6.824 - Spring 2006

6.824 Lab 3: Reading, Writing and Sharing Files

Due: Lecture 7

Introduction

In this lab, you'll continue implementing the Frangipani-like file server you started in Lab 2.

You'll start with the code you wrote for Lab 2, and add support for reading and writing file contents. You'll have to choose a representation with which to store file contents in the block server and implement the SETATTR, WRITE, and READ RPCs.

Continuing with the CCFS Server

There are no new source files needed for this lab: you will continue to implement functionality in your existing server. There is a new tester which tests for reading and writing data correctly. Download the lab tester files from http://pdos.csail.mit.edu/6.824-2006/labs/lab-3.tgz to your home directory on one of the class machines.

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

Then, copy the source files from your Lab 2 to the Lab 3 directory. For example:

```
pain$ rsync -av lab-2/ lab-3/
```

Lab 3 builds upon Lab 2 so make sure you pass the Lab 2 tester before proceeding. When you're done with Lab 3, you should be able to read and write files in your file system. For example, start up the block server on one of the class machines:

```
frustration$ cd ~/lab-3
frustration$ ./blockdbd 3772 &
```

Then, start up the file server on another:

```
pain$ ./ccfs dir frustration 3772 &
root file handle: 2d1b68f779135270
pain$ cd /classfs/dir
pain$ echo hello > a
pain$ ls -lt
total 0
-rw-r--r--  0 root  wheel  6 Feb 16 13:52 a
pain$ cat a
hello
```

```
pain$
```
If you try the above commands on your Lab 2 file server, you will get an error.

Your Lab 3 file server must support sharing the file system on other hosts via the block server. So when you're done with Lab 3 you should be able to do this on a third machine (e.g. anguish):

```
anguish$ ./ccfs dir frustration 3772 2d1b68f779135270 &
[1] 6608
root file handle: 2d1b68f779135270
anguish$ cd /classfs/dir
anguish$ ls -lt
total 0
-rw-r--r--  0 root  wheel  6 Feb 16 13:52 a
anguish$ cat a
hello
anguish$
```
Replace the fourth argument to ccfs with the root file handle that ccfs printed out on pain.

Your Job

Your job is to implement SETATTR, WRITE, and READ NFS RPCs in fs.C. You must store the file system's contents in the block server, so that you can share one file system among multiple servers.

For writes, you should ignore the stable argument and always write the file to stable storage (the block server). You should wait for the put of the file's contents and attributes to complete before sending a reply to the client. This behavior is equivalent to performing writes with the a->stable argument set to FILE_SYNC.

If your server passes the tester (see below), then you are done. If you have questions about whether you have to implement specific pieces of NFS server functionality, then you should be guided by the tester: if you can pass the tests without implementing something, then don't bother implementing it.

Please don't modify the block server or its client interface (blockdbc.C and blockdbc.h), since we may test your file server against our own copy of the block server.

Testing

You should test your file server using the test-lab-3.pl script. The script tests reading, writing, and appending to files, as well as testing whether files written on one server can be read through a second server. To run the tester, first start a block server:
```
frustration$ ~/lab-3/blockdbd 3883 &
```
Then start two instances of ccfs, with the same root file handle, on the same machine:
```
anguish$ ./ccfs dir1 frustration 3883 &
[1] 72422
root file handle: 2a868a18dad0f84e
anguish$ ./ccfs dir2 frustration 3883 2a868a18dad0f84e &
```

```
[3] 72595
root file handle: 2a868a18dad0f84e
anguish$
```

The directory argument for each instance is different, but you should pass the root file handle printed by the first ccfs as an argument to the second ccfs.

Now you can use test-lab-3.pl to test your file system by passing the two root directories that you just created:

```
anguish$ ./test-lab-3.pl /classfs/dir1 /classfs/dir2
Write and read one file: OK
Write and read a second file: OK
Overwrite an existing file: OK
Append to an existing file: OK
Write into the middle of an existing file: OK
Check that one cannot open non-existant file: OK
Check directory listing: OK
Read files via second server: OK
Check directory listing on second server: OK
Passed all tests
```

If test-lab-3.pl exits without printing "Passed all tests!", then it thinks something is wrong with your file server.

You should re-start ccfs before you run the tester. If you run the tester more than once without re-starting ccfs, the tester is likely to see stale information cached by the NFS client for the second file system. You don't need to solve this problem for Lab 3, but if you're interested, the solution is to implement mtime and ctime for files and directories.

Hints

You should continue to consult the NFS v3 specification for details on how NFS should operate. You may also want to consult the book "NFS Illustrated" by Brent Callaghan.

SETATTR

Here's how to extract the argument for the SETATTR RPC:

```
setattr3args *a = nc->Xtmpl getarg<setattr3args> ();
//nfs_fh3 a->object : the file handle that the client wants to set
attributes
//sattr3 a->new_attributes : the new attrbutes
//sattrguard3 a->guard : ignore this argument
```

Here's an example of how to set the size and mtime attributes (atime is similar to mtime):

```
fattr3 attr;
if (a->new_attributes.size.set)
   attr.size = *(a->new_attributes.size.val);
if (a->new_attributes.mtime.set == SET_TO_CLIENT_TIME)
   attr.mtime = *(a->new_attributes.mtime.time);
```

Here's how to generate a reply for the SETATTR RPC:

```
wccstat3 *res = nc->Xtmpl getres<wccstat3> ();
```

```
res->set_status (NFS3_OK);
*res->wcc = make_wcc (old_attr, new_attr);
nc->reply (nc->getvoidres ());
```

WRITE

Here's how to extract the argument for the WRITE RPC:

```
write3args *a = nc->Xtmpl getarg<write3args> ();
// nfs_fh3 a->file : the file handle where the client wants to write.
// uint64 a->offset : the offset to start the write
// uint32 a->count : the length of data to write
// stable_how a->stable : ignore
// opaque a->data : handle for the data
```

The last argument is essentially a pointer to the new data. Below is an example of how to copy its contents to a str variable.

```
str newdata (a->data.base (), a->count);
```

After a WRITE the file length should be MAX(old_length, offset+count). A WRITE should never make a file shorter. If an application wants to overwrite a file, which can cause the file to be shorter, the NFS client will first send a SETATTR and then a WRITE. The SETATTR has new_attributes.size.set=true, and new_attributes.size.val=0, which should cause your server to set the file length to zero. The WRITE then writes the number of bytes specified by the client. The resulting file can have a shorter or longer length than the original file.

Note that you'll have to update the size of the file (in its fattr3) as a side effect of most WRITE RPCs and when SETATTR explicitly asks your server to do so.

READ

Here's how to generate a reply for the WRITE RPC:

```
write3res *res = nc->Xtmpl getres<write3res> ();
res->set_status(NFS3_OK);
res->resok->file_wcc = make_wcc (old_attr, new_attr);
res->resok->count = /* The number of bytes of data written */;
res->resok->committed = FILE_SYNC; //Since this is how we implemented
the write.
nc->reply (nc->getvoidres ());
```

Here's how to extract the argument for the READ RPC:

```
read3args *a = nc->Xtmpl getarg<read3args> ();
// nfs_fh3 a->file :  the file handle where the client wants to read.
// uint64 a->offset : the offset to start the read.
// uint32 a->count : the length of data to write.
```

Here's how to generate a reply for the READ RPC:

```
char value[bytes_read];

read3res *res = nc->Xtmpl getres<read3res> ();
res->set_status (NFS3_OK);
res->resok->eof = /* true, if already end of file; otherwise, false.
*/;
```

```
res->resok->file_attributes.set_present (false);
res->resok->count = bytes_read;
res->resok->data.set (value, bytes_read);
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-3-handin.tgz` produced by running these commands in your ~/lab-3 directory.

```
% tar czf lab-3-handin.tgz *.[TCchx] Makefile
% chmod og= lab-3-handin.tgz
```

Copy this file to `~/handin/lab-3-handin.tgz`. We will use the first copy of the file that we find after the deadline.