

Project 2: Part 4: MKDIR, REMOVE and Locking

Due: 11:59PM Thursday, March 24, 2011

1 Introduction

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

- Add handlers for the MKDIR and REMOVE FUSE operations.
- Add simple locking. Locking is required to ensure that concurrent modifications to the same file or directory occur one at a time.

2 Getting started

As in part 3, copy the additional files found in the part4 subdirectory of the handout into your working directory. **Make sure you choose to keep the new Makefile.**

As in Part 3, begin by uncommenting the relevant lines at the bottom of fuse.cc::main (such as fuse_server_oper.unlink = fuse_server_unlink;) so that you point FUSE to call the appropriate functions that you will fill in in this part.

Ensure the code from Part 3 passes all tests for Part 1, 2, and 3 before starting in on this part.

Part 4 has two phases:

3 Phase 1: MKDIR, REMOVE

3.1 Your Job

Your job in phase 1 is to handle the MKDIR and REMOVE FUSE operations. This should be a straightforward extension of your Part 3 code. Make sure that when you choose the inumber for a new directory created with MKDIR, that inumber must have its most significant bit set to 0 (as explained in Part 2 of the Project, unless you changed the way YFS tells files and directories apart). When you're done with phase 1, the following should work:

```
% ./start.sh
% mkdir yfs1/newdir
% echo hi > yfs1/newdir/newfile
```

```
% ls yfs1/newdir/newfile
yfs1/newdir/newfile
% rm yfs1/newdir/newfile
% ls yfs1/newdir
% ./stop.sh
```

If your implementation passes the test-lab-4-a.pl script, you are done with phase 1. The test script creates a directory, creates and deletes lots of files in the directory, and checks file and directory mtimes and ctimes. Note that this is the first test that explicitly checks the correctness of these time attributes. A create should change both the parent directory's mtime and ctime. Here is a successful run of the tester:

```
% ./start.sh
% ./test-lab-4-a.pl ./yfs1
mkdir ./yfs1/d3319
create x-0
delete x-0
create x-1
checkmtime x-1
...
delete x-33
dircheck
Passed all tests!
% ./stop.sh
```

4 Phase 2: Locking

Next, you are going to ensure the consistency of your file system when many clients simultaneously perform file system operations on the same file system image via different `yfs_client` processes. Your current implementation does not handle concurrent operations correctly. For example, your `yfs_client`'s create method probably reads the directory's contents from the extent server, makes some changes or additions, and stores the new contents at the extent server. Suppose two clients issue simultaneous CREATEs for different file names in the same directory via different `yfs_client` processes. Both `yfs_client` processes might fetch the old dir contents at the same time and each inserts the newly created file for its client and writes back the new dir contents. As a result, only one of the file would be present in the dir in the end. The correct answer, however, is for both files to exist. The CREATE example is just one of the "race conditions". Many others exist: e.g. concurrent CREATE and UNLINK, concurrent MKDIR and LOOKUP, etc.

To fix the race conditions, the `yfs_client` must use locks to ensure that the two operations that access the same file or directory happen one at a time. For example, a `yfs_client` would acquire a lock before starting the CREATE, and only release the lock after finishing the write of the new information back to the extent server. If there are concurrent operations, the locks force one of the two operations to delay until the other one has completed. Because each `yfs_client` can run as a separate process on a different machine, all `yfs_clients` have to acquire locks from the same lock server. Now you can see why the lock server implementation from Part 1 comes in handy!

4.1 Your Job

Your job is to implement locking for `yfs_client` to ensure that concurrent operations from different `yfs_clients` proceed correctly. The testers for this part of the lab are `test-lab-4-b` and `test-lab-4-c` (The source files are `test-lab-4-b.c` and `test-lab-4-c.c`). The testers take two directories as arguments and issue concurrent operations in the two directories and check that the results are consistent with the operations executing in some serial order. Here's a successful execution of the testers:

```
% ./start.sh
% ./test-lab-4-b ./yfs1 ./yfs2
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.
% ./stop.sh
%
% ./start.sh
% ./test-lab-4-c ./yfs1 ./yfs2
Create/delete in separate directories: tests completed OK
% ./stop.sh
```

If you try this before you add locking, it will fail at “Concurrent creates” test in `test-lab-4-b`.

After you are done with phase 2, you should also test with `test-lab-4-a.pl` to make sure you didn't break anything. You might also test with `test-lab-4-b` with the same directory for both arguments, to make sure you handle concurrent operations correctly with only one server before you go on to test concurrent operations in two servers.

4.2 Detailed Guidance

- What to lock? 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 okay, 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 or directory's inumber as the name of the lock (i.e. pass the inumber to `acquire` and `release`). The convention should be that any `yfs_client` operation should acquire the lock on the file or directory it uses,

perform the operation, finish updating the extent server (if the operation has side-effects), and then release the lock on the inumber. You must be careful about releasing the locks in all circumstances upon return from `yfs_client` operation.

You'll use your lock server from Part 1. Our original template for the `yfs_client` constructor that we gave you in Part 2 included the destination address of a lock server, so it should be very easy to add a `lock_client` object to the `yfs_client` and simply call its `acquire` and `release` methods.

This is also the first lab that writes arbitrary data to the file, rather than null-terminated C-style strings. If you used the standard `std::string` constructor in `fuse.cc` to create a string to pass to your `yfs_client`, (i.e., `std::string(buf)`), you will get odd errors when there are characters equal to the termination character in the buffer. Instead, you should use a different constructor that allows for char buffers of arbitrary data: `std::string(buf, size)`.

5 Handin

Please submit all the files necessary for running Part 4, including the Makefile to:

```
/afs/andrew/course/15/440-sp11/handin/proj2/your_andrew_id/part4/
```

Please follow the same guidelines outlined in Part 1 for multiple submissions.

6 C++ Tutorials and Resources

- C++ Tutorial
<http://www.cplusplus.com/doc/tutorial/>
- C++ Reference
<http://www.cppreference.com/wiki/start>