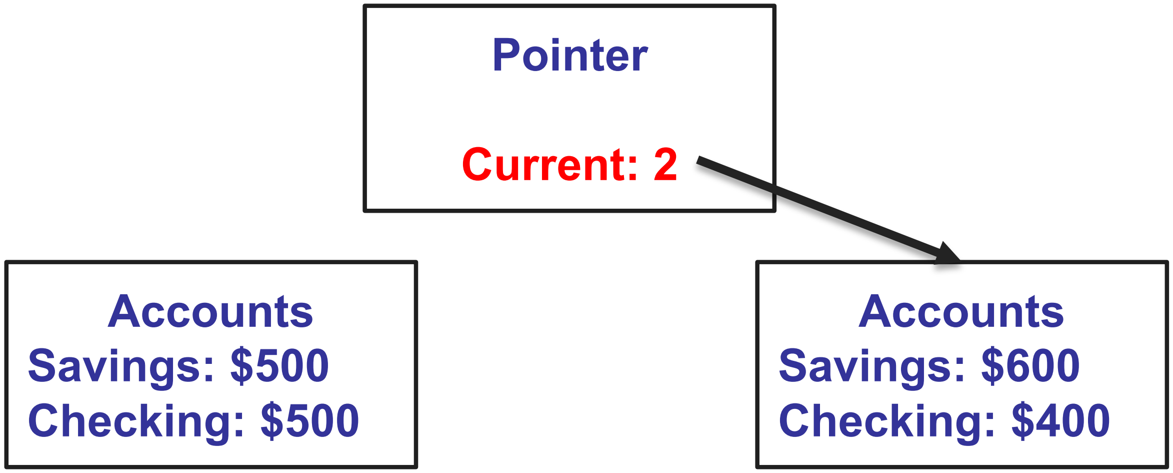
- Transaction updates the backup (shadow)
  - Next remove $100 from checking



Note: modifying “unreachable” block

Shadowing

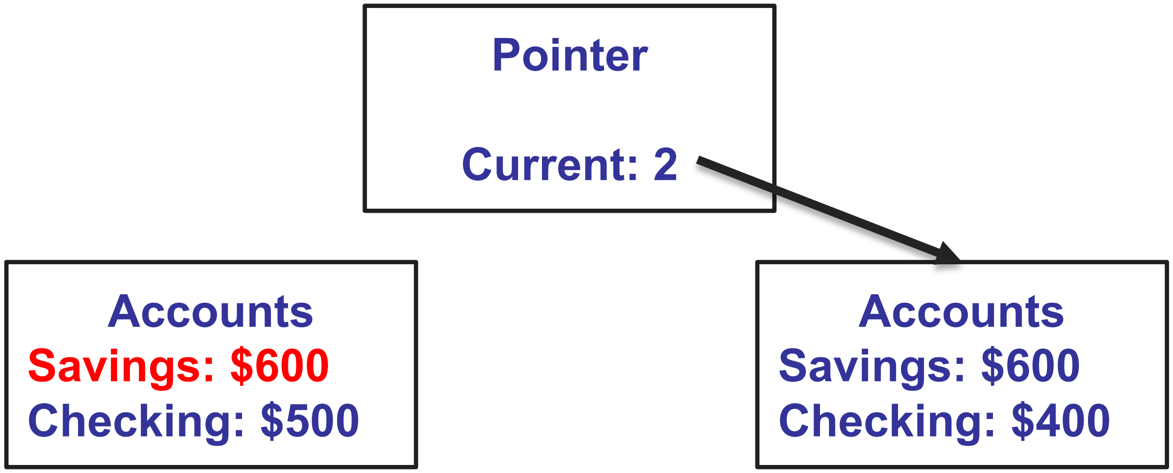
- Transaction commit switches the pointer
  - At this point updates become durable



Note: updating single block = atomic update

Shadowing

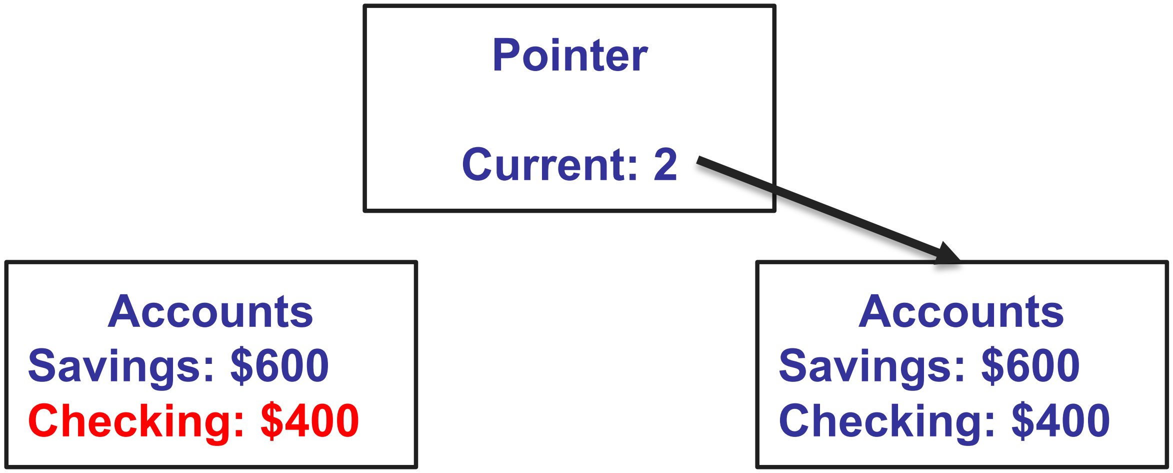
- Finally, must update new shadow
    - First, update savings



Note: again, updating unreachable block

Shadowing

- Finally, must update new shadow
  - Next, update checking



Pointer

Current: 2

Accounts
Savings: $600
Checking: $400

Accounts
Savings: $600
Checking: $400

Note: again, updating unreachable block

Shadowing summary

- Can make arbitrary set of updates in txn
  - Pointer switch is always atomic commit

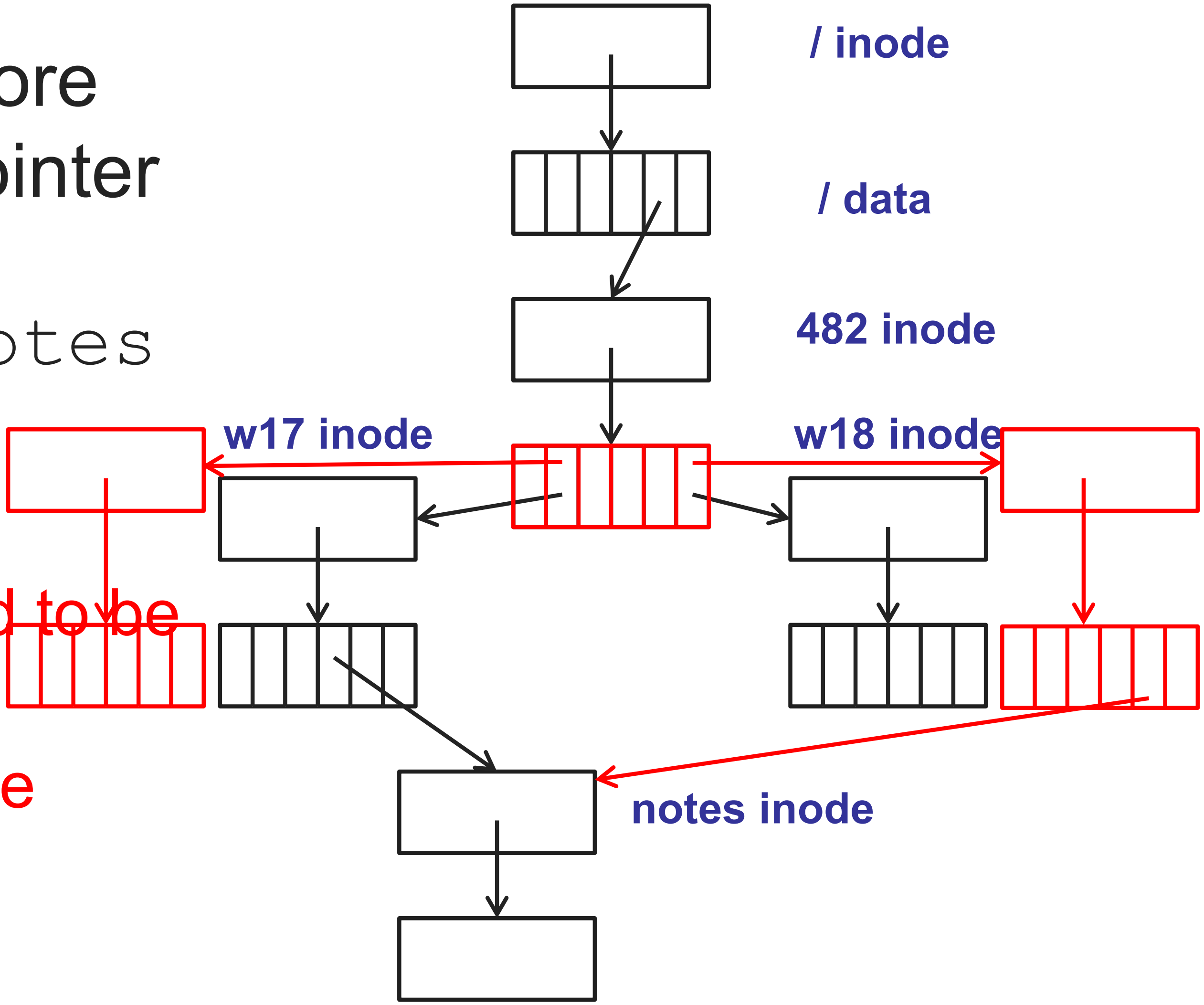
- Downside?
  - Requires replicating data store

- Can reduce cost by shadowing on demand
  - Sometimes called shadow paging
  - Used in modern file systems (WAFL, ZFS, ...)

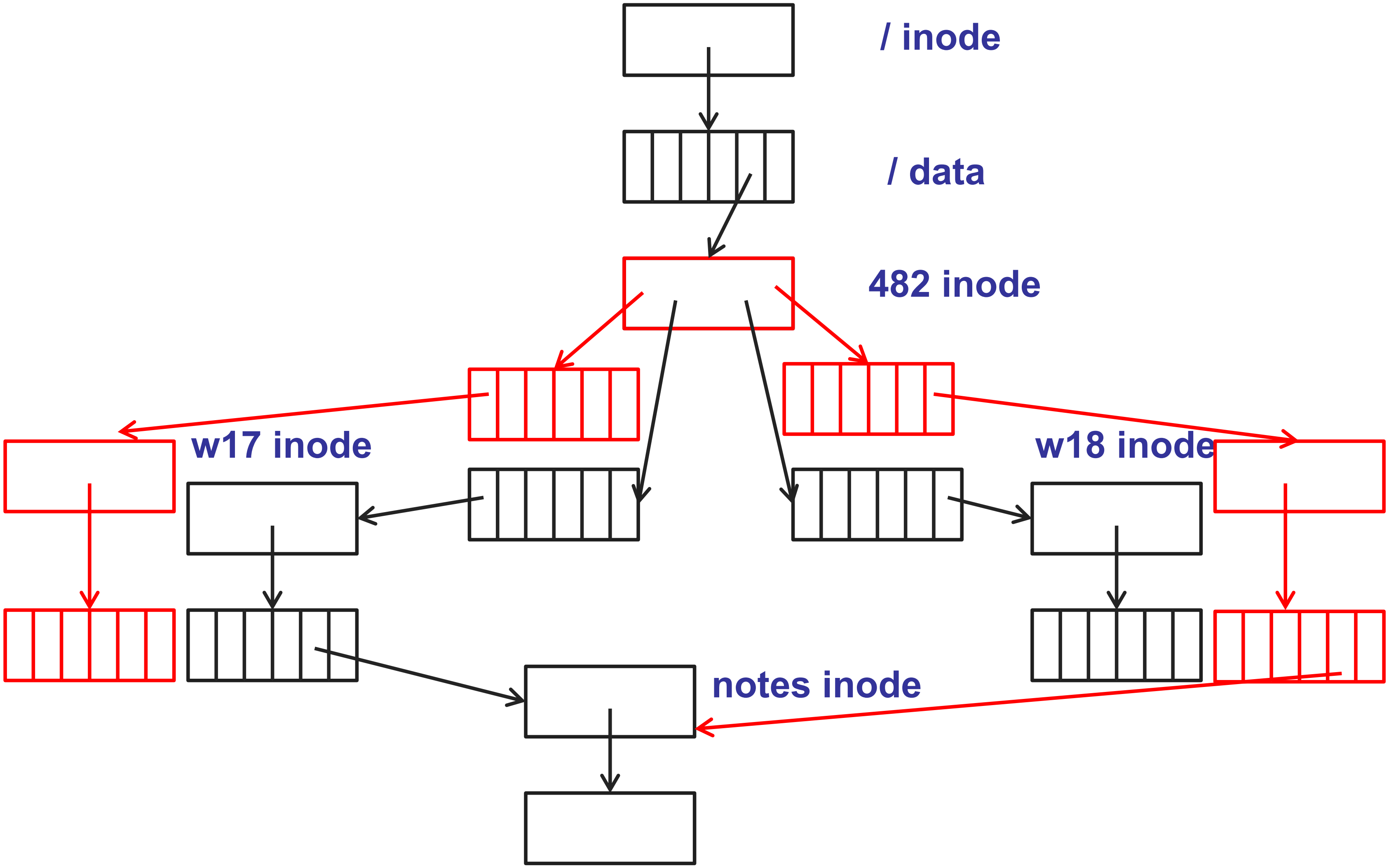
Optimizing shadowing

- 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?



Optimizing shadowing



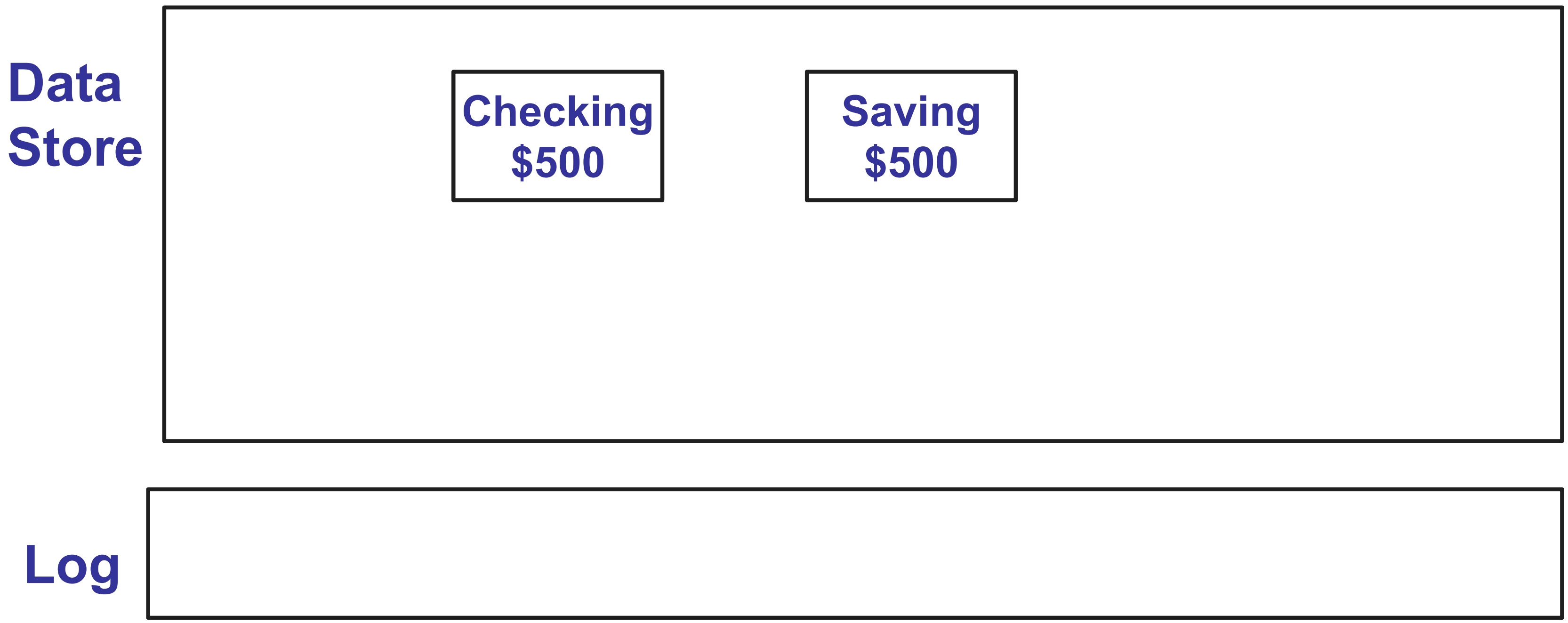
Shadowing summary

- Need to propagate shadowing up tree
  - Can stop at common ancestor
  - May be 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

Transactions via Logging

- Divide storage into:
  - Data store: Persistent copy of data
  - Log: Sequential region that enables txn updates



Data Store

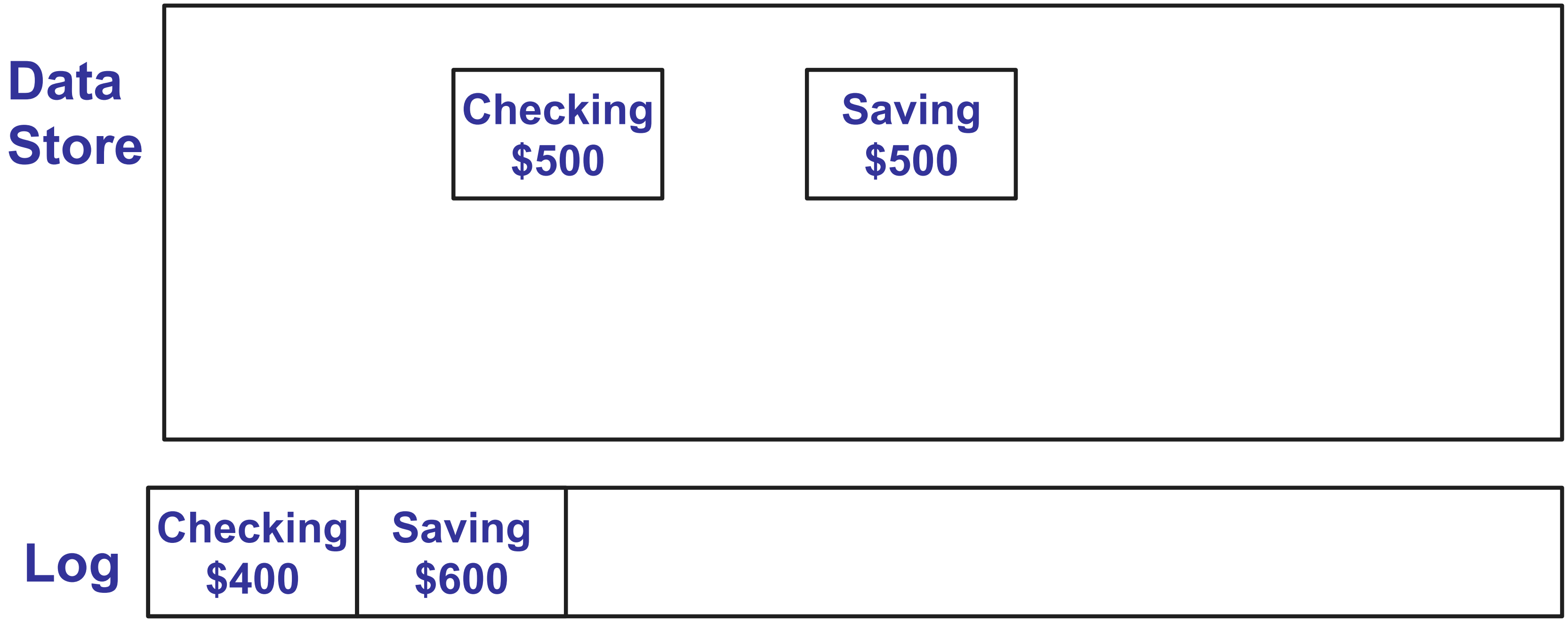
Checking
$500

Saving
$500

Log

Logging example

- Step 1: Append updates to log
  - E.g., <LBN, data> tuples (value logging)
  - Data store not updated, so no changes if crash



Data Store

Checking $500

Saving $500

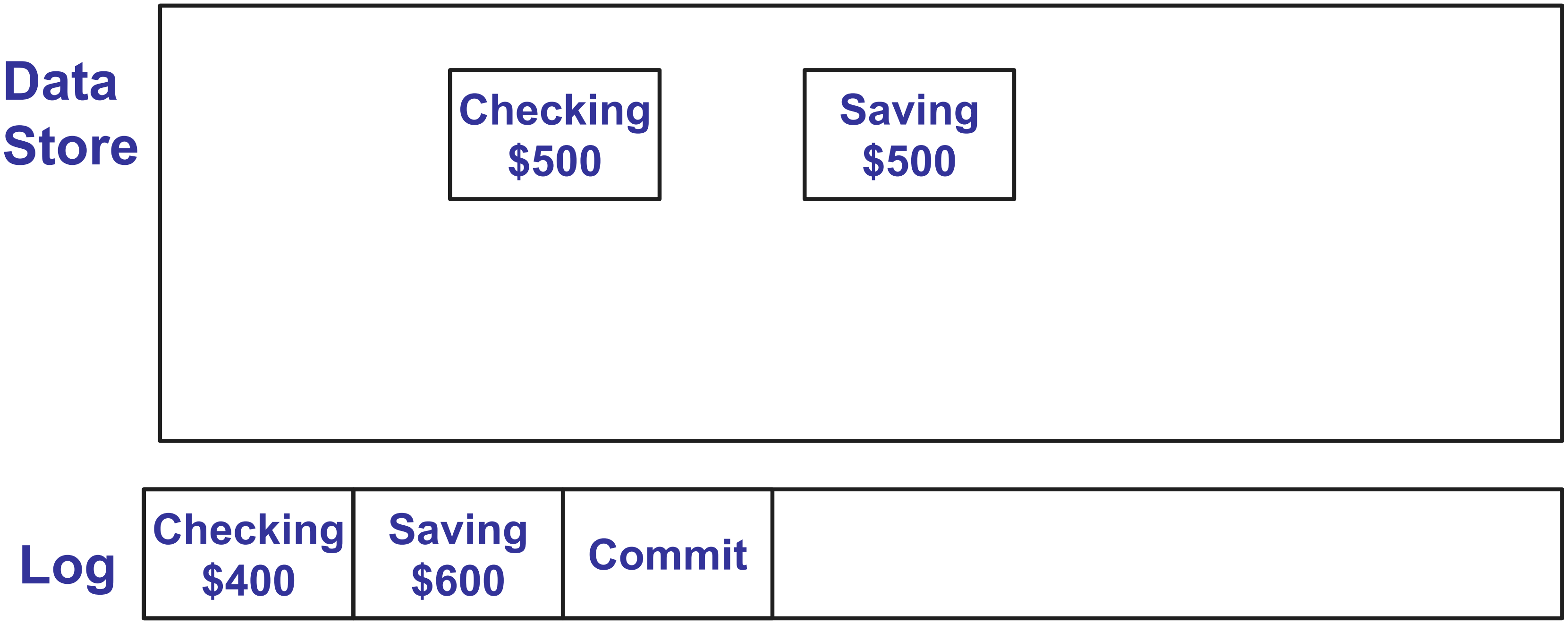
Log

Checking $400

Saving $600

Logging example

- Step 2: To commit transaction
  - Append “commit” record to log
- Step 3: Apply updates in log to data store



Data Store

Checking $500

Saving $500

Log

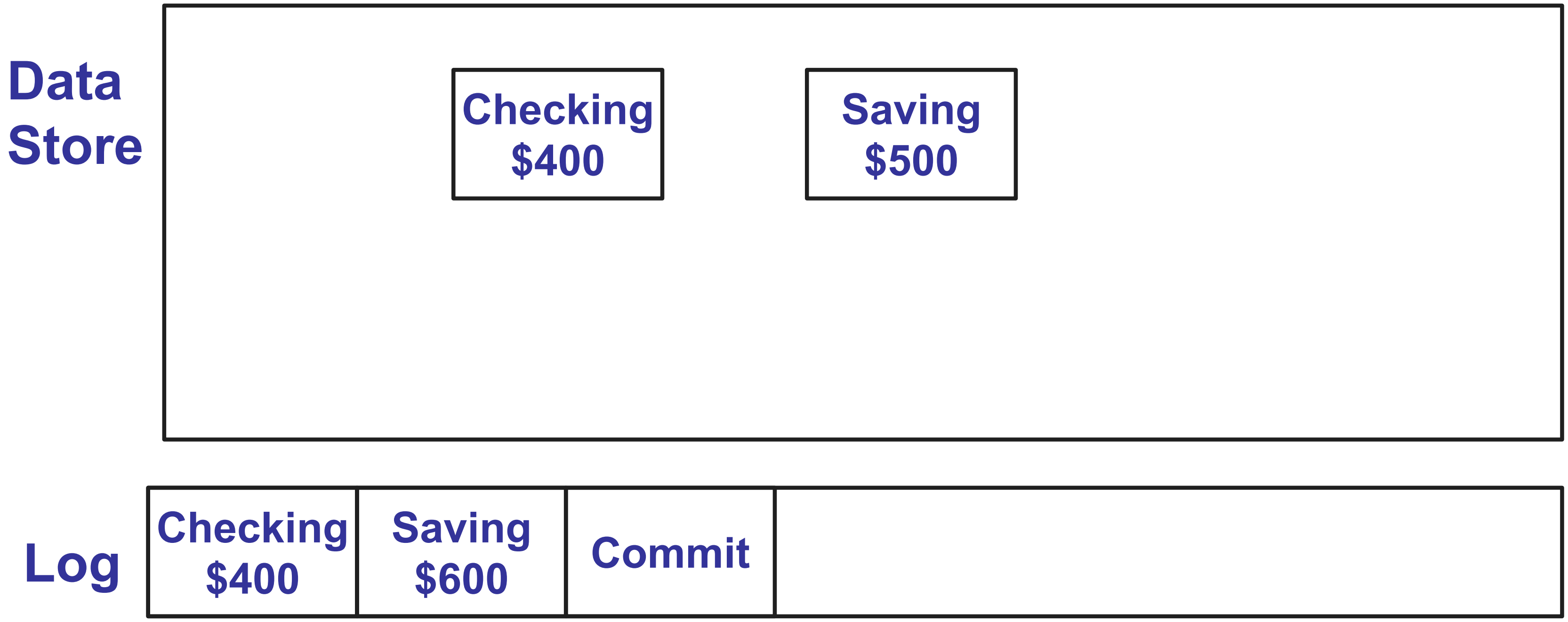
Checking $400

Saving $600

Commit

Logging example

- What if we crash before applying all updates?
  - Upon restart, apply all updates in log until last commit record



Data Store

Checking $400

Saving $500

Log

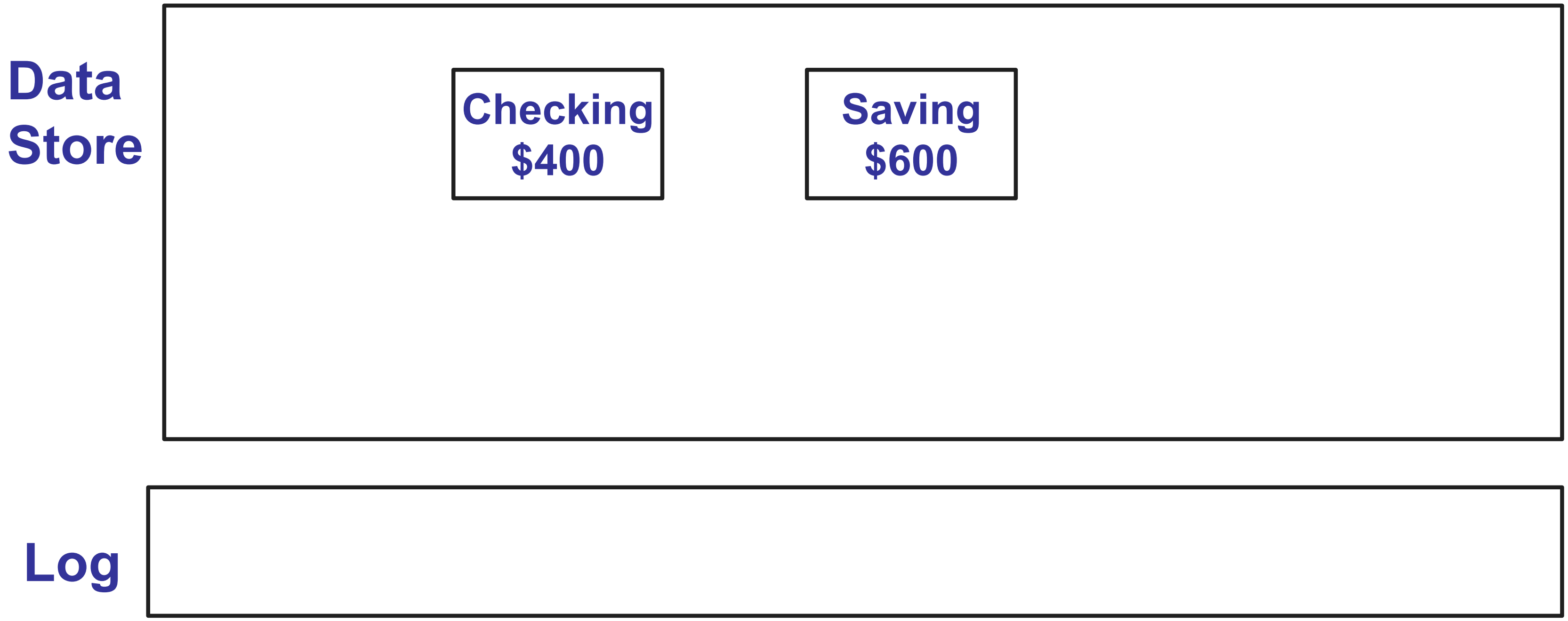
Checking $400

Saving $600

Commit

Logging example

- After applying updates
  - Checkpoint log (remove records written to store)



Data Store

Checking $400

Saving $600

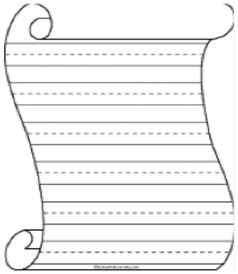
Log

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

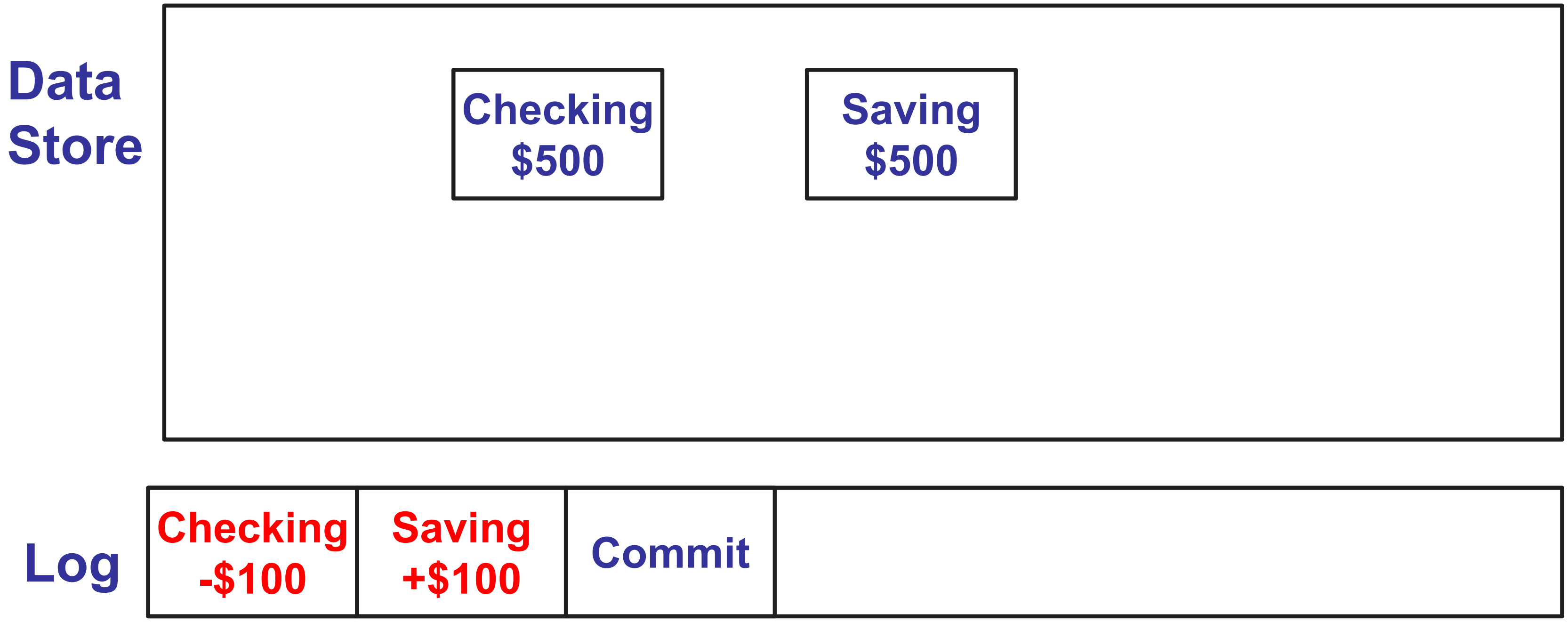


Transactions with logging

- 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 the following logging incorrect?
  - Crash after updating checking = lose $100!
  - Log operations should be idempotent



Data Store

Checking $500

Saving $500

Log

Checking -$100	Saving +$100	Commit	

Journaling

- Many file systems implement txn 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:
    - » Journal all updates
    - » Journal just metadata (**default**)
    - » No journaling

Project 4

- 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 project 3
- Bad news: more code than project 3

- Make sure to try out Friday's lab question

Log-structured file system

- Goal: Make (almost) all I/Os sequential
  - File system can write to any free disk block
  - In general, not possible for reads; leverage caching

- Basic idea: Treat disk as an append-only log
  - Append all writes to log, no data store

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

LFS Write Example

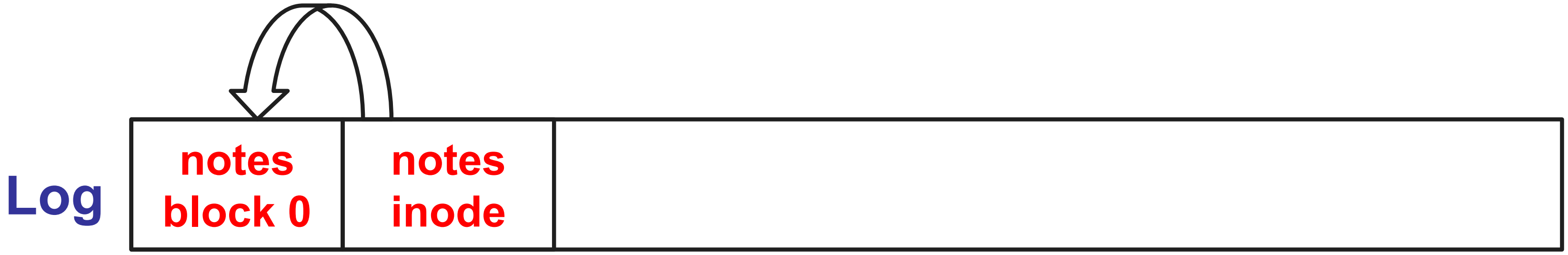
- What does it take to update the data in `/home/harshavm/482/notes`?
  1. Write data block for notes
     - » But, now inode points to wrong block

Log

notes block 0

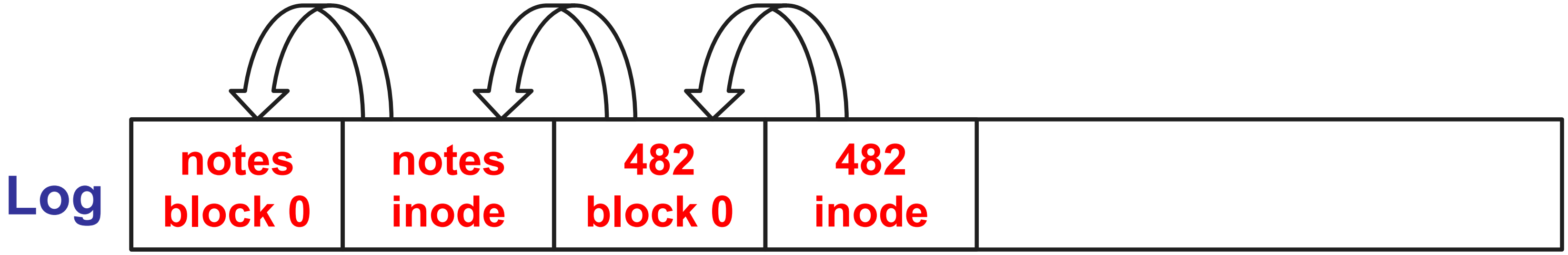
LFS Write Example

- What does it take to update the data in `/home/harshavm/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/harshavm/482/notes`?
  1. Write data block for notes
  2. Write inode for notes
  3. Write data block, inode for 482
  4. Etc. all the way to root inode



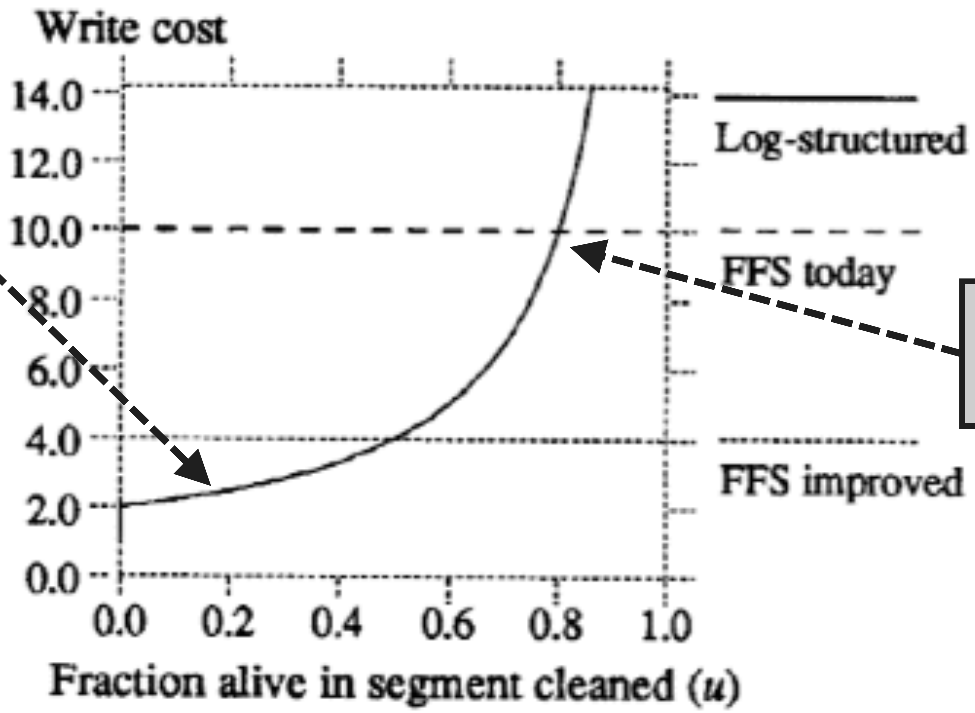
Finding data in LFS

- 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



Write cost of 2 if 20% full

Write cost of 10 if 80% full

LFS on SSDs

- LFS rarely used for hard drives
- But characteristics of SSDs perfect for LFS
  - Random reads very cheap, writes expensive
    - » LFS optimizes for write performance
  - Need to erase large chunks before overwrite
    - » LFS log cleaning enables background erase
  - SSDs have wearout after too many writes
    - » Log structure does automatic wear leveling
- Flash Translation Layer essentially an LFS