Design Document

Author: Eric Yu
Student ID: 1671640

Project Description

httpsever is an implementation of a simple local server that handles GET, PUT and HEAD requests from the client (curl requests). The output of the implementation is a response sent from the server back to the client.

Program Logic

httpserver will take a request and parse the request accordingly depending on the type of the request. If the request is GET, the server will parse the request, and retrieve the file name. If the file name is not of fifteen alphanumeric characters, it will return a 400 ‘Bad Request” status code. If the file meets the naming criteria, it is then checked for read permissions. If the file is unreadable, it sends a 403 “Forbidden” status code. If the file is readable, then it is processed and stored into a buffer until it is sent back to the response. If the request is HEAD, the server will parse the request, and retrieve the file name. If the file name is not of fifteen alphanumeric characters, it will return a 400 ‘Bad Request” status code. If the file meets the naming criteria, it is then checked for read permissions. If the file is unreadable, it sends a 403 “Forbidden” status code. If the file is readable, then the response will return the same result for GET, but without the contents of the body. If the request is PUT, the server will parse the data for the file to read and the contents to write into. The name of the file being written into is checked if it is made of fifteen alphanumeric characters. If it is not made of the specified characters, a 400 status code is sent. The file is then checked for writing permissions, if it is denied, the server will respond with a 403 status code. If the filename is valid, but the file does not exist, it is then created and written into. The server will retrieve the content of the first file and store it in a buffer. When the file that is to be written into is ready, then the buffer will be written into the to-be-written file. If the to-be-written file was created, the server will respond with 201 “Created” status code. In all three cases of GET, HEAD, and PUT, if the file being read or written already exists and proceeds as expected, the status code used in the response will be 200 “OK”. If the type of request is not GET, HEAD, or PUT, the server will return a 501 “Not Implemented” status code. If there exists an error that none of the current error status code could catch, the server will return a 500 status code.

Notable features that are included in this implementation consists of logs and threads, making the server that was previously constructed for assignment one more complex. First the server should be initialized with any order of port number, log name, and thread number. While it has to contain the port number for the server to function, log name and thread number are simply a feature that can be utilized. With the new features of threads and logging, the server is now more capable to efficiently process each request provided by the client. The way the server functions with threads is that through a queue, it enqueues connections and dequeues a connection from the queue to process when a thread is available. With the use of mutex locks, this prevents data race. Data race is essentially when multiple processes are accessing the shared memory at once, leading to conflict. Using the mutex lock, whenever a process is

enqueued/dequeued, or executed via handling of the connection, it would theoretically keep the program thread safe, or simply put, no data races. With the logging feature, which each request, we can write and store an instance of a request try/execution. What this does is that allows the server to keep track of requests that have been processed and the response given in return for the request.

Data Structures

httpserver makes use of arrays, a queue, and a log. Arrays are used to pass data into a memory allocation such that it can be read later by a separate instruction. These memory allocations, or buffers, would then be freed to deallocate the memory after use. The queue is used to process requests such that the processor is not overwhelmed with many requests. Essentially, while a worker is available to work, it dequeues the request from the buffer. While all workers are busy, the queue will spin until workers are free for more requests. The log data structure is to identify whether or not the log feature is considered in the execution of the server. If the log feature is used for the execution of the server, it holds the shared memory of the log file name, and the port number.

Functions

- Log newLog(char* name, uint16_t port)

  This function initializes a new log object to create space for shared memory accessible by the different requests. Specifically, it allows access to the log file name and the port number used by the server. This is primarily to create log statements to write into the log file. The log will be set to 0 if the log feature is not executed by the server.

- Node newNode(int *i)

  This function initiates a new node with its value pointing at the request pointer, and a pointer looking at the next node in the queue

- void enqueue(int *client socket)

  This function reads in the request, and creates a new node with the request as its value. It first checks if the tail is NULL, if so, it means that the current node that was made is the current head of the queue. If the tail is not NULL, then it can be said that there is a node in front of the currently made node. The new node is then set as the tail, and is pointed at by the node in front of it.

- int* dequeue()

  This function takes the front of the queue and returns its value while redirecting the pointer of the node that is behind it. What this means is that the queue head is not pointing at the node behind the current head node to allow a safe deallocation of memory.

- uint16_t strouint16(char number[])

  This is a helper function designed to convert a string and convert it into a uint16_t type variable The method used to achieve this is using the strtol system call.

- void * handle connection(void *p_connfd, void *p_log)

  This function handles the request and returns a response based on the response. It takes in a pointer to the request and a pointer to a log object. The pointers are then typed casted to what they are meant to be, and are used to process the requests. Based on a request, the handle connection will handle the request accordingly based on its type(get, head, put), version, and file name. Once the request is handled, the function then returns a statement that identifies whether or not there were issues with the request. If the request fails, it would return the error status message depending on the type of failure. Similarly, based on the conditions that the requests are correctly processed, it will send an appropriate response. Based on whether or not the log feature is executed by the server, there will be a log statement that will be written into the log file stating the state of the request that has been processed. How certain status codes are used and printed are down accordingly as mentioned in the program logic.

- void * worker (void *arg)

  This function takes in a log pointer, and is processed to handle connection. This function manages The mutual exclusion aspect of the program to prevent any data races. Mutex locks are used such that only one set of instructions can happen at a time, to prevent data to be overwritten by another instruction in parallel. If a thread is busy, it is told to spin with thread conditions and wait until there is a signal given that the thread is free to accept processes.

Question

- Question 1:
  - With multithreading, the requests are processed at a much faster rate. The observed speedup is dependent on the amount of threads present, and eventually cusps at a certain number of threads when a threshold of threads is met.
- Question 2:
  - The bottleneck would be the physical hardware since the CPU only holds so many cores to allow threads to proceed into the cache. Furthermore, the larger the file, the more space is needed, and larger caches would result in decrease in speed. In regards to concurrency, uses atomic read and writes to access data. This means that it forces the compiler to not be smart and enforce certain computations which results in the prevention of data races. To increase concurrency, one method that can be used is to do a set of instructions at a time, instead of processing instructions one at a time. To clarify, it enforces multiple instructions to be processed in the same chunk such that it doesn't require a back and forth for each individual instruction.
- Question 3:

    - The reason why logging information is useful is because in case there is a connection disruption, it would be difficult for the client to determine whether or not there was no response due to connection error, or something happening on the server end, or whether the request ever got to the server in the first place. Some other information that would be nice to log would be the time used to process the request to check if the request has timed out on the server end, or to simply check the efficiency of the server. This information would be important because it provides a scope of improvement, and identifies whether certain requests take longer than others.
- Question 4:
    - Changes that would be required include a mutex and a scheduler, such that whenever a request interacts with a file, it won't conflict with whenever another request is accessing it.