Gail Dylan Salak
CSE 130

Multi-Threaded HTTP Server with Logging Design Document

Overview

Goal: This program builds upon the httpserver.c from assignment 1. The original program acted as a server for a client through one thread. The server handled PUT, GET, and HEAD requests from the client and sent back the appropriate response. The goal of this assignment is to take this fully functional httpserver.c program and multiplex it to handle requests from multiple clients at once through multithreading. In addition, it will have the ability to maintain a log file which keeps a record of the sequence of requests with response information and data. A health check can be invoked to examine the log file and measure performance through the number of errors registered. Using locks and condition variables, we will need to maintain synchronization between the multiple threads as well as during our recording to the log file.

This design document assumes the reader is familiar with the prior design document found at:
https://git.ucsc.edu/cse130/spring20-palvaro/gsalak/-/tree/master/asgn1

Usage: ./httpserver [port#]

- **Flags:**
    - -N [#threads]
        - Without -N flag, program defaults to 4 threads.
    - -l [log_filename]

Program Design

High-Level Description: Since this program builds off of the design of httpserver.c from assignment 1, this program uses all the same functions to handle requests from the clients. Namely, these functions are readResponse(), processRequest(), and constructResponse(). It will use the same libraries, global variables, and structs. It will accept connections from the client(s) in the same manner. We will add more functions, variables, and libraries on top of this existing foundation.

Perhaps most importantly, the biggest addition in this program is the pthread library, <pthread.h>. We will use this library to create threads and pass information between the threads. We will also be using this library to set mutex locks and condition variables that will maintain synchronization between the threads when entering critical regions.

It is worth mentioning that a lot of this design has been inspired by the sections held by Michael Covarrubius. We will create N worker threads that will contain their own variables declared in a struct, worker. Our main() function will act as the dispatcher thread which will delegate work and send data to the worker threads through a while loop that will accept connections and pass the socket descriptors. There will be another function, handleRequest(), which the worker threads will call that essentially performs the reading, processing, and constructing of responses just as in assignment 1. The difference is that this function will also handle the mutex locks and condition variables to put the worker thread to sleep, enter critical sections, and signal the dispatcher when data has been sent. After processing the request, there is also another critical section where the thread allocates space in the log file based on its request's entry.

This function will also call the logging() function which will take care of the actual writing to the log file. Throughout writing to the log file, the function will also keep count of the number of entries and failures through incrementing global variables, ENTRY_COUNT and ERROR_COUNT. A healthcheck will return these counter variables and log itself.

Global Variables:

- ssize_t ENTRY_COUNT = 0
    - will be incremented with every entry to the log file, regardless of failure or not
- ssize_t ERROR_COUNT = 0
    - will be incremented every time there is a failure being logged
- ssize_t LOG_SIZE = 0
    - will be used as an offset to tell each thread where to start writing their entry in the log file

struct worker: This struct is used for each worker thread so they can hold all these different variables.

- struct httpObject message
    - httpObject to handle each thread's request
- ssize_t ID
    - number of each thread
- ssize_t client_sockd
    - client socket sent to each thread by dispatcher
- pthread_t workerID
    - thread's ID
- pthread_cond_t condition
    - conditional variable which the dispatcher uses to signal the threads
- pthread_mutex_t* lock
    - lock for threads to wait until they are signaled
- pthread_cond_t* dispatchCond
    - conditional variable for dispatcher to be signaled when a thread is done
- pthread_mutex_t* dispatchLock
    - dispatcher lock where the dispatcher sleeps until it is woken up
- int logFD
    - contains file descriptor of log file if it was given, contains -1 otherwise
- char logBuffer[16384]
    - logBuffer to hold strings before they are printed into the log file
- pthread_mutex_t* logLock
    - lock to maintain the atomicity of allocating space in the log file

main(): Main function will function largely the same as the main function from assignment 1. Since we are using this function as the dispatcher thread, we are still accepting client connections in this function but will be passing off the work to the worker threads.

First, we must check the command line for arguments to check for any flags or errors. We will do so through the getopt() function. If the -N flag is given, it must be followed by a number or else we throw an error and exit. If the -l flag is given, it must be followed by a filename or else we throw an error and exit. We check for non-option arguments with the "-:" flag which returns the case "\1". From here we can check for the port number which is a mandatory argument.

After parsing the arguments, we then connect to the port given. We then initialize all the variables in the worker threads through creating an array of worker threads called taskForce. After this initialization, we enter the while loop which accepts requests sent across the client socket. There is a count variable which increments with every request, and an accompanying target variable which is count % the number of threads so that the target will always be among the existing threads.

Before signaling the workers, the dispatcher must determine the thread availability. Thus, we enter a lock in which we loop between all the threads until we find a thread which is available. This is done through checking the client socket. With each iteration of the loop, we increment a busy counter variable. If "busy" is equal to the number of threads, the dispatcher waits on the conditional variable dispatchCond to be signaled by a worker thread when it is done with its task. Once we know that there is an available task, we can pass the client socket and signal its "condition" variable.

void handleRequest(void* thread): First, we de-address the pointer of the thread given as a parameter into a struct worker variable we call wThread. Then, we enter a while loop that infinitely loops since each thread will be waiting for a task until it is given one. To ensure mutual exclusion and avoid busy waiting, we have an inner while loop that waits on a signal from the dispatcher thread that sends a client socket which is sending a request. Upon receiving a client socket from the dispatcher, we will exit this while loop and follow the same processing flow from assignment 1 to read, process, and construct a response to a request.

In addition, we check for healthcheck requests so we can alter the process accordingly. For PUT/HEAD requests, we return a 403 error. For GET requests, there must also be logging enabled or else we throw a 404 error. If it's a valid GET healthcheck request, we send the valid response to the client and set a boolean variable, isHealthcheck, to true to let the logging function know that it is a healthcheck entry.

After the completion of the request, we check if logging is enabled through checking the logFD. If there is a log file, we then enter the log mutex lock. Based on the request, it will then calculate how many bytes of space need to be allocated in the log file into the variable, reservedSpace. During this calculation, we can also sprintf the first line of the log entry into the logBuffer which will be used in the logging() function. Next, we will set the start-write variable to the current LOG_SIZE and increment the LOG_SIZE variable by the reservedSpace which we just calculated. It's important that this reservation of space happens in this mutex or else different threads could try to reserve the same space in the log file. The mutex then unlocks and then we call the logging() function. Lastly, when the logging function returns, we reset wThread->client_sockd to INT_MIN and signal the dispatcher that the thread is now available for another task.

void logging(void* thread, ssize_t reservedSpace, ssize_t start_write, bool isHealthcheck): This function begins by writing the first line of the log entry which was printed into the logBuffer in handeRequest(). All writes to the log files are done using pwrite(). It then checks to see if it's a valid GET/PUT request to proceed logging the lines of data. If it's a HEAD request or failure, it will simply log the "--------\n" string to end the entry. With a valid GET/PUT request, we first check the isHealthcheck variable to see if it is a healthcheck request. If so, we sprintf the ERROR_COUNT and ENTRY_COUNT into a buffer called vibeCheck. Importantly, before we put these variables into the buffer, we first decrement the ENTRY_COUNT by 1 into a temporary variable so that we can log this variable without counting the healthcheck itself as an entry. Then, through a for loop, we translate the buffer vibeCheck into hex values into the logBuffer. We pwrite() the logBuffer to the log file, logging the error and entry data.

Logging the body of a GET or PUT, we first open the file from which we will read the body of data. We initialize two counter variables to 0, charCount and dataBytes. In the following loop, charCount will increment by 20 each time to represent the 8-byte character count at the beginning of each data line. Simultaneously, dataBytes will count the bytes read from the file and will cause the loop to break once we have logged the entire message->content_length. We then enter the while loop which begins by reading in 20 bytes from the file. First, we log the charCount padded by zeros through "08zu". Then, through a for loop I got from Clark Hibbert on Piazza, we will translate the 20 bytes read from the file into 2-byte hex characters and log them into the log file. Once we have logged the entire body of data, the loop will exit and we will close the logFD.