Assignment 2 Design Document

Andi Zhao

Goals

The goal of this assignment is to implement a multithreaded server that handles HTTP GET,PUT,and HEAD requests. The server will accept one mandatory argument(the port number) and two optional arguments number of threads to create and logfile to log output. On a PUT request, the server should create a file with the proper contents on the hard drive. GET requests should return the file contents to the client and HEAD requests should return only the header to the client.

1. **Consistency in multithreading**
   As with any multithreaded program, there is no guarantee about which thread goes first or which thread finishes first. Therefore there must be mechanisms in place to make sure each request always gives the expected output
2. **Concurrent logging**
   The server needs to log all requests while it is up, and multiple threads should be able to write to the logger at the same time
3. **Handling HTTP GET PUT HEAD requests**
   The server should know how to handle GET,PUT, and HEAD requests for all types of files.

Design

There are 4 main components in this design.

1. **HTTP server setup**.
   This includes how the server is listening to connections and how many connections can the server queue up before stalling
2. **Data Consistency in multithreading**
   This will be achieved using locks to ensure mutual exclusion and condition variables to avoid busy waiting
3. **Concurrent logging**
   This will be achieved using pwrite and an incrementally increasing offset, checked per request before writing the request.

4. **Health check**
   This is a special case of the get command, the server should only respond with a count of errors and total

HTTP Server setup

The first part is the standard server setup using sock_stream in which a server socket will bind to a specified port and listen indefinitely for client connections with http requests.

Data Consistency in multithreading

The second part is the multithreaded server with a dispatch main thread and a number of worker threads that will run in parallel to handle incoming connections.
I create a set number of threads in an array before my main function begins to listen for connections. Creating a thread pool avoids excessive creation of threads.

I chose to use a FIFO queue to store incoming connections. I implemented the queue with a linked list to prevent the server from stalling and making the clients wait. Although a circular buffer takes less space, there are more things to keep track of. This is the first shared resource between threads, so I will lock the enqueue operation from the main thread and the dequeue operation from worker threads by putting a mutex lock just around the enque and deque operations. The threads themselves will be controlled with condition variables to avoid busy waiting.

Concurrent logging

The third part is the log file, which will be implemented with pwrite so that multiple threads can write to a file at the same time. This allows multiple writers to be on the file at the same time. The offset will be tracked with a global variable. At the start of the server, this offset is set to the end of the logfile if necessary. Then, when each thread is done with their request, it calculates exactly how much space it needs to write to the log. The thread will then atomically set the offset back by that many bytes before starting to write at the previous offset. Due to the unpredictability of when threads finish requests, I decided to log files in the order that requests finish.

Health Check

For the health check part of the program, I keep track of the number of incoming connections in a variable in my main function. This is the total number of connections

my server has handled. This does not need to be locked because the main function is the only one changing this variable. For the number of errors, I have a global count. Each time one of my threads encounter an error, it will atomically add to the count. When a health check is issued, it will look at the current errors/requests and send a get request. This way, I would not have to parse the log file every time a health check is issued.

Use cases

Use case 1: Client sends a GET request

If the client sends a get request, the client should receive the contents of the requested file in terminal without any changes. If logging is enabled, the log file should contain a PUT entry with the contents of the created file in hex format.

Input: int clientSockd, fileName
Output: contents of fileName
Logic:

1. Open the file
2. If the file exists and has read permission return 200 OK
3. Write contents of the file to clientsockd

Use case 2: Client sends a PUT request:

A new file should be created with the contents of a preexisting file. If logging is enabled, the log file should contain a PUT entry with the contents of the created file in hex format.

Input: int clientSockd, fileName
Output: create a file named fileName with the same contents on disk
Logic:

1. Open fileName, if it does not exist, create a new file named fileName with write permissions
2. Open the file with O_TRUNC flag
3. Write contents to fileName
4. Return 201 created message and content length on completion

Use Case 3: Client sends a HEAD request:

The client should receive the content-length of the requested file, if logging is enabled, the log file should have a entry with the header of the head request

Input: int clientSockd, fileName
Output: sends the content length of file to clientSockd
Logic:

1. Open fileName
2. If the file exists and has read permission return 200 OK
3. Use fstat to get the content-length
4. Write 200 OK message followed by content-length to client

Functions

Main function to start HTTP server

The main function sets up an http server listening on a user specified port. It will create a user-specified number of threads, if not specified, it will create 4 threads. When a connection is received, the main function will lock a queue mutex, add the client onto a queue, then unlock the mutex.

Input : port number optional -N and -l flag
Output: None (-1) if program fails at any point.
Logic:

1. Parse the command line for port number, numthreads, and logfile
2. Create and initialize an array of pthreads as worker threads
3. Set up server socket with socket()
4. Set socket options with setsockopt()
5. Bind to port bind()
6. Listen for connections listen()
7. When connection is made, atomically put the connection in a queue
   Wake up a thread
   Go back to listening

Utility Functions

Parsing the HTTP Header

This function takes a file and will parse the file with a combination of scanf and strtok_r. On the first line, the Header will set the fileName, and the request type. If the request type is a put request, then the function will continue to look for the content-length tag

Function: parseHeader
Input: char* fileName, char* requestType, size_t length, int clientSockd
Output: assigns proper values to fileName, requestType and length
Logic:

1. Loops through the header using str_tok_r for the string"/r/n/r/n"
2. For the first instance, use sscanf to assign values to fileName, and requestType
3. If the request type is PUT, loop through the rest of the header to look for content-length and assign length

Validate HTTP Header

Reads data from the socket and checks for "/r/n/r/n", indicating end of header, if there is no /r/n/r/n, it will continue to read from the socket.

Function: validateHeader
Input: char* fileName, char* requestType, size_t length, int clientSockd
Output: assigns proper values to fileName, requestType and length
Logic:

1. Read from client socket and check for "/r/n/r/n"
2. If not found continue reading from the socket
3. If the size of the header is greater than 4096 bytes return an error

Thread Execution Function

This is the function that threads run when they are created using pthread_create. It runs in an infinite loop so the thread never exits and can be reused for other requests

Function: runThread
Input: logfile fd
Output: does not return
Logic:

1. All threads will run this function indefinitely

2. Function will wait until woken up
3. Function will try to take work off the queue
4. If successful handle the connection

Handle Client Connection

Threads will call this function to handle the request that the threads got off the queue

Function: handleConnection
Input: client_socket fd, logfile fd
Output: void
Logic:

1. Calls validate header
2. Call parseHeader
3. Validate the fileName
4. this works the same as a single threaded server
5. Call appropriate command
6. If the log option is enabled, will call **writeToLog**

Write To log

This function will change the global offset and write the request separator to the log file

Function: writeTolog
Input: int logfilefd,char* fileName,char* requestType,ssize_t fileSize
Output: void
Logic:

1. Lock the offset
    Store the offset in a local variable as the place to start writing
    Call **calculateSpace** and add the result to the current offset
2. Unlock offset
3. Loop through the file in increments of 20.
4. I chose to do this because the size of each line of the log file is 20
5. Convert the buffer into Hex representation by calling **writeFormatter.**
6. Pwrite the last 9 characters of the log entry

Format Buffer For Logging

This function takes in a buffer and converts it to formatted hex representation before writing the converted buffer to the logfile

Function: logFormatter
Input: char* writeBuffer,int byteCount,int fd,size_t currentOffset, size_t bytesRead
Output: writes hex representation to logfile
Logic:

1. Creates a buffer of sufficient size
2. Loops through writeBuffer and write the hex representation of each character in the array.
3. Create a separate buffer for the zero padded byte count
4. Combine the two buffers using sprintf
5. Pwrite the combined buffer to logfile fd starting at currentOffset
6. Add bytes written to current offset

Calculate Space For Log Entry

This function calculates the exact space required to log the entire log file including the separator.

Function: calculateSpace
Input: char *fileName, char *requestType, ssize_t fileSize
Output: space required for the request in logfile
Logic:

1. This function calculates the exact space required to to write a given command to logfile.
2. If the command is a GET or PUT command, the body of the file in question will follow in a format similar to hexdump.
3. The space for the file is calculated with a formula and the total space allocated is returned.

Deadlock Prevention

WIth this design, my server should not experience any deadlock scenarios. Deadlock in the most basic form happens when a process requests multiple requests and holds on to locks that it already has. In my case, the threads never need more than one lock at a time. Therefore there should not be any chance of deadlock.

Error Handling

This server should be relatively robust and able to throw appropriate errors when necessary. The error codes that are implemented are:

- 404 not found(if file is not found)
- 403 Forbidden (if fileName is not supported)
- 400 Bad Request (malformed syntax)
- 500 Server Error

Error Code: 404

If the fileName does not exist, a 404 error will be thrown for read.

Error Code: 400

1. The header is greater than 4096 bytes, the header length is out of range and thus will throw a 400 error.
2. The fileName is not valid by length > 27 or contains invalid characters

Error Code: 403

1. GET or HEAD: User requests a file that the user does not have read permissions for.
2. PUT: The user requests a file that the user does not have write permissions for.

Error Code: 500

1. My server does have capabilities to throw a 500 error, but as bad headers including valid POST or FETCH requests are considered a 400 error