Design Document

Author: Miguel Avalos

Student ID: 1704078

Project Description

httpserver is a program that creates an http server that receives requests from clients and attempts to complete their request. After the attempt, the server sends a response to the client containing the code indicating the result and possibly data related to the request. There are three types of requests: PUT to store data into a file in the server, GET to retrieve a file back to the client, and HEAD retrieving file size. This server will be multi-threaded to handle multiple requests at once.

Program logic

httpserver will listen for a connection from a client on the main thread. Once it receives one, the main waits until at least one worker is available. The connection will be handed to the next available worker which is on a separate thread and the number of available workers decreases by 1. The worker thread will be allowed to proceed with its process. The number of worker threads will be 4 unless the user specifies a different valid amount through the -N flag option. For every request on the connection, the worker will attempt to: complete it, respond to the client, and log the result in a file if the user indicates through the -l flag option. While the worker is busy, the main thread will loop. Once the worker is finished, the number of available workers is increased by one and will wait for a new connection file descriptor

Processing requests from a connection:
A request from the connection will be parsed. Then error checking will start. The server will respond with a 400 error and close the connection if one of the following happens: if the request is missing information required for the type of request, fails formatting requirements, has a body when it shouldn't have, or is not http 1.1. 501 error response will be sent if the request isn't HEAD, PUT, or GET. For a PUT request, it will check if the file is invalid by checking if there already exists a file but lacks permissions to overwrite it or if there exists a directory with the same name. Both will respond with a 403 error . For a GET or HEAD request, it will also check if a file is invalid like the PUT request does, but they have the extra requirement that the file needs to exist in the directory. If this requirement is failed, then the server will respond with a 404 error. If the request passes this error checking, then it will likely execute the request successfully with a 200 or 201 code response (with attached data if necessary). Unless the last request had a 400 error, it will check if the client has sent other requests on the same connection; If so, it will process them the same way as above. Once the client closes the connection, the server will close the connection too and listen for a new one. If the connection is closed by the client at any time during this process, the server will not send a response and the connection will be closed.

Data Structures

A character array will be used as a buffer to store data received from a client and data from files that will be sent to the client.

file_log struct
- sem_t* lock - locks file when in use
- char* filename- containing a filename

worker_tools struct
- sem_t* lock - makes worker wait for a job from the dispatcher
- sem_t* waiting - a semaphore that stores the amount of worker available
- int conn_fd - contains connection_fd (if fd is zero then worker is available)
- file_log* - contains a shared file log.

Functions

- void* worker(void * object)
  This function will be in a waiting state until a new connection fd is handed to it from the dispatcher. Once that happens, the function will enter into a busy state and will call handle_connection to process the connection and its data. Once the connection is closed by handle_connection, the function will revert back to the waiting state above. The parameter will be a pointer to worker_tools object.

- int filerror(int option)
  This function is responsible for determining what error occurred while interacting with a file and returning an integer value that will be used as a code for the response to the client. The caller must call this function using the errno(3) return value as the argument for **option**.
  - If the program lacks the permissions to access the file, then 403 will be returned.
  - If the file doesn't exist, then 404 will be returned.
  - For any other error, 500 will be returned.

- void err_response(int code, int conn_fd, bool body)
  This function is responsible for sending http responses to the client if an error occurs. If **body** is true, a body will be sent containing a brief description of the error.
  The responses vary by code:
  - 400 - Bad Request
  - 403 - Forbidden
  - 404 - Not Found
  - 500 - Internal Server Error
  - 501 - Not Implemented

- void discardbody(int conn_fd, int length)
  This function is responsible for discarding **length** bytes of data that are reported by the caller to be part of the body of a request that is not needed.
  If the **length** is less than 1 the function will return. Otherwise, it continues to discard the data up to length or when eof is reached.

- word_length(char* data, int position)
  Will return the length of a word within a string. If the word in the requested position does not exist, it will return 0. (“word-0 word-1 word-2 … word-3”)

- int write_word(char* data, char* buffer, int position, int file_fd)
  Will write a specific word of the requested position from a string to a file. If the word in the requested position does not exist, it will not write anything return 0. If writing the word is successful the number of characters written is returned. (“word-0 word-1 word-2 … word-3”)

- void writelog(int number, bool fail, file_log* log, char* header, char* host, char* buffer)
  Will write a log description of the request attempted to be serviced into the log file if a file name is provided by the user. Uses lock design instead of pwrite.

- int readline (int conn_fd, char* line)
  It is responsible for reading a line from a header request into an outside buffer. Will indicate if the line was filled with other characters, the line was blank, or the connection was cut.

- bool process_request(int conn_fd, file_log* log)
  This function is responsible for parsing the request for the client and calling the necessary function to complete the request if no errors were found with the request. It will log every request result if a log name is provided. A return value of true indicates that the connection needs to be closed. If the connection is closed by the client at any time during this process (including all function calls), the server will not send a response and the connection will be closed..

\

1. Attempt to get the first line into a buffer of 4001 bytes (one space is dedicated to ‘\0’ character, receive byte by byte to find a newline character that ends the first line.
2. While reading all data before the body, search the remaining request data for the content-length component and host component.
3. Let variable bad be false.
4. If the filepath is not exactly 16 characters (it includes the /), a 400 response will be sent.
5. If the host doesn’t exist or the http version is not 1.1, set bad to true.
6. Scan for command and /filepath in the buffer. If they are not detected, set bad to true.
7. (If bad isn’t already true) If not all of the characters in pathname are alphanumeric, set bad to true.
8. Match command to head, put, or get. If none match and the request wasn’t bad, respond with a 501 error with a body, discard the remaining body and return true. If the request was bad, respond with a 400 error with a body.
9. Depending on the type of request:
    a. For PUT requests:

            i. Respond with a 400 error with a body and return true if:
                1. bad is true
                2. content-length's value: does not exist, is negative or contains a non-digit.
            ii. Otherwise, call the putfile function with the filepath**,** content-length's value, and **conn_fd**. If error occurs with request, discard all body contents and send an appropriate error response.
        b. For HEAD requests:
            i. Respond with a 400 error without a body and return true if:
                1. Bad is true
                2. Context-length's value exists
            ii. Otherwise, call headfile function with the filepath and **conn_fd**. If error occurs with request, send an appropriate error response.
        c. For GET requests:
            i. Respond with a 400 error with a body and return true if:
                1. bad is true
                2. Context-length's value exists
            ii. Otherwise, call getfile function with the filepath and **conn_fd**. If error occurs with request, send an appropriate error response.
    10. Return false to caller if this point is reached

- void handle_connection(int connfd)
  This is responsible for handling the connection to the client.
    1. Will Call process_request to handle the request data sent unless one the following happens:
        a. (Does not apply for the first request of established connection) The last request has a 400 error
        b. the connection was closed by client
        c. If there is no more data that will be received from the client
    2. Close the connection with the client and the server reverts back to listening for a new connection.

These functions are called after the request head is processed, the body of the request is still unread.

- int headfile (char* filepath, int conn_fd)
  This function is responsible for sending the file size in response to head request if no errors are encountered.

    1. This function will first check if there are errors with the file path provided before proceeding in reading and will return a negative integer error code if there is an error detected:
        a. -403 - Lacking permission to access the file
        b. -404 - The file does not exist
        c. -403 - The path refers to a directory
        d. -500 - Other errors
    2. The response header will be sent until the Context-Length's value.

3. The file size will be measured then sent to the client.
4. Any remaining “\r\n” will be sent.
5. Returns the file size as a positive integer- indicating success to the caller

- int getfile(char* filepath, int conn_fd)
  This function is responsible for sending data from a file on the server to a client as a response to a get request if no errors are encountered.

  1. Will call headfile(**filepath**, **conn_fd**) to handle error checking and sending a header to the client and return if error code is received
  2. All file data will be sent to the client as the response’s body
  3. Returns the file size as a positive integer- indicating success to the caller

- int putfile(char* filepath, int length, int conn_fd)
  This function is responsible for receiving data from the client to be saved into a file if no errors are encountered. If the file exists and can be accessed, then the file will be overwritten.
  1. This function will first check if there are errors with the file path provided before proceeding in reading and will return an error code if there is an error detected:
     a. 403 - Lacking permission to access the file
     b. 403 - The path refers to a directory
     c. 500 - Other errors
  2. Will check if the connection was closed by client
  3. The file object will be opened or created
  4. Data received from the client will be read and written to the file object until **length** bytes are processed or the connection is closed by the client.
     a. Return -5 - indicating a premature closed connection
  5. Return: 200 - indicating success to caller or 201 - indicating success with file creation to the caller.

TESTING: Whole-System

- Logs: in each test keep track of the log that it generates to insure good output recording
- individual responses/connections: (Assignment 1 requesments)
  - testing persistent connections
    - check if PUT file, GET file, and HEAD file requests sent in the same connection in the order do not produce any errors or non-responses.
  - any type of request without errors works normally with any size file
  - bad request on on every implemented command:
    - file path is not 15 characters or not completely alphanumeric
    - Host component does not exist
    - http version is not 1.1
  - unimplemented request
  - unimplemented request that is also a bad request
  - no valid context length in a put request: doesn’t exist, value is negative, contains non-digit characters

    - get, head, or put request on a file without the proper permission
    - get or head on a file that does not exist
- concurrency:
    - run multiple get requests on different terminals to see if on is receiving data
    - commands in different orders. put command first and head/get second or vice versa.
    - run requests that are fast to process during slow to process requests to see if the fast ones finish first:
        - Slow: put or get request on an extremely large file that will take more than a minute to complete
        - Fast: put or get request on a small file or head request on any file or an implemented or a bad request.

Question

- Comparing using assignment 2: single-thread using one single worker and multi-threading using multiple workers because a1's code has less additional overhead and is allowed to use more memory.
    - ./httpserver 8080 -N 1
    - ./httpserver 8080 -N 8

- **Repeat the same experiment after you implement multi-threading. Is there any difference in performance? What is the observed speedup?**
  There is better performance in the multi-threaded code. Using 100 MiB size files, the speedup observed is that the multi-threading shaves a bit of time off the single threaded time.

- **What is likely to be the bottleneck in your system? How do various parts (dispatcher,worker, logging) compare with regards to concurrency? Can you increase concurrency in any of these areas and, if so, how?**
  A bottleneck could be logging by using locks for logging. I could improve that by using pwrite to write a file using offset to insure it's atomic. The dispatcher or worker could cause a bottleneck if it's designed with a busy-wait implementation by wasting CPU resources. Using semaphores unneeded threads can be put to sleep until they are needed. CPU resources can be saved.

- **For this assignment you are logging only information about the request and response headers. If designing this server for production use,what other information could you log? Why?**
  The time request received and the time that the request was processed to account for the order that the requests were received.

- **We explain in the Hints section that for this Assignment,your implementation does not need to handle the case where one request is writing to the same file that another request is simultaneously accessing. What kinds of changes would you have to make to your server so that it could handle cases like this?**

I would need to account for the files that are being interacted with at the current time and block access to them until that resource is released.