6.824 - Spring 2006

6.824 Lab 5: Cache Consistency

Due: Lecture 12

Introduction

Now it's time to add caching to ccfs in order to improve performance. First you'll add a local write-back block cache to each ccfs server. Then you'll make the caches consistent by forcing write-back of the dirty cached blocks associated with a filehandle (and deletion of the clean blocks) when you release the lock on that filehandle. Finally you'll add lazy release (i.e. caching) of locks so that each ccfs can serve NFS RPCs without contacting the block or lock servers, when possible.

Your server will be a success if it manages to operate out of its local block and lock cache when reading/writing files and directories that other hosts aren't looking at, but maintains correctness when the same files and directories are concurrently read and updated on multiple hosts.

Getting Started

```
$ cd
$ mkdir lab-5
$ rsync -av lab-4/ lab-5/
$ cd lab-5
$ wget -nc http://pdos.csail.mit.edu/6.824/labs/test-lab-5.c
$ cc -o test-lab-5 test-lab-5.c
```

Testing Performance

Our measure of performance is the number of puts and gets that your server sends to the block server. You can tell the block server to print out a line every 100 puts/gets telling you the current totals:

```
% ./blockdbd -v 35983 &
30 70
...
3200 6900
```

You may need to comment out debugging statements in the block and lock servers in order to see these print statements. You should change the definition of DBG from 1 to 0 in lock_server.C.

Each line indicates the number of puts and the number of gets so far. The workload on which we'll evaluate your server's performance is generated by test-lab-5. It creates two

sub-directories and creates/deletes 100 files in each directory, using each directory through only one of the two servers.

```
$ ./ccfs dir1 localhost 35983 &
root file handle: 4d1def4be444163c
$ ./ccfs dir2 localhost 35983 4d1def4be444163c &
$ ./test-lab-5 /classfs/dir1 /classfs/dir2
Create/delete in separate directories: OK
$
```

Now look at the last pair of numbers printed by blockdbd. Your code from lab 4 will probably generate a few thousand puts and perhaps twice that many gets. Your goal is to reduce those numbers to about a dozen puts and at most a few hundred gets.

Of course your server must also remain correct, so we will require the server you hand in to pass the Lab 4 testers as well as getting good performance on the Lab 5 tester.

Step One: Block Cache

In Step One you'll add caching to your block client (blockdbc.C and blockdbc.h), without cache consistency. This cache will make your server fast but incorrect.

get() should check if the block is cached, and if so return the cached copy. Otherwise get() should fetch the block from the block server, put it in the local cache, and then return it to the file server. put() should just replace the cached copy, and not send it to the block server. You'll find it helpful for the next section if you keep track of which cached blocks have been modified by put() (i.e. are "dirty"). remove() should delete the block from the local cache.

Look in blockdb.h and blockdb.C for example code that stores blocks in an ihash. Warning: ihash.insert() adds a key/value pair to the hash table without checking whether that key is already present, so you should check if the key is there and remove it before inserting a new value.

You may find that your fs.C code does not work correctly if get() and put() call the callback immediately (as they would if they execute the operation in the cache, without consulting the block server). You can delay the callback:

```
void
blockdbc::get_xcb(callback<void, bool, str>::ref cb, str v)
{
  (*cb)(true, v);
}

void
blockdbc::get(str key, callback<void, bool, str>::ref cb)
{
  if(key is in the cache):
    str value = ...;
    delaycb(0, wrap(this, &blockdbc::get_xcb, cb, value));
```

When you're done, run test-lab-5 giving the same directory twice, and watch the numbers printed by the block server. You should see zero puts and somewhere between zero and a few hundred gets (or perhaps no numbers at all, since blockdbd only prints a line for every 100 puts/gets). Your server should pass test-lab-4-a.pl and test-lab-4-b if you give it the same directory twice, but it will probably fail test-lab-4-b with two different directories because it has no cache consistency.

Step Two: Cache Consistency

In Step Two you'll ensure that each get() sees the latest put(), even when the get() and put() are from different ccfs servers. You'll arrange this by ensuring that your server writes a file's modified (dirty) cached blocks back to the block server **before** the server releases the lock on that file. Similarly, your server should delete unmodified (clean) blocks from its cache when it releases the lock on the relevant file.

You will need to add a method to the block client to let your file server ask for a block to be eliminated from the cache. This "flush" method should first check whether the block is dirty in the cache, and send it to the block server if it is dirty. Blocks that your server has removed (with the block client's remove() method) should also be removed from the block server if the block server knows about them. The flush method should take a callback argument that it calls when it is done, so that your server can learn when the flush has finished.

Your server will need to call flush() just before releasing a lock back to the lock server. You could just add flush() calls to fs.C before each release(). However, you'll find that Step Three is easier if you instead arrange for the lock client to call a callback in fs.C when the lock client is about to send a RELEASE RPC to the lock server. A clean interface might involve fs::fs() registering a callback with the lock client, which the lock client calls with a lock name whenever it about to send a RELEASE RPC.

So you could add this code to fs.C:

```
fs::fs(blockdbc *xdb, lock_client *xlc)
{
  db = xdb;
  this->lc = xlc;
  this->lc->set_releasing_cb(wrap(this, &fs::do_release)); // ADD
```

And add code like this to lock_client.h:

```
class lock_client {
public:
  void
set_releasing_cb(callback<void,str,callback<void,void>::ref>::ptr cb){
    releasing_cb = cb;
  }
private:
  callback<void, str, callback<void,void>::ref>::ptr releasing_cb;
```

You would then modify release() to call releasing_cb before sending the lock back to the lock server:

```
void
lock_client::call_releasing_cb(str name)
{
  (*releasing_cb)(name, wrap(this, &lock_client::really_release,
name));
}

void
lock_client::release(str name)
{
  ...;
  delaycb(0, wrap(this, &lock_client::call_releasing_cb, name));
```

The effect will be that release() will call fs::do_release(lockname,cb), your do_release() should call the block client's flush() routine for attribute and content blocks protected by the lock, and when the flush is finished your server should call the cb passed to do_release() in order to tell the lock client that it's OK to send a RELEASE RPC to the lock server.

You may need to add locking to NFS RPC handlers that don't modify anything, in order to make sure that they see fresh data from the block server.

When you're done with Step Two your server should pass all the correctness tests (test-lab-4-a and test-lab-4-b with two servers). test-lab-5 should execute correctly with two servers, but with a large number of puts and gets.

Step Three: Lazy Lock Release

In Step Three you'll cache locks in the lock client, and only send a RELEASE RPC to the lock server when the lock server receives an ACQUIRE for the lock. Thus, when fs.C calls release(), the lock client should just mark the lock as released in its local cache, and not send an RPC to the lock server. If fs.C calls acquire() for that lock, the lock client can grant the lock out of its own cache. If the lock server needs the lock back, then the lock client should first call releasing_cb, then send a RELEASE RPC to the lock server.

You'll need to add another RPC type, REVOKE, to the lock protocol, so that the lock server can ask a client to RELEASE a cached lock. Add this line to your lock_proto.x:

```
    bool LOCK_REVOKE(lname) = 4;
```

Here's the code you need to send a REVOKE from the lock server to the lock client:

```
void
LS::send_revoke(str name, struct sockaddr_in sin)
{
  bool *r = new bool;
  struct sockaddr_in *xsin = new sockaddr_in;
  *xsin = sin;
```

```
  lname xn;
  xn = name;
  c->call(LOCK_REVOKE, &xn, r,
          wrap(this, &LS::dummy_cb, r, xsin),
          (AUTH *) 0,
          (xdrproc_t) 0, (xdrproc_t) 0,
          (u_int32_t) 0, (u_int32_t) 0,
          (struct sockaddr *) xsin);
}
void
LS::dummy_cb(bool *r, struct sockaddr_in *xsin, clnt_stat err)
{
  delete r;
  delete xsin;
}
```

Add this to lock_client::dispatch():

```
  case LOCK_REVOKE:
    lc->revoke(sbp);
    break;
```

Now you need to modify your lock server to send a REVOKE when it receives an ACQUIRE for a lock that is currently held. You should modify your lock client's acquire() and release() to keep locks in a local cache, and only talk to the lock server when acquire() can't find the needed lock in the cache. You should also modify your lock client to send a RELEASE when it receives a REVOKE for a lock it has cached. If fs.C is currently holding the lock, the lock client should wait until fs.C has called release() before it sends the lock back to the lock server. The lock client should call releasing_cb before sending the RELEASE.

When you're done, your server should pass test-lab-4-a.pl and test-lab-4-b with two servers. test-lab-5 should require only five or ten puts to the block server and a few hundred gets.

Hints

If the lock server receives two ACQUIREs for the same lock from different clients, it may end up sending a GRANT immediately followed by a REVOKE. These RPCs may arrive out of order at the lock client; chances are you'll then ignore the REVOKE and never RELEASE the lock back to the lock server. It may help for the lock server to delay sending the REVOKE until it has received the RPC reply to the GRANT, to ensure that the GRANT arrives before the REVOKE.

Your lock client will probably have to keep an internal queue of requests so that it can correctly serve locks for concurrent NFS RPCs to the same file server. Thus your lock client will need to contain queue code similar to that in the lock server.

Since you're now using locks for both serialization (atomicity) and cache consistency, NFS operations that read data from the block server will need to lock the data first in order to ensure that they get a fresh copy.

Don't use a nfscall structure after you send a reply or error. The RPC library de-allocates the structure when it sends the reply. You may be tempted to use the arguments in the nfscall structure after replying in order to release locks: don't. Instead, make a copy of the arguments, or release the locks before replying.

Collaboration policy

You must write all the code you hand in for the programming assignments, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution (and you're not allowed to look at solutions from previous years). You may discuss the assignments with other students, but you may not look at or copy each others' code.

Handin procedure

You should hand in the gzipped tar file `lab-5-handin.tgz` produced by running these commands in your ~/lab-5 directory.

```
$ tar czf lab-5-handin.tgz *.[TCchx] Makefile
$ chmod og= lab-5-handin.tgz
$ cp lab-5-handin.tgz ~/handin
```

We will use the first copy of the file that we find after the deadline.