CS 134
Operating Systems

March 27, 2019

Linux ext3 crash recovery

Outline

- Logging for crash recovery
  - xv6 log: slow and immediately durable
  - ext3: fast but not immediately durable
- Trade-off: speed vs. safety

Example problem

- Appending to a file
- Two writes
  - Mark block non-free in block bitmap
  - Add block # to inode addrs array
- We want atomicity
  - Both or neither
- So, we cannot do the FS writes one at a time

Why logging?

- Atomic system w.r.t. crashes
- Fast recovery (independent of disk size). No more hours-long `fsck`

Review of xv6 logging

- Each system call is a transaction
- System call updates cached blocks, in memory
- At end of system call:
  - Write modified blocks to log on disk
  - write blocks #s and “done” to log on disk—the commit point
  - Install modified blocks to FS on disk
    - If we crash midway-through, recovery can replay all writes from log
    - rule: don’t start FS writes until all writes are committed to log
  - Erase “done” from log

Homework: `echo hi > a`

- `commit()` hacked to ignore one of the writes, crash after commit+install and recovery disabled
- Why does `cat a` (after crash) produce: "panic: ilock: no type"
  - broken commit() updated dirent but not inode
  - So, dirent is on disk and contains the inode #
  - But, the inode is marked free (type=0)
- After recovery, why does `cat a` produce empty file?
  - Recovery wrote inode in the right place
  - But, create and write are separate system calls
    - `echo` never called `write()`: crashed during create

What’s wrong with xv6’s logging? It’s slow

- Immediate commit: after every syscall
- Immediate write to FS after every commit
  - Must do this in order to reuse on-disk log
- All new syscalls (that use the FS) block during any `commit()`
  - So, not much concurrent execution
- Every block is written twice to disk: log, FS
  - Not so bad for meta-data blocks
  - Painful for big files
  - These writes are synchronous: xv6 waits
  - Creating an empty file takes 6 synchronous disk writes=60ms
  - Only 10-20 disk update system calls per second

Linux's ext3 design

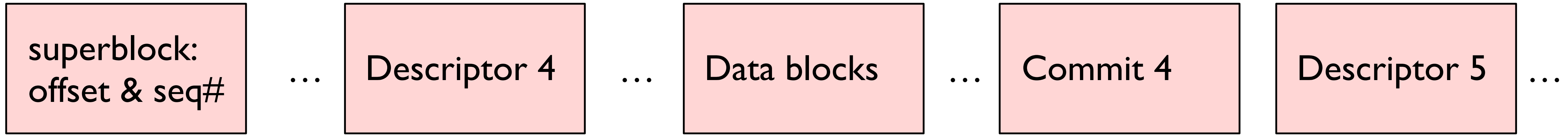
- Case study of the details required to add logging to a filesystem
- Stephen Tweedie 2000 talk transcript "EXT3, Journaling Filesystem": http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
- ext3 adds a log to ext2, a previous log-less FS
- Has many modes:
  - Start with "Journaled data"
    - Log contains both metadata and file content blocks

ext3 structures

- In memory:
  - Write-back block cache
  - Per-transaction info
    - set of block #s to be logged
    - set of outstanding *handles*—one per syscall
- On disk:
  - FS
  - Circular log

What’s in the ext3 log?

- Log superblock: log offset and starting seq # of earliest valid transaction
  - this is not the FS superblock; it's a block at start of log file
- Descriptor blocks: magic, seq, block #s
- Data blocks (as described by descriptor)
- Commit blocks: magic, seq


superblock: offset & seq# … Descriptor 4 … Data blocks … Commit 4 Descriptor 5 …

How does ext3 get good performance?

- Batching
  - Commits every few seconds, not after every system call
  - So, each transaction includes many system calls
- Why does batching improve performance?

1. Amortize fixed transaction cost (descriptor and dat blocks) over many transactions
2. *Write absorption*
   - Many syscalls in the batch may modify the same block (inode, bitmap, dirent), thus one disk write for many syscall updates
3. Better concurrency—less waiting for previous syscall to finish commit

Note: system calls return before they are safely on disk

- This affects application-level crash recovery situation
- For example, mail server that receives message, saves it to disk, then responds “OK”

ext3 allows concurrent transactions and syscalls

- There may be multiple transactions:
  - Some fully committed in the on-disk log
  - Some doing the log writes as part of the commit
  - **One** *open* transaction that's accepting new syscalls

ext3 sys call code

```
sys_open() {
  h = start()
  get(h, block #)
  modify the block in the cache
  stop(h)
}
```

- start()
  - Tells logging system to make writes atomic (until `stop()`)
  - Logging system must know the set of outstanding system calls
    - Can't commit until they're all complete
  - `start()` can block the sys call if needed
- get()
  - tells logging system we'll modify cached block
    - added to list of blocks to be logged
  - pins block in memory until transaction commits
- stop()
  - transaction can commit iff all included syscalls have called `stop()`

Committing a transaction to disk

1. Block new syscalls
2. Wait for in-progress syscalls to stop()
3. Open a new transaction, unblock new syscalls
4. Write descriptor to log on disk w/ list of block #s
5. Write each block from cache to log on disk
6. Wait for all log writes to finish
7. Write the commit record
8. Wait for the commit write to finish
9. Now cached blocks allowed to go to homes on disk (but not forced)

Can syscall B read uncommitted results of syscall A?

- A: rm x
- B: echo > y—re-using x's freed i-node
- Could B commit first, so that crash would reveal anomaly?
- Case 1: both in same transaction—ok, both or neither
- Case 2: A in T1, B in T2—ok, ext3 commits transactions in order

Can syscall B read uncommitted results of syscall A?

- Case 3: B in T1, A in T2
  - in T1: |--B--|
  - in T2: |—A--|
  - Could B see A's free of y's i-node?
    - after all, A writes the same cache that B reads
    - bad: crash after T1 could leave both x and y using the i-node
  - no: ext3 waits for all syscalls in prev xaction to finish
    - before letting any in next start
    - thus B (in T1) completes before ext3 lets A (in T2) start
    - so B won't see any of A's writes
      - T1: |-syscalls-|
      - T2: |-syscalls-|
      - T3: |-syscalls-|

Can syscall B read uncommitted results of syscall A?

- The commit order must be consistent with the order in which the system calls read/wrote state.
- Perhaps ext3 sacrifices a bit of performance here to gain correctness

Is it safe for a syscall in T2 to write a block that was also written in T1?

- ext3 allows T2 to start before T1 finishes committing—can take a while
  - T1: |-syscalls-|-commitWrites-|
  - T2: |-syscalls-|-commitWrites-|
- The danger:
  - a T1 syscall writes block 17
  - T1 closes, starts writing cached blocks to log
  - T2 starts, a T2 syscall also writes block 17
  - Could T1 write T2's modified block 17 to the T1 transaction in the log?
  - Bad: not atomic, since then a crash would leave some but not all off T2's writes committed

Is it safe for a syscall in T2 to write a block that was also written in T1?

- Ext3 gives T1 a private copy of the block cache as it existed when T1 closed
- T1 commits from this snapshot of the cache
- It's efficient using copy-on-write
- The copies allow syscalls in T2 to proceed while T1 is committing
- The point:
  - Correctness requires a post-crash+recover state as if syscalls had executed atomically and sequentially
- ext3 uses various tricks to allow some concurrency

When can ext3 re-use transaction T1's log space?

- Log is circular
- Once:
  - all transactions prior to T1 have been freed in the log, and
  - T1's cached blocks have all been written to FS on disk
  - free == advance log superblock's start pointer/seq#

What if not enough free space in log for a syscall?

- Suppose we start adding syscall's blocks to T2
- Half way through, realize T2 won't fit on disk
- We cannot commit T2, since syscall not done
- We cannot back out of this syscall, either
  - there's no way to undo a syscall
  - other syscalls in T2 may have read its modifications

What if not enough free space in log for a syscall?

- Solution: reservations
  - syscall pre-declares how many block of log space it might need
  - ext3's `start()` blocks the syscall until enough free space
  - may need to commit open transaction, then free older transaction
  - OK since reservations mean all started sys calls can complete + commit

Performance?

- `rm *` in a directory with 100 files
  - xv6: over 10 seconds—six synchronous disk writes per sys call
  - ext3: about 20 ms total
- `rm *` repeatedly writes the same same direntry and inode blocks
  - until commit, just updating the cached blocks, no disk writes
- Then one commit of a few metadata blocks
- How long to do a commit?
  - log a handful of blocks (inodes, dirents)
  - wait for disk to say writes are on disk
  - then write the commit record
  - two rotations, or about 20ms total

What if a crash?

- Crash may interrupt writing last transaction to log on disk
- So disk may have a bunch of complete transactions, then maybe one partial
- May also have written some of block cache to disk
  - but only for fully committed transactions, not partial last one

How does recovery work?

1. Find the start of the log—the first non-freed descriptor
    - log "superblock" contains offset and seq# of first transaction (advanced when log space is freed)
2. Find the end of the log
    - scan until bad magic or not the expected seq #
    - go back to last commit record
    - crash during commit → no commit record, recovery ignores
3. Replay all blocks through last complete transaction, in log order

What if block after last valid log block looks like a log descriptor?

- Perhaps a descriptor block left over from previous use of log?
  - seq # will be too low
- Perhaps some file data happens to look like a descriptor?
  - Logged data block cannot contain the magic number!
  - ext3 forbids magic number in logged data blocks:
    - Replace magic number with 0
    - Set flag for that block in descriptor

“Ordered data” mode

- Logging file content is slow, every data block written twice
- Can we entirely omit file content from the log?
- If we did, when would we write file content to the FS?
  - Can we write file content blocks at any time at all?
    - No: if metadata committed first, crash may leave file pointing to unwritten blocks with someone else's data
- ext3 "ordered data" mode:
  - Don't write file content to the log
  - Write content blocks to disk *before* committing inode with new size and block #

“Ordered data” mode

- If no crash, there's no problem—readers will see the written data
- If crash before commit:
  - Block has new data
  - Perhaps not visible, since i-node size and block list not updated
- No metadata inconsistencies
  - inode and free bitmap writes are still atomic
- Most people use ext3 ordered mode

Correctness challenges with ordered mode

1. `rmdir`, re-use block for `write()` to some file
   - Crash before rmdir or write committed
   - After recovery, as if rmdir never happened,
     - But directory block has been overwritten!
   - Fix: don’t re-use freed block until freeing syscall committed
2. `mkdir`, commit, `rmdir`, commit, reuse block in file, ordered file write, commit,
   - Crash+recover, replay mkdir and rmdir
   - File is left w/ directory content e.g. . and ..
     - Since file content write is not replayed
   - Fix: put "revoke" records into log, prevent log replay of a given block (rmdir will add revoke for direntry block)

- Note: both problems due to changing the type of a block (content vs meta-data)

Summary of rules

- The classic write-ahead logging rule:
  - Don't write meta-data block to on-disk FS until committed in on-disk log
- Wait for all syscalls in T1 to finish before starting T2
- Don't overwrite a block in buffer cache before it is in the log
- Don't free log space until all blocks have been written to FS
- Ordered mode:
  - Write data block to FS before commit
  - Don't reuse free block until freeing syscall is committed
  - Don't replay revoked syscalls

Another corner case: open fd and unlink

- Open a file, then unlink it
  - unlink commits
  - file is open, so unlink removes dir entry but doesn't free blocks
- Crash
  - Nothing interesting in log to replay
  - inode and blocks not on free list, also not reachably by any name
  - Will never be freed! oops
- Solution: add inode to linked list starting from FS superblock
  - Commit that along with remove of dir ent
  - Recovery looks at that list, completes deletions

Checksums

- Recall: transaction's log blocks must be on disk before writing commit block
  - ext3 waits for disk to say "done" before starting commit block write
- Risk: disks usually have write caches and re-order writes, for performance
  - Sometimes hard to turn off (the disk lies)
  - People often leave re-ordering enabled for speed, out of ignorance
- Bad news if disk writes commit block before the rest of the transaction

Checksums

- Solution: commit block contains checksum of all data blocks
  - On recovery: compute checksum of data blocks
    - If matches checksum in commit block: install transaction
    - If no match: don't install transaction

- ext4 has log checksumming

Does ext3 fix the xv6 log performance problems?

- Synchronous write to on-disk log—yes, but 5-second window
- Tiny update → whole block write—maybe (if syscalls permit write absorbtion)
- Synchronous writes to home locations after commit—yes

- ext3/ext4 very successful!