Design Document for

Multithreaded Web Server

Author: Sunny Zhang

Project Description

Httpserver is a simple program written in C that mocks the behavior of a web server, it can process request types such as GET, HEAD, and PUT and can send feedback to the client if there is anything wrong regarding the request or the server. What's more, users can send multiple requests within 1 connection or multiple requests in multiple connections. Additionally, httpserver can also handle requests coming in different connections in a multi-threaded fashion, with the options for users to choose how many threads to implement and whether to log the outcome of the request or not. The program works in the Linux Ubuntu 20.04 environment. To send requests to the server, users could use curl or ncat to send whatever GET, HEAD, or PUT requests they want.

Program logic

Program will first receive a certain amount of data each time until substring "\r\n\r\n" is found in the data, then parse the data using tokenization to get the request type, file path, http version, and potentially content length if there is any. Then after parsing, it will check the data collected and see if they match the general requirement, if so, it will proceed to dispatch the information to corresponding helper function depending on the request type, if incorrect, then it will send back 400 bad request or 501 not implemented to the client and close the connection. For a get request, it will attempt to read the file from the file path, if nothing goes wrong, then it will read the file content and send it back to the client, if something is wrong, it will send back an error message according to the error type. For a head request, it will attempt to read the file from the file path, if nothing goes wrong, it will send back the length of the file content to the client, if something goes wrong, it will then send the error message back according to the error type. For a put request, it will first check if the file exists or not, if exists, it will attempt to write to the file with data from the body of the put request, and send back 200 OK at the end, if the file does not exist, it will then create a file and write the data from the body of put request into that file and return 201 Created at the end to the client. After successfully processing the request, it will check if there is a second request in the same connection, if not, it will close the connection and wait for the next connection.

For multi-threading, the server will first initialize the thread pool and set up mutual exclusion tools such as mutex and conditional variables. What's more, the server will then use a Queue data structure to queue up the connections for different threads to take. Each thread can only take 1 connection out of the queue and only if the queue is not empty, if the queue is empty, then the conditional variable will put the thread to wait until signaled that there is a new connection in the queue, and all of these are done inside a critical region locked by the mutex. After taking the connection out of the queue, the thread then can start processing the requests from that connection, and log the outcome and the end of each processing. When logging, the server will take the file descriptor out of the queue data structure and set the file descriptor inside the queue to be -1, meaning it cannot be accessed by other threads at that time, and set the file descriptor in the queue to its original value when it is done logging. There is also a flag inside the queue data structure to indicate whether the user wants to log or not, so when the server tries to log it will first check the flag.

My system is thread-safe because I did not use any function that is not multi-threaded safe, and I used mutual exclusion tools such as mutex and conditional variables. The variable that is shared between threads is the queue data structure, and the data inside the queue, which is the connections, log file descriptor, and the flag for logging. The Queue is modified whenever connections and log file descriptors are modified, and when connections are modified, there is either a new connection coming in the queue or when one of the threads is grabbing the front connection out of the queue. The log file descriptor is modified when one of the threads is done with a request and try to grab the descriptor for the log file to log the outcome, which will change the log file descriptor data inside the queue to be -1, and it will be modified again when the thread is done with the logging and put the descriptor back to the queue. The critical region is when the server is trying to enqueue or dequeue connection from the queue. It is also a critical region when the thread try to grab the file descriptor from the queue or put it back in. My design for logging each request contiguously works because only 1 thread can access the file descriptor for the log file at one time, whenever the file descriptor for the log file is grabbed from the queue, it will turn to -1 which would tell other threads to wait until it returns back to normal.

Data Structures

The program used 1d arrays to store all the information gathered from the client with the usage of dynamic memory allocation. There is 1 main array used to store a certain amount of data received from the client which is about 10000 byte large, and 1 “passer” array that is used to read data from the client and pass them to the main array, and it is about 50 byte large. Depending on the size of the request, the main array can dynamically allocate more memory for storage if needed.

For multi-threading, the program utilizes a queue data structure with linked list implementation for storing and directing the traffic of incoming connections. What’s more, inside the queue data structure, it also stores the file descriptor of the log file and the flag for logging for convenient reasons.

Functions

- getRequest(int request, char* filePath, char* httpVersion, char* hostName, struct Queue* q)

This function takes the file descriptor of the request, file path, http version, hostName, and the queue data structure as inputs. It first checks if the file exists or not, if file does not exist it will send back 404 error, if the file does exist, it will then check if it has permission to read the file, if we do not have permission to read, then send back 403 error. If nothing goes wrong, it will first send back the 200 OK response then start reading the content of the file using a temporary buffer and read 10000 bytes at a time and write to the client immediately after reading. After finishing reading the file, the function will log the outcome and return back to the main program.

- putRequest(int request, char* filePath, char* httpVersion, char* hostName, char* requestBuffer, int content_length, int requestBuffer_Byte_size, int requestBuffer_size, struct Queue* q)

This function takes the file descriptor of the request, file path, http version, host name, the buffer that stores the information of the request, content length, the current size of the buffer, the capacity of the buffer, and the queue data structure as inputs. It

first checks if the file exists or not, if the file exists, it will then again check if it has permission to write, if no permission, then it will return 403 error, if it has permission, then it will proceed to open the file for write. And if the file does not exist, it will create a file according to the file path. Then it will starts reading from the buffer to search for the body to write into the file, it keeps track of the size of the buffer and the content_length that still needed to read, if the data size inside the buffer is bigger than content_length, it will take out the body from the buffer and directly write that to the file. If the data size inside the buffer is smaller than content_length, it will first transfer everything inside the buffer into the file and then read more from the request and read up to the capacity of the buffer at a time and write that into the file immediately. When content_length is 0, it will stop reading from the buffer and return 200 OK if file originally exists and 201 Created if not. When everything is done, it will log the outcome and return to the main program.

- headRequest(int request, char* filePath, char* httpVersion, char* hostName, struct Queue* q)

  This function takes the file descriptor of the request, file path, http version, host name, and the queue data structure as inputs. It first checks if the file exists or not, if file does not exist it will send back 404 error, if the file does exist, it will then check if it has permission to read the file, if we do not have permission to read, then send back 403 error. If nothing goes wrong, it will first send back the 200 OK response then start reading the content of the file using a temporary buffer and read 10000 bytes at a time and keep track of the size. After finishing reading the file, the function will send the content length of the file back to the request and log the outcome and return back to the main program.

- thread_func(void* arg)

  This function is where the worker threads are dispatched to and work forever until the program ends. The function will take in one input of void and transform it to its original structure which is the queue that is used to store the connections. Then there is a while loop in which the worker thread will constantly wait for a new connection to come out of the queue using conditional variables and mutex. If an available connection

is found then it will get the connection out of the queue and pass that to the handle_connection function to handle the requests.

- putFileBackQueue(struct Queue* q, int f)
  This function puts the file descriptor back to the queue.
- getFileFromQueue(struct Queue* q)
  This function grabs the file descriptor for the log file from the queue and sets the data in the queue to -1 to indicate that it is being occupied.
- enQueue(struct Queue* q, int c)
  This function is a typical enqueue function for the queue data structure, it first checks if the queue is empty then try to append the new found connection c at the end of the queue.
- deQueue(struct Queue* q)
  This function is a typical dequeue function for the queue data structure, it first check if the queue is empty, if it is empty then returns -1, if it is not empty, it will pop the first node of the queue and update the order of the queue, then it will return the file descriptor for the connection stored inside the node.
- getLogFile(struct Queue* q)
  This function would first check if the server needs to log by checking the flag stored inside the queue data structure, if the log is needed, then it would proceed to grab the file descriptor of the log file from the queue data structure by calling the getFileFromQueue function. If the log is currently being occupied by other threads, meaning getFileFromQueue returns -1, the condition variable for log will then let the thread wait until it is signaled that the log file is free. The critical region in which the function is grabbing the log file is locked by the mutex for log file.
- log_to_file(int successful, int log_file, int err_code, long int content_length, char* requestType, char* filePath, char* httpVersion, char* hostName, struct Queue* q)
  This function is used to implement the logging, it will first check if the log is needed, if not it will simply return by doing nothing, else it will proceed to see if the request to log is successful or not, if it is successful it will log the request using the success format, else it will log the file using the fail format. All of the

input such as err_code, content_length, requestType, filePath, httpVersion, and hostName are information needed to log. After logging, it will put back the file descriptor of the log file back to the queue data structure, and the action of putting it back is locked by the mutex for the log file as a critical region.

Testing:

When testing the program, I did unit testing and test 1 function at a time. For example, when I was testing my getRequest function for GET request, I will comment out the part for HEAD and PUT and isolate the testing to only the GET part. After finishing testing every function, I used curl and ncat to send multiple requests in a single connection and different connections to see if my program can handle persistent connections. What's more, I used flags such as -Wall -Wextra -Wpedantic -Wshadow to see if there are any warnings in my program.

When testing the multi-threading part of the function, I used a test.sh script to try to simulate a situation where multiple connections are trying to connect to the server all at once, the logging functionality is also a great tool for me to verify the correctness of the logic of my multithreaded server implementation.

The script inside the test.sh is similar as follow:

```
curl localhost:8080/test0file012345 &
curl localhost:8080/0123456789abdec \
    localhost:8080/abc123456789abc -I &
curl localhost:8080/1234 &
curl 127.0.0.1:8080/abcdde012345678 -T test.sh &
```

There are 4 connections in the script using curl for different get, put, and head requests.

Improvement on having a multithreaded web server

The difference is significant, performance wise it is better since multithreading is CPU intensive, meaning we increase the usage of the CPU for this program and it processes tasks much faster and efficiently due to that reason. The observed speedup varies for different

numbers of threads, but in general it is about 2 times faster when having 2 or 3 threads than not having multiple threads.

Bottleneck

The bottleneck in my system is most likely the logging, there is not a very efficient way to kind of “reserve” the space for writing, the only one that makes sense is probably filling in a bunch of 0s beforehand and replacing those parts later with the real logs. However, in theory, the time it takes to write the character ‘0’ and other characters such as ‘a’, ‘b’, ‘c’, and ‘d’ should be about the same. Therefore it is not much more efficient to fill in the 0s if you already know the amount of log content to write and what to write. Thus when logging happens in one thread, all the other threads can only wait for that thread to finish writing into the file then can take over and write their logs. However, this bottleneck only happens when every thread is done with their requests so fast that they are all waiting to write the logs and if the logs take a long time to write. Other than that case, the concurrency of dispatcher, worker, and logging work just as fine. I can definitely increase the concurrency if I can come up with a clever way to “reserve” the space for logging so that other threads won’t need to wait for the write time.

Also, my multithreaded web server cannot deal with the case where multiple connections are trying to process put requests to the same file, which would easily cause race conditions and data conflict.

Possible production use improvement

If I am designing this server for production use, I would also try to log the time when the request is finished processing, and maybe attach a serial number to record the requests so I can know if there is any request that I missed or lost during processing.

Additional possible improvements

For handling multiple put requests writing to the same file, I could make priority queues so that whenever a put request is followed up by another put request accessing the same file in different threads or different connection at the same time, it will put the later request into the queue after the first put request, when the first put request is done, it will then start processing the second put request. Whether they should overwrite the file or append to the file depends on the implementation and the purpose of the put.