6.181: Filesystems (pt. 2)

Adam Belay <abelay@mit.edu>

Agenda

- Last week:
  - Filesystem basics
  - Filesystem implementation in xv6
- Today:
  - Crash recovery
  - Issue: Crashes leave disk in inconsistent state
  - Solution: Logging
- Note: Last lecture on xv6 is today
  - Up next: Research papers on OSes

What is crash recovery?

- Imagine you are using the filesystem
- Power is suddenly lost
- The system reboots
- Is the filesystem still usable?
- Is your data still there?

Why is this a hard problem?

- Filesystems perform multi-step operations
  - E.g., reserve an inode, then reserve a bit in the bitmap, then update a directory, then fill in an inode, then write data, etc.
  - A crash could leave invariants violated
- After rebooting:
  - Bad outcome: Crash again due to corrupt FS
  - Worse outcome: Silently read/write invalid data

Suppose we create a file

$ echo hi > x

// block write trace from last week

```
bwrite: block 33 by ialloc     // allocate inode (block 33)
bwrite: block 33 by iupdate    // update inode (e.g., set nlink)
bwrite: block 46 by writei     // write directory entry
bwrite: block 32 by iupdate    // update directory inode with new len
```

Suppose we create a file

$ echo hi > x

// block write trace from last week

bwrite: block 33 by ialloc	// allocate inode (block 33)
bwrite: block 33 by iupdate	// update inode (e.g., set nlink)
bwrite: block 46 by writei	// write directory entry
bwrite: block 32 by iupdate	// update directory inode with new len

Crash

What happens?

- Not much bad happens
- Inode allocated and wasted, never usable in future

What about this order?

$ echo hi > x

// block write trace from last week

bwrite: block 46 by writei // write directory entry

Crash

bwrite: block 32 by iupdate // update directory inode with new len

bwrite: block 33 by ialloc // allocate inode (block 33)

bwrite: block 33 by iupdate // update inode (e.g., set nlink)

What happens?

- Disaster!
- Inode could be reallocated again
- Directory points to uninitialized inode

What order could really happen?

What order could really happen?

- Kernel (and maybe the disk too) reorders writes to minimize seeks
- In general, any order is possible
- Similar issue to memory model and locking in prior lecture

What about write?

1. inode addrs[] and len
2. indirect block
3. block content
4. block free bitmap

crash: inode refers to free block -- disaster!

crash: block not free but not used -- not so bad

What about unlink?

1. block free bitmaps
2. free inode
3. erase dirent

What should we hope for?

After reboot, run recovery code

1. Internal FS invariants must hold
   - e.g., no block is both free and used by an inode
2. All but the last few operations stored on disk
   - Data I wrote yesterday should be there!
   - But perhaps data at the time of crash will be lost
3. No reordering of data writes
   - echo 99 > result ; echo done > status

Correctness and performance

- Often at odds with one another!
- Disk writes are very slow
- Safety: Write data right away
- Speed: Wait and batch together writes

Crash recovery

- Arises in all storage systems, e.g., databases
- Many clever solutions
- Performance/correctness tradeoffs

Solution today: Logging

- Very popular, also known as journaling
- Goal: Atomic system calls w.r.t. crashes
- Goal: Fast recovery (no hour-long fsck)

- xv6: minimal design for safety
- ext3: adds more speed

Logging basics

- Atomicity: All of system call's writes applied or none are
- Each atomic op is called a transaction

Three phase operation:

1. Log phase: Record all the writes the system call will perform on disk
2. Commit phase: Record done on disk
3. Install phase: Do the actual disk writes

Crash recovery w/ logging

- If ”done” found in log, replay all writes
- If “done” not found, ignore entries in log
- Called write-ahead logging

Rules:

- install ***none*** of a transaction's writes to disk
- until ***all*** writes are in the log on disk, and the logged writes are marked committed

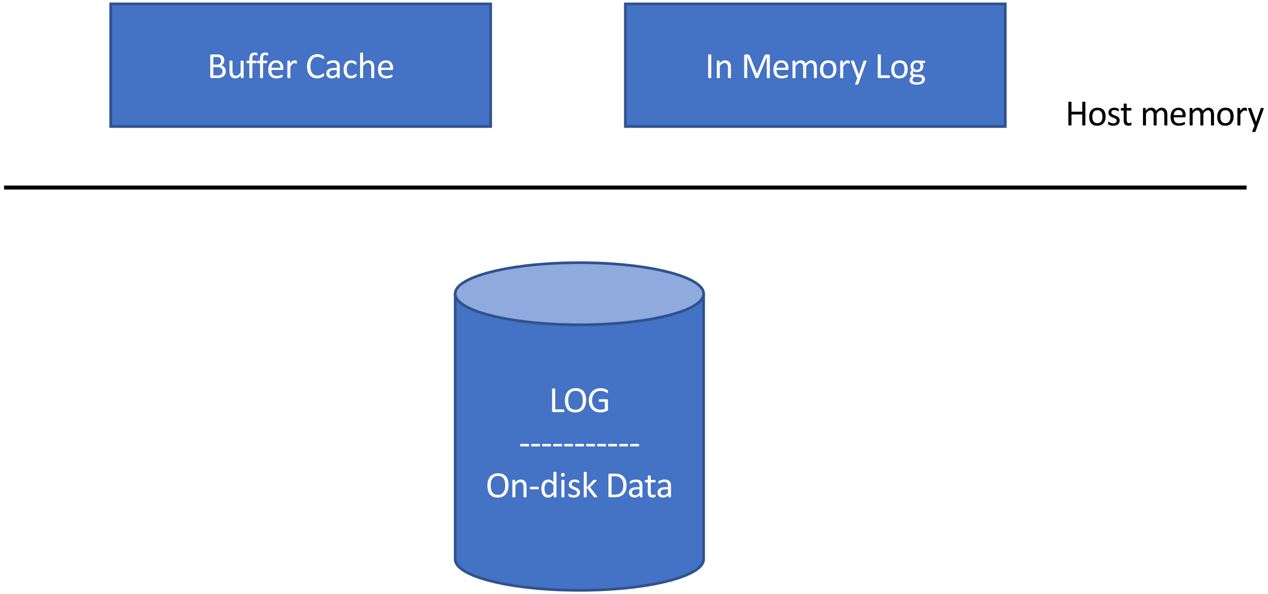
Why this approach?

- One we’ve installed a transaction on disk…
- We have to do all its writes
- This ensures the transaction is atomic
- Log allows us to detect if all steps in the transaction are there
- If not, we can safely abort the transaction

The magic of logging

- Crash recovery of complex mutable data structures is hard
- But logging makes it easy, can retrofit on top of existing FS designs
- Compatible with high performance too (w/ some effort)

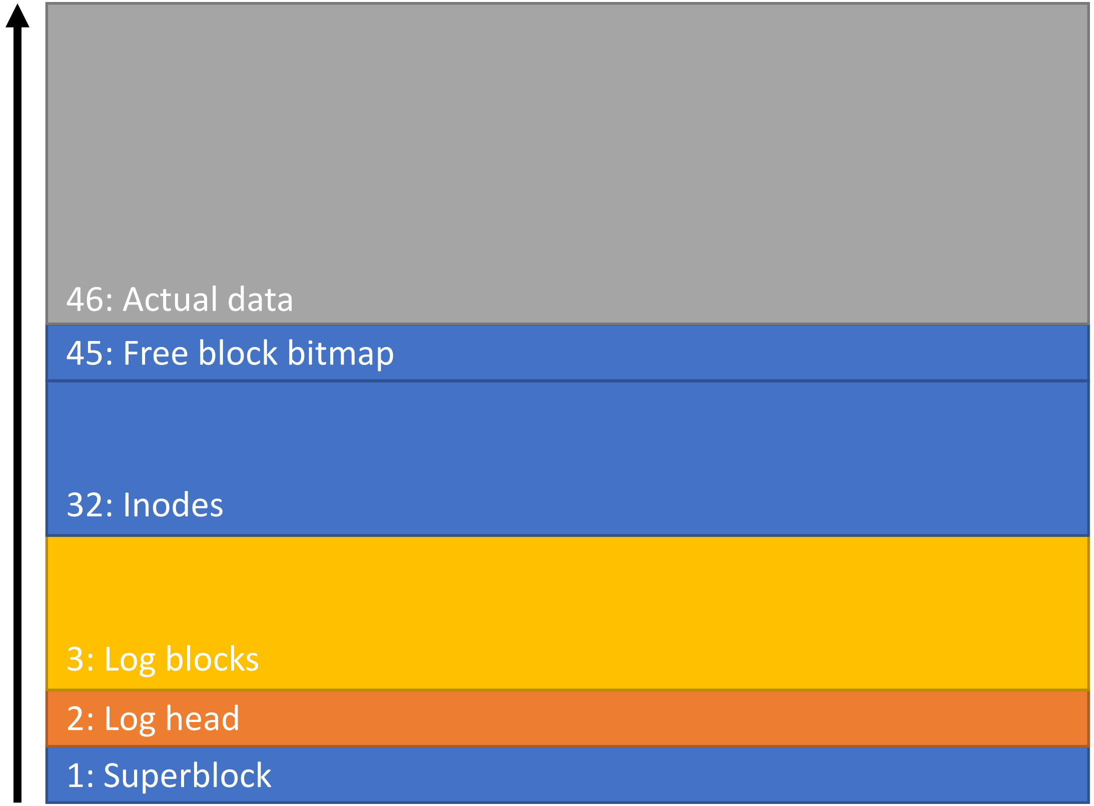
Logging in xv6



xv6 logging steps

- On write:
  - Add blockno to in-memory array
  - Keep the data itself in the buffer cache (pinned)
- On commit:
  - Write buffered log to disk
  - Wait for disk to complete writing (synchronous)
  - Write the log header sector to disk
- After commit:
  - install (write) the blocks in the log to their location in FS
  - Unpin the blocks in the buffer cache
  - Write zero to the log header sector on disk

Recall: On-disk layout



Log header?

- An “n” value on disk indicates the commit point
- Nonzero: Indicates a valid transaction is committed on disk
- Zero: Not committed, may not be complete
- Also records block #s that were updated
  - Why?
- And the number of blocks in log

Demo: xv6 logging

What happened?

1. Create inode
2. Write ‘hi’ to file x
3. Write '\n’ to file x

Create file x

```
bwrite 3     // inode, 33
bwrite 4     // directory content, 46
bwrite 5     // directory inode, 32
bwrite 2     // commit (block #s and n)
bwrite 33    // install inode for x
bwrite 46    // install directory content
bwrite 32    // install dir inode
bwrite 2     // mark log "empty"
```

Write “hi”

```
bwrite 3     // bitmap update (45)
bwrite 4     // actual data (746)
bwrite 5     // inode update (33)
bwrite 2     // commit (block #s and n)
bwrite 45    // bitmap
bwrite 746   // a (note: bzero was absorbed)
bwrite 33    // inode (file size)
bwrite 2     // mark log "empty”
```

Write “\n”

```
bwrite 3     // actual data (746)
bwrite 4     // inode update (33)
bwrite 2     // commit (block #s and n)
bwrite 746   // \n
bwrite 33    // inode (file size)
bwrite 2     // mark log "empty”
```

Note xv6 assumes disk is fail-stop

- Either an entire sector is written or it is not (no partial writes)
- Difficult to achieve in practice, source of many bugs
- No decay of sectors (no read errors)
- No read of the wrong sector (seek errors)
- Sometimes real disks fail in subtle ways!

Challenge: Prevent write-back

- Buffer cache holds in-memory copies of disk blocks
- Can’t let the buffer cache write back until logged
- Tricky because cache could run out of memory
- xv6’s solution:
    - Ensure buffer cache is big enough
    - Pin dirty blocks in buffer cache
    - After commit, unpin blocks

Challenge: Data must fit in log

xv6 solution:

- Compute upper bound on number of blocks each system call could write
- Set the log size to be greater than this upper bound
- Break up some system calls into several transactions
  - E.g., really large write()'s
  - Large writes not atomic, but crash will leave correct prefix

Challenge: Concurrent syscalls

- Must allow several system calls to log concurrently
- On commit, must write them all to log
- But can’t write data if still in middle of transaction
- xv6 solution:
  - Don’t allow new system calls to start if not enough space in log
  - Wait for concurrent calls to complete and commit
  - Commit happens when in-progress calls reaches zero
  - Then wake up any waiting calls

Challenge: Writes to same block

- Same block could be modified several times in a transaction
- Actually fine because only the last write reflects the final state of the block
- So installed blocks reflect the overall committed transaction
- Called “write absorption”, **improves** performance

Conclusion

- Logging makes file system transactions atomic
  - Either they complete fully or not at all
- Write-ahead logging is the key solution in xv6
  - Log written in batches, good for performance
  - But now each disk write happens twice!
  - And have to wait (synchronous) for disk writes
  - Trouble with operations that don't fit in log
- Overall, performance is quite a bit worse
  - Next lecture: How can we make this fast?