Carlos III

Exercise 1

Austin Ramos

Distributed Systems

The following requirements impacted my design decisions:

- All the operations in the api must be done atomically in the server side
- Developed server must be concurrent
- There must be no imposed limit on the number of elements that can be stored
- Client code must be linked to the static api library
- Client code may **only** contain calls to the api, can not establish any message queues directly.

Firstly, I decided to hold the tuples in memory as a dynamic array. I found a vector Implementation here: http://eddmann.com/posts/implementing-a-dynamic-vector-array-in-c/, which was straight forward to add to my makefile, link to my main program(on the server side), begin using to store the tuples "request" struct. The struct is defined in the "message.h "header file. It contains an integer key, a float value, a string value, and one more string value to hold the name of the client message queue. While there were more roundabout ways to have access to the client message queue name, I found this to be the simplest from a readability point of a view as well as from a pragmatic one.
To store tuples, I first declared and initialized an empty vector on the server side called "messages". The index of the vector is not relevant to the key value in the struct. I created a simple helper function on the server side called "vector_index()" which will take a key value as an argument, and return the index of the struct containing that key. This is used in a variety of the api method functionalities. It is a simple linear search.

Client side

As specified in the requirements, the clients are set up in such a manner where all you need to do is use the associated api in the static library described below. The last argument of each and every api method call is a char * qname, to pass along the name of the client queue to the keys.c code as well as the server side eventually.

Keys.c and libkeys.a static library

While it seems cumbersome to pass the client queue name to each call, it does a great deal of simplifying on the server side. It also gives the extra (potential_ benefit of being able to see which client has added what message to the server side vector.

Conceptually, I assigned each of the methods to be implemented in keys.c an integer value from 1-6, shown below:

1 – init()
2 – set_value()
3 – get_value()
4 – modify_value()
5 – delete_key()
6 - num_items()

Each of these method calls will open a client and server queue to begin client server communication. At first, it will always send a fake "dummy" struct, with the client message

queue name, and the integer **corresponding to the 1-6 values above** stored in the "key" value of the struct. This is because the server side is at first waiting to receive this "dummy" struct, so it can read this key value to determine what type of call it is expected to do(as the keys.c code has to be done atomically on the server side). I receive the response from the server and return it as an integer in most cases to the client.

Server Side

On the server side, everything up until the while(1) permanent loop was more or less copied from the slides in class. One difference is I created an array of 5 pthreads, one for each potential thread process function(modify value is simply a call to delete_key and a subsequent call to set_value, and is not treated as its own function really).
As Described above, once the while(1) permanent loop begins, the server awaits the initial "dummy" struct, which will indicate what function it is supposed to execute, or rather, what changes(or information to return) it will make to the messages vector.
There is then a 5 part if-else branch conditional, for each of the api method calls above(except for modify value which is treated as a compound of delete and set).
Within this conditional, I will create the pthread to its corresponding thread process function. Sometimes it is necessary to await a second struct(which will contain the actual information, for set_value, for example) before going into this alternate thread process.
These processes will send responses back to the client on their side and then close the client queues.

Generating executables:

I have found, at least on my virtual machine(linux mint) that the message queues are very finicky. There is a default msg_max size less than the size of my request struct, so it is important to run the following shell command
" sudo sh -c 'echo 512 > /proc/sys/fs/mqueue/msg_max' "

I was also able to solve many issues by deleting the leftover empty queues from the disk(as they are stored on the actual linux os). They can be found in the directory "/dev/mqueue" and can be removed with a simple "rm" command.

To build the two executables, all that is necessary is to run the command "make" in the linux terminal. This will compile the server executable, compile the keys.c code, archive it into a static library, and handle all the static library and pthread linking. The demos would then be available by running "./server" on one terminal, and "./client#" on another.

Test plan:
There are two demo client1, client2 , which show different functionalities

Client1.c
will check the the number of items is correctly 0 on start.
It will add 1 tuple, show that the method on the server side returned that it was a success.
Then it will check the new total number of elements stored, which will be 1.
It then adds another tuple.
Shows totalnum =2;
It deletes 1, shows total num = 1;
Then it will init()
And show server side has been reset and totalnum is at 0 again.

Client2.c-
Will add 2 tuples. Will get 1 tuple and print it out.
Will change a tuple and print it out.