Design Document: httpserver.cpp

Cristian C. Castillo

1.0 Introduction

Multithreading is all about allowing a process to generate threads that exploit the processor with the help of the operating system to achieve phenomenal throughput in parallel, greatly mitigating the response time of a client's request to either retrieve or send content information via the web. Prior to such techniques, there was that of the HTTP Server implementation, where only one process could be presented to the server(main thread). By allowing the OS to manage these threads, share resources with a process, these threads can achieve massive improvements when it comes to throughput and possibly shorter latency. You can think of it as a superworker with additional limbs, simultaneously working together on an event; like a superpower! Nevertheless, to pull off such great techniques synchronization highly matters, along with the right design to make this technique feasible!

1.1 Goals and objectives

The goal is to make a Multi-threaded HTTP/1.1 server with the option of redundancy. The number of threads and the option of redundancy is all decided by the host (default of 4 threads if host did not specify). This server needs to respond to GET(read) and PUT(write) requests to "files" with the file's name length being 10 and consist of only ASCII characters. The server will store its files in the directory of the current server location, where they shall persist (live within the HTTP Server directory) and be present upon invocation. However, if redundancy is asserted, there would be 3 copies of the file in which each of the three reside in the directory's "copy1", "copy2", and "copy3"(the 3 directories reside within the HTTP Server directory). A design document is expected, along with a concise README.md file in the Git directory. The code must build httpserver.c/cpp using the make command produced by a Makefile. The server must be tolerant of varying client requests, respond adequately to a errorones request, and appropriately generate the needed content upon a successful HTTP request.

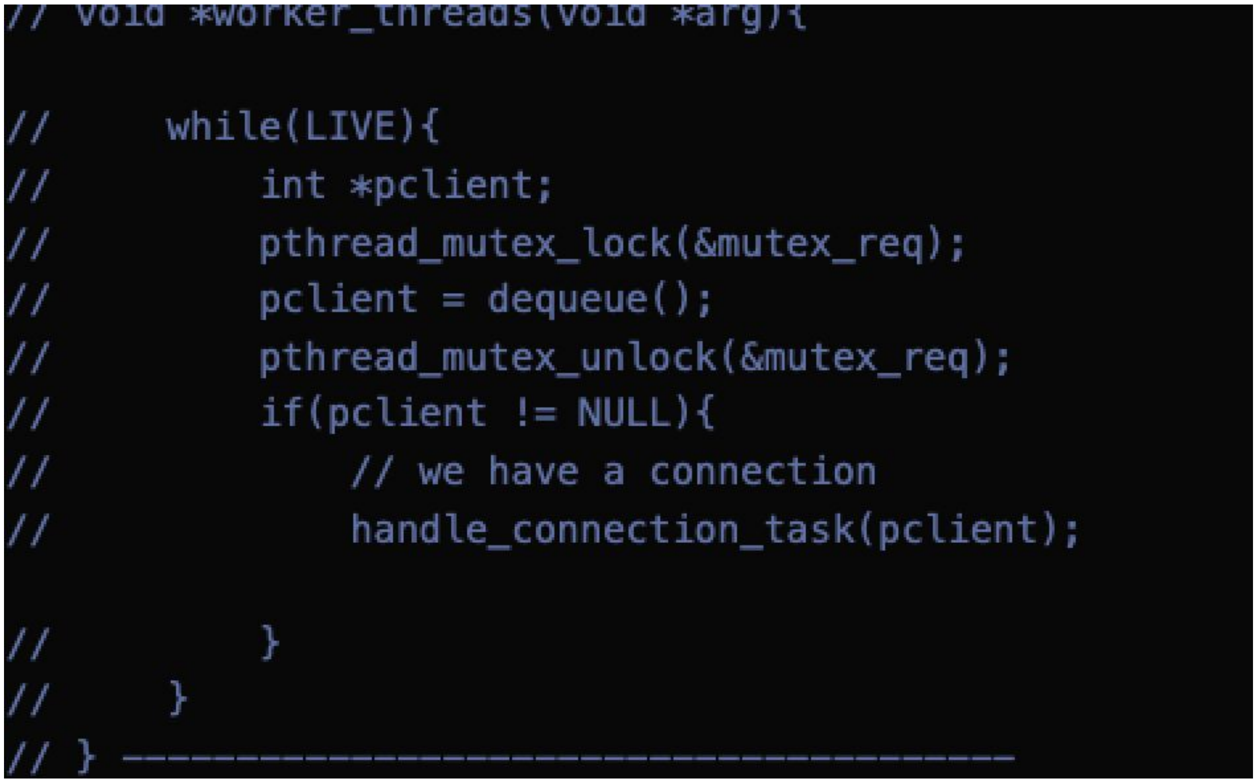
1.2 Assumptions

From the previous assignment, we shall further advance our knowledge and technical abilities by successfully integrating threads and redundancy, providing a functional client to http server bilateral relationship that will transmit messages via text files. It is most presumed that this will be a daunting challenge that involves extensive research, one which requires applying the theoretical teachings by integrating systems and designs of complex systems to mitigate complexity through modularization, abstraction, naming, hierarchy, and finally layering. By completing this project one will have acquired ample knowledge regarding multi thread programming/concepts, how a more realistic HTTP requests functions, and acquire the minimal ability to solve race condition problems along with scenarios of a corrupt file.

1.3 Assignment Question

1. Q: If we do not hold a global lock when creating a new file, what kind of synchronization problem can occur? Describe a scenario of the problem.

ANS: First and foremost the correctness of a concurrent program shouldn't depend on accidents revolving around timing. Through concurrent manipulation of shared mutable data, dangerous bugs can arise; hence, race conditions become difficult to debug as complexity rises, but to mitigate this cognitive burden one needs a way for concurrent modules to share memory via synchronization from locks, one that acquires and releases. Conceptually you could think of it as a bathroom key, as discussed in Daniel's discussion where one person obtains the keys (acquires), goes in the critical region (bathroom), leaves these critical regions and releases the lock for future threads to use. By using such methodology of locks, we inform the compiler and processor that we are using shared memory concurrently, so that registers, caches, etc… will be flushed from shared storage. This avoids the problem of what is known as reordering! Hence the owner at the time being is always looking for the most current up to date information. WIthout the locking mechanism we would encounter problems such as **Deadlocks!** However, to make this concrete lets first dive right into a conceptual example on why global locks are needed in the first place. Conceptually, we can take a bank as an example, where the bank itself is considered the shared memory, and you have cash machines A,B,C,.. and so on which all link up to the essential shared memory bank, aka the bank. The bank is the memory concurrency, whereas the cash machines are all operational with reads/writes happening. As you can see coordination is involved amongst the shared memory bank example to such cash registers (in my program the structs holding the data are like the cash registers of http content client request), but without any coordinations for the reads/writes, we could have money (relate to data) write into the wrong balances, read from the wrong balances concurrently for the reads/writes, hence leading to a horrible dire situation! Nevertheless, a problem such as this is resolved with locks, and with these locks we can protect each bank account (cash registers of structs) where each one needs to acquire their own locks for transaction (reads/writes) updates. Nevertheless, to achieve the general principle of the correctness of a concurrent program we need to consider confinement, immutability, existing thread safe data types (strtok_r for example), and lastly synchronization. The bank is the global lock in this case, and each cash register must first acquire the lock to access and update their accounts from the Bank! By conceptually implementing this we ensure cash registers A,B,C, and so on run independently on a different account. To conclude, when we use locks carefully we can prevent race conditions, but in doing so some locks require threads to wait, but another scenario that could present itself is that 2 threads are waiting for each other, and no thread makes progress. This would present itself as cash register A (who wants to do an operation at the shared memory bank) updating B, and B updating A simultaneously by both acquiring their locks from their respective bank accounts, and now A is awaiting for shared memory region2 and B is waiting for memory shared memory region 1, ending in an abrupt stalemate known as a “deadly embrace”. Needless to say, this is the significance of having a global lock to avoid what we call a **cycle of dependencies,** where A is waiting for B, which is waiting for C, which in return is waiting for A, leading toward a dead end of no progress, or previous mentioned scenarios. This deadlock scenario comes in many forms and can be presented even without using locks in message passing systems when buffers fill up. Locks also help us with the busy waiting situation… observe the following code:



```
// void *worker_threads(void *arg){

//     while(LIVE){
//         int *pclient;
//         pthread_mutex_lock(&mutex_req);
//         pclient = dequeue();
//         pthread_mutex_unlock(&mutex_req);
//         if(pclient != NULL){
//             // we have a connection
//             handle_connection_task(pclient);

//         }
//     }
// } ----------------------------------------
```

Busy waiting is a highly inefficient way to access the critical region to put a client connection into the queue. Locks work magnificently better, and with global locks we can help manage local lock access to shared memory locations!

2. Q: As you increase the number of threads, do you keep getting better performance/scalability indefinitely? Explain why or why not.

ANS: The answer is Yes, No, and is dependent on the expected traffic flow between client/server. If the number of clients is low and the frequency of the number of client requests is low, having many threads present is redundant. This causes the threads to stay idle waiting, which hugs up a lot of CPU space doing nothing. If your hardware system is not robust/made for multi -threading, this will potentially make the server run slower(diminishing returns). This is why a design document is necessary before implementing a server for your clients. However if your server is expecting many connections along with a high frequency number of requests from clients, having many threads is a must. Knowing prior data of your client audience and the amount of clients would help with building a hardware system that will support such traffic. This is another reason why having a design document is necessary. To sum up everything, if your hardware is built for an abundant amount of threads to run smoothly, an increase of threads will definitely increase performance indefinitely (if you don't hit the mac cap). However, this could cause a lot of complexity. Nevertheless, to summarize it all if the threads are conducting any I/O operations, synchronization of any sorts, then running 1 thread should suffice per core, and will generate best performance, though adding more threads does improve throughput, as mentioned the law of diminishing returns will become transparent at some point. Hence, there is a sweet spot for the increase of threads per core, and one would think that producing something like 3k-5k threads would yield upon scalability, but it could reproduce reverse actions that produce tons of context switches; resulting in an actual increase in wait time! In one case I generated 5k threads, and data returned relatively too slow, or not at all. Hence, as mentioned there is a sweet spot for the number of threads dispatched and the relationship between scalability becomes transparent as throughput increases to an extent that the laws allow us too!

2.0 Internal data structure

The *argv[] is declared within the int main() parameters and is used to help access the files and keep track of the number of inputs given by the user in this program. *argv[] is an array containing the names of all the inputs made through the terminal when executing the ./httpserver program.

The code must use fixed-size buffers, and may allocate no more than 4KiB of Memory for each HTTP headers both sent and received (either via malloc() or as a direct variable declaration). The data that follows it (for a PUT) may be much longer. Similarly, response headers will be less than 4 KiB, but the data that follows may be longer.

`struct sockaddr_storage` : This structure is used to define a socket address which is used for the following functions: bind(), connect(), getpeername(), getsockname(), etc… On line 63 of the HTTP Server program, this storage structure defined the address of the socket in reference to the original 1st socket that supports protocol-specifics. Upon a sockaddr_storage cast, this structure maps onto the sa_family, but when as a protocol-specific address...it shall map onto a structure that is of type sa_family_t and identifies the protocol’s address family. This enabled us to create processes that were bound to clientSocket.

`struct addrinfo` : This structure is used by getaddrinfo(), an essential component to setting up our connection configurations. The structure contains the following features:

- ai_flags
- ai_family
- ai_socktype
- ai_protocol
- ai_addrlen
- ai_addr

All were used to set up, excluding ai_flags. The ai_family set up the AF_INET, the socket was of type SOCK_STREAM which maps to ai_socktype (vs other methods, for e.g. DGRAM connectionless implementation). The ai_protocol which helped specify our protocol for the returned socket addresses. The ai_protocol was set to IPPROTO_TCP (0), which in case must be set as is to conform with SOCK_STREAM, and zero meaning TCP implementation. Lastly, a pointer to the socket address is placed in the ai_addr field, along with the length of the socket address to ensure accept() functionality is invoked accordingly.

`socklen_t` : An unsigned opaque integral type of length of at least 32 bits, where values of 2^32 -1.

`getaddrinfo` : The get address feature was a wonderful treat that was integrated to the HTTP Server program on line 44. The used parameters are self explanatory: getaddrinfo(host_address, portNumber,&hints,&res); The host_address is e.g localhost/127.0.0.1, the user also has the ability to establish a custom port number, in this case holding the second parameter. The hints and res we will go into depth soon, but prior to this function, gethostbyname() was used to do DNS lookups (associating various information to domain names) by loading the data into a struct sockaddr_in. However, in this program we used this function to mitigate lines of code and to keep up with industry standards. In regards to the hints, this points to a struct addrinfo which helped us assign us to the American Familly_INET (which is an address family that helped designate the type of addresses that the socket may communicate with, in this case our IP addresses). Also tying into our SOCK_STREAM features via hints, which is known as TCP (Transmission Control Protocol which establishes a connection between the client and server prior to the data being sent). Finally the res, is the carrier one might say, presenting itself as a pointer to a linked list, which when you run valgrind in its entirety will become apparent as it requires freeaddrinfo(res) to successfully free the linked list; res* in our program.

`struct stat` : This function was used in our program to return and access information about a file. Stat requires a const char path, which in this case is our file name, and a pointer to a factitious buffer that is filed by the stats which is pointed by stat. In this program buf.st_size was used to store our content length (known as child->content_length), which was a detrimental key feature that helped accurately send the correct expected amount to our HTTP Server program upon a Put Request. The bus.st_size in this program returned the total size in bytes of our referenced file, and stored the bytes in integers.

`fstat` : This function is identical to stat(), except that the file to be stat-ed is specified by the file descriptor fd, in this case corresponding to line 138 of our HTTP Server's source code, where stat pertained to fetch_file_fd (a file descriptor that fetched a file). The significance was on par to how we implemented stat for our Put Request functionality, but in this regards fstat was vital to delivering our content length upon a HTTP/1.1 200 OK Request!

`struct http_node` : A generic simple node struct that contains only the content length, and surprisingly one of the most significant pieces to the success of this program. This node stores content-length in the main program and can access via the struct with an arrow to access the member from that stores the data throughout the program's lifespan.

`unordered_map<std::string,int>hashing` : This data structure was an essential piece in mapping each unique thread id to a client's file. This enabled O(1) for best case scenario, which is explicitly visible in our program and on average will perform O(n), during a lookup. Nevertheless, when a client file is hashed to a unique id, we iterate from beginning to end via a for-loop lookup, and determine through a string parse if the client's request was either a PUT/GET. The following hash format is presented as so:

```
/*
  hash id [client file n ] = id & convert to c-str() due to sensitivity
  hashing [string_to_parse.c_str()] = node_parser->id

*/
```

2.1 Global data structure

Queue is used in our program to store clientSocket(s) to the dispatcher pool for threads to use. These clients are globally locked, signaled to be processed, and ensures that mutual exclusion is compliant by using mutex locks and mutex conditions via the thread pool and vice versa.

- **queue (stack)** is a Queue in the Standard Template Library (STL). it has a First In First Out(FIFO) type data structure. Items/objects are inserted at the back (top of queue) and are deleted from the front (bottom of the queue)

- **push()** function adds an item, client in this case, to the end of the queue (top of queue).

- **pop()** deletes the first item of the queue(bottom of queue), or removes a client from the queue.

- **empty()** Returns whether the queue is empty or not, are there clients who need work?

- **vectors (stack)** are the same as dynamic arrays with the ability to resize itself automatically when an item/object is inserted or deleted. In the face of new demand it's hard to reason incommensurate scaling, hence we used the vector for an increase in client demands.

- **push_back()** pushes the item into a vector from the back, or in this case a client makes a request and must wait in line like any other shopper at a store.

- structures to hold data regarding the client/server connect, thread’s individual data, and number of threads and redundancy

- **Master http_obj** only stores content length
  struct **http_node**

- Holds redundancy (passes message to worker thread)
  struct **marshall**

- Holds information of number of threads to dispatch and whether or not to assert redundancy into the httpserver
  struct **command_line_inputs**

- Holds data for each individual thread. Like a bank client going to the bank to get money that is associated with himself only. Therefore this struct also holds the passcode (mutex key) to his bank account

  struct **http_object**
  This struct is in the header's file and is considered an http object that contains the necessary client data that is unique to one. This struct, like an individual person, contains its unique buffer, client connection, id, protocol, and even its own pthread_mutex_t master_key lock that it acquires to access the critical region area.

2.2 Global variables

- *pthread_mutex_t:* A type used to initialize statically pthreads. Used to create in our program a dynamic dispatcher of n workers.

- *pthread_cond_t:* Used for appropriate functions for waiting and later of process continuation. This variable enables threads to suspend execution, voluntarily cease to keep the processor until a condition presents itself as true. To avoid race conditions that were created by one thread, we associated pthread_cond_t with a mutex. While that one thread is preparing to wait, an additional thread may signal the condition before the first thread actually waits on the resulting deadlock. Hence, the thread will wait for a signal that is never sent and any mutex can be used as there is no link amongst mutex and the condition variable.

- *Pthread_mutex_t lock_put* : The key for local put modules so that the thread can utilize it to access the critical region area.

- *Pthread_mutex_t lock_get* :The key for local get modules so that the thread can utilize it to access the critical region area.

- *Redundancy* : Is initialized once and for all. The absence of race conditions will not be present with this variable since no data manipulation is involved. It is simply used as a switch to assert redundancy to the overall program if the host decides to have the httpserver to include redundancy or not.

2.3 Local variables

- *pthread_t:* The data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier. In our program this was used to help create the dispatcher of threads (array).

- *pthread_t_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr):* A function that initialises the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Uon successful initialization, the state of the mutex becomes initialized and unlocked. Any attempts to initialize an already initialized mutex results in undefined behavior.

- *pthread_mutex_lock(pthread_mutex_t *mutex):* The mutex object referenced by mutex is locking by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

- *pthread_mutex_unlock(pthread_mutex_t *mutex):* This function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when this function is invoked, this results in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.

- *pthread_cond_signal(pthread_cond_t *cond):* Unblocks at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).

- *pthread_mutex_unlock(pthread_mutex_t *mutex):* This function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when this function is invoked, this results in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.

- *pthread_cond_signal(pthread_cond_t *cond):* Unblocks at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).

- *pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex):* Used to block on a conditional variable. They are called with mutex locked by the calling thread or undefined behavior will result. These functions atomically release mutex and cause the calling thread to block on the condition variable cond.

2.4 Functions

`char *strtok(char *str, const char *delim)`

A macro used to break apart a string dependent upon a pattern(delimiter) that is in the string *str itself. This function returns a pointer to the first value up to where the delimiter is found in the string, if nothing in front of the first delimiter however, the first values after the delimiter is returned. A null pointer is returned if there are no values left to retrieve.

`char *strtok_r(char *str, const char *delim, char **saveptr)`

Serves the same purpose as *strtok but *strtok_r is Multi-Thread safe. The third argument saveptr is a pointer to a char * variable data type in which iis used to maintain context between successive calls that parse the same string.

`void *memset(void *str, int c, size_t n)`

Used to fill a block of memory with a particular value. Where the starting pointer to the address of memory is to be filled with the provided value, given the size. Memset was declared in our dog definitions to re-initiate the state of our buffer; setting all bits to zero by using '\0' for the second argument. The third argument is simply the size for this buffer in question; 4Kib as stated in requirements.

`int strcmp(const char *str1, const char *str2)`
Takes two string arguments, compares them, if equal, returns integer value 0 indicating equivalence. In our driver program dog.cpp, this feature was implemented as the first argument corresponds to the file arguments, and the second argument a string dash argument.

`int sscanf(const char *str, const char *format, ...)`
This function reads/parses data from a string rather than standard input or input from the keyboard. The first argument is a string from where we want to read/parse the data from. The second argument is the format/style in which we want to parse from the first argument. So if we want to parse for example 3 strings from the first argument(%s %s %s), the last arguments 4,5,6 represented by "…" are getting those parsed strings in that order respectively. The output is the number of items read from the string, and returns -1 if the parsing is unsuccessful.

`char *strcpy(char *dest, const char *src)`
This functions Copies the string from input ***src** to puts it into input ***dest**. The result of this function is the ***dest** yielding the same string as ***src**.

`char *strcat(char *dest, const char *src)`
This function appends the input string src to the end of the input string dest. The output of this function is the string dest concatenated with src.

`void freeaddrinfo(struct addrinfo *ai)`
This function frees the linked list ***res** which is dynamically allocated memory. The function should also free **addrinfo** structures returned by **getaddrinfo()**, along with any additional storage associated with those structures.

`getopt(int argc, char *const argv[], const char *optstring)`
This function takes the number of command line arguments, the command line arguments itself, and the flags to look out for (*optstring) created by the host. If an element of *argv* starts with '-', then the letter following that is checked with *opstring, in which if there is a match a procedure will occur. It will return -1, when no more options to process. If the letter following the dash '-' is not in *optstring returns '?' to show that this is an unrecognized option.

void f_void_setsocket_error(int *int_setsocket*)
The input is a struct pertaining all the properties necessary to set up the server socket. If the server socket address is not correctly linked, this function prints/outputs the exact reasoning to what caused the event.

bool f_bool_permissions(char **file*)
Checks if the input file grants authorization for the server to open the file. Return True if permission granted, false otherwise.

bool f_bool_check_request(char *request)
Checks if the the passed in input *request string contains the string “GET” or “PUT”. returns true if *request matched, returns false otherwise.

bool check_http_protocol(char **protocol*)
Checks if the passed in input *protocol string contains the string “HTTP/1.1”. Returns true if the string matched, returns false otherwise.

bool f_bool_check_file(char **parseFile*)
This function checks that all resource names passed from the client to the server must be 10 ASCII characters long, must consist only of the upper and lowercase letters of the (English) alphabet (52 characters), and the digits 0–9 (10 characters). If a request includes an invalid name, the function will respond accordingly.

bool f_bool_check_file_len(char **fileLength*)
Checks if the file name being PUT to the server by a client is of 10-ASCII character length. If a request includes an invalid name length, the function will respond accordingly.

void f_void_server_arg_error()
This function will be triggered when the user(you) sets up the server the wrong way and will respond to the user accordingly. After you compile the server you will have a binary called httpserver, which you will execute with ./httpserver (but is optional and will default to 80 when not specified).

void f_void_bind_error(int *bind*)
This function is called upon when the bind connection on the server side is unsuccessful, it will return an integer which indicates an error with -1. The specific code in the global variable errno will respond to the user the problem accordingly indicating the problem and close the bind socket.

`void f_void_listen_error(int socket_listen)`
This function is called upon when the listen connection on the server side is unsuccessful, it will return an integer which indicates an error with -1. The specific code in the global variable errno will respond to the user the problem accordingly indicating the problem and close the listen socket.

`int fstat(int fd, struct stat *buf)`
This function helps us obtain the content length of a file that's passed through a file descriptor. However the buffer is obtained by creating a structure (struct stat buf) which will contain the content length itself

`void f_client_req_found(int file,int clientsocket)`
This function is for when the user sends a GET(read) request to the server. The server checks if the file exists, if so, the server terminates the child process and responds to the client with "HTTP/1.1 200 OK\r\n\r\n" along with the contents of the file(Note: "\r\n\r\n" is new line). On the new line the client will see "Content-Length: X \r\n\r\n" where X is a number written in the form of the data type "char"(ex: "278").

`void f_client_req_created(int int_client_sockd)`
This function is called upon when the client sends a PUT request to the server. The server will respond by printing to the client terminal via the dprintf function "HTTP/1.1 201 Created\r\nContent-Length: 0\r\n\r\n".

`void f_void_400(int clientsock)`
This function is called upon when the client makes a 400 bad request to the server. The server will respond to the client by printing to the client terminal via the dprintf function "HTTP/1.1 400 BAD REQUEST\r\nContent-Length: 0\r\n\r\n".

`void f_void_client_error_forbid(int int_client_sockd)`
The input is the socket connection between server and client. This function is called upon when the client triggers a 403 Forbidden on the server side. The server will respond to the client by printing to the client terminal via the dprintf function "HTTP/1.1 403 FORBIDDEN\r\nContent-Length: 0\r\n\r\n".

`void f_void_client_error_not_found(int int_client_sockd)`
The input is the socket connection between server and client. This function is triggered when the client is requesting(GET) a file from the server that doesn't exist, hence, making the server print on the client “HTTP/1.1 404 FILE NOT FOUND\r\nContent-Length: 0\r\n\r\n”.

`void f_void_intr_error(int int_client_sockd)`
The input is the socket connection between server and client. This function is triggered when the client tries to communicate with the server and the server is unable to fulfill the request due to an error on the server, hence, making the server print to the client "HTTP/1.1 500 Internal Service Error\r\nContent-Length: 0\r\n\r\n"

`void f_void_kill_request(int clientsock,struct addrinfo *res)`
The input is the socket connection between server and client and the linked list pointer containing all the information regarding the connection. This function gets triggered when the request that is sent to the server is not a PUT or a GET request, which results in the function making the server client socket connection to close, free all the dynamically allocated memory(socket address structures returned in the form of a list pointed to by res), and exit the program.

`void f_void_path_error(int int_client_sockd)`
The input is the socket connection between server and client. This function is triggered when the system call read(2) fails to read information(client request) from the client and prints a 500 internal error message "HTTP/1.1 500 Internal Service Error\r\nContent-Length: 0\r\n\r\n".

`void f_void_getaddrinfo_error(int status,struct addrinfo *res)`
The get address feature was a wonderful treat that was integrated to the HTTP Server program on line 44. The used parameters are self explanatory: getaddrinfo(host_address, portNumber,&hints,&res); The host_address is e.g localhost/127.0.0.1, the user also has the ability to establish a custom port number, in this case holding the second parameter. The hints and res we will go into depth soon, but prior to this function, gethostbyname() was used to do DNS lookups (associating various information to domain names) by loading the data into a struct sockaddr_in. However, in this program we used this function to mitigate lines of code and to keep up with industry standards. In regards to the hints, this points to a struct addrinfo which helped us assign us to the American Familly_INET (which is an address family that helped designate the type of addresses that the socket may communicate with, in this case our IP addresses). Also tying into our SOCK_STREAM features via hints, which is

known as TCP (Transmission Control Protocol which establishes a connection between the client and server prior to the data being sent). Finally the res, is the carrier one might say, presenting itself as a pointer to a linked list, which when you run valgrind in its entirety will become apparent as it requires freeaddrinfo(res) to successfully free the linked list; res* in our program.

`void f_void_error_on_accept()`
A generic message informing the user that connection is reestablishing via the server.

`Int f_fetch_file(char *char_file)`
The input is the file name that the client requested to get back using GET. This function opens a file for reading only using the open(2) system call and returns the file descriptor associated with the file that was just opened.

`void f_void_socket_error(int int_socket)`
The input for this function is the information regarding the failed state(less than 0 return value)) of the system call . Inside the function errno will find out what caused this problem, when found, outputs to the terminal the reasoning for the failed connection.

`bool f_bool_check_vitality(char *file_name,char *protocol,char *request,int client_connect)`
The function input is the file's name, the HTTP protocol and request, along with the client connection with the server. This function checks for file permissions, length of the file's name, the character set in which the name is composed of, the request made by the client, and the HTTP protocol. If either of these fail, a HTTP error message will be sent to the client.

`bool f_regex_overkill_parse(char * req, char *file, char *protocol)`
The inputs are the request, file name, and the protocol sent to the server by the client. This module's job is to check for correctness/validity of the of the request by straining the standard for the request to be either a "GET" or a "PUT", the file name to consist of an optional forward slash, with only ASCII characters lowercase a to z, uppercase A to Z, or consisting of numbers 0 to 9, all combing to a strict length of 10 ascii characters(forward slash not accounted for). The last thing is to check if the protocol is exactly "HTTP/1.1". If all passes, a "true" will be returned indicating the correctness of the client's request, otherwise returning a "false".

void f_make_copy(char *file_name, int content_length, int clientSocket)

This function is for redundancy for the PUT module. The input is the file’s name, the content length regarding that file, and the clientsocket. The resulting output are three potential copies of the files uncorrupted, and one copy is in each of the three copy directories (copy1, copy2, copy3).

void *f_check_copy(char *file_name, int clientSocket)

This function is for redundancy for the GET module. The input is the file’s name, and the client socket connection. The resulting output is sending to the client a file that is not. If no two files differ in content, size and with correct permissions, that indicates a file that is not corrupted ,so we send that to the client. If the condition is not met, a HTTP error message will be sent to the client. The HTTP error message protocol will be numbered in respect to the problem at hand to notify the client the problem.

void *f_void_put_module(char *buffer, char *file_name, int clientSocket)

The inputs of this function are the buffer in which data has been read in from the client socket, the name of the file, and the client socket itself. This function is called upon with a “PUT” request made by the client. The result/output of this function is that It writes the file’s contents and its desired name into the current server directory in which it will be stored. If write is not possible, an appropriate HTTP procool will be sent to the client.

void *f_void_get_module(char *buffer, char *file_name, int clientSocket)

The inputs of this function are the buffer in which data has been read in from the client socket, the name of the file, and the client socket itself. This function is called upon with a “GET” request made by the client. The result/output of this function is sending the file that is previously saved on the server to the client. If a file has inappropriate permissions or doesn't exist, an appropriate HTTP protocol will be sent to the client indicating the problem at hand.

void create_worker(pthread_t *workerArr, int *thread_array, int numWorker)

The create_worker thread models the worker/boss model in terms of pthread implementation, with the signature for loop known as the dispatcher function. Hence threads are created, awaken, and sent off when needed. In addition this function initiates attributes (2nd parameter) that are safe in terms of thread stack size, which was passed which each created thread via pthread_create(&workerArr[index], &attr,

workther_thread,&thread_array[index]). The fourth parameter is vital to creating unique thread id, which were binded (late binding) in the worker_thread function with their unique file names, allowing us to distinguish between client files from one another. The first parameter is the pthread_t dispatcher function from posix (allowing us to pass void * args - “unique data of the thread”), invoking pthreads to be of use via the program. Lastly, the worker_thread parameter (3rd) is a function that is invoked each time a thread is ready to go to work on a clients behalf. This function also assures that these allocated threads on the stack are free’d so that memory leaks do not become present.

`void *worker_thread(void *args)`

The worker function is the master processor of client request, where a thread arrives and is screened through various rules and regulations. The HTTP request is stripped of its contents via file name, buffer contents, and http protocol version. In addition the HTTP request goes through an overkill of parsing via REGEX, to ensure that the robustness of user inputs is flexible, but strict on outputs. Hence, we maintain a safety of margin along the way by checking file permissions, length, and allowed ASCII values; these hard checks are modularized into a black box of abstraction known as the f_bool_check_vitality, where should any of these conditions fail, will generate the appropriate response. Also, note that we extract the pthread_t critical information upon arrival of the create_worker (dispatcher) and cast it to a unique id, to distinguish it individuality from other threads, and associate it with its rightful client owner via an unordered map (a built in std from c++ that resembles a hash map). When a thread arrives, only one thread is allowed in the critical region by global lock. In the critical region area, if there are no clients (with request) in the queue one will be queued to sit patiently, while the incoming queue bypasses the waiting thread. Once that thread is complete, it is signaled on the other end of the httpserver.cpp, and released into the wild; meaning that the thread goes to work! Also note that each request is caudal into a struct of data, hence mitigating any loss of data by uniquely storying each client request to their unique http struct nodes. Once the screening process is complete the clients connection request, will be handed onto the thread to either perform a PUT/GET request.

2.5 Helper Functions

The helper functions are non-trivial, where the majority are specific to a sole single operation purpose. For example, the helper function f_void_400, indicates a naming convention that anyone with minimal computer science fundamentals can connect to that this function indeed returns a client response of 400 Not Found. In addition we also have a f_bool_check_vitality black box module, which is a great helper function that screens a client's request to the tee. Observe:

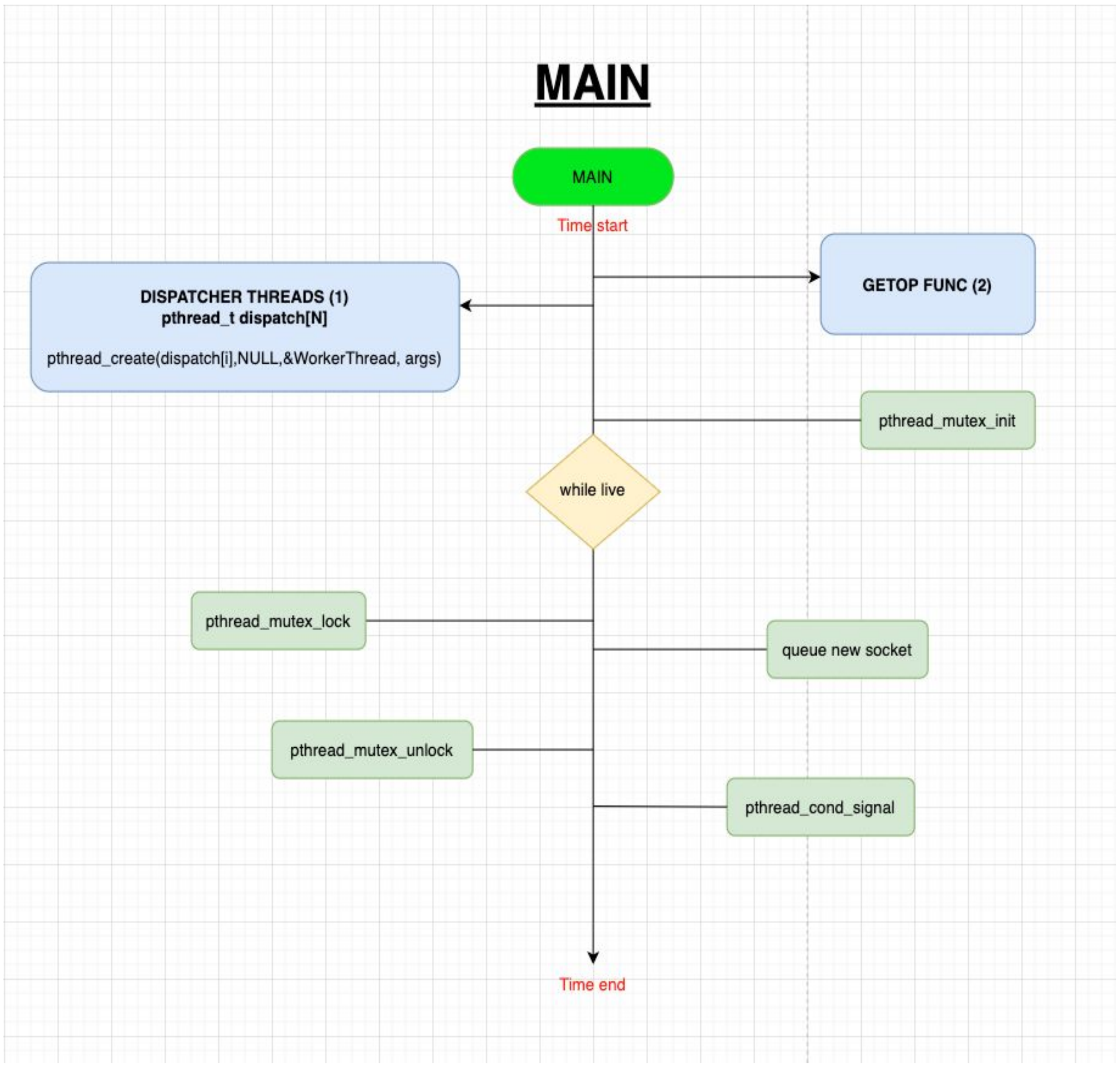
```
f_bool_check_vitality(char *file_name,char *protocol,char *request,int client_connect){

                              /* Do you have rights to the goods ? */

      bool check_file_access = f_bool_permissions(file_name);

      /* Check length of file must be = 10 ASCII */
      bool check_file_length = f_bool_check_file_len(file_name);

      /* Checking valid resource names A-Z/a-z/0-9 */
      bool check_file_format = f_bool_check_file(file_name);

      /* Check file permissions */
      bool check_request = f_bool_check_request(request);

      /* Check HTTP/1.1 version */
      bool check_protocol = check_http_protocol(protocol);
```

Naming is key here and through these isolated helper functions we break down each check so that we can mitigate the burden of debugging should an error be encountered. It is extremely vital to ensuring that the appropriate error checking protocols are established early on to ensure productivity. Through this black box of what I like to call TSA Screening, we check if a file is exactly 10 characters, does it meet the file formats, does one have permissions, and is it the right version… Once the client goes through the TSA screening (airport example), they are on their way for processing!

3.0 Architectural and Component-level design Driver

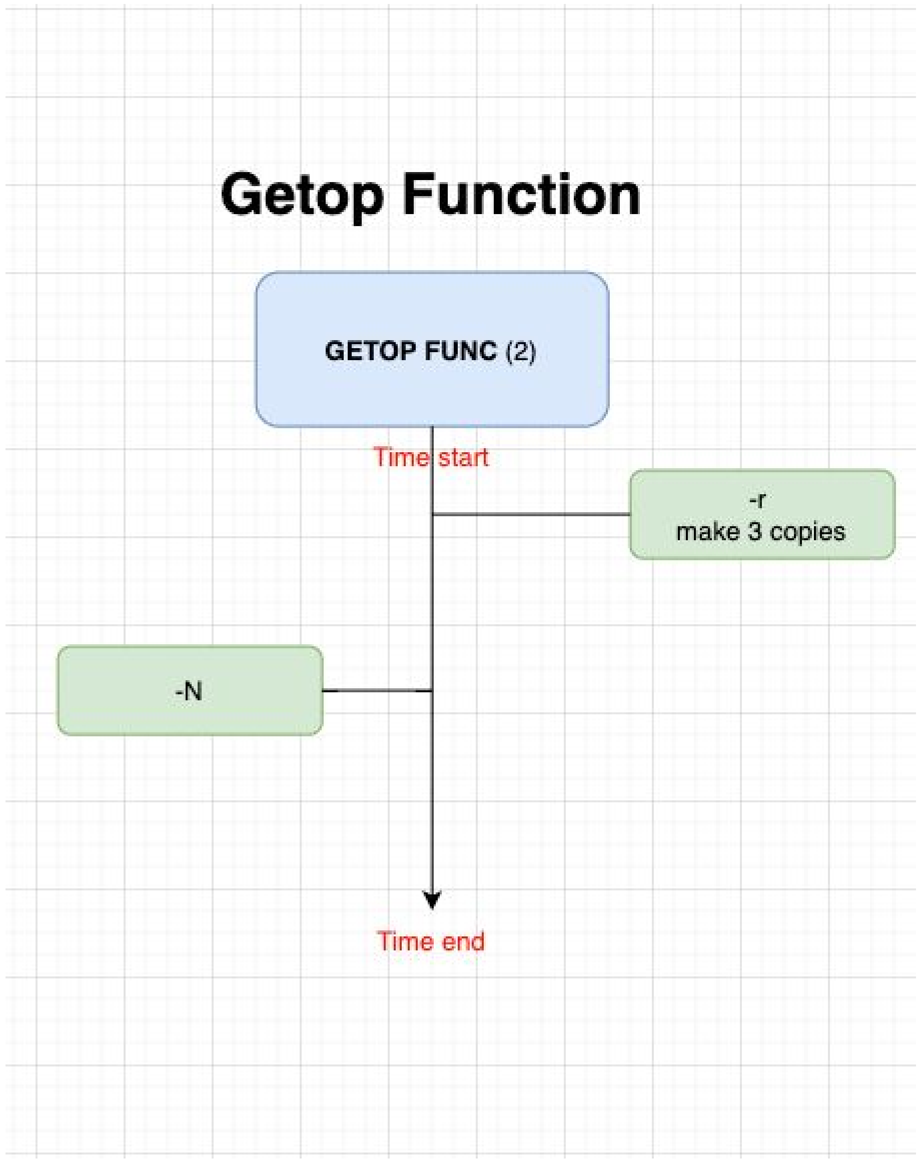
Visit the design flow-chart: Multi-threading-Flow-Chart
https://drive.google.com/file/d/1jAIC_GTAwp5UVUfCE3Oi95_wHc0RH-XV/view?usp=sharing

The Main httpserver diagram flow from a high level point of view dictates the process of the driver program and how multithreading is initialized. The GETOP parses the command line for function ar arguments, then the client connections are pushed into a queue, where they will sit in the thread pool (worker thread) until they are ready for work.



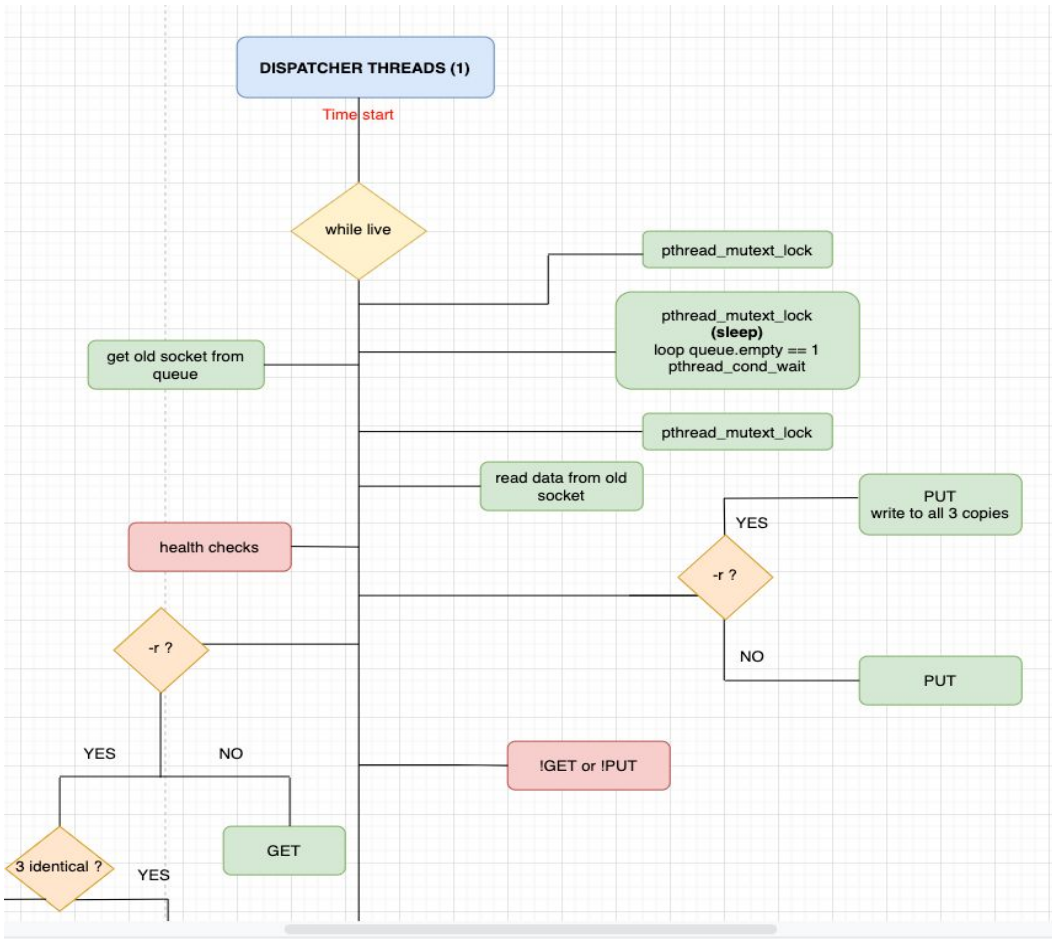
3.1 Architectural and Component-level Design Getop

The GETOP function function parser for command line flags such as -r, for redundancy, and -N for the number of threads. This piece is vital to the program, which allows our threads to be dictated by the user. The user may also invoke the redundancy feature upon providing the -r flag. A switch was used to signal a particular flag with the getopt c library feature.

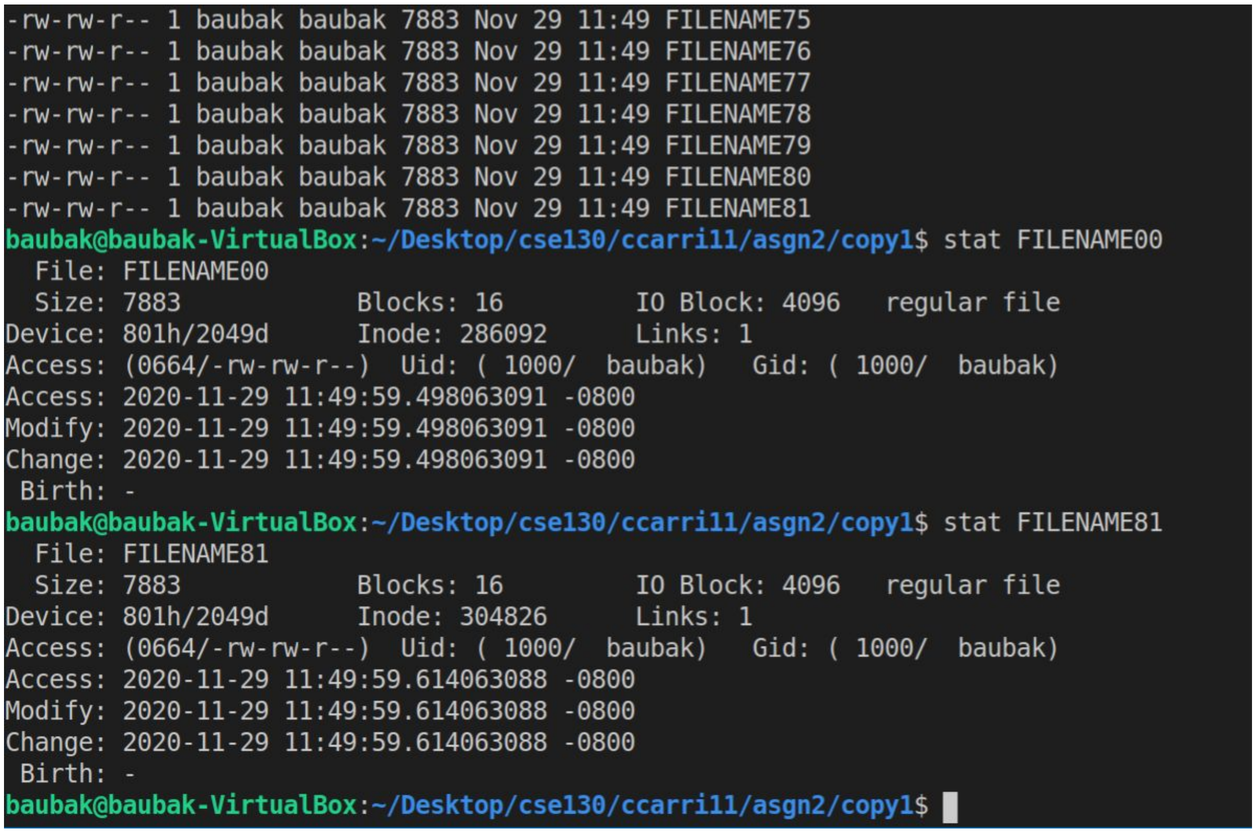


3.2 Architectural and Component-level Design Definitions and Flow

Once the client connection is received in the queue, it signals that a request has arrived to the dispatcher function (thread pool). This worker/boss model will create unique ids and associate each thread with this unique identification. Global locks ensure that the program and its entirety communicate across diagram flow regions and assure synchronization is followed as strictly through a flagSignal, which acts like a lightswitch in the critical region areas. These signals are shared across all major diagrams, in particular in the main driver program, during a client request, and when a worker arrives at the workstation. Once the boss states it's time for work, the worker is clocked in from the worker function and processed. The last thing to note is that each GET/PUT modules, upon file creation and retrieval have their own unique locks via a struct, this ensures that content is never lost upon transmission!



4.0 Screen images



```
-rw-rw-r-- 1 baubak baubak 7883 Nov 29 11:49 FILENAME75
-rw-rw-r-- 1 baubak baubak 7883 Nov 29 11:49 FILENAME76
-rw-rw-r-- 1 baubak baubak 7883 Nov 29 11:49 FILENAME77
-rw-rw-r-- 1 baubak baubak 7883 Nov 29 11:49 FILENAME78
-rw-rw-r-- 1 baubak baubak 7883 Nov 29 11:49 FILENAME79
-rw-rw-r-- 1 baubak baubak 7883 Nov 29 11:49 FILENAME80
-rw-rw-r-- 1 baubak baubak 7883 Nov 29 11:49 FILENAME81
baubak@baubak-VirtualBox:~/Desktop/cse130/ccarril1/asgn2/copy1$ stat FILENAME00
  File: FILENAME00
  Size: 7883            Blocks: 16         IO Block: 4096   regular file
Device: 801h/2049d      Inode: 286092      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/  baubak)   Gid: ( 1000/  baubak)
Access: 2020-11-29 11:49:59.498063091 -0800
Modify: 2020-11-29 11:49:59.498063091 -0800
Change: 2020-11-29 11:49:59.498063091 -0800
 Birth: -
baubak@baubak-VirtualBox:~/Desktop/cse130/ccarril1/asgn2/copy1$ stat FILENAME81
  File: FILENAME81
  Size: 7883            Blocks: 16         IO Block: 4096   regular file
Device: 801h/2049d      Inode: 304826      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/  baubak)   Gid: ( 1000/  baubak)
Access: 2020-11-29 11:49:59.614063088 -0800
Modify: 2020-11-29 11:49:59.614063088 -0800
Change: 2020-11-29 11:49:59.614063088 -0800
 Birth: -
baubak@baubak-VirtualBox:~/Desktop/cse130/ccarril1/asgn2/copy1$ 
```

The image above further assures us that we have implemented multithreading correctly and observing the power of multithreading and what it could do for us. We ran a bash.sh script with 82 PUT requests done simultaneously by using “&” (without piping) to the host server. Looking further into detail at the first file “FILENAME00” and the 82th file “FILENAME81”, we see that the timing in which these two files were created were the exact same, down to even the second. This gives us further confidence that we are implementing multithreading like it is supposed to.

5.0 Restrictions, limitations, and constraints

- Syscall param socketcall.accept(addrlen_in) points to uninitialized byte(s) at 0x578E791: accept (accept.c:26) is presented when you run (any of) the valgrind commands, varying compilers may have an offset. This offset if not accounted for... could impact generated data upon client request. This program application did not account for such offset, hence the following two commands will fail upon input via the terminal:

  1) Client Put Request : curl -T foo http://localhost:8080/FILENAME[0-N] -v

  2) Client Get Request : curl http://localhost:8080/FILENAME[0-N] -v

- Per assignment, each thread is only allowed 16Kib, however this is only regulated should the content-length not be present. The Put/Get modules do accommodate a buffer size of the allotted quantity dynamically should a content-length be provided.

6.0 Testing Issues

The following test cases were implemented to ensure program functionality, design, and performance:

- Program is tested vigorously by Valgrind for memory leaks.

- In the Test-Case directory you will find a brief description of what each file content contains. See TestList document for file content attributes.

- Unit testing of modularized components/segments were completed successfully upon each box ranging from the Regex parsing, helper functions, not limited to exhaustive amounts of data over multiple iterative GET/PUT requests of vast amounts. The goal was to break the program down by any means, and reinforce a monolithic structure that sustains a strong lifespan.

- Each component was modularizatized into abstract boxes and placed into the httpserver_definitions.cpp file. The design of each box was to constrain complexity from within the driver program, so that fellow programmers may comprehend this source code with ease. Hence, mitigating lines of code from the driver program to ensure that the hierarchy of abstraction is visible while achieving successful results.

- Program is tested vigorously by Valgrind for memory leaks.

- Testing can be done by typing in terminal: valgrind ./httpserver 127.0.0.1(optional --leak-check=full). Resulting in total heap usage: 3,703 allocs, 3,688 frees, 127,413 bytes allocated. Server is not in run mode.

- Testing can be done by typing in terminal: valgrind ./httpserver localhost 8080 --leak-check=full. Resulting in total heap usage:3,703 allocs, 3,688 frees, 127,413 bytes allocated. Server is not in run mode.

- Testing can be done ./httpserver localhost 8080 with no flags, while the server is running with the following command: valgrind ./httpserver localhost 8080. Since the program is concurrently in run mode, 6574 bytes will be in use in 1 block, a total heap usage of 3645 allocs, 3598 frees, and 106,763 bytes allocated. The amount reachable is 6574 bytes in 47 blocks, but this due to getaddrinfo(), a linked list, that has yet to be freed because the HTTP Server is active while conducting this test. Should the HTTP Server not be in run mode, you will have all no leaks as expected.

- Binary files (various sizes) were tested and compared with the diff -s foo1 foo2. Both are identical when generated and compared to the original file. You may also compare the original file interchangeably with foo1 and foo2.

- Files persist with this program should the client disconnect from the server. To test this feature, start the server, and go into a remote directory or desktop. In the terminal type in the following command: curl -T foo localhost:8080/1234567890 -v. This command will put the file, along with its contents, in the httpserver directory from your remote location. You may then retrieve this file from your location by typing in the following command in the terminal: curl localhost:8080/1234567890 > new_file01. To compare if your contents indeed were retrieved successfully, you may apply the following command: diff foo new_file01 -sq. For further reassurance, you may locate file 1234567890 within the Multithreaded HTTP Server directory, and compare all three contents interchangeably to ensure contents were recorded accurately.

- The Multithreaded HTTP Server program is to remain connected during all requests, unless ended by the client manually with control+c or any kill mechanism. The following test checks for prolonged connection persistence, in the Test-Case directory you will find various unit testing curl commands that send an amount of curl request via 1 command to illustrate the power and speed of a multithreaded server. Note, these commands were built using chmod +x foo.sh for the purpose of efficiency. In the terminal type the following:

- ./daniel_get_test.sh
- ./daniel_put_test.sh
- ./getWork.sh
- ./putWork.sh
- ./putWork_curl.sh
- ./getWork_curl.sh
- ./remove.sh (removes all the files generated in the http dir and Test-Case dir).
- etc

The daniel command contains 82 files that are being transmitted from client to server vice versa, each file contains 16 Kib of content for this test, the max allotted content for a thread as indicated per assignment pdf documentation. The majority of chmod commands replicate the following two curls:

- curl -T foo localhost:8080/FILENAME00 > cmd0.output1 &
- curl localhost:8080/FILENAME00 -v > cmd0.output1 &

These commands will generate a Get/PUT Request, returning the content sequentially in the order that it was called, with the content, status msg e.g 200,201,404, etc..., while threading, all while maintaining a connected status.

- The following commands may be inputted in the terminal,as presented by TA's Daniel Santos & Yiming Zhang:

client side:

- echo Hello World > t1
- head -c 80000 /dev/urandom | od -x > t2
- head -c 200 /dev/urandom > b1

server side:

- echo Hello World > SmallFile1
- head -c 80000 /dev/urandom | od -x >LargeFile1
- head -c 200 /dev/urandom > SmBinFile1

./httpserver localhost 8001 -N threads &

- (curl -T t1 http://localhost:8001/Small12345 -v & \
- curl -T t2 http://localhost:8001/Large12345 -v)

./httpserver localhost 8003 -N threads -r &

- (curl -T t1 http://localhost:8003/Small12345 -v & \
- curl -T b1 http://localhost:8003/binaryfile -v)

./httpserver localhost 8005 -N threads &

- (curl http://localhost:8005/SmallFile1 -v > out1 & \
- curl http://localhost:8005/LargeFile1 -v > out2)

./httpserver localhost 8004 -N threads -r &

- (curl http://localhost:8004/SmallFile1 -v > out3 & \
- curl http://localhost:8004/SmBinFile1 -v > out4)

All the following commands can be implemented gracefully via the terminal, with the intended anticipated passing results.

- The Multithreaded HTTP Server program was tested with the bible's length of 4.5 million bytes on a (Put Request) along with various files included for a consecutive 8 requested threads, supporting more than 16 Kib if needed per thread due to dynamic memory allocation. Should content length not be provided, the default buffer size of 16Kib will be adhered to as indicated. These consecutive threads sustain successfully keeping the connection intact to the server. This command may be implemented as ./putWork_curl.sh. See Testlist document in Test-Case for further details on implementation.

- The Multithreaded HTTP Server program was tested with the bible's length of 4.5 million bytes on a (Get Request) along with various files included for a consecutive 8 requested threads, supporting more than 16 Kib if needed per thread due to dynamic memory allocation. Should content length not be provided, the default buffer size of 16Kib will be adhered to as indicated. These consecutive threads sustain successfully keeping the connection intact to the server. This command may be implemented as ./putWork_curl.sh. See Testlist document in Test-Case for further details on implementation.

- The Multithreaded HTTP Server program when prompted with multiple Get Request/Put Request will remain open during the entire process, patiently awaiting for the next curl upon finishing its objective, and will respond accordingly with the appropriate response should a file be present, internal error, forbidden, created, etc... all while keeping the connection alive.

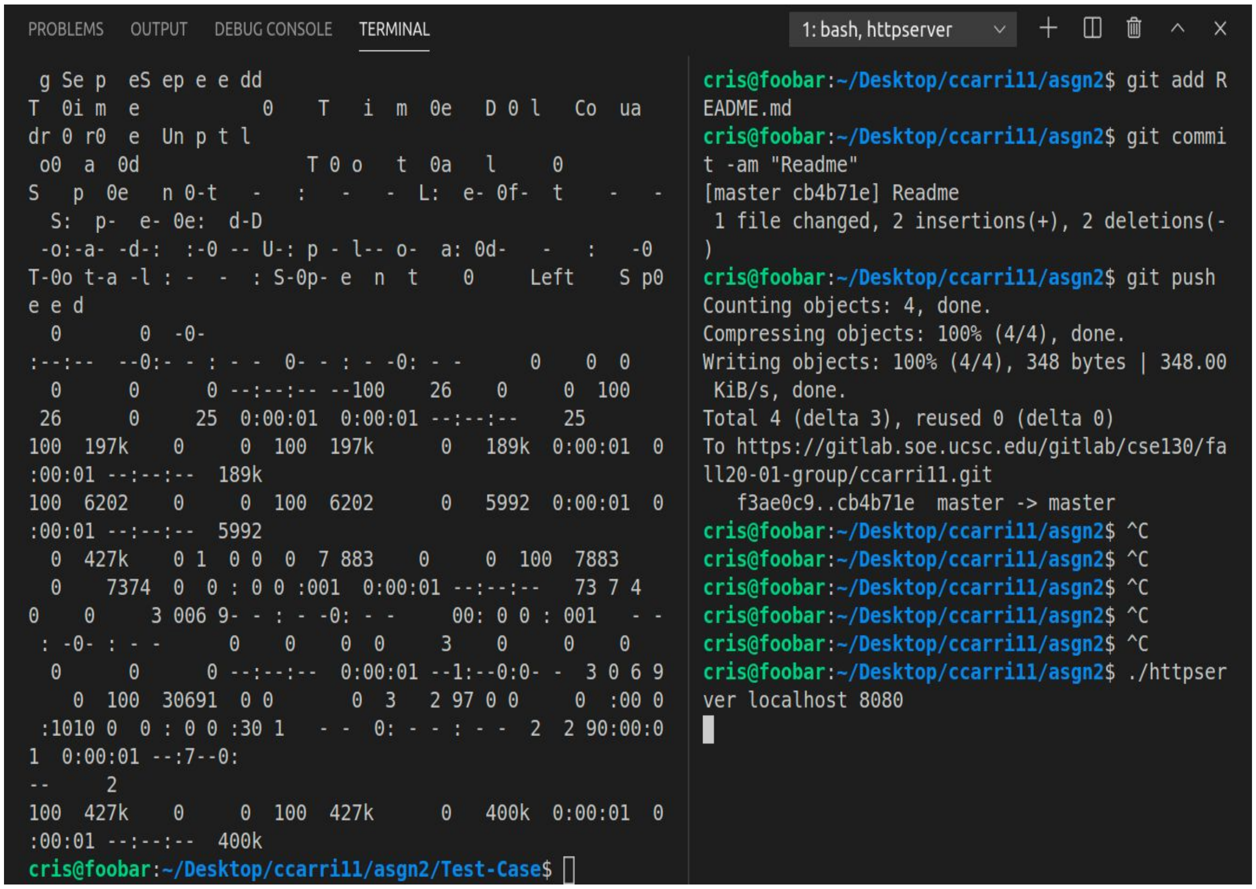
- The Multithreaded HTTP Server program will acknowledge a file of content-length 0, binary, null upon a Put Request, and will retrieve them successfully upon a Get Request.

- This program sustains a Worker/Boss model implementation (thread-pool flow style). Threads will wait until invoked by the user, synchronized accordingly (Global and Local locks), and process in parallel upon any request presented.

- **NOTE:**

```
#!/bin/bash

array0=(1 2 3 4 5 6 7 8)

for i in "${array0[@]}";do

            curl -T "$i" localhost:8080/123456789"$i"  > cmd"$i".output"$i" &

done
```

The PUT/GET request will generate compiled random generation (see below) randomly. These commands will present themselves in a hanging state… However, this is not the case!! **Drag the terminal in any direction (resizing it just a small bit) and you should see a successful PUT/GET that has closed on its own accordingly!**



```
 g Se p  eS ep e e dd
T  0i m  e           0    T   i  m  0e   D 0 l   Co  ua
dr 0 r0  e  Un p t l
 o0  a  0d               T 0 o   t  0a   l     0
S   p  0e   n 0-t   -   :   -   -  L:  e- 0f-  t    -   -
  S:  p-  e- 0e:  d-D
 -o:-a- -d-:  :-0 -- U-: p - l-- o-  a: 0d-    -   :   -0
T-0o t-a -l : -  -  : S-0p- e  n  t    0     Left    S p0
e e d
  0       0  -0-
:--:--  --0:- - : - -  0- - : - -0: - -       0    0  0
  0      0      0 --:--:-- --100    26    0      0  100
 26      0     25  0:00:01  0:00:01 --:--:--     25
100  197k    0      0  100  197k      0   189k  0:00:01  0
:00:01 --:--:--  189k
100  6202    0      0  100  6202      0   5992  0:00:01  0
:00:01 --:--:--  5992
  0  427k    0 1  0 0   0  7 883    0      0  100  7883
  0    7374  0  0 : 0 0 :001  0:00:01 --:--:--   73 7 4
0    0     3 006 9- - : - -0: - -      00: 0 0 : 001    - -
 : -0- : - -       0    0    0  0     3    0      0    0
  0      0      0 --:--:--  0:00:01 --1:--0:0- -  3 0 6 9
   0  100  30691  0 0       0  3   2 97 0 0     0  :00 0
 :1010 0  0 : 0 0 :30 1   - -  0: - - : - -  2  2 90:00:0
1  0:00:01 --:7--0:
--     2
100  427k    0      0  100  427k      0   400k  0:00:01  0
:00:01 --:--:--  400k
cris@foobar:~/Desktop/ccarri11/asgn2/Test-Case$ 
```

```
cris@foobar:~/Desktop/ccarri11/asgn2$ git add R
EADME.md
cris@foobar:~/Desktop/ccarri11/asgn2$ git commi
t -am "Readme"
[master cb4b71e] Readme
 1 file changed, 2 insertions(+), 2 deletions(-
)
cris@foobar:~/Desktop/ccarri11/asgn2$ git push
Counting objects: 4, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 348 bytes | 348.00
 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
To https://gitlab.soe.ucsc.edu/gitlab/cse130/fa
ll20-01-group/ccarri11.git
   f3ae0c9..cb4b71e  master -> master
cris@foobar:~/Desktop/ccarri11/asgn2$ ^C
cris@foobar:~/Desktop/ccarri11/asgn2$ ^C
cris@foobar:~/Desktop/ccarri11/asgn2$ ^C
cris@foobar:~/Desktop/ccarri11/asgn2$ ^C
cris@foobar:~/Desktop/ccarri11/asgn2$ ^C
cris@foobar:~/Desktop/ccarri11/asgn2$ ./httpser
ver localhost 8080
```

(Drag me just a bit in any direction)

6.1 Expected software response

- On a successful Put request a 201 Created status shall be generated and presented to the client.
- On a successful Get request a 200 OK status shall be generated and presented to the client.
- On a Put request that was interrupted internally the client shall be notified of a 500 Internal Response Message.
- On a Get/Put request that exceeds that is not an exact amount of ten ASCII values, a 400 Bad request shall be sent to the client.

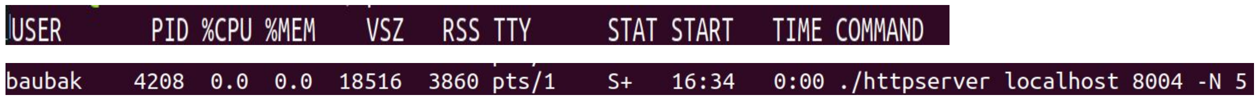
- On a GET/PUT request that seeks to retrieve a privileged file, permissions will be checked and should those permissions not be granted, the client will be notified of a Forbidden 403 Message
- Generic responses will be generated should the Multithreaded program fail prior to connection such as a listening, bind, socket, etc with the appropriate error message.
- Each request will be screened for HTTP/1.1 Version protocol, and strictly adhere to this rule, should a request not meet this standard, a 400 message of “Not Found” will be generated.
- Should a client prompt for a Get request, and the file does not persist on the server, the client will be presented with a 400 Not Found response.

6.2 Identification of critical components

To make sure our threads are being successfully created, we dug into the ubuntu kernel to explore and validify that the threads are indeed created and that the number of threads are identical to the number of threads indicated by the host upon server start. Below are the steps we took to observe this (Note: server needs to be running):

1. Run the server program ./httpserver localhost 8080 -N 5 then do a GET or a PUT request, continue to step 2 when request is fulfilled and completed.

2. Go to a separate terminal and type "ps aux" which prints all the processes owned by the user (you). Search under the command column for “./httpserver localhost 8080 -N 5” (Below is the image of the row. data changes and is different for each user)



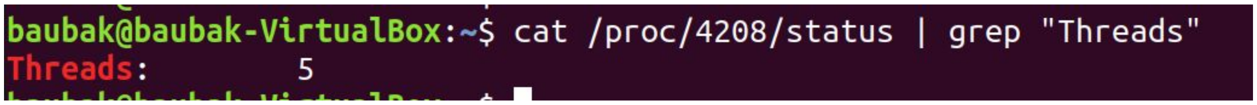
```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
baubak    4208  0.0  0.0  18516  3860 pts/1    S+   16:34   0:00 ./httpserver localhost 8004 -N 5
```

3. Look for the running process’s PID. In the case of the image above, it’s the numbered value 4208.

4. Then enter to the terminal the command :
   cat /proc/<PID>/status | grep "Threads"
   In our case from the image above it will be

cat /proc/4208/status | grep "Threads"

Below is the results that the command yields:



```
baubak@baubak-VirtualBox:~$ cat /proc/4208/status | grep "Threads"
Threads:        5
```

7.0 Pseudocode

Procedure httpserver.cpp

```
Struct command_line_inputs *cmd, get_cmd
cmd = &get_cmd

const char *portNumber
const char *host_address

counter_left_off = 0

FOR i = 1 TO i < argv DO
    IF strlen(argv[ j ]) >= 7 AND  strlen(argv[ j ]) <= 15 DO
        match_result = regex_match(argv [ j ] ,
"(localhost|[0-9]{1,3}[.]{1}[0-9]{1,3}[.]{1}[0-9]{1,3}[.]{1}[0-9]{1,3})" )
        IF match_result == 1 DO
            host_address = argv[ j ]
            counter_left_off = j
            break
        Else DO
            perror(“Invalid <ip address> format\n”);
            exit(EXIT_FAILURE)

IF counter_left_off == 0 DO
    perror(“Invalid <ip address> or <ip address> not entered.\n”)
    exit(EXIT_FAILURE)

portNumber = “80”
FOR j = counter_left_off  TO  j < argc DO
    IF strlen(argv[ j ]) == 4
```

```
                    portNumber = argv[ j ]
                    break

struct sockaddr_storage client_addr
 struct addrinfo hints, *res
 socklen_t addr_size
  enable = 1



memset(&hints, 0, sizeof(hints))
hints.ai_family = AF_INET
hints.ai_socktype = SOCK_STREAM
hints.ai_protocol = IPPROTO_TCP

IF  return_val = getaddrinfo(host_address, portNumber, &hints, &res) != 0 DO
          CALL f_void_getaddrinfo_error(return_val, res)

CALL f_getopt(argc, argv, &get_cmd)

serverSocket = socket(res->ai_family, res->ai_socktype, res->ai_protocol)

IF serverSocket < 0 DO
          CALL f_void_setsocket_error(serverSocket)

IF return_val = setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR, &enable,
sizeof(int)) < 0 DO
          CALL f_void_setsocket_error(return_val)

IF return_val = bind(serverSocket, res->ai_addr, res->ai_addrlen) < 0 DO
          CALL f_void_bind_error(return_val)

pthread_mutex_init(&mutex, NULL)
pthread_cond_init(&client_in_queue, NULL)

IF return_val = listen(serverSocket, BACKLOG) == -1 DO
          CALL f_void_listen_error(return_val)

IF cmd->Redundancy == 1 DO
```

```
        message->is_redundant = cmd->Redundancy
        (GLOBAL VAR) Redundancy = 1

*threader = ((int*)malloc(sizeof(int)*cmd->N_threads))

pthread_t *dispatcher_thread = ((pthread_t*)malloc(sizeof(pthread_t)*cmd->N_threads))

sleep(1.5)
counter = 0

WHILE(TRUE) DO

        IF cmd->Redundancy == 1 DO
                clientSocket = accept(serverSocket,(struct sockaddr *)&client_addr,
        &addr_size)
                create_worker(dispatcher_thread,threader,&get_cmd)

                pthread_mutex_lock(&mutex) DO LOCK

                        flagSignal = 0
                        IF StackItems IS empty DO
                                flagSignal = 1

                        StackItems push (clientSocket)

                        IF flagSignal == 1 DO
                                pthread_cond_broadcast(&client_in_queue)

                pthread_mutex_unlock(&mutex) DO UNLOCK
                continue



        clientSocket =  accept(serverSocket,(struct sockaddr *)&client_addr, &addr_size)
        StackConnections push_back (clientSocket)
        CALL create_worker(dispatcher_thread, threader, &get_cmd)

        pthread_mutexd_lock(&mutex) DO LOCK
                flagSignal = 0
        IF StackItems IS empty DO
            flagSignal = 1

        StackItems push (StackConnections[counter])

        IF flagSignal == 1 DO
            pthread_cond_broadcast(&client_in_queue)
    pthread_mutex_unlock(&mutex) DO UNLOCK
    counter = counter + 1


freeaddrinfo res //clears res's linked list
exit(EXIT_SUCESS)
```

Procedure for create_worker (dispatcher pool)

```
int numWorker = configs->N_threads;
free(&configs);

rc = pthread_attr_init(&attr)

IF rc == -1 DO
    perror("error in pthread_attr_init:")
    exit(1)

s1 = 4096  //max header size
rc = ptrhead_attr_setstacksize(&attr, s1)

IF rc == -1 DO
    perror("error in pthread_attr_setstacksize")
    exit(2)

FOR index = 0 TO index < numWorker DO
    thread_array[index] = index
    return_val = pthread_create(&workerArr[index[, &attr, worker_thread,
&thread_array[index])

    IF return_val  == 1 DO
```

```
        sprintf(msg, “Failed on pthread_create()%d\n”, return_val)
        exit(EXIT_FAILURE)

Free &thread_array
Free &workerArr
```

Procedure for worker_thread

```
struct http_object *node_parser,get_http_content
node_parser = &get_http_content

int id = *((int *)args)  // argos == thread_array[index]

WHILE(TRUE) DO
    clientSocket = -1
    pthread_mutex_lock(&mutex) DO LOCK
        WHILE StackItems IS empty DO
            pthread_cond_wait(&client_in_queue, &mutex)

        clientSocket = StackItems.front()
        StackItems.pop()
    pthread_mutex_unlock(&mutex)


buffer[4096]
msgsize = 0

WHILE bytes_read = read(clientSocket, buffer+msgsize,sizeof(buffer)-msgsize-1) > 0
DO
    msgsize = msgsize + bytes_read
    IF msgsize > 4096-1 || buffer[msgsize-1] == ‘\n’ DO
        break

buffer[msgsize-1] = 0 //null terminate message and remove \n


char request[4], file[50], protocol[10]
sscanf(buffer,"%s %s %s",request,file, protocol)
```

```
node_parser->buffer = buffer
node_parser->protocol = protocol
node_parser->request = request
node_parser->file_name = file
node_parser->client_connect = clientSocket
node_parser->id = id

match_value = CALL
f_regex_overkill_parse(node_parser->request,node_parser->file_name,node_parser->p
rotocol)

IF match_value != 1 DO
    CALL f_void_400(node_parser_client_connect)
    return NULL

*theFile  = file
*restOfString = theFile
*file_name = strtok_r(restOfString,"/",&restOfString)
node_parser->file_name = file_name
node_parser->client_connect = clientSocket

die_on_failure = CALL
f_bool_check_vitality(node_parser->file_name,node_parser->protocol,node_parser->re
quest,node_parser->client_connect)

IF die_on_failure == FALSE DO
    memset(node_parser, 0, sizeof(http_object))
    continue

ss << node_parser->file_name
ss >> string_to_parse

std::unordered_map<std::string,int>hashing

hashing[string_to_parse.c_str()] = node_parser->id

std::unordered_map<std::string, int>::iterator lookup
```

```
FOR look = hashing.begin() TO lookup != hashing.end() DO
          IF  strcmp(node_parser->request, “PUT”) == 0 AND
strcmp(node_parser->request,”GET”) != 0 DO
                    CALL f_void_put_module(bufferm node_parser->file_name,
node_parser->client_connect)
                    memset(node_parsor, 0, sizeof(http_object))
                    return NULL

IF strcmp(node_parser->request,"GET") == 0 AND
strcmp(node_parser->request,"PUT") != 0) DO
               CALL  f_void_put_module(bufferm node_parser->file_name,
node_parser->client_connect
               memset(node_parser,0,sizeof(http_object))
               return NULL

return NULL
```