明文 0416329

Introduction to Operating Systems

Project #2 Report

Multitasking

Testing Environment

Host OS: PopOs (Ubuntu based)

Host Hard: i9 9900kf (16 threads), 32GB ram

Software: Oracle VM Virtualbox

Guest OS: Ubuntu

Guest Hardware: 2 cores ,4GB ram

The guest environment is the same the Tas provided

Code: client.js, handler.js (provided by TA)

sample\_server.c modified from code provided

sample\_server\_multithreaded.c multithreaded version of sample\_server.c

Task

On this project we are asked to implement a multithreaded server which communicates with clients sending data to it. The client first sends a random string of length 5,call it message. After receiving the data, the server creates a different random string, also of length 5, call it seed. The server concatenates the received message and the created seed into a new string which is sent to the SHA-256 function. Which the sent string the function creates a hash value of length 64. The server proceeds to check if the created hash starts with 4 zeros. If it doesn’t then it proceeds to create a new hash and repeats such process until it creates a hash that starts with 4 zeros. After the required hash is created, we send the client a string of the following format, “message,seed,hashvalue” , commas included. The client receives the data and it confirms that it was made with the original message it sent.

Single Threaded Performance

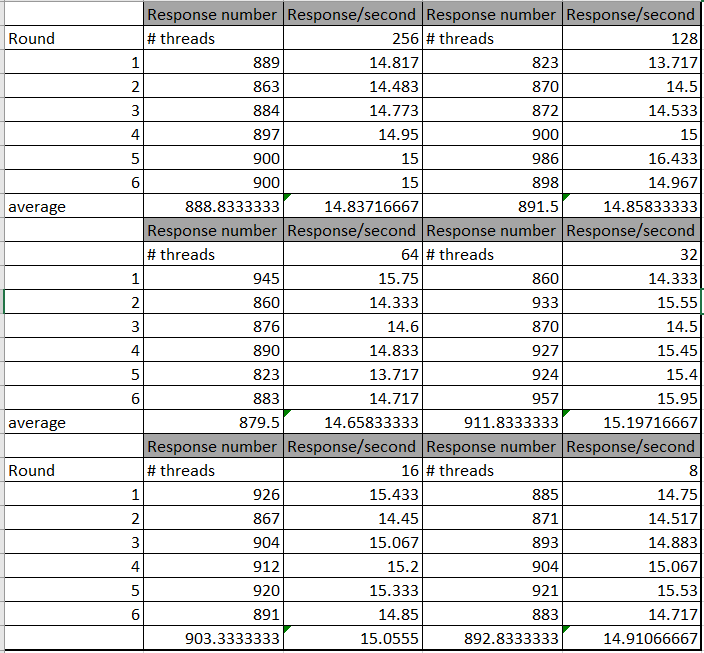
A close up of a sign

Description automatically generated

Multithreaded implementation

I used the POSIX threads library to make this a multithreaded task. I first create a pthread\_t pointer that can hold the CONN\_NUM, constant defined at the beginning of the file, amount of threads. Then I also create the same amount of int pointers that will hold the one return value from accept\_connection(sock), one pointer holds one value not the same as the pthread\_t pointer which holds all CONN\_NUM amount of pthreads\_t. It’s called client\_fd. I also separated the process of receiving the data, creating the seed, hash and sending it back to the server into one function called handle\_client. Then on every iteration a new \*client\_fd is created, it accepts a connection and finally I create one thread with pthread\_create with the current pthread pointer position, NULL, a reference to the handle client function and the current client\_fd. This creates a thread that will handle a call to the handle\_client function. The thread closes itself when the length of the data received is equal to zero. Since the client program runs for 60 seconds, our server can now have a pool of threads receiving and sending back data at the same, by this we mean while one thread is beginning is running the function for one data, another thread receive the next data without waiting. Instead of only having one thread receiving, performing the task, and sending it back. To run the server please add the additional flag -pthread when compiling it (ex: “gcc sample\_server\_multithreaded.c -lssl -lcrypto -pthread”) then run the a.out file (./a.out).

I tried various sizes for the thread pool and have listed the performances in the following table.



I wouldn’t dare to say that this data is significant as every config was only run 6 times and there no major differences in the performance. Even when running the same config multiple times, the variance of the results is like that of the other configs. The real change was when I ran the program on my host computer, not the environment on virtualbox.

The results here:

A screenshot of a cell phone

Description automatically generated

I believe the improvement is because I am running 16 physical threads instead of 2 on the environment. Even though I don’t understand why a pool of 8 threads works at least 100-200 more responses it is still the case as the response number never went below 1400. I also took some screenshots of the CPU usage during both 16 and 8 thread pool configs. The 16 thread pool config uses 100% of all 16 threads while the 8 thread pool config uses 100% but of only 3 threads. I can only say that the bottleneck is probably my code and it is not optimized to properly use all threads.

Here are the screenshots of the CPU usage during both 16 and 8 thread pool configs:

8 thread pool config

A screenshot of a video game

Description automatically generated

16 thread pool config

A close up of a scoreboard

Description automatically generated

Finally I would like to add that sometimes the server fails to receive the data from the client. This was the case in both the single and multithreaded implementations. I did not know if we were allowed to change the server setup for the connections and from what I saw on the template it didn’t seem to have errors on the setup. If it doesn’t work at first, close the server and restart it, also waiting 2 o 3 seconds after the server starts seems to work all the time.

EDIT AFTER SHA-256 REPLACEMENT:

After replacing the old sha-256 function, my program keeps running as before. I wasn’t encountering errors before so I assumed there would be no noticeable difference. One observation is that the program might be running after as I now average around 15.200 Response number per second on the environment. Before it would be below 15, it might be random good runs so in the end I don’t feel there is a big impact.