Name: Noha Ahmed Ahmed Morsy
ID: 18012515

Socket Programming

Assumptions:

- There is a master socket waiting for incoming connections.
- There can be at most 3 connections waiting for being accepted.
- When a connection is accepted the connection handling is delegated to a new thread. .
- The Client parse requests existing in commands.txt file in the Client directory.
- An OK Message is sent by the server to the client after the handling of each request informing the client to send the successive request.
- At the end of the connection the client sends a CLOSE message then the server closes the connection.
- In case of GET request if the file is NOT FOUND the request is neglected and the next requests are handled.
- The buffer size in the Client and Server is 1024
- Port Number of the master socket is entered as an argument when running the server and client programs.

Approaches Chosen:

- **Multithreading / Multiprocessing**: I implemented the multithreading approach because:
    - Threads are lightweight which take less overhead in creating and termination.
    - Threads can share resources and memory in the same program easily.
    - Threads can be synchronized easily using Mutexes.
- **Time-out:** a timeout is set to each connection opened equals to 10 seconds divided by the number of the opened connections.

How to run:

- Server:
    - Open terminal in Server folder.
    - Compile -> gcc server.c -o server -lpthread
    - If port is used -> fuser -n tcp -k [port-number]
    - Run -> ./server [port-number].
- Client:
    - Open terminal in Client folder.
    - Compile -> gcc client.c -o client
    - Run -> ./client [ip-address] [port-number]

Program Flow:

Server

Functions:

```
void createMasterSocket();
int acceptConnection();
void closeConnection(int sd);
void sendMessage(int sd, char *msg);
void parseMessage(char *msg, char *type, char *path);
bool isCloseMessage(char *msg);
void receiveFile(char *path, int sd);
void sendFile(char *path, int sd);
void handlePostRequest(char *path, int sd);
void *handle_connection(void *);
```

Data Structures:

- Strings (Array of char) used as a buffer.
- struct sockaddr_in used to store ip protocol , ip address and port number of the server.
- Mutex used in forcing mutual exclusion on modifying the global variable of the active sockets counter.
- struct timeval used to store timeout value.
- Fd_set used to store active sockets to keep track of the timeout.

Main:

```
int main(int argc, char const *argv[])
{
    // storing IP and Port of the server socket
    PORT = atoi(argv[1]);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    addrlen = sizeof(address);
    FD_ZERO(&fds);

    createMasterSocket();

    while (1)
    {
        // accepting connections
        int new_socket = acceptConnection();

        // increment counter of used sockets
        pthread_mutex_lock(&lock);
        FD_SET(new_socket, &fds);
        counter++;
        pthread_mutex_unlock(&lock);

        // delegate connection handling to a new thread
        pthread_t thread_id;
        int *pclient = malloc(sizeof(int));
        *pclient = new_socket;
        pthread_create(&thread_id, NULL, handle_connection, pclient);
}

return 0;
}
```

Connection Handling:

```
void *handle_connection(void *socket_desc)
{
```

```
pthread_mutex_lock(&lock);
timeout.tv_sec = 10 / counter;
pthread_mutex_unlock(&lock);
// Get the socket descriptor
int sd = *((int *)socket_desc);
int read_size;
char client_message[BUFFER_SIZE], *type = malloc(256), *path = malloc(256);
while (1)
{
    int t = select(FD_SETSIZE, &fds, NULL, NULL, &timeout);
    if (t == 0)
    {
        printf("\n[+] Timeout\n");
        closeConnection(sd);
        return 0;
    }
    read_size = read(sd, client_message, BUFFER_SIZE);
    printf("\n[+] Client %d : %s \n", sd, client_message);
    if (isCloseMessage(client_message))
        closeConnection(sd);
        break;

    parseMessage(client_message, type, path);
    if (strcmp(type, "POST") == 0)
        // printf("IN POST \n");
        handlePostRequest(path, sd);
    else if (strcmp(type, "GET") == 0)
        //  printf("IN GET \n");
        handleGetRequest(path, sd);
}

pthread_mutex_lock(&lock);
counter--;
timeout.tv_sec = 3 / counter;
pthread_mutex_unlock(&lock);
return 0;}
```

Client

Functions:

```
void createRequest(char *req, char *type, char *path, char *header_lines, char *body);
void initiateConnection();
void readServerMessage();
void parseCommand(char *command, char *request, char *type, char *path);
bool isOK();
void sendFile(char *path);
void receiveFile(char *path);
void sendGetRequest(char* request, char *path);
void sendPostRequest(char* request, char *path);
void sendRequests();
```

Data Structures:

- Strings (Array of char) used as a buffer.
- struct sockaddr_in used to store ip protocol , ip address and port number of the server.

Main:

```
int main(int argc, char const *argv[])
{
    PORT = atoi(argv[2]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    initiateConnection();
    sendRequests();

    return 0;
}
```

Sending Requests:

```
void sendRequests()
{
    // reading commands from file
     fp = fopen("commands.txt", "r");

    while ((r = getline(&line, &len, fp)) != -1)
    {
        parseCommand(line, request, type, path);
        if (strcmp(type, "POST") == 0)
            sendPostRequest(request , path);

        else if (strcmp(type, "GET") == 0)
            sendGetRequest(request , path);
        }
    }
    createRequest(request, "CLOSE", "", "", "");
    send(server_socket, request, strlen(request), 0);

    fclose(fp);}
```

GET Request:

```
void sendGetRequest(char* request, char *path )
{
    send(server_socket, request, strlen(request), 0);
    readServerMessage();
    if( isOK() ){
        receiveFile(path);
    }
}
```

POST Request:

```
void sendPostRequest(char* request, char *path)
{
    send(server_socket, request, strlen(request), 0);
    readServerMessage();
    if( isOK() ){
        sendFile(path);
        readServerMessage();
        if( !isOK() ){
            sendPostRequest(request,path);
        }
    }
}
```