xv6 File System Consistency — loggin' & lockin'

Computer Science 432 — Lecture 18 — Duane Bailey

April 25, 2022

Announcements

- Lab today: Lab 7 — Supporting Large Files. *Due next Wednesday*.
  This will be our last lab.
- Small group meetings *will be held* this week and next.
- Office hours as normal: T1-3, F9-10:30.
- O/S Conference begins next Friday (~ 10 days). Details on PDF delivery next week.
- Wednesday: Shared library support.
  Kelly lecturing next Monday, Duane next Wednesday.

File System Consistency

- The consistency of file systems is very important.
  - If the file system structure breaks, it's often *permanent*.
  - A race condition in a computation is an inconvenience.
  - A race condition in a file system is a very bad, long term problem.

- We'll look at two xv6 file system features that ensure consistency:
  - The logging system. This guards against crash-related inconsistencies.
  - Locking at various levels. This guards against races that involve files.

The Logging System

- Support for crash recovery.
  - File systems take time to update.
  - Systems crash.
  - The combination requires some mechanism for ensuring file system updates are *atomic* with respect to crashes.

- Logging mechanism:
  - All disk operations (transactions) are *logged* and then *committed*.
  - If crash happens before commit, the transaction does not happen.
  - If a crash happens during commit, transaction will finalize on boot.

Observations about Logging

- Things to note about logging:
  - Logging does not know about file system structure.
    It's low level: just about blocks written.
  - Transactions are simply guaranteed to leave the file system consistent.
    Not necessarily "the way you want it."
  - The way to think about "consistent state"? Think about invariants:
    - No inode points to a block that is unallocated, or
    - inode nlink counts number of directory entries that point to inode.

Logging Transaction Granularity

- File system call typically leads to a logging transaction:
  - start_op()
  - 1 or more log_write()s
  - end_op()
- Logging commits only happen when there are no operations in progress.
- Each file system operation has an expected budget of blocks written
  - If logging area is insufficient, start_op() waits (sleep locks) until space
- log_write() detail:
  - schedules a block to be written to the logging area
  - “pins it” to make sure it doesn’t leave cache (even if brelse) before write

Logging Details

- All log entries are written to a dedicated area of disk (see superblock)
  - A transaction header keeps track of
    - # of blocks to be written (0 until commit, then #), and
    - an array of block addresses
  - Blocks that follow are the (unique*) updated disk blocks
- At end of transaction:
  - Blocks in logging area are the final list, block count > 0
  - Blocks are then written to the disk at target locations
  - Block count is then set to 0.

Logging Commit Details

- A commit for a transaction is a four step process:
  - write_log() — moves blocks from cache to log area
  - write_header() — **commit point.** Non-zero log header written.
  - install_trans() — copy blocks from log area to disk.
  - write_header() — write 0 to header, finalizing transaction

- Useful to think about outcome if you crash at any of these points.

Crash Recovery Details

- What happens at a crash?
  - If the transaction header has a count of 0, do nothing.
  - If the count is > 0:
    - Write blocks to disk in appropriate locations (possibly unnecessary)
    - Write a 0 in the count in the transaction header.

- With a little thought:
  - File system transactions are *atomic* with respect to crashes:
    - The transaction is fully written, or
    - Nothing is written

Thinking About Locking

- There are many locks associated with the file system
- Each lock supports an invariant for a specific level of filesystem
- Consistency <=> maintained invariants
- Sometimes consistency at higher level of file system depends simply on lower level consistency

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids interm
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DM
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Disk-level Locking

- struct disk has a vdisk_lock
  - Held while communicating transactions with the virtio disk
  - Disk operations are then atomic
  - They're serialized

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids interm
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DM
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Buffer-Cache Locking

- Buffer cache has two locks:
  - bcache.lock: only one process may update cache at a time
    Does not handle full cache…
  - buf->lock: only one process may work with a block at a time
    Sleep-lock acquired in bread()
    released in brelse()

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids interm
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DM
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Logging Level Locking

- Log.lock forces serialization of access to logging information
- Log makes heavy use of consistency from bcache ie. bread()/brelse()

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids interm
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DM
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Inode Level Locking

- Inodes in memory are cached
  - icache lock serializes allocation
    - inodes are uniquely in cache
    - refcounts are accurate
  - ip->lock (sleep)
    exclusive access to inode fields

- Lock-like characteristics:
  - ref > 0 in inode keeps inode in $
  - unlink: inode not freed until
    nlink == 0.

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids interm
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DM
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Inode Level Locking

- Inode layer lock-related entry points:
  - ialloc - allocates a new inode on disk, igets and returns inode from $
  - iget - increases the reference count in inode $, reads from disk if nec.
  - ilock - locks inode in inode $
  - iupdate - updates change to inode on disk.
  - iput - decreases reference count; calls itrunc if ref == 0 and nlink = 0.

- Other entry points, not directly involving locks:
  - bmap() - fetch block address from inode
  - readi(), writei() - read or write data to file by inode
  - stati() - get inode metadata

Directory and Path Consistency

- The consistency at directory and pathname levels is result of consistency at lower levels

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids interm
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DM
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

File Descriptor Consistency

- File descriptors are kept in global table.
  - Consistency maintained by ftable.lock

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids interm
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DM
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Repeat after me…

"I do not let locks define me. I define locks."

"I will not talk about locks any more. I am happy."