USER INSTRUCTIONS

Using the program is very simple: once you’ve build the project by make command, run the created executable file with command ./executable (when the executable is in your current directory, otherwise navigate to it). You can optionally give the amount of servers to be created as a command line argument. For example ./executable 7 would create 7 server threads (instead of the default 4).

Running the executable opens up two extra terminals:

-One for server side outputs, describing what is happening with the server
side program, which was opened in the executable file by running the file server_executable through system call.

-The other one is for server responses to your requests.

You continue to type the commands in the terminal in which you ran the executable in the first place, under the field "Give serve-request:".

There are quick commands: h, which prints out all the available commands, and e to exit the program (can be also done by pressing ctrl + c). To submit requests press enter.

FEW IMPORTANT NOTIONS:

SOMETIMES THE CONNECTION IS REFUSED BY THE SERVER THE FIRST TIME TRYING TO CONNECT. WHEN THAT HAPPENS, JUST TRY TO RUN THE EXECUTABLE AGAIN.

WHEN CREATING AN ACCOUNT, YOU WON'T GET SERVER RESPONSE. YOU CAN ASSURE THAT THE ACCOUNT WAS CREATED BY THE COMMAND: l <id>. THE CREATION IS ALSO SHOWN IN THE SERVER-SIDE TERMINAL.

FURTHER CODE DOCUMENTATION:

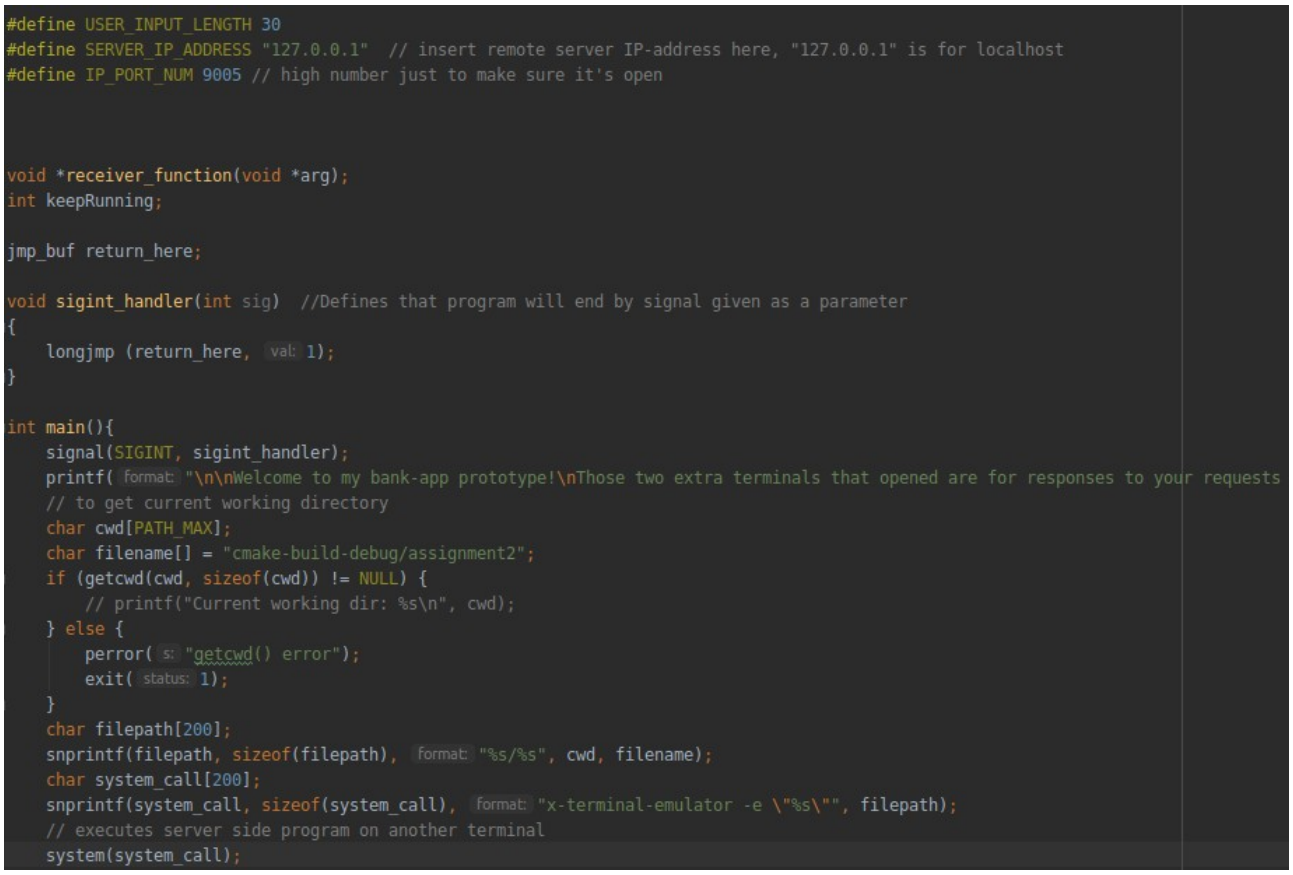
Quick links to implementations of the specific requirements:

Communication between processes or programs

Thread creation

Use of mutex (r/w-locking in my program)

Thread synchronization

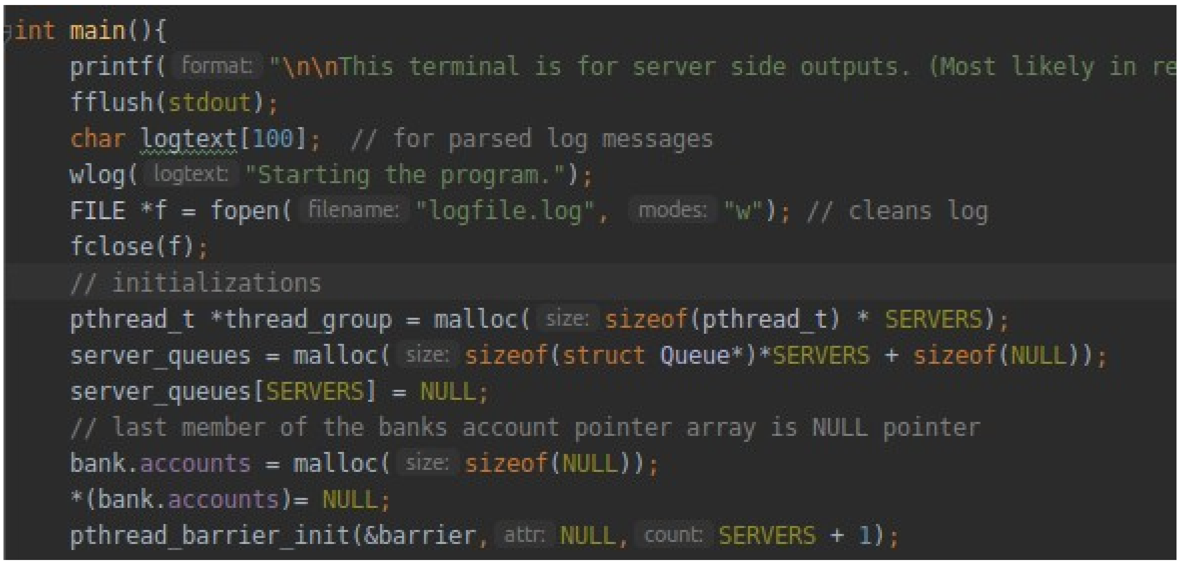


```
#define USER_INPUT_LENGTH 30
#define SERVER_IP_ADDRESS "127.0.0.1"  // insert remote server IP-address here, "127.0.0.1" is for localhost
#define IP_PORT_NUM 9005 // high number just to make sure it's open



void *receiver_function(void *arg);
int keepRunning;

jmp_buf return_here;

void sigint_handler(int sig)  //Defines that program will end by signal given as a parameter
{
    longjmp (return_here,  val: 1);
}

int main(){
    signal(SIGINT, sigint_handler);
    printf( format: "\n\nWelcome to my bank-app prototype!\nThose two extra terminals that opened are for responses to your requests
    // to get current working directory
    char cwd[PATH_MAX];
    char filename[] = "cmake-build-debug/assignment2";
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        // printf("Current working dir: %s\n", cwd);
    } else {
        perror( s: "getcwd() error");
        exit( status: 1);
    }
    char filepath[200];
    snprintf(filepath, sizeof(filepath),  format: "%s/%s", cwd, filename);
    char system_call[200];
    snprintf(system_call, sizeof(system_call),  format: "x-terminal-emulator -e \"%s\"", filepath);
    // executes server side program on another terminal
    system(system_call);
```

The beginning of the main in TCPclient program, which is the code the the executable is running, does:

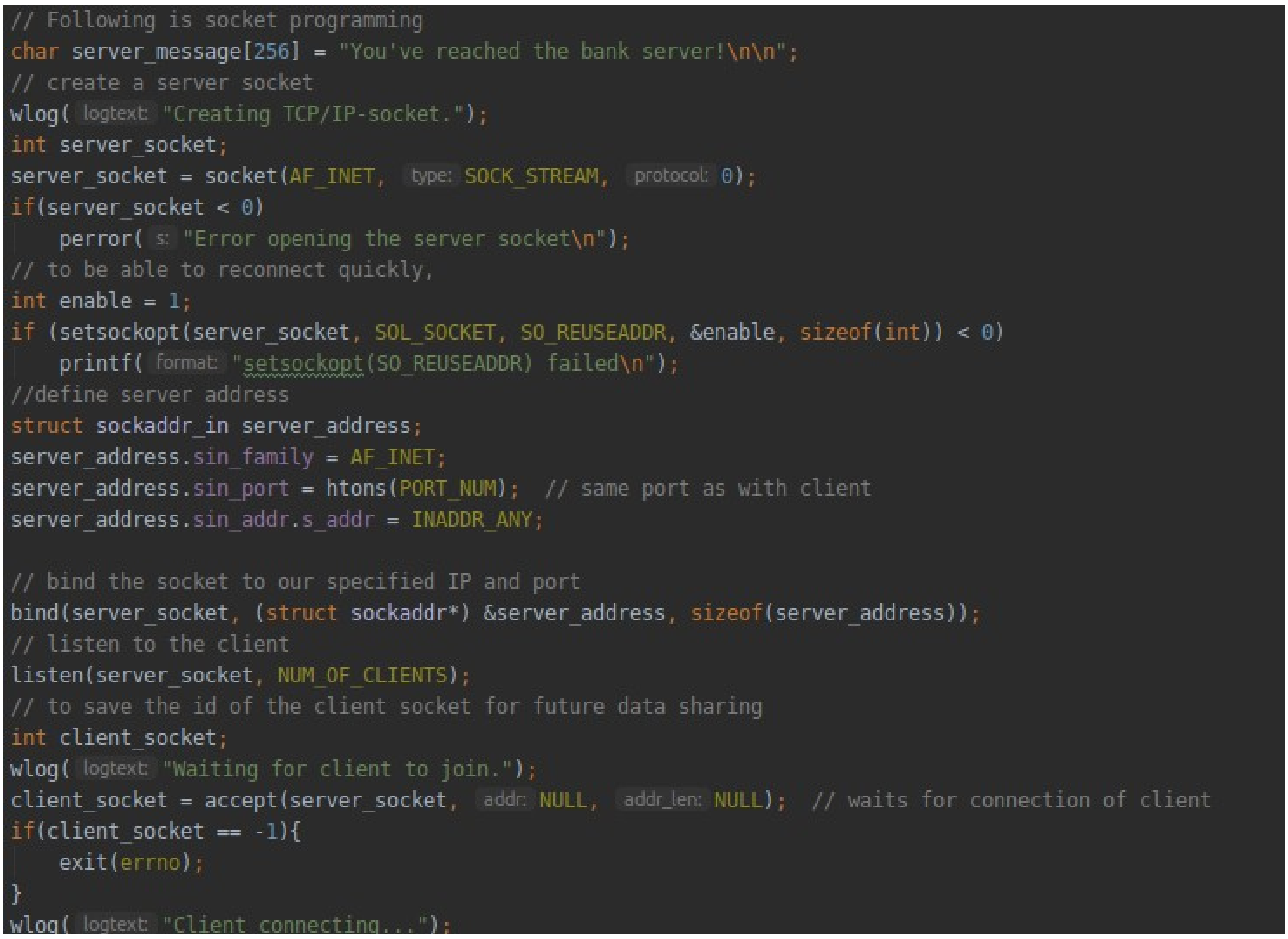
- Replace SIGINT’s (signal send to program when user presses CTRL + C) normal signal handler with own implementation, that relies on moving the execution of the code to block that ends the program by concept of jumping.

- Execute the server side program by running the file server_executable in the directory by system call. Let’s move to that program code since it is ran first.



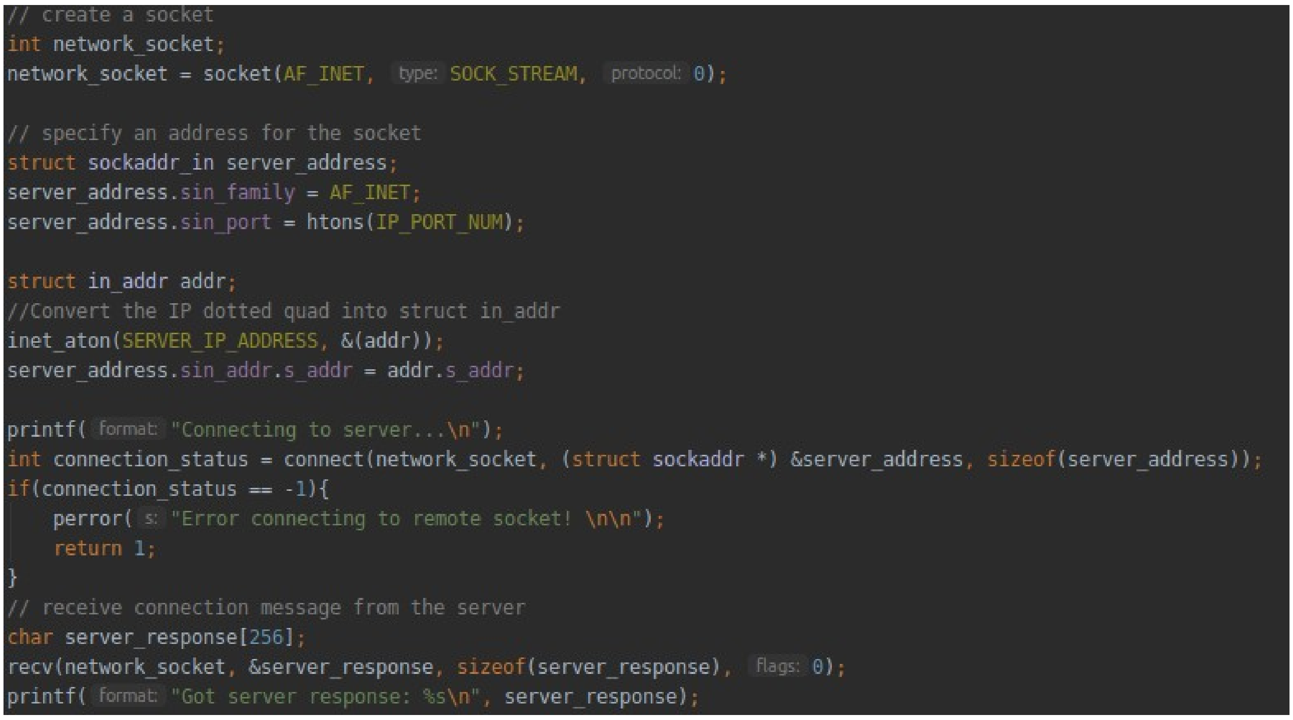
```
int main(){
    printf( format: "\n\nThis terminal is for server side outputs. (Most likely in re
    fflush(stdout);
    char logtext[100];  // for parsed log messages
    wlog( logtext: "Starting the program.");
    FILE *f = fopen( filename: "logfile.log",  modes: "w"); // cleans log
    fclose(f);
    // initializations
    pthread_t *thread_group = malloc( size: sizeof(pthread_t) * SERVERS);
    server_queues = malloc( size: sizeof(struct Queue*)*SERVERS + sizeof(NULL));
    server_queues[SERVERS] = NULL;
    // last member of the banks account pointer array is NULL pointer
    bank.accounts = malloc( size: sizeof(NULL));
    *(bank.accounts)= NULL;
    pthread_barrier_init(&barrier, attr: NULL, count: SERVERS + 1);
```

Above is the beginning of the main of the server-program, ran by the server_executable. It is just memory reservation for the logging tools, threads, and the structs for queues and the bank (implementations of these structs are found in c files named accordingly).



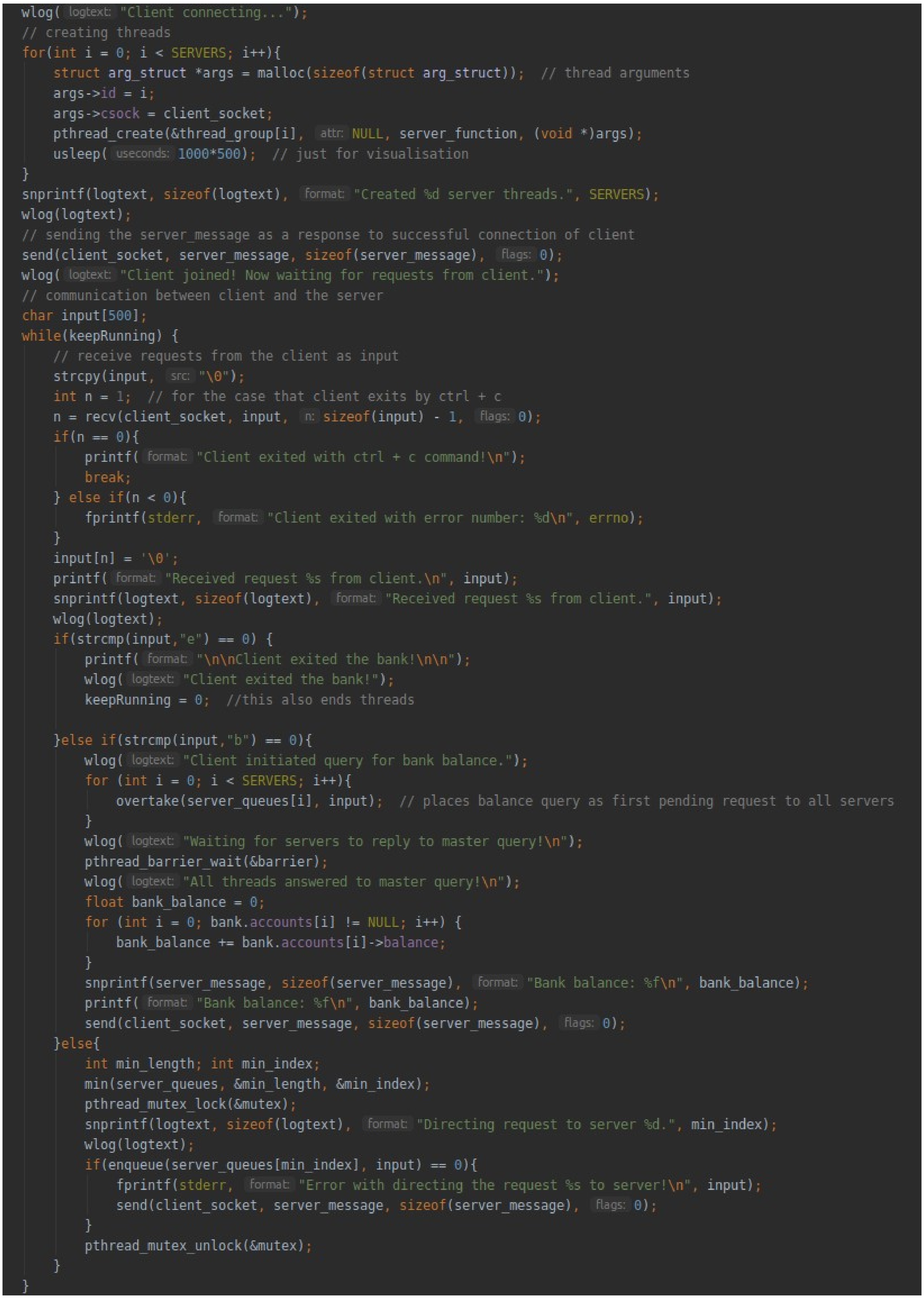
```
// Following is socket programming
char server_message[256] = "You've reached the bank server!\n\n";
// create a server socket
wlog( logtext: "Creating TCP/IP-socket.");
int server_socket;
server_socket = socket(AF_INET,  type: SOCK_STREAM,  protocol: 0);
if(server_socket < 0)
    perror( s: "Error opening the server socket\n");
// to be able to reconnect quickly,
int enable = 1;
if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
    printf( format: "setsockopt(SO_REUSEADDR) failed\n");
//define server address
struct sockaddr_in server_address;
server_address.sin_family = AF_INET;
server_address.sin_port = htons(PORT_NUM);  // same port as with client
server_address.sin_addr.s_addr = INADDR_ANY;

// bind the socket to our specified IP and port
bind(server_socket, (struct sockaddr*) &server_address, sizeof(server_address));
// listen to the client
listen(server_socket, NUM_OF_CLIENTS);
// to save the id of the client socket for future data sharing
int client_socket;
wlog( logtext: "Waiting for client to join.");
client_socket = accept(server_socket,  addr: NULL,  addr_len: NULL);  // waits for connection of client
if(client_socket == -1){
    exit(errno);
}
wlog( logtext: "Client connecting...");
```

Above code is my implementation of socket programming, that is commentated so well, that it needs no further explanation. When the code reaches the accept method, it is blocked and left to wait for the client connection.



```c
// create a socket
int network_socket;
network_socket = socket(AF_INET, type: SOCK_STREAM, protocol: 0);

// specify an address for the socket
struct sockaddr_in server_address;
server_address.sin_family = AF_INET;
server_address.sin_port = htons(IP_PORT_NUM);

struct in_addr addr;
//Convert the IP dotted quad into struct in_addr
inet_aton(SERVER_IP_ADDRESS, &(addr));
server_address.sin_addr.s_addr = addr.s_addr;

printf( format: "Connecting to server...\n");
int connection_status = connect(network_socket, (struct sockaddr *) &server_address, sizeof(server_address));
if(connection_status == -1){
    perror( s: "Error connecting to remote socket! \n\n");
    return 1;
}
// receive connection message from the server
char server_response[256];
recv(network_socket, &server_response, sizeof(server_response), flags: 0);
printf( format: "Got server response: %s\n", server_response);
```

Now, the samekind of socket programming is also happening in the client side, and it is trying to connect to the server socket, which address we have configured above. The defined variable SERVER_IP_ADDRESS is the key part if wanting to connect to a remote machine. Now it is just set to 127.0.0.1 in the definitions (after imports) which is just the local host, so it loops back to own computer. The program is left blocking until it receives green light from the server by receiving the first server response.



```
wlog( logtext: "Client connecting...");
// creating threads
for(int i = 0; i < SERVERS; i++){
    struct arg_struct *args = malloc(sizeof(struct arg_struct));  // thread arguments
    args->id = i;
    args->csock = client_socket;
    pthread_create(&thread_group[i],  attr: NULL, server_function, (void *)args);
    usleep( useconds: 1000*500);  // just for visualisation
}
snprintf(logtext, sizeof(logtext),  format: "Created %d server threads.", SERVERS);
wlog(logtext);
// sending the server_message as a response to successful connection of client
send(client_socket, server_message, sizeof(server_message),  flags: 0);
wlog( logtext: "Client joined! Now waiting for requests from client.");
// communication between client and the server
char input[500];
while(keepRunning) {
    // receive requests from the client as input
    strcpy(input,  src: "\0");
    int n = 1;  // for the case that client exits by ctrl + c
    n = recv(client_socket, input,  n: sizeof(input) - 1,  flags: 0);
    if(n == 0){
        printf( format: "Client exited with ctrl + c command!\n");
        break;
    } else if(n < 0){
        fprintf(stderr,  format: "Client exited with error number: %d\n", errno);
    }
    input[n] = '\0';
    printf( format: "Received request %s from client.\n", input);
    snprintf(logtext, sizeof(logtext),  format: "Received request %s from client.", input);
    wlog(logtext);
    if(strcmp(input,"e") == 0) {
        printf( format: "\n\nClient exited the bank!\n\n");
        wlog( logtext: "Client exited the bank!");
        keepRunning = 0;  //this also ends threads

    }else if(strcmp(input,"b") == 0){
        wlog( logtext: "Client initiated query for bank balance.");
        for (int i = 0; i < SERVERS; i++){
            overtake(server_queues[i], input);  // places balance query as first pending request to all servers
        }
        wlog( logtext: "Waiting for servers to reply to master query!\n");
        pthread_barrier_wait(&barrier);
        wlog( logtext: "All threads answered to master query!\n");
        float bank_balance = 0;
        for (int i = 0; bank.accounts[i] != NULL; i++) {
            bank_balance += bank.accounts[i]->balance;
        }
        snprintf(server_message, sizeof(server_message),  format: "Bank balance: %f\n", bank_balance);
        printf( format: "Bank balance: %f\n", bank_balance);
        send(client_socket, server_message, sizeof(server_message),  flags: 0);
    }else{
        int min_length; int min_index;
        min(server_queues, &min_length, &min_index);
        pthread_mutex_lock(&mutex);
        snprintf(logtext, sizeof(logtext),  format: "Directing request to server %d.", min_index);
        wlog(logtext);
        if(enqueue(server_queues[min_index], input) == 0){
            fprintf(stderr,  format: "Error with directing the request %s to server!\n", input);
            send(client_socket, server_message, sizeof(server_message),  flags: 0);
        }
        pthread_mutex_unlock(&mutex);
    }
}
```

A big chunk of server side code, where:

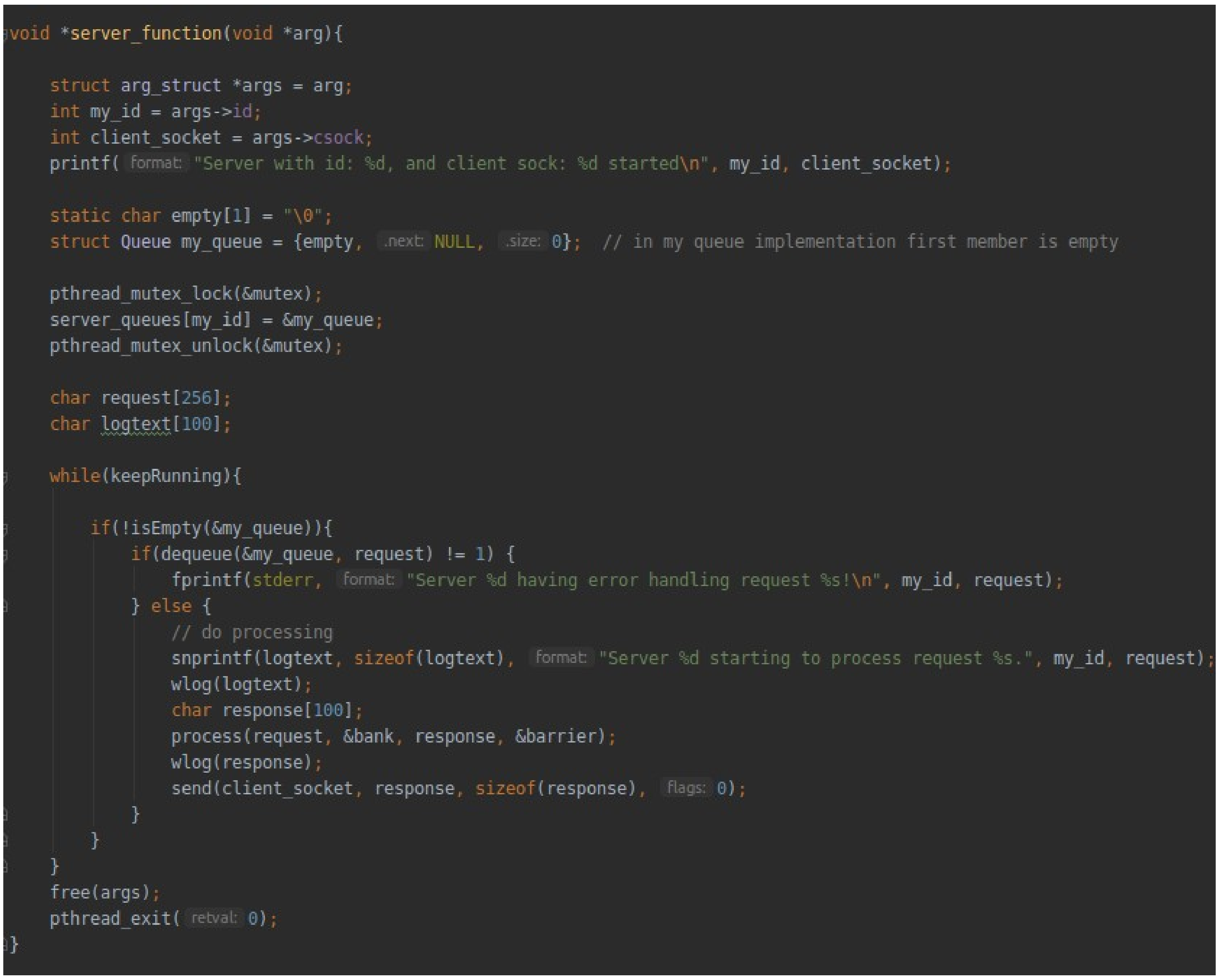
The threads (=servers) are created, with them getting the client socket (for sending back server responses) and server id as arguments. After that, a connection response is send to client program, so it can start sending the requests, and the program proceeds to enter while loop where it receives these requests and acts accordingly:

The special requests, b for bank query and e for exit, are handled differently than the rest of the commands:

Request e just exits the while loop (by changing global variable keepRunning to equal 0 which also ends the threads). Main then waits that the threads also close by pthread_join and frees all the reserved memory and closes the socket.

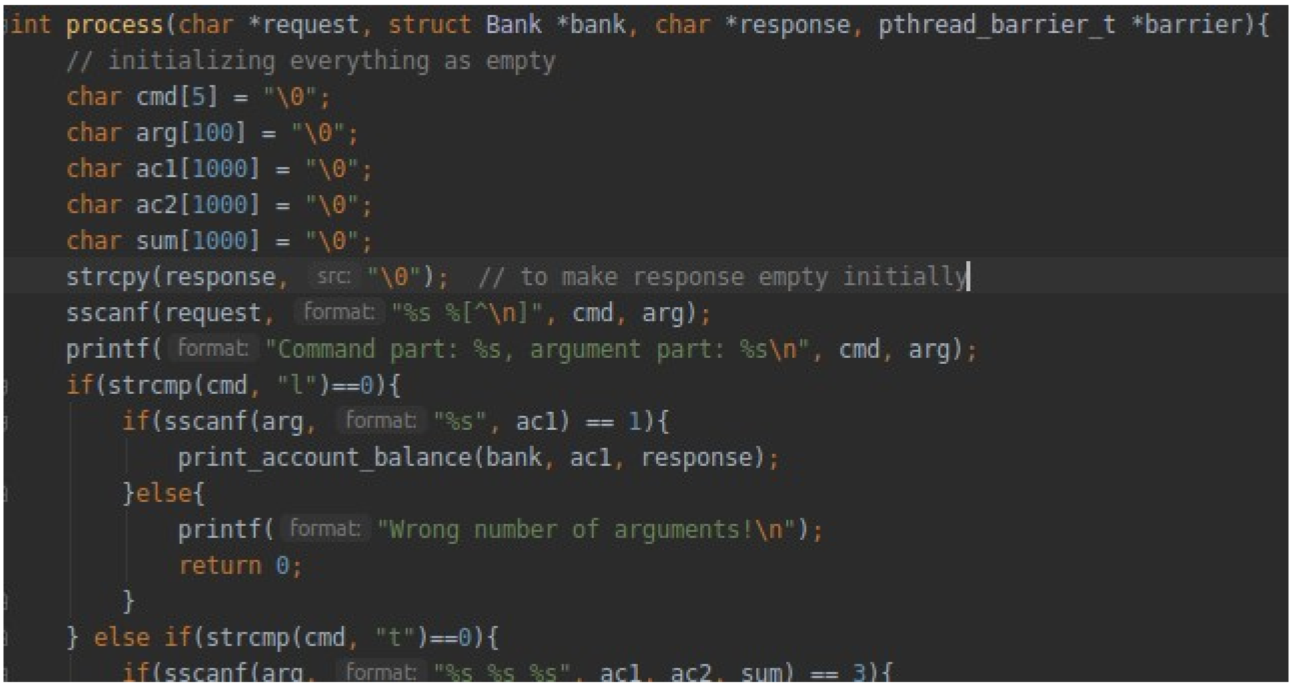
Request b initializes bank balance query and overtakes the queues by placing the request b as first to serve in them. After that it is blocked by the pthread_barrier_wait command, which needs SERVERS + 1 amount of threads to execute the same function call with the same barrier, until anyone of them can continue pass it.

Other request are just placed in the minimum length queue among the server queues (found with the min function call, implemented in the miscellanous code file).



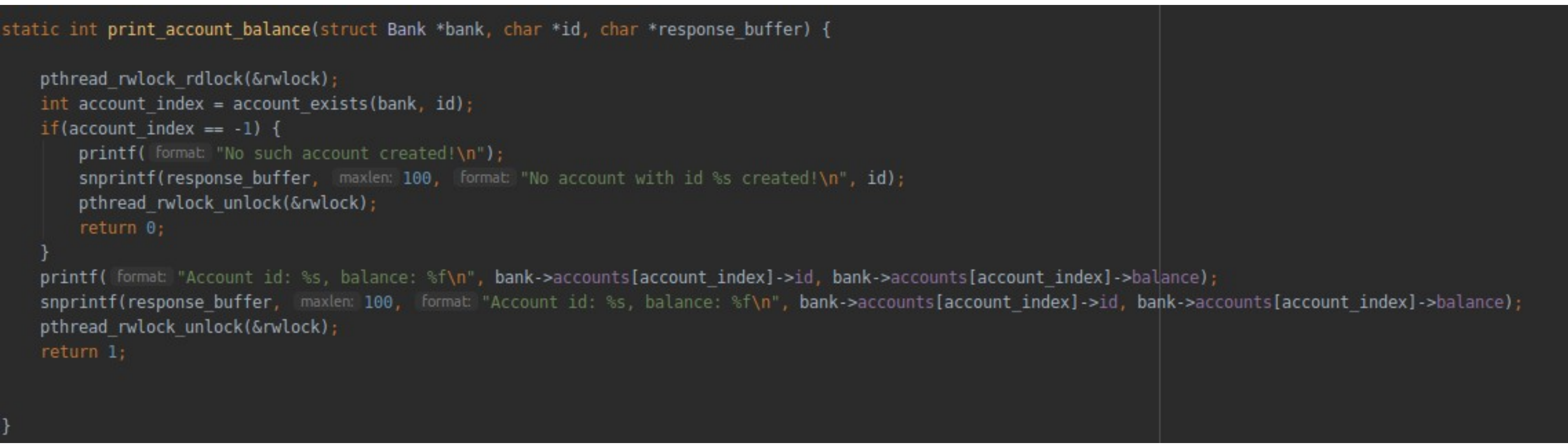
```c
void *server_function(void *arg){

    struct arg_struct *args = arg;
    int my_id = args->id;
    int client_socket = args->csock;
    printf("Server with id: %d, and client sock: %d started\n", my_id, client_socket);

    static char empty[1] = "\0";
    struct Queue my_queue = {empty, NULL, 0};  // in my queue implementation first member is empty

    pthread_mutex_lock(&mutex);
    server_queues[my_id] = &my_queue;
    pthread_mutex_unlock(&mutex);

    char request[256];
    char logtext[100];

    while(keepRunning){

        if(!isEmpty(&my_queue)){
            if(dequeue(&my_queue, request) != 1) {
                fprintf(stderr, "Server %d having error handling request %s!\n", my_id, request);
            } else {
                // do processing
                snprintf(logtext, sizeof(logtext), "Server %d starting to process request %s.", my_id, request);
                wlog(logtext);
                char response[100];
                process(request, &bank, response, &barrier);
                wlog(response);
                send(client_socket, response, sizeof(response), 0);
            }
        }
    }
    free(args);
    pthread_exit(0);
}
```

This is the function that the server threads start to execute, and which I hope is so simple that needs no explanation. The request processing happens with the process function call, which is implemented in a separate server file. It is a lot of code, but the idea is simple: parsing the request string to get the right information out of it (which command is it, which accounts does it operate with…) and then calling the static functions in that file to actually do the work. Rather than going through the whole code I try to demonstrate it with processing the request “l ac1” which should list account 1 details if there exists an account with that id.

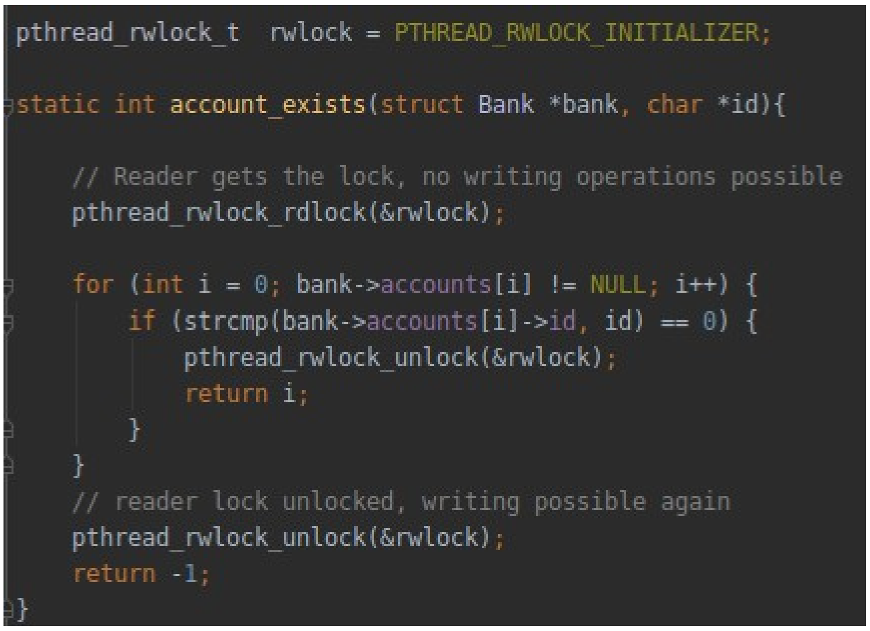


```c
int process(char *request, struct Bank *bank, char *response, pthread_barrier_t *barrier){
    // initializing everything as empty
    char cmd[5] = "\0";
    char arg[100] = "\0";
    char ac1[1000] = "\0";
    char ac2[1000] = "\0";
    char sum[1000] = "\0";
    strcpy(response, src: "\0");  // to make response empty initially
    sscanf(request, format: "%s %[^\n]", cmd, arg);
    printf( format: "Command part: %s, argument part: %s\n", cmd, arg);
    if(strcmp(cmd, "l")==0){
        if(sscanf(arg, format: "%s", ac1) == 1){
            print_account_balance(bank, ac1, response);
        }else{
            printf( format: "Wrong number of arguments!\n");
            return 0;
        }
    } else if(strcmp(cmd, "t")==0){
        if(sscanf(arg, format: "%s %s %s", ac1, ac2, sum) == 3){
```

First the request is parsed to command part and arguments. If the command part is l, we are moved inside the first if block. The right amount of arguments is made sure, with another sscanf command and then we call the static function print_account_balance in the same file.



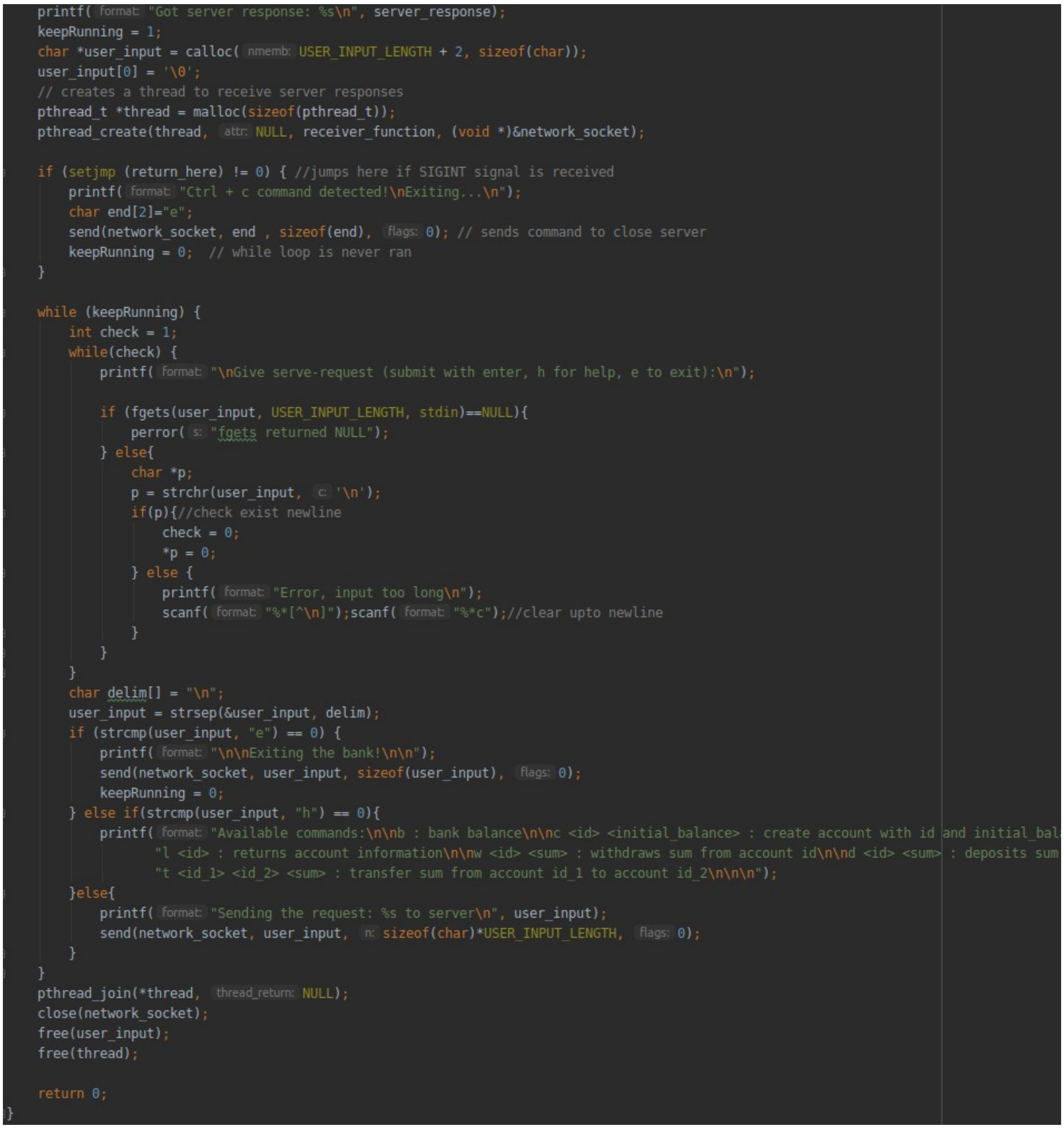
```c
static int print_account_balance(struct Bank *bank, char *id, char *response_buffer) {

    pthread_rwlock_rdlock(&rwlock);
    int account_index = account_exists(bank, id);
    if(account_index == -1) {
        printf( format: "No such account created!\n");
        snprintf(response_buffer, maxlen: 100, format: "No account with id %s created!\n", id);
        pthread_rwlock_unlock(&rwlock);
        return 0;
    }
    printf( format: "Account id: %s, balance: %f\n", bank->accounts[account_index]->id, bank->accounts[account_index]->balance);
    snprintf(response_buffer, maxlen: 100, format: "Account id: %s, balance: %f\n", bank->accounts[account_index]->id, bank->accounts[account_index]->balance);
    pthread_rwlock_unlock(&rwlock);
    return 1;


}
```



```c
pthread_rwlock_t  rwlock = PTHREAD_RWLOCK_INITIALIZER;

static int account_exists(struct Bank *bank, char *id){

    // Reader gets the lock, no writing operations possible
    pthread_rwlock_rdlock(&rwlock);

    for (int i = 0; bank->accounts[i] != NULL; i++) {
        if (strcmp(bank->accounts[i]->id, id) == 0) {
            pthread_rwlock_unlock(&rwlock);
            return i;
        }
    }
    // reader lock unlocked, writing possible again
    pthread_rwlock_unlock(&rwlock);
    return -1;
}
```

Remark on the code above:

Here we can see my implementation of the R/W-locks that are supposed to protect account data from getting corrupted by threads trying to read and write it at the same time. Rwlock presented here is a reader lock rdlock, which gives access to following code only if the rwlock hasn’t been already taken by writer lock wrlock (which can be found for example in create_account function in the same file).



```
    printf("Got server response: %s\n", server_response);
    keepRunning = 1;
    char *user_input = calloc(USER_INPUT_LENGTH + 2, sizeof(char));
    user_input[0] = '\0';
    // creates a thread to receive server responses
    pthread_t *thread = malloc(sizeof(pthread_t));
    pthread_create(thread, NULL, receiver_function, (void *)&network_socket);

    if (setjmp (return_here) != 0) { //jumps here if SIGINT signal is received
        printf("Ctrl + c command detected!\nExiting...\n");
        char end[2]="e";
        send(network_socket, end , sizeof(end), 0); // sends command to close server
        keepRunning = 0;  // while loop is never ran
    }

    while (keepRunning) {
        int check = 1;
        while(check) {
            printf("\nGive serve-request (submit with enter, h for help, e to exit):\n");

            if (fgets(user_input, USER_INPUT_LENGTH, stdin)==NULL){
                perror("fgets returned NULL");
            } else{
                char *p;
                p = strchr(user_input, '\n');
                if(p){//check exist newline
                    check = 0;
                    *p = 0;
                } else {
                    printf("Error, input too long\n");
                    scanf("%*[^\n]");scanf("%*c");//clear upto newline
                }
            }
        }
        char delim[] = "\n";
        user_input = strsep(&user_input, delim);
        if (strcmp(user_input, "e") == 0) {
            printf("\n\nExiting the bank!\n\n");
            send(network_socket, user_input, sizeof(user_input), 0);
            keepRunning = 0;
        } else if(strcmp(user_input, "h") == 0){
            printf("Available commands:\n\nb : bank balance\n\nc <id> <initial_balance> : create account with id and initial_bal
                   "l <id> : returns account information\n\nw <id> <sum> : withdraws sum from account id\n\nd <id> <sum> : deposits sum
                   "t <id_1> <id_2> <sum> : transfer sum from account id_1 to account id_2\n\n\n");
        }else{
            printf("Sending the request: %s to server\n", user_input);
            send(network_socket, user_input, sizeof(char)*USER_INPUT_LENGTH, 0);
        }
    }
    pthread_join(*thread, NULL);
    close(network_socket);
    free(user_input);
    free(thread);

    return 0;
}
```

Now that we understand the basic structure of the server, lets continue in the client program, which continues after receiving the connection affirmation response from the server.

It creates a second thread (main is the first) to receive the server responses to requests. The receiver thread function basically just: 1. creates a text file to store received server responses, 2. opens a new terminal, again with the same kind of system call as in the beginning, and instead of starting a executable file, it starts to tail the created file, which makes the output in the terminal update every time the received responses are stored in the file.

After this there is defined the block where the program execution jumps if user presses CTRL + C. As we can see, if the execution goes inside the if block, the while loop after it is never ran.

Normally, however, we move inside the while loop where user inputs are read and send through socket to server program.

User inputs are read with fgets which is a safe way to read user inputs compared to scanf which is vulnerable for stack overflow with too long input. User can test to send too long input (>100 characters) and see that my program just sends error as a response.

The requests are send through the socket to the server side. Request e also exits the while loop (and thus ends the program) after sending the exit message to the server side.

After exiting the while loop, the program waits for the receiver thread also to finish with the pthread_join command and the does the necessary closing of the socket and frees' of memory before returning 0 and closing the program.