Project Four Write-Up

Written by Sandile Keswa

Overview

The overall goal of the project was to achieve a client-server system with shared constant resources - all within the same process. I dare say that my implementation achieves all the aforesaid with gusto. In just over 200 lines of code, I devised a system in coherence with the requirements. At the core of my implementation is the posix message queue. Essentially the server listens for requests and returns in one message queue, puts pending requests in another message queue and lastly puts notifications in each of the four message queues for the threads. As a result, by this design, the same implementation could operate with very minor alterations in a multi-process environment as opposed to the multi-threaded one it exists inside right now.

Types

Message
An abstraction for the messages being sent between client and server

- byte - type - represents what action the message calls for; can be MESSAGE_TYPE_PUT, MESSAGE_TYPE_GET, MESSAGE_TYPE_GOAHEAD
    - MESSAGE_TYPE_PUT - this message is returning resources
    - MESSAGE_TYPE_GIVE - this message is taking resources
    - MESSAGE_TYPE_GOAHEAD - this message is a callback for client messages
- size_t - count - the size for the current operation (put or get how many)
- byte - senderId - represents who sent the message

Functions

void *client(void *args);
Function executes client thread logic

- Parameters
    - void * - args - the arguments for the client thread
- Return Value
    - Returns NULL

void *server(void *args);
Function executes server thread logic

- Parameters
    - void * - args - the arguments for the server thread
- Return Value
    - Returns NULL

struct mq_attr *newAttr();
Function generates new message queue attributes

- Parameters
- Return Value
    - A pointer to a new message queue attributes object

Message *newMessage(byte type, size_t count, byte senderId) ;
Function creates a new message

- Parameters
    - byte - type - the type of the message
    - size_t - count - the number of resources affected
    - byte - senderId - the sender of the message
- Return Value
    - Returns the new message

Message *cloneMessage(Message *existingMessage);
Function clones an existing message

- Parameters
    - Message * - existingMessage - the message to clone
- Return Value
    - Returns a new shallow copy of the existing message

float randomFloat();
Function generates random number between 0 and 1

- Parameters
- Return Value
    - Returns a random positive float less than 1

Testing

Testing this project was a tedious affair since it was an exercise in concurrency. I started off putting log statements everywhere to catch bugs. However, this became futile once I got going because there were so many log statements. I ultimately started to use gdb which proved awesome since it tells you when there is a context switch. So what I would do normally is compile my program with debug flags and manually step through and check every value at every point using 'next' and 'print' commands. In the end my big issue was waiting for the wrong amount of time (years instead of seconds) and that issue was corrected. My workflow was made much simpler by putting together a make file to start up gdb and recompile in one fell swoop (make debug) - I even had a make function for commiting to git (make commit).

Execution

To run and compile my code, simply employ:

make run

The rest will be taken care of by the make file