Design Document - Assignment 2

Derrick DeBose

1 Goal

The goal of this assignment is to create a multithreaded HTTP protocol server in which we can use clients like curl in order to make get and put requests.

2 Assumptions

We are assuming that this code will be run on Ubuntu 18.04 VM. So, this code may not work if it were to run on a Mac or Windows system. We are going to assume that we are using an IPv4 address. Assume, no bad arguments will be given on the server side when starting up the server. Also assume that when starting up the server the host name is always given before the port number

3 Design

I will start the socket program by creating a server socket. I will then assign attributes of the address from the command line arguments like ip address and port number. We will be able to detect the -N and -l flags from the getopt function. I can then bind the address to server file descriptor. Then we will listen for a client to connect to the server.

Then we will start a while loop where we can accept the client's connection. Every client socket that gets accepted will enter a critical region where that client socket will get pushed onto the queue. When the server leaves the critical region a signal will be sent to the waiting thread to tell them that there is a request in the queue that needs to be processed. When the thread is woken up, it will already be in a critical region where the thread will pop off the clients file descriptor from the queue, leave the critical region and process the request.

Processing the request will start by reading the request from the client that we can parse the request on newline in order to figure out the command, the filename, the HTTP/1.1 header for the response header and the content length if the request was a put request.

If not a get or put request, send a 500 status code.

We will check to see if the filename used is invalid. If invalid then return a 400 error. If we have a valid filename we can determine if request was Get or Put.

If a get request then first get the content length of the file and send the response header. Then we can actually send over the data to the client of the file they want from the server.

If a put request then we want to know the difference between a 200 or 201 status conformation but send a 403 if we can not truncate the file that we are trying to overwrite. Then we will use the content length number we found when scanning the put request in, to determine how much data we need to recv from the client and write to the file.

Close the client socket, and continue to the top of the while loop in order to accept another connection from a client.

Lastly we need to close the server socket.

4 Pseudocode

We will use the Server algorithm as our main method where we will use the functions: get, put, client_reqs, queue_socket_id for the program. We will need the global values: global_log_offset, queue, mutex_lock, and pthread_cond_t

```
if argc ==1 then
    error(”Not Enough Arguments”);
end
create socket file descriptor;
address.family = IPv4 address type;
if getopt(argc, argv, ”N: l:” then
    switch case do

    end
     do
        N
    end
    : nflag = true;
    case l do
        :
    end
    log = true;
end
set N for number of threads;
set logFile name;
set hostname or port number;
for n = 0 ->N do
    pthread_create(thread_id[n], NULL, client_reqs, (void)arguments);
end
bind the server file descriptor to the address;
listen for a client;
while 1 do
    Accept client’s connect;
    mutex_lock;
    push(socket_id);
    mutex_unlock;
    signal_thread;
end
close server;
return 0;
```

Algorithm 1: Server

```
This function is called from pthead_create.
while 1 do
    int new_socket = queue_socket_id();
    set new_socket;
    process_requests(arguments);
end
return 0;
```

Algorithm 2: client_reqs

```
pthread_mutex_lock();
if requests.empty() then
    // this waits for a signal to know that we can pop from the queue;
    pthread_cond_wait();
end
int sock_id = requests.front();
requests.pop();
pthread_mute_unlock();
return sock_id;
```

Algorithm 3: queue_socket_id

```
read client's request;
split request on new line;
scan lines for command, filename, and content length;
if not get or put request then
    error(500);
end
if invalid filename then
    error(400);
    close client socket;
    continue;
end
if command is a get request then
    content_length = get_filesize();
    get();
end
else if command is a put request then
    call put function;
    print(content length);
end
close client socket;
```

Algorithm 4: process_requests

```
open the file;
if 404 status then
    return 0;
end
else if 403 status then
    return 0;
end
while read_count >0 do
    read_count = read(file);
    if read_count >0 then
        content_length += read_count;
    end
end
return content_length;
```
Algorithm 5: get_filesize

```
open the file;
while content length >0 do
    read the file;
    sent count = send the file to client socket;
    content length -= sent count
end
```
Algorithm 6: Get

```
open/create the file;
while content length >0 do
    receive the data from the client socket;
    write count = write to the file on the server;
    content length -= write count;
end
```
Algorithm 7: Put