6.824 - Spring 2005

6.824 Lab 2: Basic File Server

Due: Lecture 5

Introduction

In this lab you'll start implementing the Frangipani-like file server originally outlined in Lab 1:

```
----------------     -------------
|              |     |           |
|   App   FS---|-----| block srvr|----- FS on other hosts
|    |    |    |  |  |           |  |
|--------------|  |  -------------  |
|    |    |    |  |  -------------  |
|     Kernel   |  |  |           |  |
|  NFS Client  |  ---| lock srvr |---
|              |     |           |
----------------     -------------
```

We supply you with a block server, block server client interface, and a skeletal file server that knows little more than how to parse NFS RPC calls. Your job in this lab is to design and implement directories. You'll have to choose a representation with which to store directories in the block server and implement the CREATE, LOOKUP, and READDIR NFS RPCs. You will need to consult the NFS v3 specification for information about what the NFS RPCs mean and what arguments/responses are used for each RPC. You can also consult the book "NFS Illustrated" by Brent Callaghan.

You can learn more about NFS loopback servers and asynchronous programming in the loop-back NFS paper. You can find the sources for this software at www.fs.net or in /u/6.824/src/sfs1 and /u/6.824/src/classfs-0.1. The NFS protocol is specified in XDR in /u/6.824/include/sfs/nfs3_prot.x and the output of the RPC compiler is available as nfs3_prot.h.

The CCFS Server

Download the lab starter files from http://pdos.csail.mit.edu/6.824-2006/labs/lab-2.tgz to your home directory on one of the class machines.

```
% wget -nc http://pdos.csail.mit.edu/6.824-2006/labs/lab-2.tgz
% tar xzvf lab-2.tgz
% cd lab-2
% gmake
```

Now you should start the block server on one of the class machines. You'll need to choose a UDP port number that other students aren't using. If, for example, you choose to run the block server on host frustration on port 3772, you should type this on frustration:

```
frustration% cd ~/lab-2
frustration% ./blockdbd 3772 &
```

At this point you can start up the file server, called ccfs. By default, the file server initializes a new file system in the block server, chooses a new random file handle for the root directory, and prints out the root handle. You can optionally specify an existing root handle. (The block server keeps its state in memory, so you can't use a root handle after you re-start blockdbd.)

When ccfs starts, it mounts itself as a file server for a sub-directory under /classfs, as specified on the command line. (Each student has their own view of /classfs so you do not need to pick a unique directory name.) You must also tell ccfs the host name and port number of the block server so that it knows where to get and put data. The following example starts ccfs and mounts a new filesystem on /classfs/dir using the block server you just started on frustration:

```
pain% ./ccfs dir frustration 3772 &
root file handle: 2d1b68f779135270
pain% touch /classfs/dir/file
touch: /classfs/dir/file: Input/output error
pain%
```

Clearly the file server is not very functional. It implements a few NFS RPCs (FSINFO, GETATTR and ACCESS) and returns error replies for most other RPCs (e.g. CREATE, LOOKUP, and READDIR). When you're done with the lab, you should be able to run commands like this in your file system:

```
pain% ./ccfs dir frustration 3772 &
root file handle: 01a2d816f726da05
pain% cd /classfs/dir
pain% ls
pain% touch a
pain% ls -lt
total 0
-rw-rw-rw-  0 root  wheel  0 Feb  9 11:10 a
pain% touch b
pain% ls -lt
total 0
-rw-rw-rw-  0 root  wheel  0 Feb  9 11:10 b
-rw-rw-rw-  0 root  wheel  0 Feb  9 11:10 a
pain%
```

If all goes well, your file server should also support sharing the file system on other hosts via the block cache. So you should then be able to do this on frustration:

```
frustration% ./ccfs dir frustration 3772 01a2d816f726da05 &
frustration% cd /classfs/dir
frustration% ls -lt
total 0
-rw-rw-rw-  0 root  wheel  0 Feb  9 11:10 b
-rw-rw-rw-  0 root  wheel  0 Feb  9 11:10 a
```

```
frustration%
```
The extra argument to ccfs (01a2d816f726da05) is the root file handle printed by the ccfs on pain. It tells ccfs on frustration where to look for the file system in the block server.

Your Job

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

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. For example, you don't need to implement the exclusive create semantics of the CREATE RPC.

Please modify only fs.C, fs.h and fsrep.x. You can make any changes you like to these files. 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 can test your file server using the test-lab-2.pl script, supplying your directory under /classfs as the argument. Here's what a successful run of test-lab-2.pl looks like:
```
pain% ./test-lab-2.pl /classfs/dir
create file-69301-0
create file-69301-1
...
Passed all tests!
```
The tester creates lots of files with names like file-XXX-YYY and checks that they appear in directory listings.

If test-lab-2.pl exits without printing "Passed all tests!", then it thinks something is wrong with your file server. For example, if you run test-lab-2.pl on the fs.C we give you, you'll probably see an error message like this:

```
test-lab-2: cannot create /classfs/dir/file-69503-0 : Input/output
error
```

This error message is the result of this line at the end of fs::nfs3_lookup(), which you should replace with a working implementation of LOOKUP:

```
  nc->error(NFS3ERR_IO);
```

The Block Server

The goal of the block server is to provide a centralized storage location for all the data representing your distributed filesystem, much like a hard disk would. In later labs you will be serving the same file system contents on multiple hosts, each with its own file server. The only way they can share data is by reading and writing the block server.

The block server stores key/value pairs, with writes limited to a maximum of size of 8K (8192 bytes); Both keys and values are byte arrays; the block server does not interpret them. The block server supports put(key,value), get(key), and remove(key) RPCs. Your code will use the client interface provided by blockdbc.h and blockdbc.C. The skeleton file server class already contains an instance of the client interface that you can use via db->put() and db->get(). Have a look at fs::get_fh() and fs::put_fh() in fs.C for examples.

This block server is somewhat simplistic in that it only stores data in memory; if you restart it, all the data previously stored will be lost.

File System Representation

In this lab you must choose the format for file and directory meta-data. Meta-data includes per-file information (for example file length) and directory contents. In future labs you'll have to choose a format in which to store each file's contents. The format you choose for your data will need to reflect the 8K block-size limit of the block server.

NFS requires a file system to store certain generic information for every file and directory, such as owner, permissions, and modification times. This information corresponds to an i-node in an on-disk UNIX file system. The easiest way for you to store this information is to store an NFS fattr3 structure in the block server, using the file handle as the key; there is already some code provided that works along these lines. Then when an RPC arrives with the file handle as argument it is easy to fetch the corresponding file or directory's information. The file server we give you uses the file handle as key and an fattr3 structure as the block value, but you are free to change this setup.

The other meta-data that you must store in the block server are the contents of each directory. A directory's content is a list names, each with a file handle. Keeping this information allows you to handle CREATE, LOOKUP and READDIR RPCs: The CREATE RPC must add an entry to the relevant directory's list, the LOOKUP RPC must search the list, and the READDIR RPC must return each entry from the list.

Since you're storing this information in the block server, you have to choose a key under which to store the information, and a format for the information. We have provided you with a suggested representation in fsrep.x and the XDR machinery for translating between an in-memory representation and one that can be sent to the block server:

```
str data; // Initialized from something like db->get ()
fs_dirent dirent;
if (!str2xdr (dirent, data)) {
  fprintf (stderr, "Unable to unmarshal data\n");
  return;
}
```

```
  // Now can use dirent.allocated, dirent.fh and dirent.filename.

  // Create a new directory and add a directory entry
  fs_directory dir;
  dir.entries.push_back (dirent);
  str dirrep = xdr2str (dir);
  // Flush it under key (stored in xxx) to the block server.
  db->put (XXX, dirrep, wrap (this, ....));
```

Thus, you can directly declare fs_dirent and fs_directory structures in your C++ code and then use the xdr2str and str2xdr functions to convert between an in-memory and serialized version of these datastructures.

The design of the NFS RPCs assumes that the server implements a file system similar to that of UNIX. You may want to look at Sections III and IV of The UNIX Time-Sharing System by Ritchie and Thompson to learn about the UNIX file system. Section IV of the paper explains how UNIX represents the file system on a hard disk, which might help you think about how to represent a file system in the block server.

Hints

Implementing the NFS Protocol

Many of the fields in the fattr3 structure aren't very important. You can see an example of initializing a new fattr3 structure (which your CREATE RPC will have to do) in fs::new_root(). Since you're creating an ordinary file you'll want to set the type field to NF3REG, not NF3DIR. Here's some sample code to initialize a fattr3 structure for an ordinary file in a CREATE RPC:

```
    int mode = 0;
    if(a->how.obj_attributes->mode.set)
      mode = *a->how.obj_attributes->mode.val;
    nfs_fh3 fh;
    new_fh(&fh);
    fattr3 fa;
    bzero(&fa, sizeof(fa));
    fa.type = NF3REG;
    fa.mode = mode;
    fa.nlink = 1;
    fa.fileid = fh2fileid(fh);
    fa.atime = fa.mtime = nfstime();
```

Your CREATE RPC should check to see if the file already exists. It it does, it should just return the existing file handle.

You will probably need to provide a directory entry for "." in the root directory that refers to the root directory itself. The ls command sometimes needs to list files from ".". You could try to fake "." on the fly in the LOOKUP RPC implementation, or create a real directory entry in fs::new_root_cb().

Look in fs.C at fs::nfs3_getattr() and fs::nfs3_access() for examples of complete RPC handlers. You can change them if you want to. Look at fs::nfs3_create(), fs::nfs3_readdir(), and fs::nfs3_lookup() for sample code that uses the RPC arguments and formats an RPC reply. You will need to change these last three functions, and you will also have to add one or more callbacks to each as part of reading and writing blocks.

General hacking and debugging

You can learn more about the asynchronous programming library (wrap, callbacks, str, and strbuf) by reading the Libasync Tutorial.

You may be able to find memory allocation errors with dmalloc by typing this before running ccfs:

```
pain$ dmalloc low -l dmalloc.log -i 10
```

You can see a trace of all RPC requests that your server receives, and its responses, by setting the ASRV_TRACE environment variable, like this:

```
env ASRV_TRACE=10 ./ccfs ...
```

You may find ASRV_TRACE useful in helping you figure out what to work on first: you'll want to implement whatever RPC shows up first in the trace output with an error reply (or no reply at all). Chances are the first such RPC will be a LOOKUP.

You can find out where a program crashed if you run it under gdb (the GNU debugger).

```
pain% gdb ccfs
(gdb) run dir localhost 5566
```

Then run the tester, or whatever it is that makes ccfs crash. gdb will print something like this when the program crashes:

```
Program received signal SIGSEGV, Segmentation fault.
0x806a108 in fs::nfs3_create (this=0x8144600, pnc=0x814d240) at
fs.C:749
749       *((char *) 0) = 1;
(gdb)
```

This means that the program crashed in function nfs3_create(), in file fs.C, at line 749. Often the problem is actually in the calling function. You can see the entire call stack with gdb's where command, which shows you function names and line numbers. You can move up and down the call stack with gdb's up and down commands. You can print arguments and variables of the current point in the call stack with the print command.

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-2-handin.tgz` produced by `gmake handin`. Copy this file to `~/handin/lab-2-handin.tgz`. We will use the first copy of the file that we find after the deadline.