I. Introduction

Implement a multi-threaded HTTP server with logging and a health check functionalities. Here are some discussions:

Discussion: A shared mutex is ready. With the shared mutex locked, common resources will not be written by different threads at the same time. File descriptor for log_file should be shared, since every worker thread needs it to perform logging. One critical region is where we update the offset. Offset for logging is shared, while this variable is locked by mutex when each worker tries to update it simultaneously. Mutex is locked once the worker gets the client socket from the message queue, the worker will then check the resource of the message to calculate the offset and update it. After the offset is updated, the mutex gets unlocked. Once the mutex is unlocked, offset for this request is already well set, in which case logging each requests can contiguously work from the correct offset. Another critical region is where the worker tries to write a file, since we won't know how many workers are trying to do the same thing. I believe my system is thread safe because by locking critical regions, those common resources won't be modified at the same time within a lock.

Request	Functionalities
GET	This will send back a **200 OK** response followed by content of the file if succeed. Can accept a health check request to send back requests status if log flag is enabled.
PUT	This will send back a **201 CREATED** response and create a new file on server side, with content of the file on the client side, if succeed.
HEAD	This will send back response of corresponding information on existing files.

II. Functional Design

Data Structure:

char arrays will be used as buffers to temporarily store HTTP requests and file content.

Struct *Queue* will be used to store incoming HTTP requests.

Struct *httpObject* will be used to store the information of the request.

Struct *ThreadArg* will be used to store the information of each thread.

Function Design:

int bad_request(httpObject):*

- check if the validation of the http version or method or filename of the request. -1 is returned if it is a bad request. 0 indicates a success.

int read_http_request(client_sockd, httpObject):*

- receive HTTP request and parse it. Method, filename, http version, and content length is stored in *httpObject* struct.

int resource_check(httpObject):*

- check the status/permission of resource (filename). For GET and HEAD, file size will be returned if no error code is found, otherwise -1 is returned. For PUT, content length of the to-be-put file is returned if no error code is found, otherwise -1 is returned.

int ppwrite():

- loop over the message buffer and use *pwrite* to log file content. New offset is returned.

void process_request(client_sockd, httpObject):*

- process PUT requests. If this request is errored when doing *resource_check()*, it will trash out the content in client socket, otherwise it will write the content to a file on the server side.

void construct_http_response(client_sockd, httpObject):*

- This will take client socket, HTTP version, status code, and content length, and concatenate all of them together in a format as a response being sent to client.

void send_response():

- process GET requests. It will write the file content to the client side. If the request is a healthcheck, the corresponding response will be sent.

void offset_calc(httpObject, error_flag):*

- This will calculate the offset of the request. Error flag will help determine the correct offset that will used for *pwrite*.

void consumer(*obj)*:

- This will be the worker thread of the program. Once it receives its condition variable from dispatcher thread, it will start to process the requests.
- This will get the client socket from the queue and process the request with the functions described above. Once it finishes the process, it will send a signal to the dispatcher.

void producer(*obj):*

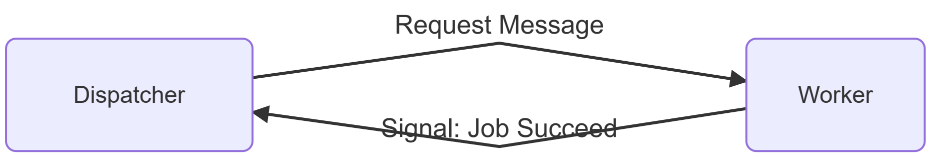
- This will be the dispatcher thread. Once no workers are available, it will wait until someone sends the condition variable back.
- If the message queue is not empty, it will dispatch message(s) to worker that is available by sending signal.

*int main (int argc, char** argv):*

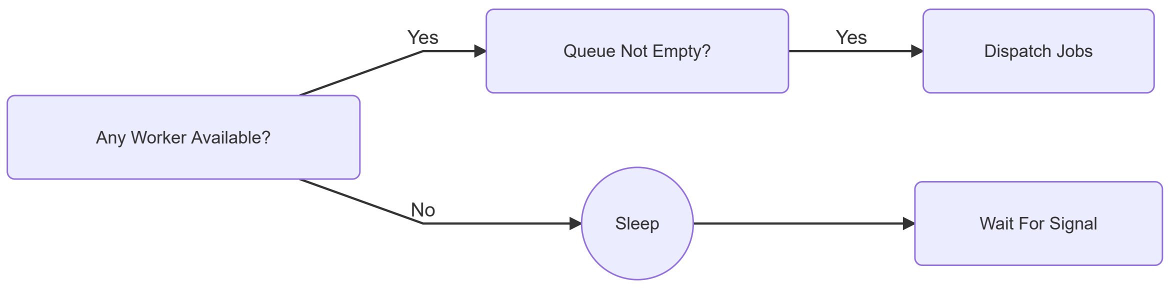
- Main will take command line arguments via *getopt()*. -N indicates number of threads and -l indicates log_file enabled.
- This will setup sockets to connect server and client.
- Worker threads and producer thread will be created.
- HTTP requests will be continuously received in a *while(true)* loop, and requests will be pushed to the queue.

III. Architectural Design

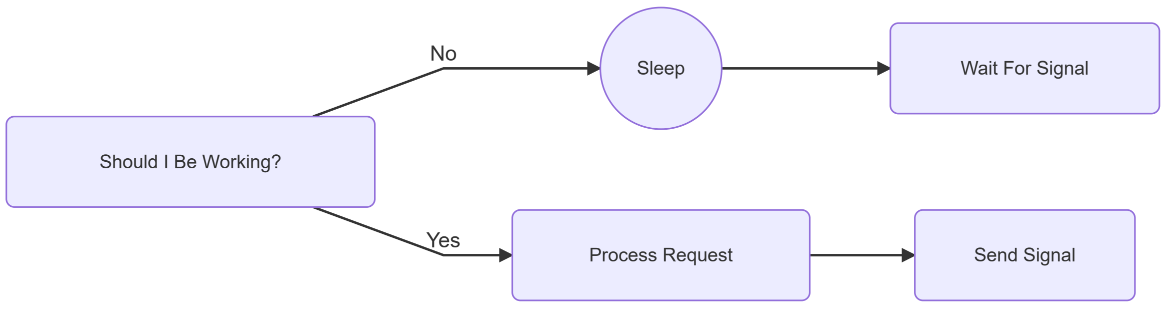
Overall, requests are dispatched by the dispatcher (producer thread) to any available worker (worker thread). Once multiple threads are created in main(), they start running.



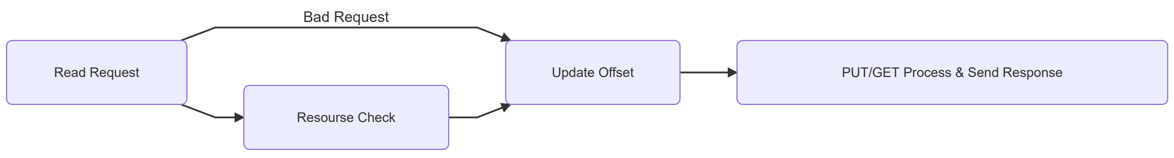
Below is the implementation of *Producer Thread*:



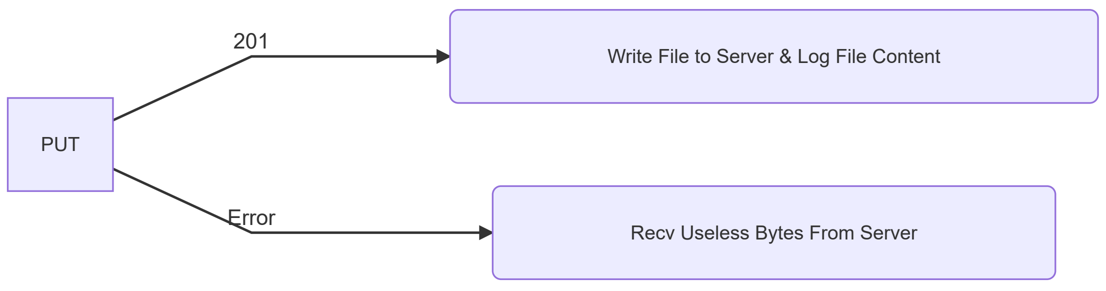
Below is the implementation of *Worker Thread*:

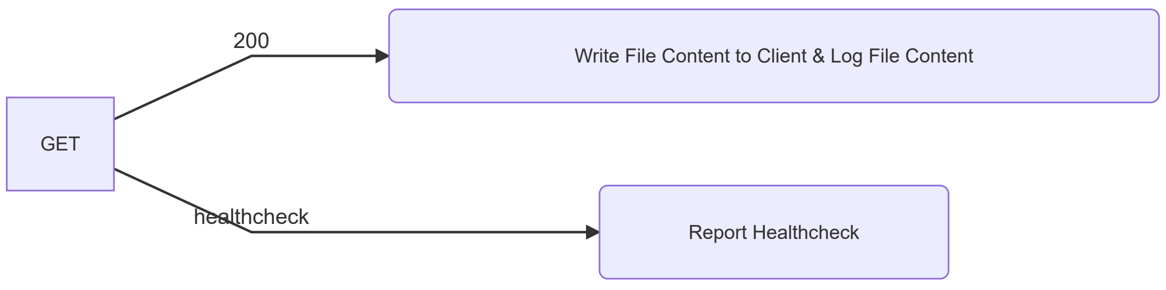


Below is the implementation of *Process Request*:



Below is the implementation of *PUT/GET Process:*


PUT
201
Write File to Server & Log File Content
Error
Recv Useless Bytes From Server



IV. Testing

curl will be used as testing commands. Concurrent requests will be sent by using "&" between every two *curl* requests.

GET:

```
curl http://localhost:8080/filename
```

```
curl http://localhost:8080/healthcheck
```

PUT:

```
curl -T file.txt http://localhost:8080/filename
```

HEAD:

```
curl -I http://localhost:8080/filename
```