Atticus Deutsch
CSE130 Asgn2
Document Design
W20

httpserver - mulithreaded

WHAT IS IT

...

Objective:

This project is an extension to *Asgn1 httpserver*. *Asgn1 httpserver's* documentation can be found in the *Asgn1 httpserver design document*

The goal of this program was to multi-thread the simple http server built in Asgn1. Multi-thread means that the program is split up into multiple threads of execution or lists of instructions. This is beneficial because two threads can be running at the same time if the program is compiled and ran on a machine which supports multiple threads of execution. If the running machine doesn't support multiple threads of execution, then the threads that the program was split up into will be scheduled to run one at a time in a manner decided by the OS. Additionally this program aspired to have a log which logs each request that comes in to the server. This is very useful for debugging the server and understanding what's happening while the server is running. The log is non-trivial because it is shared data and requires synchronization to avoid two threads racing on it.

Purpose:

The purpose of multithreading the simple http server from Asgn1: *Speed.*

In particular, *throughput*.

This program is a specification of the single producer multi-consumer problem in computer science. This problem is known as embarrassingly parallel because the consumers do not have to worry about what each other are doing, and each one doesn't have to worry about what the producer is doing besides when they exchange data with the producer once at the rendezvous point. In Particular, "workers not having to worry about what each other are doing" means they are not sharing any data. Throughout is thus increased because workers can each be given a thread of instruction which does not share data with any other worker thread of instructions. The fact that they don't share data means that they can run at the same time without causing data races, while linearly scaling throughput.

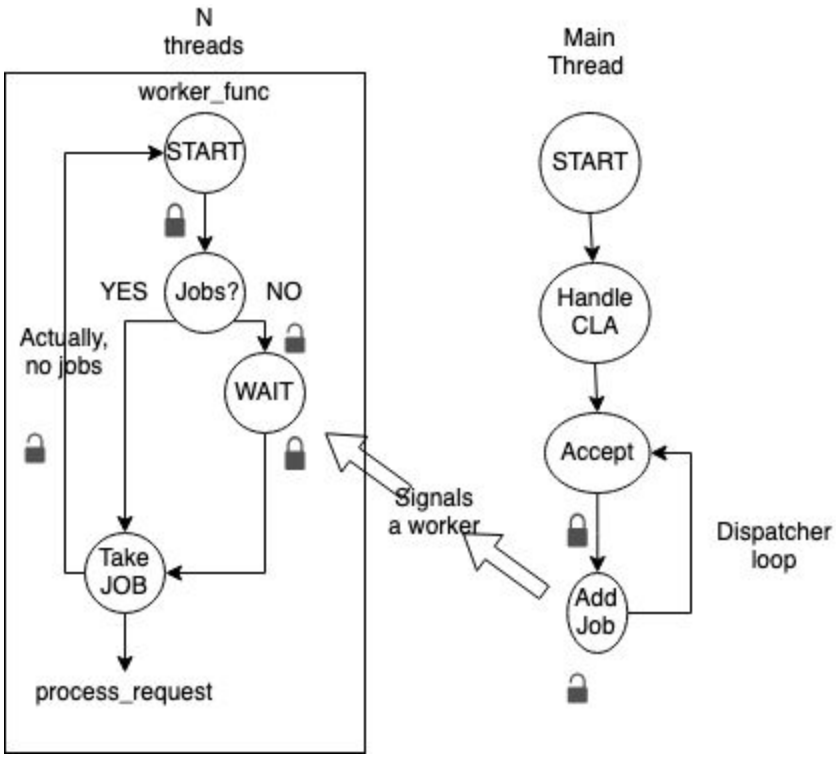
Constraints:

This program will be written in C. This program will not be using any high level http libraries. This program will not support concurrent requests to the same resource.

HOW IT WORKS

...

High level Multithreading:



When the program starts, like any C program it begins in main() on the main thread. Next I handle the command line arguments.

Handle CLA:

Handling the command line arguments is entirely abstracted away into a function called `uint8_t handle_cla(int argc, char * argv[], char ** log_file, uint16_t * num_threads, char  ** port)`

handle_cla() uses getopt to look for the -N and -l options based on the total number of CLAs (argc-1) and the number of options it finds. There should be one CLA left which is the port number, else an error. The data of the options and the port number is allocated to the heap, then pointed two by the parameters of handle_cla() for access outside of handle_cla()

handle_cla() returns a 0 or a 1. 1 meaning there was an error. After returning, if main sees that handle_cla() returned a 1 it will exit the program with status code 1. If handle_cla() returns a 0, the heap values of the options and the port number is localized then freed.

Prior to the main thread entering the dispatcher loop the http socket is set up and begins to listen() in a similar fashion to asgn1 (TAs code). Additionally, the N worker threads are created using pthread_create() and are sent to in the the start of the worker_func()

Dispatcher Loop:

The Dispatcher Loop simple calls accept(), gets a file descriptor used to communicate with a client. Then calls add_job() with that file descriptor passed in. Then repeats forever.

add_job(): add job locks creates a linked list node on the heap with the file descriptor as its data. Then locks the linked list mutex, adds the node to the tail, unlocks the mutex, and signals using a condition variable.

Worker_func:

The entire function is wrapped in an infinite while loop. Upon entering the while loop, the workers lock the **linked list mutex** then check the number of jobs in the linked list. If there are some jobs in there, a worker will proceed to take one using the function take_job().

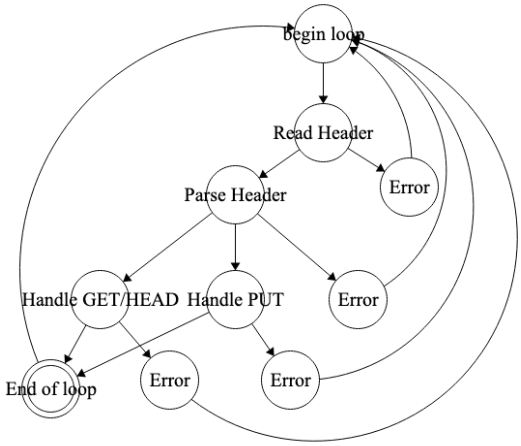
If there aren't any jobs in the **jobs linked list** when a worker checks, they proceed to call pthread_cond_wait which automatically unlocks the **linked list mutex,** and waits for a signaling from the dispatcher. Upon receiving

the signaling, the thread automatically locks the **linked list mutex,** and calls take_job().

take_job(): take _job() will remove and free a node from the head of the **jobs linked list** and return the file descriptor of the request (job) to handle.tak_job() will then unlock the **linked list mutex.** If the **jobs linked list** was empty for some reason, take_job() will return 0.

If take_job() returns 0, the worker tries to get another job by continuing to the next iteration of the infinite while loop. Else, the job is processed by the worker using process_request().

Additional Changes from Asgn1:



proccess_request(), as mentioned above, is essentially the same as the “Main Loop” as shown above from Asgn1.

A major structural difference was Handle GET/HEAD was logically split into two functions Handle GET and handle HEAD.

Logging:

The greatest change to the way that requests are handled is the fact that every request must be logged.

Logging must be done in the form described in the spec (see asgn2 specifications).

The log function (either log_no_header() or log_func()) is called everywhere right after the res() function is called (res() responds to the client()

Logging an absent header:

If the read_header() function from Asgn1 fails. We know that we failed to read a full header. The first 42 characters of whatever was sent is used as replacement for the first line of the header, when logging the error. This is done by calling log_no_header();

Logging a malformed header:

If parse_header() returns a Request struct whose error field is not 200 or 201, then the header is malformed. After the response is sent log_func is called passing in the Request struct.

Logging a PUT request:

A key point to notice is that when handling a PUT, one shouldn't respond with 201 until all the data has successfully been stored on the server. This means that one cannot begin logging until all the data has successfully been stored on the server because we do not know if we have a 500 or a 201 yet so we can't log. After responding 201, we cannot read the data from the client socket again, and we can't allocate all of the data on the heap while PUTing it on the server due to a violation of spec restrictions (the amount of data being sent in the PUT could be too large). Thus, this implementation has decided to store the data in temporary files. **A temporary file has a name which is some integer.** An incrementing integer is used so that every request has a different temporary file. This integer is atomic so that workers don't race on it. The temporary files are written to as the data is being read from the socket.

After every call to res(), log_func() is called with the arguments being the temp file name and the updated Request struct, It is possible, upon an

error, that the temp file won't have been written to when its name is passed into log_func, but that is okay.

Logging a GET request

Similar to that of a PUT request, a temporary file is opened with an integer as its name. If the resource cannot be opened or we cannot determine the size of the requested resource then we call log_func() after we res() to those two cases.

Else, In the case of a GET request whose request resource can be opened and the file size of the resource can be determined, we simply write the data to the tempfile as we send it to the client. After all the data has been sent to the client we call log_func() with the arguments being the temp file name and the updated Request struct.

Logging a HEAD request

For a HEAD request we do not open a temp file. However, Similar to the other GET we call log_func() with the arguments being the temp file name and the update Request struct after we res().

log_func():

If a log file was not specified in the command line arguments, log_func() simply deletes the temp file that was created while handling the request, then **returns**.

If the Request struct passed into log_func() had a code field that was not 200 or 201 then we format the FAIL to be logged using snprintf. Snprintf will return the amount of bytes in the formatted string. Next, we __atomic_fetch_add() to the atomic log file offset integer. This integer is used to tell threads where their reserved space in the log file is. Lastly, We pwrite() the FAIL log to the log file.

If the Request struct passed into log_func() had a code field that was 200 or 201

> If the type was a HEAD we know we have no tempfile to get the data to log from we we simply use snprintf to format the response

200, Next, we __atomic_fetch_add() to the atomic log file offset integer.

Else, we need to open the temp file and write its contents to the log. This is done by formatting the first line of the log using snprintf, then passing that and the tempfile name int space_to_reserve() which will calculate the total number of bytes this log entry will take. Next we. Next, we __atomic_fetch_add() that value to the atomic log file offset integer, and localize a variable called my offset which is our offset to write to the log at for this request. The logging goes as follows.

```
pwrite() the first line which we already
formatted

Increment the local offset.

Num_lines = get_num_lines(tempfilename)

for(num_of_lines)

     pwrite(the 8 digit decimal data counter)

     Increment the local offset

     Read 20 bytes from the temp file

     for(each of those bytes)

          pwrite(it in hex)

          Increment the local offset

     pwrite(a new line)

     Increment the local offset

pwrite(the 9 byte ending ========\n)
```

Finally we delete the temp file and **return**.

Health check:

Health check is a new feature of the server. Upon a proper GET request to the resource healthcheck, the server will respond with the number of invalid requests so far followed by a new line followed by the number of total requests so far. These two values are global atomic integers that are incremented when logging in the logging functions log_func() and log_no_header(). This response to the client is coded into the handle GET function. If a valid head or a put request is made to the resource healthcheck, the server responds with 403 and calls log_func() passing in the name of the temporary file and the Request struct (error = 403) as arguments.