Design

Overview:

So the design of this program is fairly simple. So the design is broken up into header files and the main HTTP server file as described below including the functions in each file:

- Httpserver.c
  - Dispatcher
  - Worker_threads
  - main
- RequestHanlder.h
  - Head
  - Get
  - Put
  - Strtouint16
  - Handle connection
  - Create listen socket
- RequestLog.c
  - Waitlist struct definition
  - Create waitlist
  - Count_bytes
  - Log_request
  - Log_fail
  - Validate_log
  - healthcheck

These functions make up the HTTP server program. I created separate functions for each request because this will make the code more legible to read and create a more organized code. Below I will describe the function and the way I designed them.

Httpserver.c

Handles connfd connections coming in and uses dispatcher and worker threads design.This is where the main thread creates the worker threads and handles connfd when they are coming in and dispatches them to the worker threads using a queue to handle each connection filediscriptor. For locks I used semaphore to increase readability and easy of use.

Dispatcher function

- This function is for the main threads and add new connection fds to the queue.
- Checks if the queue isn't empty, if so wait, does this using a semaphore
- Checks if lock is set if not set lock
- Adds the connfd to the queue and increase the tail of the queue
- Post the lock so other threads can access the lock
- And post the full semaphore the increment the lock if the queue is full

Worker_threads function

- This function is where the worker threads pull the new connection fds from the queue.
- Set a while loop so that a thread continuously grabbing connections that are waiting in the queue
- Checks if the queue isn't empy if so wait otherwise check if the lock is open, does this using a semaphore
- If the lock is open, lock it and continue
- Acces the queue and pull the first connection
- Increment the head
- Post the lock so other threads can access the queue
- Post the empty semaphore so that the dispatcher knows when it is empty
- Call handle connection with the connection fd pulled from the queue also send the shared data struct within the function

Main function

- Checks if the arguments and options are given are valid
- Returns errors if arguments given are not correct
- Checks if logging is requested and checks if log file given already exists, if so checks if the logfile follow guidelines. If log file does not exist it creates the log file.'
- Creates the shared data struct for the threads.
- Creaetes a list of threads, and loops through the list and calls create threads for each threads sending it to the worker thread function.
- This is the main function where the port is set and listen_socket gets called to start listening for connections.
- After setting up the socket and the socket started listening, there is a infinitely loops listening till it connects with a client and calls dispatcher function where it adds the new connection fd to the queue and if the queue is full wait till it isnt, which handles the request made.
- After adding the conneciton fd to the queue the code loops till it get another client connection.

RequestHandler.h
This header has all the functions need to handle the requests made, including setting put the socket.

Head function

- This function handles the head requests
- It first checks if the file requested has the right permissions and if it is even located in this directory, if not it will return either a 403 forbidden message or 404 file not found the message to the client. In either case it will also log the failed request if logging enabled.
- Locks the file with flock, but it is a shared lock so that multiple get and head requests can read the file.
- After it initializes a length integer this will count the number of bytes read for the response to the client.
- Then there is a loop so it reads bytes from the file till it reaches the end.
- I used a buffer of 1000 this allows for a fast read but not any memory issues
- Unlocks and closes the file.
- In the end, it will return a response message that the file is there and the length of the file requested. It also logs the success full request by calling log_request which is defined in LogRequest.h.

Get function

- This function handles get requests.
- This design is fairly similar to the head request function besides the fact the function also has to send the file to the client.
- The first half of the function is the same as the whole head function.
- After this function has looped through the file to calculate the number of bytes, it will send the header for the get response so the client knows the file is there and how many bytes they will be expecting.
- Then the get function will reopen the file and start reading the bytes but instead of counting the bytes it will use send() to send the bytes of data read from the file directly to the client, which will be the body of the Get request. The buff will be dynamically allocated, with the type set an int so binary files can also be read.
- When it finished reading the file it will close the file and unlocks the file.
- In the end, it also logs the success full request by calling log_request which is defined in LogRequest.h.

Put function

- Handles the Put requests
- This function is drastically different from the head and get function.

- Checks if the file is forbidden if so sents a 403 request. It will also log the failed request if logging enabled.
- The function starts off with deleting the file if it is going to overwrite the file. This only happens if the file is there.
- Then the length variable which is the length of the file that is given through the function, sent by the client, will be converted to an integer because it is a string when given in the function.
- Then using open() with flags O_RDWR | O_CREAT, 0666, a new file will be created, setting permissions so it can be read and written to.
- Loops through the buffer given and finds the end of the header
- Then a new buffer is allocated using calloc to set the buffer to all zeros. The type is an int so binary files can also be read.
- Then copies the buffer from when the buffer ended and stores it in another buffer
- Then the file locks
- It writes to the file if there was something after the header in the first recv()
- And int r is initialized which is where the amount of byte read is going to be stored
- Then a while loop goes till bytes is less than zero, which is when all the bytes are read using recv() and written to the new file.
    - Every loop r which is the number of bytes read by the client's file gets subtracted from bytes which are the total bytes that the client sent.
    - Checks if r ever returns anything less than 1 meaning that connection has lost and closes and deletes the file and returns to handle_connection and closes the connection, going back to listening.
- Then it frees the buffer, closes the file, and eventually sends a 201 created response header to let the client know that the new file has been created or overwritten with the file that the client sent. It also logs the success full request by calling log_request which is defined in LogRequest.h.
- The response is sent using a buffer called response which has been dynamically allocated this is to prevent memory from being leaked and memory overflow.

Strtouint64 function

- This code was given in the start code from assignment 1.
- It converts a string to a 16 bits integer.
- This is called the main function of the code.
- It is used to convert the given port number into a 16-bit number since to create a listening socket the port has to be in 16-bit format.

Handle_connection function

- This function handles the client's header parsing and calling the right request function. Also checking if the message sent by the client is okay.
- It first allocates memory for the following variable: host, HTTP version, request type, file, temp file, and buff for reading the client's header. This is dynamically allocated to prevent the overflow of memory.
- There is an infinite loop around the following code because if there is an issue, the code uses the break statement to break out, this allows for fewer lines of code to free the allocated memory, instead, it is outside of the while loop so when there is any issue or reaches the end there is a break statement.
- Within the infinite loop
    - It first reads the header sent by the client
    - Using sscanf it parses through the buff which holds the bytes for the client's header
    - It checks if the file given by the client has the allowed characters
        - It does this by looping through each character and checking using isalnum() to check if it is a number or a letter. It also allows for the characters '.' and '_'.
        - If a character is not any of the allowed characters it will set a variable check_file to 1 and break the loop
    - Then it loops through the given host
        - It loops through the string checking if there are no spaces
        - If there is a space, it will set a variable check_host to 1 and break the loop
    - After that, it creates a temp HTTP version variable that is set to the required HTTP version which is HTTP/1.1.
    - Then it uses an if statement to check :
        - If the given HTTP version is not the same as the variable we allocated space for and set the required HTTP version
        - If either the check_host or check_file are set to 1
        - Or the length of the file is smaller than 2 or bigger than 20 and does not start with '/'
        - If any of these are satisfied it will send the client a 400 bad request response
    - Then, I have for loop that loops 19 times so that it copies the file given to the temp file holder where it copies the whole string besides the first one:

```
for(int i = 0; i < 19; i++){
  f[i] = file[i+1];
}
```

    - It sets a variable to the string "GET"
    - Then checks if the a heathcheck is request

        - If it isn't a get request asking for a health check, returna 403 forbidden request repsonse and log it if enabled
        - If logging is not enabled but it is a get request but logging is not enabled return a file not found request
        - If neither are the case call healthcheck which is defined in LogRequest.h
    - Then it creates two variables that are set to "HEAD", and "PUT"
    - This is so the program can use strcmp() to compare the given request type to those variables so the code know what function to call
    - After creating the variables it will check if any of the variables are equal to the request given by the client
        - If so it will call that request function
        - If it is put it will scan the buffer one more time to get the content length of the file sent by the client
        - Else it will send a 500 not implemented message header to the client saying that the given request type is not implemented
    - Then there is a break statement to break out of the loop
- When the while loop is excited all the dynamically allocated memory is freed and the connection between the client and the host is closed.

Create_listen_socket function

- This code was given in the starter code.
- But get called in the main function to create a listen socket, opening up for clients.
- It uses the 16-bit port number and creates the socket and sets the address and port for the socket
- Returning a listen file descriptor

RequestLog.h

This header has all the functions to handle logging for the server. This header also includes the functions and struct definition for the shared data between threads. It is implemented like this since the shared struct is used both in httpserver.c and RequestHandler.h which is also the case for RequestLog.h so implementing the struct definition and create function within RequestLog.h is the most logical.

In RequestLog.h it defines the struct which has the following attributes:

- A queue for the connection fds
- Integers for the front and the end of the queue and the sizes of the queue
- Also an integer that is set to 1 if logging is enabled
- A string that will hold the log file name
- An offset variable that holds the offset for the logfile
- And four semaphores

    - One that checks if the queue is full
    - One that check if the queue is empty
    - One that locks if the queue is accessed
    - One that locks when logging

The functions that are defined within LogRequest.h

CreateWaitlist

- Creates the waitlist struct
- Takes in size for the queue, a log file name, if logging is enabled, and the end of the log file
- Allocates memory for the struct
- Sets queue with given size
- Sets the variables and initializes the semaphores
- Semaphore full is set to 0 and empty is set to the size of queue

Count_bytes

- Counts the number of the bytes
- Takes in log file name.
- Opens the given file and reads to a buffer and adds the amount of bytes read to the counter till the end of file is reached
- Returns the counter that counts the bytes

log _request

- Logs executed requests
- Takse in the following inputs host, request type, buffer of the first 1000 bytes of the file, length of the file, and the shared data struct
- Initialize a log and a hex array to store the hex and the total log that needs to be logged
    - Hex has allocated 2000 since the buffer can be max 1000 and each byte is two hex bytes
    - And the log is 3000 to account for the log entry
- If the buffer isn't zero meaning that it was not a head request
    - Parse the buffer and convert each byte into temp array of three
    - Than create a for loop that make sure for every byte read the two bytes in hex will, be written right. As seen below.

```
for(int i = 0; i < strlen(buffer); i++){
  char temp[3];
  snprintf(temp, 3, "%02hhx", buffer[i]);
  int x = 0;
  for(int j = i*2; j < (i*2)+3; j++){
    hex[j] = temp[x];
    x++;
  }
```

    - After converting the buffer into hex storing it in hex using sprintf create the log entry and store it in log
- If it is a head request
    - Create the long entery using sprintf without the buffer since that is not necessary for head requests
- Then uses the logging lock to lock
- Then write and find new offset using byte count
- Post the lock so it can be accessed by others
- Close file and free allocated memory for log

Log_fail

- Logs a failed request
- Takes in file name, request type, the code error that needs to be logged, http version, and shared data struct
- Allocates memory for the log that needs to be logged
- Using sprintf creates the log entry
- Then uses the logging lock to gain acces
- And then find the new offset using count bytes and write to the file using pwrite
- Post the lock so it can be accessed by others
- Close the file and free the memory allocated for the log entry

Validate_log

- Takes in a log file name
- Return one if log file is valid else returns 0
- Returns 1 if number of bytes in file is 0
- Opens file and loops through file reading to the buffer
- Counts the number of tabs
- If there is a new line it increment the new line counter
    - If amount tabs is smaller than 2 or greater than four within a new line
        - Return 0
    - Else increment the amount of newlines

    - And reset tabs to 0
- After reading file checks if tabs doesn't equal zero or amount of new lines is not zero
    - This mean that either if there are any tabs that the file did not end with a new line or there were only tabs in the file but no newlines at all
- If it passed all the test return 1 meaning that the log file is validated

Healthcheck

- Checks the amount of entries and failed request are logged
- Sets a lock on the logging to prevent issues when reading log file
- Checks if the log file is a valid
- If corrupted
    - releases the locl
    - Create and send the response to the client that there was an internal server error
    - Log the failed request in the log
    - And return the function
- Open the fiel
- Allocate space for the buffer and read from the file
- Initialize the variables entries and fails and newlines
- Loop till the size of the buffer equals 0
    - Loop through the buffer, if there is a new line increment entries and new_line
    - If newline equals one and buffer[i] == "F"
        - Increment fails
    - Read new 10000 bytes form the file and set the amount of bytes read to bytes
- Free buffer
- Release the lock using post
- Close fd
- Create the body which is the amount of fails and the amount of entries
- Create the response which uses the body
- Send the response and free the response
- Also log the request