Maiah Pardo
CruzID: mapardo

Design Doc: Asgn2: httpserver.c

An Introduction of the Assignment:

This assignment is a continuation of asgn1: the HTTP Server with multithreading, a log, and health checker.

- Multithreading: allows us to perform operations (multiple curl requests) simultaneously.
- Log: Keeps track of each performed request, their status (success or failure), and data transmitted dumped as hex (for put and get operations only).
- Health check: returns the number of operations that resulted in a failure status (which can be found in the log).

Program is called by: **./httpserver port# -N numthreads -l logfilename**

- **-N numthreads**: if omitted, program will be run with 5 threads
- **-l logfilename**: if omitted, program will not run with logging
- the optional parameters can appear in any order

Some discussion questions:

- **Why is your system thread safe?**
  - When a worker thread is accessing a critical area, other worker threads will not be able to access the same area at the same time. (This is done by using mutexes) This is also discussed in the design of my functions.
- **Which variables are shared between threads?**
  - The log file offset is the only shared variable between threads that is not of constant value. I have some other shared variables that are not of as much importance since they are of a constant value (log file name, a Boolean: 0 if not logging, 1 if logging, and the number of threads in the program).
- **When are they modified?**
  - This question is a large discussion, which has been discussed in detail in the design of my functions below, but I will touch on it briefly here.
  - The number of total requests and request errors are modified during a GET healthcheck case. The log file offset is modified in when updating the offset (when we are about to the write to the log file) and when we have no worker threads working, and check that the size of the file matches the offset to ensure we do not have a problem if there are changes to the log file. isLogging is a check to see if we are infact, performing logging. These variable must be shared because all threads must have access to them.
- **Where are the critical regions?**

        - The critical regions are where multiple threads can access a shared variable. The most important shared variable is the log file offset. If it's value were to be incorrect during an access, we may end up with a corrupt log file.
    - **Explain why your design for logging each request contiguously actually works**
        - We can use pwrite with an offset of our choosing so that we can write to different parts of a file without interruption simultaneously. Using an offset allows us to choose the next available place of writing so that (as long as requests aren't sent in parallel) we are most likely to stay in order. When accessing the log file offset and updating it, we use a mutex to lock this critical region, so that no two threads can read the offset and update it at the same time. If two threads were to read the offset at the same time, they would be writing to the same place in the log file, which would cause write errors.

Data Structures:

- **Struct which holds the attributes for each worker thread:**

```
struct workerThreadAttributes
        pthread_t thread_id;
        int log_fd;
        size_t logfile_offset;
        int status;
        char requestType[];
        char filename;
        long contentlen;
        int portnum;
        char first_thou_file_bytes[];
        char putBuffer[];
        char host[];
        char req_line_from_client[];
        int num_requests;
        int num_request_errors;
} workerThread;
```

- Variables and their meaning in the struct:
    - **pthread_t thread_id**
        - a 64-bit integer value that is given by the creation of a pthread_t structure, for each individual thread created
        - value is given by: pthread_t val = pthread_self()
    - **int log_fd**
        - log file file descriptor
    - **size_t logfile_offset**
        - unsigned type that will store the offset of the log file
    - **size_t logfile_offset;**

        - used to determine how many bytes we will write to the log file (not including the hex content)
    - **int status;**
        - holds the status code of the current request being processed by the thread
    - **char requestType[];**
        - PUT, GET, or HEAD
    - **char filename;**
        - filename used for PUT, GET, or HEAD operation
    - **long contentlen;**
        - content length of the current request being processed by the thread
    - **int portnum;**
        - port number of the server
    - **char first_thou_file_bytes[];**
        - holds the first thousand file bytes as hex of the data being PUT or GET
    - **char putBuffer[];**
        - used when we have the special case: file bytes were sent along in the first receive (recv()) in which we are given the headers. This will only occur in the case of put.
    - **char host[];**
        - holds the host name (ie. localhost:8080)
    - **char req_line_from_client[];**
        - holds the request line given to us by the client (for FAIL cases of printing to the log file.
    - **int num_requests;**
        - holds the number of requests found in the log file
    - **int num_request_errors;**
        - holds the number of request errors found in the log file

Functions:

- **main()**: Contains our "dispatcher"
    - Checks to make sure we have the correct number of arguments
        - Must have 2, 4, or 6 arguments only
        - Any other number of arguments is invalid
    - Loops through all arguments and checks for port number, and the optional flags: -N (flag for number of threads), and -l (flag for log file)
        - Check that port number is valid (> 1024)
            - Convert port number to a 16 bit unsigned integer using strtouint16() function
            - Create a listening socket given the port number (call create_listen_socket())
        - Save number of threads if -N flag present
            - Check that number of threads is valid (> 0)

        - Save log file name if -l flag present

    - If we have logging to do:
        - Check the log file name given is not null
            - If null, exit with error
        - Check if log file already exists
            - If it does exist:
                - If we do not have write or read permission, throw an appropriate error message
                - Set a bool flag indicating file was not created (already existed)
        - Open the log file with flags: O_RDWR (read/write permission), O_CREAT (optionally create it if it does not exist), O_APPEND (append to file if it already exists)
            - The log file needs to be able to be modified during execution, so open with flag 0664
        - If the log file already existed:
            - Check that every line in the log file follows the specified format:
                - Call helper function: log_format_check() with the log file descriptor
                - If file does not follow format, exit with error message
    - Dispatching:
        - Create the appropriate number of worker threads
            - Create an array of worker threads which is of the appropriate size (either the default size 5, or if a -N flag given, the number of threads specified by the user)
        - Initialize worker thread attribute variables for each thread:
            - Set all worker thread attribute variables (discussed above) to its initial value.
        - Enter infinite loop which we use to wait for new connections:
            - Once we have received a client
            - Lock the queue mutex (pthread_mutex_lock(&mutex))
                - This is done so that only one thread may mess with the queue (enqueue or dequeue) at one time
                - Note: A queue is not a safe data structure, if two threads enqueue, dequeue at the same time, we will have issues, so we must protect the queue with a mutex lock
            - Signal to one of the threads currently waiting that there is now work to do (pthread_cond_signal(&cond))
            - Now we can put the new connfd on the queue (call enqueue)
            - Now we can unlock the queue (pthread_mutex_unlock(&mutex))
            - Note: our worker thread is now active and will enter doWorkFcn()

        - Check if no threads are working: if not, check the size of the log file offset compared to the size of the log file. If they differ, update offset to be the size of the file. This is how we ensure that if a log file is changed while the server is running, we will be able to continue to write to the same place.

- **doWorkFcn():** When a thread becomes active, it immediately enters this function
    - Enter infinite loop:
        - Lock the queue mutex (pthread_mutex_lock()
            - Again, this is done so that only one thread may mess with the queue at one time
        - Dequeue off the queue, protected by a mutex, now we have a connection file descriptor
            - The queue is not a safe data structure. If we call dequeue() at the same time or enqueue at the same time, we may end up with errors
        - If there are no items in the queue:
            - call pthread_cond_wait() with the dispatch condition variable (This will make the threads wait until there is work to do, meaning another connection file descriptor in the queue)
        - Now we can handle the connection (by calling handle_connection()) and begin performing the operation our client asked for. If a healthcheck was asked for, then we will find this out when parsing the headers from our request in handle_connection().
        - When returning from handling the connection, we check if we are meant to perform a healthcheck. We only perform a healthcheck if in a GET request type case, where we check that the format of the log file is valid (log_format_check()) and send the client an OK message if so.
            - If we do not have a valid format, we have a different request type or if we are not logging, we send the appropriate status.
        - If there is logging to do:
            - We update the offset of the log file by calling update_offset()
            - We write to the log file by calling logWriter()
            - Both of these functions are discussed later
        - Lastly, we finish by resetting all of the worker thread's attributes so that we do not carry old data with us when our worker thread works again for a new client.

- **update_offset():**
    - Updates the worker thread's attribute called "logfile_offset" depending on if our operation has failed or succeeded.
    - We also create a "request line" used when we write to the log file, which contains information about our operation. (ie. FAIL\t…)

- **doHealthCheck():**
  - This function opens the log file for reading
  - Here we use a flock() exclusive lock on the file so that we may read it without being interrupted by a write to the log file. Reading while writing may cause errors so we use flock() to ensure that this does not happen.
  - If a line contains "FAIL", we add 1 to our count of the number of requests processed that resulted in error. For each line, we add 1 to the number of requests processed.
  - Lastly, we close our file, and save our number of errors and number of requests processed into our appropriate thread attribute variable.

- **logWriter():**
  - The logWriter function opens the log file, calls flock() on the log file descriptor so with a shared lock so all threads to write to the log life concurrently. Once the worker thread has reached the log file, it should carry a status code. If the status code is above 201 or 0, we consider it an error.
  - In an error status code case:
    - We print a "FAIL" request line, with the error code and no content.
  - In a GET, where filename is "healthcheck" case:
    - we grab the number of errors and number of requests that appear in our log file (this value was updated in doHealthCheck() (called in doWorkFcn()), before logWriter was called in the doWorkFcn()), convert this value to hex, and place into our hex buffer titled "first_thou_file_bytes" temporarily.
  - Given offset of our request line (stored in a thread attribute variable called logfile_offset, which was updated inside of update_offset()) and the offset of our content (the first thousand file bytes that has now been converted to hex) plus one new line. We can now add this value to our current log offset of the log file. This will then be the size of the log file after our write.
  - Next we print the request line to the log file, followed by the content, followed by a single new line.

- **log_format_check()**
  - First we open the log file, put an exclusive lock using flock() on the file, and grab the log file's size. If the size is 0, there is no need to check if it is valid. If the size is not 0, we parse through the contents of the log file and check for 3, 4, 5 tab spaced elements. I do this by reading in 4096 bytes of the log file at a time. I check each character in the log file for a newline, if it is, I know it is the end of a line. I ignore the first next character (considered the first element) and then check for subsequent tabs. If a tab exists, I continue looping until the next element is not a tab. The first next element which is not a tab is considered a valid element. I count the number of elements which exist on each line.

    - If we have finished reading the file without returning 0, the log file follows correct format, return 1

- **handle_connection()**
    - Handle connection is mainly a parser function which parses through the request from the client. First I do the first receive from the connection file descriptor. I check that there are no extra file bytes that came in with the first receive, if there are I handle these accordingly:
        - I check if there is content after the \r\n\r\n
        - If there is, I store the file bytes into my worker thread attribute's buffer called putBuffer. I can carry this around with me to my handler functions when processing puts. A PUT case will be the only time we must worry about this, as it is the only time we receive file bytes.
    - I store the request line from the client in case we must use this when printing to our log file in an error case
    - Next I use a string tokenizer to grab the content length, request type, filename, host name, and headers. I check that the headers follow a valid format with a ":" in between.
    - I check that the version HTTP/1.1 is correct
    - I check the file name for validity (> 19 characters, ., _, alphanumerics only)
    - If any errors have been found, we return from this function back to doWorkFcn() to immediately write the error to the log file
    - If no errors have been found, we continue to process our HEAD, PUT, or GET request

- **strtouint16():** (given to us)
    - Converts a string to a 16 bits unsigned integer
    - returns 0 if the string is malformed or out of range

- **create_listen_socket():** (given to us)
    - creates a socket for listening for connections
    - Closes the program and prints an error message on error

- **handlePut():**
    - This function is given as arguments: the connection file descriptor, filename, content length, an int called isMoreContents which will tell us whether or not there is was file data received in the first recv(), and the worker thread.
    - If we have received extra file contents from the first receive, we handle these by converting the first 1,000 bytes to hex and storing in our worker thread's attribute buffer called first_thou_file_bytes
    - Check if file exists using access**,** if the file does not have the right permissions, update the worker thread's attribute with the error status, send an error message to the client and return back to handle_connection().

    - Open the file, then lock the file descriptor using a call to flock() with an exclusive lock. We must use flock() to lock the file, because if we were to write to a file while another thread may try to read (GET case) or even grab the size (HEAD case) of the same file we are writing to, then we may end up completing an invalid operation, or end up with errors.
    - If there were more contents from the first recv(), we first write this into the file.
    - Next we call receive and write the bytes to the file, while making sure we have saved 1,000 btyes into our first_thou_file_bytes (converted into hex).
    - Once we have finished, we flush the file, close the file descriptor (which unlocks the file locked by flock()), send the appropriate message OK or CREATED back to the client, close the connection fd and return.

- **handleGet():**
    - This function is given as arguments: the connection file descriptor, filename and the worker thread.
    - Given connection file descriptor and filename
    - Check if file exists using access, If error upon opening: send the appropriate message back to the client and update our worker attributes status code with the error number. Then return back to handle_connection().
    - Open the file, calling flock() as a shared lock on the file descriptor.
        - We use a shared lock because we are not changing the file during reads. If another thread is doing a HEAD or GET on the same file, this will not disrupt this thread's read since we are only reading, not writing.
        - We must keep other threads from accessing the file descriptor concurrently because if a PUT was taking place at the same time, we could potentially receive the wrong size of the file.
    - grab the file size
    - send an OK message to the client, with the Content-Length value as the size of the file, store this status into our worker thread's status variable
    - While reading the file, grab the first 1,000 btyes of the file and convert to hex, storing the result in our worker thread attribute's first_thou_file_bytes buffer.
    - Close the connection
    - Return back to handle_connection()

- **handleHead():**
    - This function is given as arguments: the connection file descriptor, filename and the worker thread.
    - Check if file exists using access, If error upon opening: send the appropriate message back to the client and update our worker attributes status code with the error number. Then return back to handle_connection().
    - Open the file, calling flock() on the file descriptor.
        - We use a shared lock because we are not changing the file during reads. If another thread is doing a HEAD or GET on the same file, this will not disrupt this thread's read since we are only reading, not writing.

        - We must keep other threads from accessing the file descriptor concurrently because if we were to read the file while another thread is writing to it, we will have the wrong contents and potential errors.
    - We retrieve the size of the file, and send this back to the client