6.824 - Spring 2006

6.824 Lab 4: MKDIR, REMOVE, and Locking

Due: Lecture 9

Introduction

In this lab, you will continue to add functionality to your file server. You will:

- add handlers for the MKDIR and REMOVE RPCs,
- add support for the mtime attribute, and
- add simple locking.

The mtime attribute is neccessary to help the NFS client code check whether locally-cached file and directory contents are up to date. The locking is required to ensure that concurrent modifications to the same file or directory occur one at a time.

Getting Started

Your first task is to add the lock client module to your file server, and your lock server module to the block server. The latter is only for convenience, the two servers are logically separate. To do this, you will combine some files provided in lab-4.tgz with your file server from Lab 3 and your lock server from Lab 1. Then, you will make some small changes to your `fs.C` and `fs.h` in order to utilize the lock client/server.

Begin by initializing your Lab 4 directory with your implementation from Lab 3 and then updating it with the files and testers for Lab 4.

```
pain$ wget -nc http://pdos.csail.mit.edu/6.824-2006/labs/lab-4.tgz
pain$ rsync -av lab-3/ lab-4/
pain$ tar xzvf lab-4.tgz
```

Note that this will overwrite some source files. In particular, it replaces your `Makefile`, `ccfs.C`, `blockdbd.C`, and the block client code; if you have changes you wish to preserve in these files (which you should not have modified), you should merge the changes we have made to these files. For example, if you are using TAME, you may need to tweak the Makefile rules yourself to copy in the rules to generate `fs.C` from `fs.T`. You may find the `diff` command useful in determining where files differ.

Copy your lock server code from Lab 1, which will be used For example:

```
pain$ cp lab-1/lock_server.[Ch] lab-4/
```

You will need to make a series of changes to add lock client support to your fileserver.

- Edit fs.h (in the lab-4 directory) and include `lock_client.h` and enable locking support:
- `#define LOCKS`
- `#include "lock_client.h"`
- Also change the fs constructor to take a lock client argument, and add a new class member to hold it:
- `class fs {`
- `public:`
- `fs(blockdbc *db, lock_client *lc); // CHANGE`
- `...`
- `private:`
- `lock_client *lc; // ADD`
- Change the implementation of the constructor in fs.C to match:
- `fs::fs(blockdbc *xdb, lock_client *xlc) // CHANGE`
- `{`
- `db = xdb;`
- `lc = xlc; // ADD`

These changes are also available as a patch

.

At this point you should be able to run `gmake` and successfully compile your file server and a block server. Operations that worked in Lab 3 should still work. Note that you must use the new block server that is built in this lab as it also hosts the lock server; with the above changes, ccfs will expect a lock server to be available on the same transport as your block server.

The rest of this lab has two parts.

Part 1: Your Job

Your job in part 1 is to handle the MKDIR and REMOVE RPCs, and to add support for the mtime attribute.

Your server should update a file's mtime when SETATTR changes its size, during a WRITE, and when the file is created. The server should update a directory's mtime whenever an entry is added or removed from the directory. In all cases the server should set the mtime to the current time with `nfstime()`.

If your server passes the tester (see below), then you are done.

When you're done with Part 1, the following should work:

```
pain$ mkdir /classfs/dir1/newdir
pain$ echo hi > /classfs/dir1/newdir/newfile
pain$ ls /classfs/dir1/newdir/newfile
pain$ rm /classfs/dir1/newdir/newfile
pain$ ls /classfs/dir1/newdir
pain$
```

Part 1: Testing

You can test your file server using the test-lab-4-a.pl script. The script creates a directory, creates and deletes lots of files in the directory, and checks file and directory mtimes. If the tester is happy, it will say "Passed all tests!". Here's a successful run:

```
pain$ ./test-lab-4-a.pl /classfs/dir1
mkdir /classfs/dir1/d3319
create x-0
delete x-0
create x-1
checkmtime x-1
...
delete x-33
dircheck
Passed all tests!
```

Part 2: Your Job

Your server's CREATE handler probably reads the directory's contents from the block server, makes some changes or additions, and writes the new contents back. If two clients were to issue simultaneous CREATEs for different file names in the same directory to two ccfs servers sharing the same file system, chances are that only one file would be exist in the end. The correct answer, however, is for both files to exist.

A convenient way to fix this "race condition" is for the file servers to use locks to force the two CREATE operations to happen one at a time. That is, a server would acquire a lock before starting the CREATE, and only release the lock after finishing the write of the new information back to the block server. If there are concurrent operations, the locks force one of the two operations to delay until the other one has completed.

You must choose what the locks refer to. At one extreme you could have a single lock for the whole file system, so that operations never proceed in parallel. At the other extreme you could lock each entry in a directory, or each field in the attributes structure. Neither of these is a good idea! A single global lock prevents concurrency that would have been OK, for example CREATEs in different directories. Fine-grained locks have high overhead and make deadlock likely, since you often need to hold more than one fine-grained lock.

Your best bet is to associate one lock with each file handle. Use the file handle as the name of the lock (i.e. pass the file handle to acquire and release). The convention should be that any NFS operation should acquire the lock on the file or directory it uses, perform the operation, finish updating the block server (if the operation has side-effects), and then release the lock on the file handle. One reason this is convenient is that you can add locking relatively simply by calling acquire in fs::get_fh(); thus all of your RPC implementations will acquire file handle locks without you having to modify your RPC handler code. Similarly, you can modify fs::put_fh() to release the file handle lock. You'll still need to add explicit releases for RPC handlers such as GETATTR that don't write the block store, and for error cases.

You'll use your lock server from Lab 1. The lab-4.tgz included a blockdbd.C that incorporates your lock server code, so you won't have to start a separate server. blockdbd looks at the RPC program number to figure out whether it's a locking RPC or a block RPC. The new ccfs.C instantiates a lock_client and connects it to your lock server; your changes to fs.C at the start of the lab will allow you to call acquire() and release() from fs.C as in this example:

```
void
fs::acquire(nfs_fh3 fh, callback<void,void>::ref cb)
{
  lc->acquire(fh2s(fh), cb);
}

void
fs::release(nfs_fh3 fh)
{
  lc->release(fh2s(fh));
}
```

Part 2: Testing

Test your locking with test-lab-4-b. The tester takes two directories as arguments. It issues concurrent operations in the two directories and checks that the results are consistent with the operations executing in some serial order. Here's a successful execution of the tester:

```
frustration$ ./ccfs dir1 pain 5566 &
root file handle: f7dbbbadfd082018
frustration$ ./ccfs dir2 pain 5566  f7dbbbadfd082018 &
root file handle: f7dbbbadfd082018
frustration$
frustration$ ./test-lab-4-b /classfs/dir1 /classfs/dir2
Create then read: OK
Unlink: OK
Append: OK
Readdir: OK
Many sequential creates: OK
Write 20000 bytes: OK
Concurrent creates: OK
Concurrent creates of the same file: OK
Concurrent create/delete: OK
Concurrent creates, same file, same server: OK
test-lab-4-b: Passed all tests.
```

If you try this before you add locking, your server will probably fail the "Write 20000 bytes" test or the "Concurrent creates" test. The reason the "Write 20000 bytes" test might fail is that it produces three concurrent WRITE RPCs (of 8192, 8192, and 3616 bytes).

You might want to test your solution to Part 2 with test-lab-4-a first, to make sure you didn't break anything. You might then test with test-lab-4-b, but giving it the same directory twice, to make sure you handle concurrent operations in one server before you go on to concurrent operations in two servers.

Hints

Here's how to extract the arguments from a MKDIR RPC:

```
mkdir3args *a = nc->Xtmpl getarg<mkdir3args> ();
// a->where.dir : existing directory
// a->where.name : name of new directory
// a->attributes.mode.set : if non-zero, set new dir's fattr3.mode
to...
// *a->attributes.mode.val : the new directory's mode
```

Here's how to reply to a MKDIR RPC. before_attrs and after_attrs are the attributes of the containing directory before and after the MKDIR.

```
diropres3 *res = nc->Xtmpl getres<diropres3> ();
res->set_status(NFS3_OK);
res->resok->obj.set_present(true);
*res->resok->obj.handle = fh;
res->resok->obj_attributes.set_present(false);
res->resok->dir_wcc = make_wcc(before_attrs, after_attrs);
nc->reply(nc->getvoidres());
```

You may need to create, or fake, a "." entry in each new directory, referring to the directory itself. Also, if possible you should include the object attributes in the reply above as some (non-class) versions of FreeBSD will crash if it is not present.

Here's how to extract the arguments from a REMOVE RPC:

```
diropargs3 *a = nc->Xtmpl getarg<diropargs3> ();
// a->dir : the directory
// a->name : the file name
```

Here's how to reply to a REMOVE RPC:

```
wccstat3 *res = nc->Xtmpl getres<wccstat3> ();
res->set_status(NFS3_OK);
*res->wcc = make_wcc(before_attrs, after_attrs);
nc->reply(nc->getvoidres());
```

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 assigment. 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-4-handin.tgz` produced by running these commands in your ~/lab-4 directory.

```
$ tar czf lab-4-handin.tgz *.[TCchx] Makefile
$ chmod og= lab-4-handin.tgz
```

Copy this file to `~/handin/lab-4-handin.tgz`. We will use the first copy of the file that we find after the deadline.