Design Document

Author Chun Zhou

Student ID: 1698844

Project Description

This project is to modify the HTTP server that is already implemented to include two additional features: multi-threading and logging. Multi-threading will allow the server to handle requests from multiple clients simultaneously, each in its own thread. Logging means that the server must write out a record of each request, including response information. It needs to use synchronization techniques to service multiple requests at once, and to ensure that entries in the log from multiple threads aren't intermixed.

Project Logic

The program will take a port number as a parameter, but this time it will also have two additional optional parameters: "N", defining the number of worker threads, and "l," the name of the log_file to hold the contents of logging. Either of these parameters can be omitted, which would result, respectively, in the server running with a default of four threads, or without logging.

The program will use multi-threading to improve throughput. This will be done by having each request processed in its own thread. The typical way to do this is to have a "pool" of "worker" threads available for use. The server creates N **worker** threads when it starts; N = 4 by default, but the argument `-N nthreads` tells the server to use N = n threads. In addition to the worker threads the server will also have a **dispatcher** thread, which manages accepting connections and sending them to workers. Each worker thread waits until there is work to do (a connection to handle), does the work (executes the request(s) sent over the connection), and then goes back to waiting. Worker threads may not "busy wait" or "spin lock"; they must actually sleep by waiting on a lock, condition variable, or semaphore.

The dispatcher thread listens for connections and, when a connection is made, alerts (using a synchronization method such as a semaphore or condition variable) one worker thread in the pool to handle the connection. Once it has done this, it assumes that the worker thread will handle the connection itself, and goes back to listening for the next connection. If there are no worker threads available,the dispatcher must wait until one is available to hand off the request.

If the -l log_file option is provided to the program, it must log information about each request to the file log_file.

Setting up the global pthread_mutex_t for the usage in all functions. In main, initialize the mutex and use them in the sub functions for each worker. Parse the input argument using getopt to get the number of the worker thread needed and the name of the log file. After that, check the connection and use the dispatcher thread to assign each connection to each worker in their function. Call the handle_connection in each function after they receive their work. Within each handle_connection, we also need to perform the write to the log file with different log information. In this process, use the semaphore/mutex to lock the critical region in order to make each thread write on the log file separately.

Data Structures

The program uses the implementation of the HTTP server, in which it creates the socket as the server for the client to send the request and response through the Socket as well.

In addition to the last HTTP program, this time, the program deals with the idea of multithread. The programs will create the numbers of worker threads by using pthread_create which is based on the command line option -N, otherwise 4 as default. And there will be shared thread_record as the input arguments to each thread. The shared thread_record will be a type that is defined by typedef and it will contain shared memory with semaphore and lock.

After receiving the connection, the dispatcher will check if any worker thread is free with sem_wait and send the connection id to the share list. The worker will receive the id and start handling that connection. At the end of each handle connection, the worker thread will open and write to the log_file if the option -l is used. While writing to the file, the worker will use lock and unlock to maintain the critical region clean.

Every worker thread and dispatcher thread will continue to loop in there until the connection closes.

Functions

- Void writelog(char *log_file, char *method, char *file, char *hostname, int response_code, pthread_mutex_t *lock, char *http, int length, sem_t *count)

  The function that deals with the write to log_file. The function will check if the log_file is accessible and write the correct log message to the log_file. Based on the response_code and http information, it will write different messages to log_file. If the log_file is first time open, the function will truncate the log_file. Otherwise, continue writing to the next line. Before and after the writing, there will be locks that prevent other threads access to the log_file at the same time.

- Typedef struct thread_record

  In order to use the thread more efficiently. This is a class that contains all the shared memory resources for workers and dispatchers threads to use. And later will be created at main with all the share variables provided to each thread.

- Void * worker(void *arg)

  The function of the worker thread, input is the pointer to the type of shared information class. This function uses the shared information to deal with the connection/request. It will use semaphore to check if there is availability of the connection and lock the share memory of the connection id. After taking the id out, it will unlock and call handleconnection. And at the end, it will post semaphore 2 to indicate it is free and loop again to seek new connection.

- Void * dispatcher(void * arg)

  Very similar to the worker thread.

- Void handle_connection(int connfd, pthread_mutex_t *lock, char* log_file, sem_t *count)

  This function is the main function that uses the socket connfd to handle the main part of “recv” the Request message from the client. The function also parses the Request message into different parts. First it checks the format of the object name and obtains the corresponding response_code if any errors are discovered. Then it decides the request type. And call the corresponding function to deal with the Request. And lastly call the response function to send the response message back from socket. After that, it will call the writelog function with an associated argument if the log_file option is used in HTTP.

- Void put(char* file, int* response_code, char* body, int length)

  The function is to perform the PUT request from the client and change the corresponding response code that reflects any errors discovered from the process to the function handle_connection. The function takes the filename, response_code, body message and length of body which are all parsed out and verified from the handle_connection. The function will “create/open” the file based on the object name and “write” the body with the length of it onto the file. Any errors discovered will change the response_code.

- char* geth(char* file, int* response_code, char* body_get, int* size)

  The function performs both GET and HEAD requests since they are very similar. If the request format is correct, return the corresponding body for the GET request. The function takes the filename, response_code which are all parsed out and verified from the handle_connection. The function will “open” the file based on the object name and get the file size with the “stat()” and “read” body with its size into the char* body_get and return body_get to the handle_connect. Any errors discovered will change the response_code.

- Void respond(int connfd, int response_code, char* method, char* body_get, int body_length)

  The function that helps to send the response message with corresponding response_code and request method.The function takes input of connfd, response_code, method, body_get from the either geth or put function. The function will implement the response_code with each different request_method into different strings and set it as the response message and send it back to the client.

Question

Repeat the same experiment after you implement multi-threading. Is there any difference in performance? What is the observed speedup?

After implementing multi-threading, there is a dramatic difference in performance. The speed in the previous takes a few minutes to finish the whole process while after the multi-threading it will only take less than a minute to finish. The observed speedup is more than 2x.

What is likely to be the bottleneck in your system? How do various parts (dispatcher, worker, logging) compare with regards to concurrency? Can you increase concurrency in any of these areas and, if so, how?

The bottleneck in my system should be the speed of dealing with a single connection/request. Regarding concurrency, dispatcher, worker and logging seems to speed up the system a lot because it allows the parallelism working to satisfy when multiple connections happen. The one way that I can increase the concurrency in the logging is to use pwrite which can allow the different threads to log on the same file together without interrupt with each other.

For this assignment you are logging only information about the request and response headers. If designing this server for production use, what other information could you log? Why?

When designing this server for the production use, I think I can log the information about the client so I can collect the datas from different clients and create the cache or something for later usage in order for future time reduction. That the cache can directly respond to their request when the server is down or so.

We explain in the Hints section that for this Assignment, your implementation does not need to handle the case where one request is writing to the same file that another request is simultaneously accessing. What kinds of changes would you have to make to your server so that it could handle cases like this?

In order to deal with this situation, the one way I can think of is to copy the log_file each time it finishes writing and make the copy file available for another request to access so the request can write to the file while another request can access the copied file. Or just simply lock /semaphore the file to prevent another request to access at the same time.