6.824 2006 Lecture 8: Tutorial on Cache Consistency and Locking

lecture overview
  a tutorial to help you with labs 4 and 5
  lab 4: locking for correctness with multiple servers
  lab 5: caching for performance

overall goal:
  ccfs-based distributed file system
  try to increase number of clients supported by single block server
  assume that (usually) clients work w/ different files
    so let's make this case efficient using caching
  but let's also preserve correctness

start with your lab 3 ccfs
  [draw picture: two ccfs servers, one block server]

first: correctness w/ multiple servers
  suppose both servers executing a CREATE RPC on same directory
  they both get() dir contents, add a new entry, put() contents
  first put is overwritten, so one file is lost
  how do we know this was the wrong answer?
  need a definition of correctness for concurrent operations
  traditional definition: atomicity
    result of two concurrent operations must be the same as if
    they were run in some one-at-a-time order
  usual solution: serialize operations
    wait for one to finish, then start the second
  if you serialize, and each operation is correct when
    run alone, then the whole system is correct
    don't need to reason specifically about every concurrent
interleaving
  you'll serialize w/ locks in lab 4

[add lock server to picture]

what should each lock protect?
  whole file system? no: prevents concurrency that would have been OK.
  just one block? maybe, but then need one lock per dirent for
NFS3_CREATE.
  i-node + contents: perhaps this will match atomic operation
granularity.
  so let's have locks with name == file handle

what operations need to be atomic in ccfs?
  certainly CREATE, due to get()-modify-put()
  SETATTR?
  WRITE? (sub-block writes to same block, or updating block lists)
  READ? maybe confusing if size != actual amount of data
    and atime update requires read-modify-write

span of a lock in time?
  CREATE checks if file exists, creates new i-node, reads
    directory contents, writes contents, writes directory i-node
    better hold the directory lock the whole time!
  in general, acquire lock first, release when totally done
    then we get serialization

```
  lucky we're using file handle as lock name, which means we can
acquire
    lock before any get()
  can't release lock until after last put() completes
  and better not reply to RPC until put() completes

what if a single ccfs gets concurrent CREATEs in the same dir?
  must still execute one at a time
  so you actually need locks even for a single ccfs
  that's why we never wrote more than 8192 bytes in lab3 tester
    (NFS client sends WRITEs concurrently for same file)

do we ever need multiple locks?
  CREATE involves two file handles (directory and new file)
  REMOVE involves both a file and a directory
  do we need to hold two locks?
  RENAME probably requires two locks, if two directories
    deadlock, order of acquisition

what about performance?
  every NFS RPC now involves many RPCs to block and lock server
  likely to be slow

Lab 5 plan:
  want to operate out of local cache, w/ no RPCs to block/lock servers
  as long as only one ccfs is using a given file &c
  only talk to block/lock servers when others need our blocks/locks

step 1: add block caching to ccfs
  you will modify blockdbc.C and .h
  get() checks local cache first
    if in local cache: just return
    otherwise: fetch from block server, add to local cache, return
  put() *just* adds to cache, marks block as dirty
  you can copy some code from blockdbd.C: the hash table

need to know when another ccfs wants to read a block that's dirty in
our cache
  and when another ccfs wants to write a block that's clean in our
cache
  and need to ensure at most one ccfs has a dirty copy of any given
block
  we need "cache consistency"
    informally, a read sees the most recent write

here's a good rule:
  you can cache a block (dirty or not) if you hold the file's lock
  you cannot have a block cached if you don't hold the corresponding
lock
  so need to "flush" blocks before releasing lock back to server
    drop clean blocks from cache
    put() dirty blocks

but this hasn't helped performance!
  must flush data cache before each release()
  still doing many get()/put()/acquire()/release() per NFS RPC
  idea: cache the locks also!
```

```
so you need to change lock_client to cache locks locally
  so that release() just marks lock locally as released
  if you acquire() it again, no need to talk to lock server
  need to make lock server send a REVOKE if some other client is
waiting
  lock_client should tell fs.C what lock is being revoked
  fs.C should tell block client to send that file's dirty blocks
    to block server, and drop file's clean blocks
  all file's blocks: content, attribute, &c
  fs.C should tell lock_client when block server has replied to all
PUTs
  then lock_client should send a RELEASE RPC to the server

Details
  given a lock name, how to figure out keys of blocks that should be
flushed?
  lock name should be file handle (so easy to flush attributes)
  name other file block in a predictable way from file handle

typical sequencing when interacting with locks
  client #1 is caching the lock and dirty blocks
  client #2 calls acquire()
    #2 -> LS : ACQUIRE
    LS -> #2 : reply
    LS -> #1 : REVOKE
    #1 -> LS : reply
    #1 -> BS : put(fh, v)
    BS -> #1 : reply
    #1 -> LS : RELEASE
    LS -> #1 : reply
    LS -> #2 : GRANT
    #2 -> LS : reply
    #2 -> BS : get(fh)
  #1 must ensure the block server has the dirty data before releasing!

lab 5 quirks
  NFS3_READ must take the lock, not for atomicity, but to get latest
data
  NFS3_REMOVE may need the file lock to force file handle to be stale
    if you only lock the directory, you leave i-node in other caches
    so future GETATTR for file may succeed
  NFS3_CREATE may need to grab lock on new i-node
    to force others to read from our cache

What you're *not* responsible for:
  atomicity w.r.t. crashes
    recovering lock state after lock server reboot
    replicating the block server
    client crash while holding locks: un-do partial operations?
```