Maiah Pardo, mapardo

Asgn2 Writeup

A description of my testing along with interesting things I found when testing:

- I first worked on implementing the multithreading part of the project. Then I added flock() calls to lock and unlock the file descriptor and mutexes to protect critical sections and race conditions in my code. I tested that this works by creating scripts which pulled large files from u/dev/random. At first I tested just one HEAD request, then a PUT request, then a GET request (using diff on the files) to make sure my result was correct. Then I added more and more PUT, GET, and HEAD requests in my script (not in parallel) and tested this as well. I tried to "PUT" to a file, and "GET" from the same file. During this test I found that I was having read and write errors as my files were not matching up with diff. After some more research I found that flock() is not declared to be thread-safe by its documentation, and flock() was not protecting my file from being accessed by two threads wishing to read and write from it at once, and had to come up with a solution.
    - During the test I ran when I PUT to fileA, at the same time asking to GET from fileA, I added a print statement inside of the GET after the call to flock() which told me the size of fileA. The size of fileA was less than its final size when the PUT finished. So my GET did not execute properly.
- My solution was to use a mutex around all calls to flock(). At this point I was calling flock to lock and unlock, with mutexes around both of these. When I did this, however, I found that my program was hanging and figured out my threads were in a deadlock situation:
    - ThreadA acquires the mutex lock to call flock
    - ThreadA puts a lock on the file
    - ThreadA releases the mutex lock
    - ThreadB acquires the mutex lock to lock flock
    - ThreadB waits for the file lock that A has
    - ThreadA tries to acquire the mutex lock to call flock to unlock the file
    - Deadlock
- I tried to fix this by removing my mutexes around the flock() call to unlock the file, but then I had read-write errors again, since flock() is again, not declared to be thread-safe by its documentation.
- I found that the call close() on a file descriptor unlocks the file, so I then took out the flock() call to unlock the file and allowed close() to do this job instead.
- Finally I created tests which ran many variations of HEADs, PUTs, GETs in parallel, some all PUT calls, some all GET calls, etc, and tested the resulting files with diff. I used both binary and text files.

- Next, I implemented the logging and healthcheck. I tested this by creating two C scripts, for one, I copied my code for the log checker (to see if a file is valid log file) I ran this on many different large and small log files that my server created. I also created a C script which checked that my hex output was correct. It would take a file name as an argument and create two output files, one which created hex from the input file using my algorithm to create hex, and another which created hex from the input file using the system call 'xxd -p -l 1000 -c 1000' command given to us and ran a system call "diff" on the files. This was how I checked many binary and text files for the correct hex output.

- I also ran my code on the test scripts provided by other students and passed all of the tests successfully: testing.c, asgn2-test.sh, healthcheck-test.sh

Questions:

1. I started my server for asgn1. For the files I placed 8 different files from dev/urandom of sizes around 400 MiB. When I sent 8 GET requests at once the time it took was: 0.470 seconds.

   I started my server for asgn2 with 5 threads and no logging. I used the same files. Then when I sent 8 parallel GET requests at once, the time it took was: 0.129 seconds.

   The observed speed up is: 0.470/0.129 = 3.64 times faster

2. One of the bottlenecks of my system is capping the number of threads that we have. This number is chosen by the user, so we have no control over this – but it is a bottleneck because if our user chose a number of threads much smaller than the number of client requests received, then we would only be able to process the number of requests as we have threads at one time. Another bottleneck of my system is my process of dispatching: I use a queue in which I enqueue the connection file descriptor in the producer function (then signal to a waiting thread that work is to be done), then dequeue it in the consumer function. Since the queue is a critical region which must be protected by a mutex, only one thread may dequeue or access the queue at a time, and our threads can only be dispatched one at a time. This reduces the concurrency of the program, especially when there are more client connections asking to connect than there are threads, because then our clients must wait for an available thread. When we are logging, we are writing to the log file at the same time and do not wait to write to the log file, even if another thread is currently writing, but in order to do this, we must access a shared variable, the log file offset. The log file offset tells us where to write to, and then we update it with how large our write will be. Accessing this shared variable is a critical region since we don't want multiple threads to grab the same value or update it at the same time. Since the log offset can only be accessed by one thread at a time, some threads must wait for access, and we lose some concurrency. Another bottleneck of the system is that we must lock the files we access when appropriate. If we are processing a PUT, which writes to file A, and just a moment later we receive a request to

GET file A, or read from file A, then we must lock the file until the thread processing the PUT has finished writing. Blocking the access to reading until done writing to a file is necessary, but will reduce concurrency of the program.

I believe that the way to increase concurrency in the program is to remove the logging, remove the PUT operation, and not cap the number of the threads available to the program to use. If we remove the logging, we no longer must have threads wait to access the log file offset. If we remove the PUT operation, we will no longer be writing to a file, and therefore there will be no need to wait for a read after write. If we do not cap the number of threads available to our program, we can process as many client requests as we want without waiting for a free thread.