6.824 - Spring 2006

6.824 Lab 1: Lock Server

Due: Lecture 3

Introduction

This is the first in a sequence of labs in which you'll build a multi-server file system in the spirit of Frangipani. In this lab you'll supply the logic for a lock server that you'll use in subsequent labs to keep multiple file servers consistent.

At the end of all the labs, your file server architecture will look like this:

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

You'll write a file server process, labeled FS above, using the loop-back NFS toolkit. Each client host will run a copy of FS. FS will appear to local applications on the same machine by pretending to be an NFS v3 server. The FS server will store all the file system data in a block server on the network, instead of on a disk. FS servers on multiple client hosts can share the file system by sharing a single block server.

This architecture is appealing because (in principle) it shouldn't slow down very much as you add client hosts. Most of the complexity is in the per-client FS server, so new clients make use of their own CPUs rather than competing with existing clients for the server's CPU. The block server is shared, but hopefully it's simple and fast enough to handle a large number of clients. In contrast, a conventional NFS server is pretty complex (it has a complete file system implementation) so it's more likely to be a bottleneck when shared by many NFS clients. The only fly in the ointment is that the FS servers need a locking protocol to avoid inconsistent updates. In this lab you'll implement the core logic of a lock server.

Getting Started

We provide you with a skeleton RPC-based lock server, a lock client interface, a sample application that uses the lock client interface, and a tester. You should fetch this code from http://pdos.csail.mit.edu/6.824-2006/labs/lab-1.tgz to one of the lab machines using wget. pain% wget -nc http://pdos.csail.mit.edu/6.824-2006/labs/lab-1.tgz

```
pain% tar xzvf lab-1.tgz
pain% cd lab-1
pain% gmake
```

Now start up the lock server, giving it a port number on which to listen to RPC requests. You'll need to choose a UDP port number that other students aren't using. For example:

```
pain% ./server1 3772
```

Now open a second window, log into one of the 6.824 lab hosts, and run `lock_demo`, giving it the host name and port number on which the server is listening:

```
frustration% cd ~/lab-1
frustration% ./lock_demo pain 3772
Asking for the lock...
Got the lock. Sleeping...
Releasing the lock.
frustration%
```

`lock_demo` asks the lock server for a lock, waits for the server to grant the lock, pauses for a few seconds, and then releases the lock back to the server.

The lock server we've given you always grants lock requests immediately, and ignores release RPCs. As a result it will give the same lock to multiple clients, which makes for a pretty poor locking system. You can see this for yourself by starting two lock_demo programs in two different windows at the same time -- you will see that they both say "Got the lock" at the same time, rather than one waiting for the other to release.

We have supplied you with a program `lock_tester` that tests whether the server grants each lock just once at any given time. You run lock_tester with the same arguments as lock_demo. A successful run of lock_tester (with a correct lock server) will look like this:

```
frustration% ./lock_tester pain 3772
Acquire a, release a, acquire a, release a:
Acquire a, acquire b, release b, release a:
Acquire a, acquire a, release a, release a:
Acquire a and b from two clients:
lock_tester: passed tests
```

If your lock server isn't correct, lock_tester will print an error message. For example, if lock_tester complains "error: server granted lock 61 which we already hold!", the problem is probably that lock_tester sent two simultaneous requests for the same lock, and the server granted the lock twice (once for each request). A correct server would have sent one grant, waited for a release, and only then sent a second grant. The 61 is the lock name "a" in hex.

Requirements

Your job is to modify lock_server.C and lock_server.h so that they implement a correct lock server. Intuitively, correctness means that the lock server grants a given lock to at most one requester at a given time. More precisely, consider the sequence of grant and release RPCs that leave and arrive at the server for a particular lock. The grants and releases must strictly alternate in the sequence: there should never be two grants not separated by a release.

Your lock server must pass the lock_tester tests. We will test your lock_server with the standard client code, not your copies of the client code. Thus you should not make any significant changes to files other than lock_server.C and lock_server.h.

About the Lock Server

The lock server's RPC messages are defined in lock_proto.x. The protocol has three messages: acquire, release, and grant. The client sends the acquire RPC to the server when the client wants a lock. The server sends a reply back, but the reply has no meaning. At some point (perhaps immediately if the lock is free) the server will send the client a grant RPC, indicating that the client now possesses the lock. The client replies immediately to the grant RPC, but the reply has no meaning. When the client is done with the lock, it sends a release message to the server.

The lock server can manage many distinct locks. Each lock has a name; a name is a string of bytes. Names might not be printable ASCII, which is why the lock software prints lock names in hex rather than ASCII. The set of locks is open-ended: if a client asks for a lock that the server has never seen before, the server should create the lock and grant it to the client.

lock_client.C and lock_client.h implement a client-side interface to the lock server. The interface provides acquire() and release() functions, and takes care of sending and receiving RPCs. See lock_demo.C for an example of how an application uses the interface.

Different modules in a single application might ask for the same lock at roughly the same time. lock_client.C handles this by sending two separate acquire RPCs to the server. When one of the modules is done with the lock, lock_client.C releases the lock back to the server and waits for a second grant.

Tips

Coding plan

You'll need to modify class LS in lock_server.h to keep track of which locks are currently granted and which clients are waiting for locks. There is just one instance of class LS in the lock server.

You'll need to modify the LS::acquire() function in lock_server.C to check the status of a lock when an acquire RPC arrives, and only send a grant if the lock isn't currently held. acquire() should also update your record of which locks are held and what clients are waiting.

You'll need to modify LS::release() in lock_server.C in order to update the lock's status, and to potentially send a grant RPC to a waiting client.

To give you a feel for how much work should be involved, our solution adds about 40 lines of code to lock_server.C.

libasync

Both lock_client.C and lock_server.C are written in an event-driven style using the SFS libasync libraries. You can learn about event-driven programming in Section 4.4 of the NFS toolkit paper, or in Using Libasync (you can skip Section 2.2), or by reading the Libasync Tutorial. You can find the source for the libraries in `/u/6.824/src/sfs1` or at the SFS project page.

You may want to use the ihash template class to store state about the set of active locks, and the list template class to store the list of clients waiting for a lock. You can see examples of their use in lock_client.h and lock_client.C.

You can declare an ihash hash table as follows:

```
struct entry {
  ihash_entry<entry> hlink; // ihash uses this
  str name;    // the hash key of this entry
  int a_field; // data you want to store in the entry
  int another_field;
};
ihash<const str, entry, &entry::name, &entry::hlink> table;
```

ihash arranges the entries in a linked list per hash bucket, and strings the items of each list together with the hlink field. To make C++ type checking happy, you have to explicitly mention the key type, the entry type, the name of the entry field that contains the entry's key, and the name of the hlink field in the ihash<...> definition.

An ihash hash table has three operations defined on it. table[key] yields a pointer to the entry with the indicated key, or 0 if there's no such entry. table.remove(entry *) removes an entry, given a pointer to the entry. table.insert(entry *) inserts a new entry, given a pointer to the new entry. You'll need to allocate the entry with the C++ new operator, like so:

```
entry *e = new entry();
// initialize e, particularly e->name
table.insert(e);
```

Here's an example of a definition of a list, called lst:

```
struct element {
  list_entry<element> llink;
  // things you want to put in each list element...
};
list<element, &element::llink> lst;
```

The methods implemented by list include insert_head(element*) and remove(element*). The following code iterates through the elements of the list called lst. If you modify the list inside the loop, then you should save a copy of lst.next(i) before modifying the list.

```
element *i;
for(i = lst.first; i; i = lst.next(i)){
  ...;
}
```

An alternative to the list class is the vec class which is another way to manage a dynamic list of elements. Compared to list, vec does not require a structure with an internal llink object. Thus, a vec is parameterized with only one argument: the class to be stored in the vec. vecs allow random access much like an array, elements can be added to the end of the vec, and then can be removed from either the front or the back. The vec class interface is specified in `/u/6.824/include/sfs/vec.h`.

The str class implements constant, garbage-collected (reference counted) variable-length strings. The constructor str("xxxx") creates a new str object initialized from the C string "xxxx". The str method cstr() returns a null-terminated C string pointing to the contents of the str object. Equality with == is defined on str objects, and returns true iff the two strings contain the same bytes. str objects have an explicit length, so they can contain nulls.

Don't modify a str. Don't modify the result of str.cstr(), since cstr() returns a pointer to the str's internal storage.

Debugging

You may be able to find memory allocation errors with dmalloc by typing this before running server1:

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

You can tweak settings as desired; see the manual for more information (also available locally via running "info dmalloc"). Your program must exit cleanly for dmalloc to produce a report (into dmalloc.log).

You can see a trace of all RPC requests that your server receives, and its responses, by setting the ASRV_TRACE environment variable. There is a corresponding ACLNT_TRACE variable that displays RPCs made by a process and their responses. Since the lock server must both receive and make RPCs, you may want to run with both:

```
pain% env ASRV_TRACE=10 ACLNT_TRACE=10 ./server1 ...
```

Smaller numeric values will display information less verbosely.

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 code 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-1-handin.tgz` produced by `gmake handin`. Copy this file to `~/handin/lab-1-handin.tgz`. We will use the first copy of the file that we find after the deadline.