Assignment 2: Multithreaded Server **Design Doc** **Emil Ingemansson (eingeman)**

Goals:

The goal of this assignment will be to build off of what we made in assignment 1. The http server will need to add multithreading functionality with the user who starts the server specifying the amount of worker threads and the server needs to have the option of dumping the logs of a request containing the content length, address, and request along with the content of whatever was transferred

Design:

Part 1: Main function / Dispatcher

1) The server in assignment 2 will have two additional optional flags. The -N flag followed by a number will specify how many "worker" threads the http server will have. Second, the -L flag followed by a filename will specify the server to create a log and to dump the contents along with request, address and content length a file with that name.
2) First, we will need to start up the server in a similar fashion to assignment 1 by using the same code. (socket, bind, and listen).
3) Afterwards, we will want to create our threads that will be doing the multithreading. This can be done by calling
    a) Example: Create dispatcher thread
        i) "pthread_t dispatcher_thread
        pthread_create(dispatcher_thread, NULL, worker, NULL);"
4) Declare an array in our program of ints which will hold the socketfd's that need to be serviced by worker threads. This will be a **GLOBAL** variable so that our dispatcher and worker threads can access it and will be called allSocketFDs. We can also have a **GLOBAL** int variable which is equal to 0 and increment whenever a worker starts working. This can be used to see how many worker threads are currently working.
5) Then we will want to go into our while loop where we will accept a connection. This while loop will act as our dispatcher.
    a) Once we accept a connection. We will want to call a lock onto our mutex.
    b) Once we lock the mutex, we can increment the incomingConnections variable.
    c) Then, we will set the socketFD which we got from the accept and set it allSocketFDs[i] where i starts at 0 and we increment it so the next FD goes on the next slot.
    d) We can increment i+1.
    e) We can now unlock our mutex.
    f) Now we either wake up a worker thread or if they're all busy the dispatcher waits.
        i) Check if the currentlyWorking variable is equal to the number of worker threads. If this is true then we need to wait. If this variable is less than the total that means

that at least one worker is asleep and we can signal to it to wake it up to do the work.

ii) If all the workers are busy, then we can put the dispatcher thread to sleep waiting for one of the workers to signal it to wake up. (Meaning one of the workers finished their jobs and went back to sleep).

Part 2: Worker Threads

1) First thing worker thread will do will check if there are any incomingConnections. If there are no incomingConnections, then it will wait for a signal from the dispatcher function.
    a) If our worker thread makes it past here, that means it was signal to from the dispatcher thread and there must be at least one connection waiting to be worked on.
2) The worker thread will lock the mutex.
3) Once we lock the mutex, we will want to loop through out allSocketFDs array to try and find one which is not null. Once we find a file_descriptor, we will want to save it to a local variable my_socket_fd.
4) Once we save the socketfd, we can then set the allSocketFDs[i] of where we extracted our file descriptor to NULL so that it can be used again.
5) We will decrement incomingConnection.
6) We will increment currentlyWorking
7) The worker thread will unlock the mutex.
8) Using the socketFD, we can now call parseHeader() and performRequest() in a similar manner to assignment 1.
9) Now we want to lock the mutex once we have finished parsing through each header and performing each request.
10) Decrement currentlyWorking.
11) Unlock the mutex.
12) Now we can signal to the dispatcher thread in case it is sleeping waiting for a worker thread to become available.

Part 3: Server Logging

1) We will define a function called serverlog which returns voids and takes in three arguments. First, the name of the server log file. Second, the address of the file whose contents we will put into the log file. Third, the offset into the file where we want to start writing into the server log file.
2) Once we are inside of the function, the first thing we want to do is open or create the server log file and save the file descriptor to a variable. Ex: log_descriptor = open(2).
    a) We can open the file by calling open(2). We can specify the flag O_CREAT and if the file doesn't exist already then it will create it. The server will **NOT** use the O_TRUNC flag used in the write function because we do not want to delete the file if it already exists.

3) Once we have the file descriptor saved to variable log_descriptor. We can call open again to open the file at the address given in the arguments and save it's file descriptor to content_descriptor.
    a) We can open this file by calling open(2) in a similar way to the log_descirptor. We will not need any special flags for this and we want to open it O_RDONLY because we do not need to read from it.
4) Once the server have the file descriptors for both files, next we will want to do is use fstat(2) to get the content length of the content_descriptor.
    a) Once we have the content length of the content file, we can then use it to calculate how much we need to add to the global offset that was given in the arguments so that the next thread to call server log can start writing at the offset set.
    b) We can calculate the offset in another function called getOffset() which will take in the content length as its argument. It can get the amount we need to add the offset by first adding the 4 for the "PUT or GET" with a space afterwards, adding 28 for the address length with a space afterwards, 7 for the word "length " with space after.
        i) For the number after length, this can vary. We can use a helper function where we divide by the length by 10 over and over until counting each time until we can no longer divide by then then add 1 to it to for the '\n'.
            (1) EX: Count starts at 0. If length is 10254, we divide by 10, get 1025 count++, then divide by 10 get 102 count++, then divide by 10 get 10 count++, then divide by 10 get 1 count++, then divide by 10 get 0 count++. Once we reach 0 we stop and return count, which in this case will be 5.
        ii) Once we have the size of the first line of our server log, we can now specify how long the rest will be. This can be done easily by dividing the content length by 20 to see how many full lines we have and multiplying them by 69. Then we can take the modules and multiply that by 3 and add 9. (multiply each char by 3 (each hexadecimal is to two places with a space or newline afterwards) and add 9 for 8 bit address and the space
        iii) Lastly, we want to add 9 for the "========\n" to end the log request.
        iv) Add everything together for the total offset.
    c) Once we have the offset, we can lock our global offset. Increment it by += offsetAmount. Then unlock the global offset. This is done so multiple threads don't try to update the global offset at a time and cause a race condition.
5) Once here, we should have the space "reserved" the space inside our log file and can write to it. We can create a local buffer with length 20 and another buffer of length 69.
    a) First call pwrite and set its offset to the value we got from the function arguments.
    b) We can sprintf with the 69buffer[] and set it to be "(get or put) (address) length (content-length)\n"and then write that to the log_descriptor.
    c) Then while(read_count = read(content_descriptor, 20buffer, 20))
        i) while(i < read_count)
            (1) sprintf(69buffer, "%s %02x", 69buffer, 20buffer[i]);
            (2) Add the newline to the 69buffer
            (3) Write to log_descriptor using pwrite(2).

d) Once we finish reading through the whole file, 20 bytes at a time, we can then add the last line signaling the end of the log "========\n"

UPDATES TO ASSIGNMENT 1 DESIGN DOC. ALL CAPS UNDERLINED AND BOLDED IS WHAT IS DIFFERENT IN ASSIGNMENT 2.

PerformRequest(){

1) If the request is a GET
    a. First, our server will try to open the file in read only with resource name that was taken from the HTTP Header.
        i. If this read returns -1, then the file does not exist and we can return our http header with status code 404 Not Found.
        ii. Else the file exists and we can perform the get for the client.
    b. If step 5a is successful, first we need to use fstat() to get the size of our file using the file descriptor we opened.
    c. Once we know the size of the file, we can send back our HTTP header to the client containing our status code and content length.
    d. After we send the header, we can then read from the file we opened and write the bytes we read back to the client.
    e. **ONCE WE ARE DONE WE CAN LOG THE FILE BY CALLING FUNCTION LOGFILE() SINCE THE FILE HAS BEEN CREATED IN OUR SERVER'S DIRECTORY AND CLOSED.**

2) Else if the client's request is a PUT
    a. First, the server will try to open the file and if that returns a nonnegative value, we know that the file already exists.
        i. If the file already exists, we will want to overwrite it so the server will just delete the file that already on the server and create a new file with the same address with what the client is sending. This can be done by specifying the O_TRUNC value in open which will truncate the value to zero when opening a file if it already exists.
    b. Then, we call open to make a new file on the server at the address (with the name) specified by the client with flag O_CREAT and O_RDWR.
        i. Here we can send back the address because we know we can read the bytes in. We will send back a 201 Created HTTP header because we will be making a new file in the server with the address specified by the user.
        ii. We can then we read in the data from the content that the client sent.

    C. **ONCE WE ARE DONE WE CAN LOG THE FILE BY CALLING FUNCTION LOGFILE() SINCE THE FILE HAS BEEN CREATED IN OUR SERVER'S DIRECTORY AND CLOSED.**

3) Then the client sent an HTTP header with a request that wasn't "get" or "put" and we then can send back an HTTP header with status code 400 bad request.

4) Once here, the server has finished the clients request and has already sent back the applicable HTTP error code. We can then close the *new_file_descriptor*.

}