Sehej Sohal sssohal 1650056

Assignment 2 - Multithreaded HTTP-Server Design Document

1 Goal

Building off of assignment1, the goal of this assignment continues to be to create and host, from the ground up, a working HTTP server that can accept and serve content via GET and PUT requests. However, now the server must be running on multiple threads, specified by the user. The server should be thread safe, and support contiguous logging from all threads. It should be able to accept and serve any size of file, given enough time, and should not modify the data in any way.

2 Assumptions

- The open/read/write/send/recv system calls have some sort of max amount of bits they can read, regardless of buffer size, and that size differs between OS's. Meaning that to have the code work on multiple OS's, I need to keep the buffer size small.
- Usage of common HTTP paradigms, such as pausing and expecting a 100 Continue before uploading a large file.
- Maximum buffer sizes are 16kb, but we're welcome to use smaller buffers.
- Ubuntu has some issues printing the \n character. I have run into this where sometimes my \n characters are printing as new lines, and sometimes they just manifest as a space. I'm not sure whether this is just me or my virtual machine.
- As long as the server is thread safe, and does not heavily slow down performance, any sort of synchronization method is acceptable.

3 Design

The design of the core HTTP server is exactly the same as from assignment1. The documentation for that assignment is here: https://gitlab.soe.ucsc.edu/gitlab/cse130/fall19-01-group/sssohal/blob/master/assignment1/DESIGN.pdf

Building off of that assignment, I have added some features and data structures. In terms of data structures, the httpserver class has been extended, and now has a field **workQueue** that is the std::queue. A vector, or any other queue implementation here would have been sufficient, however the std::queue was immediately available.

I have also updated the workflow of the listenAndServe function, to handle the dispatching of the threads. Before, this function would open the server's socket, and accept and handle incoming connections as they appeared. However, with multithreading this is. no longer possible.

In the start of this function, before any connections are accepted, an array of worker threads is created and initialized. The amount of worker threads is specified by the user. Then, the server begins to listen for incoming connections. Once a connection is accepted, we need to add it into our workQueue. However, since the dispatcher **and** the worker threads will all be attempting to change this queue concurrently, there needs to be some form of synchronization.

This is done via usage of mutual exclusion regions through the pthread library. The start routine for the worker threads has them periodically check workQueue for new work. To do this, they attempt to access workQueue by locking a global that has previously been initialized. If they are not able to access the critical region, they will sleep and wait on a global signal, that has previously been initialized.

Before the dispatcher attempts to put new work into the queue, it locks the queue mutex. After it is done pushing its accepted connection into the queue, it unlocks the queue mutex. It then sends a signal through the pthread library to tell the worker threads that there is work to do.

Once the worker threads receive this signal, they will wake up and attempt to access workQueue, so that they may pop the first value and serve the connection.

This is an implementation of the **Bounded Buffer** solution that was covered in class. The workQueue is our buffer, our listen module is the producer, and our serve module is the consumer.

In terms of logging, two functions have been added to the httpserver class. A function called **writeHeaderToLog** and one called **writeFileToLog**. These function provide a way to contiguously log the data being processed by the server without worrying about interweaving. Synchronization is needed for all the threads to contiguously write to the specified log file, however locking the file while it is writing is a bottleneck. To solve this problem, the httpserver class has had a field called **offset** added to it, which essentially acts as a pointer to the position in the log file.

To write both the headers and the body to the log file, the program will calculate the length of what needs to be written, lock a previously initialized mutex, add the length of the header to the offset variable, and then use pwrite with the **old** offset to write the header/body.

The reasoning behind using simple mutex’s instead of semaphores or more complicated synchronization methods is that: it simply was not needed. The simplest way to achieve what was desired is by studying the bounded buffer problem, and attempting to implement it in the simplest way possible.

4 Pseudocode Bolded code is different from assignment1

```
http_server.h/cpp

void workerFunction(void ptr) {
    httpserver server
    copy voidptr into server
    while(true) {
        LOCK
        if server->work_queue is empty
            thread.sleep(cond, lock)

        pop client fd from server->work_queue
        UNLOCK
        server->serve(client fd)

    }
}

void asciiToHex {
    helper function to convert ascii char[] to hex char[] seperated by spaces.
}

class httpserver{
    public:

        httpserver(port(optional), address(optional)) {
            get socket file descriptor
            bind socket(port, address)
        }

        listen(port) {
            open log file
            create worker threadPool
            initialize worker threadPool
            listen()
            while(infinite) {
                accept(client fd)
                LOCK
                server->work_queue->push(client fd)
                UNLOCK
                SIGNAL
            }
            close(port)
            close(log)
        }
```

```
private:
    address = default 127.0.0.1
    port = default 8080

    echo(function from assign0 to echo from 1 fd to another fd)

    serve(file descriptor) {
        buffer = 4kib
        recv(client fd, buffer, 4kib)
        parse recv with util functions(getMethod, getFilename,
                                       getContentLength, getExpectation,
                                       getBody)
        if(httpMethod == get) {
            get(request info)
        } else if (httpMethod == put) {
            put(request info)
        } else {
            send 403 bad request
        }
    }

    get(http request) {
        check filename
        fd = open(filename)
        *check if fd < 0 for 500 internal error*
            or 400 bad request
            or 403 forbidden
            or 404 not found
        echo(fd, client)
        close(client)
        200 OK
    }

    put(http request) {
        check filename
        fd = open(filename)
        *check if fd < 0 for 500 internal error*
            or 400 bad request
            or 403 forbidden
        echo(client, fd)
        close(client)
        201 CREATED
    }
```

```
        writeHeaderToLog(statusCode, method, contentLength, filename) {
            if logging not enabled
                return
            int old_offset
            char header[]
            int length
            if statusCode < 400
                if method == put
                    snprintf(header, ...(PUT))
                    calculate space to reserve for body in log using contentLength
                    length = above
                else
                    snprintf(header, ...(GET))
                    length = sizeOf(header)
            else
                snprintf(header, ...(FAIL))
                length = sizeOf(header)
            old_offset = server->current_offset
            log_LOCK
            server->current_offset += length; //reserves space for writing
            log_UNLOCK
            if successful put
                pwrite(log_fd, header, sizeOf(header), old_offset)
                writeFileToLog(filename, contentLength, old_offset+headerLength)
            else
                prwite(log_fd, header, sizeOf(header), old_offset)
        }

        writeFileToLog(filename, contentLength, old_offset) {
            open filename
            char asciiline[], hexLine[]
            int bytecount
            do {
                read asciiline
                asciiToHex(asciiline, hexLine)
                bytecount += 20
                snprintf(padded bytecount, hexline)
                pwrite(buffer, old_offset);
                old_offset += sizeOf(buffer)
            } while read > 0 and contentLength > 0
        }

        util functions:

        getMethod - gets method
        getFilename - gets and validates filename
        getContentLength - gets and validates length
        getContentType - gets and validates type
        getExpectation - gets expectations if any
        getBody - gets content body

}
```

main.cpp

```
int main(argc, argv) {
    default args:
        port = 8080
        address = localhost
        threadcount = 4
        logfile = ""
        loggingEnabled = false
    check args (-p, -a, -N, -l):
        if found, replace default value

    httpserver.listen(default args)
}
```