Design Document: Multi-Threaded Server (Assignment 3)

Program Description

Given a functional http server (assignment 1), implement multi-threading and logging. The server will have 4 threads running by default, but it will take command line arguments that can specify the number of threads. If the log flag is specified in the command line arguments, logging is enabled and will keep track of all requests and errors. If client requests *healthcheck*, the number of errors and requests are sent to the client.

Objectives

1. Parse additional command line arguments
2. Implement multithreading and ensure synchronization between threads
3. Implement logging. Files must be logged in Hex and must be formatted
4. Implement health check which is a GET method that sends the number of errors and log entries to the client.

Implementation

Parsing additional command line arguments

The multi-threaded server accepts additional arguments for numbers of threads and for the log file. The original server only accepted a mandatory argument for the port number.

1. Command line arguments will be validated in a separate function called validArgs() which returns a Boolean
2. Using getopt(), optional arguments can be checked. If matching argument is found, it is checked if its format and value are valid. If valid, it will store the value. If not valid, function will return false and program will terminate.
3. If no optional arguments are found, the default value is stored
4. After optional arguments are parsed, the remaining Non-option arguments are parsed. The port number is checked while parsing and if any additional arguments are found besides port number, function returns false.
5. Port number is checked if it is between 0 and 65535

Multithreading and synchronization between threads

Multi-threading is used to improve throughput. Each thread will handle a request concurrently. The user specifies the number of worker threads available and each thread must wait until it has work to do. The threads must not busy wait and if they need to share resources they must use synchronization mechanisms. Each thread may allocate up to 16Kib of buffer space.

Required Data structures and mechanisms for multithreading and synchronization

Queue: to ensure that requests are processed in order and no requests are processed twice or missed

Struct httpObject needs an additional variable to store the client file descriptor

Mutex locks to ensure synchronization between threads

Conditional Locks to prevent busy waiting

P_thread functions to create threads

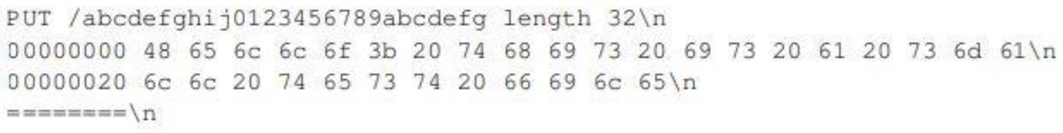
A thread function for each thread to process requests

1. Threads are created using p_thread_create() depending on the value specified by user. Threads then wait using a conditional lock and mutex lock until there is a request in thread_func().
2. A dispatcher thread creates a httpObject and adds it to the queue when there is a request. Once enqueued, it will send a signal to wake a thread to start processing.
3. The worker threads will start processing the request either when it receives a wake signal.
4. if the queue is not empty, it will not sleep and process queue items. If queue is empty, threads will sleep and wait for dispatcher thread to send wake signal.

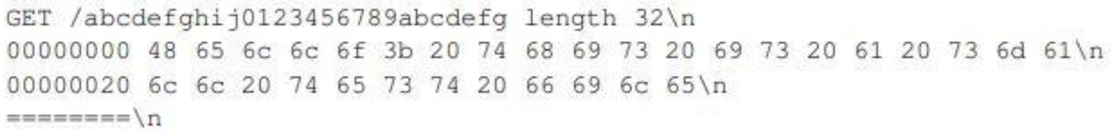
<u>Implement logging</u>

If -l log_file is specified as an option, logging is enabled, and all requests and errors must be logged to log_file. The log_file must be truncated every time the server starts and if -l option isn't provided; no logging is done.

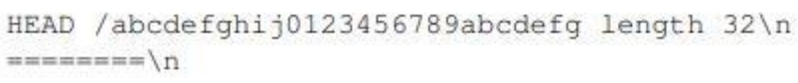
Format



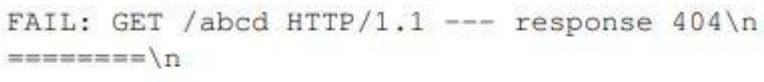
```
PUT /abcdefghij0123456789abcdefg length 32\n
00000000 48 65 6c 6c 6f 3b 20 74 68 69 73 20 69 73 20 61 20 73 6d 61\n
00000020 6c 6c 20 74 65 73 74 20 66 69 6c 65\n
========\n
```



```
GET /abcdefghij0123456789abcdefg length 32\n
00000000 48 65 6c 6c 6f 3b 20 74 68 69 73 20 69 73 20 61 20 73 6d 61\n
00000020 6c 6c 20 74 65 73 74 20 66 69 6c 65\n
========\n
```



```
HEAD /abcdefghij0123456789abcdefg length 32\n
========\n
```



```
FAIL: GET /abcd HTTP/1.1 --- response 404\n
========\n
```

Required functions and mechanisms to implement logging

A string2Hex() function is required to parse through the contents and convert it to hex in the above specified format

logContentLength() is required to calculate the amount of logging space each request will take, so that threads can simultaneously write to the file

Offset variables to keep track of where to each thread can write to. A global offset is used so that each thread knows where to write from. A local offset is used to keep track of where the current thread is writing. An error offset is used in the event there is an error and so that the thread can write over previously written log information

Mutex locks on the global offset variable to enable synchronization between threads

1. If logging is enabled, offsets are used to calculate write positions.
2. If the request is properly formatted, the logContentLength() function is called in the processReq() function. This calculates the total amount of space required to log the request. logContentLength() updates the amount of space required for the total log file and the amount of space the header needs. A mutex lock is then used, so that the global offset can be incremented by the total log file space required. The local offset and error offset's value is set to the value of global offset before it was incremented. After which, the header is written into the log file and the string2Hex() function is called and the request contents are logged in hex to the file. After this, "========\n" is written to indicate the termination of logging of that request.
3. If the request was not properly formatted, the logging is done after the request response has been sent to the client. The request header's length is calculated and added to the global offset (Mutex locks are used once again to allow for synchronization).
4. If an error is discovered while processing a properly formatted request, the local offset is replaced with the error offset so that the previously written log information can be overwritten.

Healthcheck

If the client wants to monitor the performance of the program, they can call healthcheck to see the number of log entries and the number of errors.

Required functions and mechanisms to implement healthcheck

A Boolean function healthCheck() is needed to log the number of log entries and number of errors. In this function, two static integers are used to keep track of log entries and errors. When these integers are incremented, a mutex lock is used to ensure that values aren't altered whilst another thread is trying to read or write those values. The function also serves to provide the formatted response message for healthcheck.

1. healthCheck() is called in process_Req() to increment log_entries
2. If the request were to get healthcheck, it functions formats the response message with the number of errors and log entries.

3. If the request were to get healthcheck, and logging was not enabled, an error message is formatted instead. If the request was a PUT/HEAD healthcheck, an error message is again formatted and sent.
4. If an error is discover, Healthcheck() is called and the int storing the number of errors is incremented.
5. A mutex lock is used at the beginning of the function and unlocked at the end to prevent any changes while another thread is inside the critical region.