CS232 Operating Systems
Assignment 04

Faraz Ahmed Khan (fk03983), Syed Ammar Ahmed (sa04050)

Fall 2019

1 Client:

Code from client.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <assert.h>
#define BUF_SIZE 4096


int RUNNING;

void * input(void * ptr)
{
    /*
    keeps taking input from the user
    */

    int sock = *((int *) ptr);
    char *buffer;
    size_t bufsize = 32;
    buffer = (char *)malloc(bufsize * sizeof(char));

    while(RUNNING)
    {
        //send the request
        if (getline(&buffer,&bufsize,stdin) != 0)
        {
```

```
            if(write(sock,buffer,strlen(buffer)) < 0) //sendint to clieng
            {
                perror("send");
            }
        }
    }
    printf("Exiting thread\n");
    free(buffer);
}

int main(int argc, char * argv[]){

    if (argc != 4)
    {
        printf("Incorrect format\n");
        return 0;
    }
    char * hostname= argv[1];
    short port= atoi(argv[2]);                  //the port we are connecting on

    struct addrinfo *result;        //to store results
    struct addrinfo hints;          //to indicate information we want

    struct sockaddr_in *saddr_in;  //socket interent address
    pthread_t thread;
    int s,n, i;                        //for error checking

    int sock;                       //socket file descriptor

    char * request= argv[3]; //the GET request

    char response[4096];            //read in 4096 byte chunks
    memset(&hints,0,sizeof(struct addrinfo));  //zero out hints
    hints.ai_family = AF_INET; //we only want IPv4 addresses

    //Convert the hostname to an address
    if( (s = getaddrinfo(hostname, NULL, &hints, &result)) != 0){
        fprintf(stderr, "getaddrinfo: %s\n",gai_strerror(s));
        exit(1);
    }
        //convert generic socket address to inet socket address
    saddr_in = (struct sockaddr_in *) result->ai_addr;

    //set the port in network byte order
    saddr_in->sin_port = htons(port);

    //open a socket
    if( (sock = socket(AF_INET, SOCK_STREAM, 0))  < 0){
        perror("socket");
        exit(1);
    }

    //connect to the server
    if(connect(sock, (struct sockaddr *) saddr_in, sizeof(*saddr_in)) < 0){
        perror("connect");
```

```
        exit(1);
    }
    if(write(sock,request,strlen(request)) < 0){ //sending name first
            perror("send");
    }
    n  = -1;
    while (n < 0) //waiting for the client to read the name
    {
        n = read(sock, response, BUF_SIZE-1);
    }
    // read(sock, smth)
    pthread_create(&thread, NULL, input, &sock); //creating thread that takes inp
    RUNNING = 1;

    while (RUNNING) //to keep reading from server
    {

        memset(response, 0, BUF_SIZE);
        response[0] = '\n';
        n = read(sock, response, BUF_SIZE-1);

        if(n <= 0)
        { //closed or error on socket

            printf("Client␣Closed␣With␣socket␣%d\n:", sock);
            RUNNING = 0;
            return 0;

        }
        else
        { //server sent a message
            printf("%s", response);
        }



    }

}
```

2 Server:

Code from server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
```

```
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <assert.h>

#define BUF_SIZE 4096


//code for lock taken from the book.
struct node {
  int FD;
  char * name;
  in_port_t port;
  struct node * next;
};

typedef struct _rwlock_t {
    sem_t writelock;
    sem_t lock;
    int readers;
} rwlock_t;

typedef struct node Node;

Node * head = NULL;
rwlock_t mutex;


void rwlock_init(rwlock_t *lock) {
    lock->readers = 0;
    sem_init(&lock->lock, 0,1);
    sem_init(&lock->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *lock) {

    sem_wait(&lock->lock);
    lock->readers++;
    if (lock->readers == 1)
    sem_wait(&lock->writelock);
    sem_post(&lock->lock);

    return;
}

void rwlock_release_readlock(rwlock_t *lock) {
    sem_wait(&lock->lock);
    lock->readers--;
    if (lock->readers == 0)
    sem_post(&lock->writelock);
    sem_post(&lock->lock);
    return;
}
```

```
void rwlock_acquire_writelock(rwlock_t *lock) {
    sem_wait(&lock->writelock);
    return;
}

void rwlock_release_writelock(rwlock_t *lock) {
    sem_post(&lock->writelock);
    return;
}


void add(Node * new)
{
    rwlock_acquire_writelock(&mutex);
    Node* temp = head;
    if (head == NULL)
    {
        head = new;
        new->next = NULL;
    }
    else
    {
        new->next = temp;
        new->next->next = temp->next;
        head = new;
    }

    rwlock_release_writelock(&mutex);
}




int removeFD(int FD)
{
    /*
    removes client having socket = FD
    */
    rwlock_acquire_writelock(&mutex);
    Node * temp = head;
    if (temp == NULL)
    {
        return 0;
    }
    if (head -> FD == FD)
    {
        head = head->next;
        if (temp->name != NULL)
        {
            free(temp->name);
        }
        free(temp);
        rwlock_release_writelock(&mutex);
        return FD;
```

```
    }

    Node * temp2 = head->next;
    while (temp2 != NULL && temp2->FD != FD)
    {
        temp = temp2;
        temp2 = temp2->next;
    }

    if (temp2 != NULL && temp2->FD == FD)
    {
        temp->next = temp2->next;
        if (temp2->name != NULL)
        {
            free(temp2->name);
        }
        free(temp2);
        rwlock_release_writelock(&mutex);
        return FD;
    }
    rwlock_release_writelock(&mutex);
    return 0;
}
void write_to_client(char * msg,int client)
{
    /* send msg to client having socket = client
    */
    if (write(client, msg,strlen(msg)) < 0)
    {
        perror("send");
    }

}

void list_connections(int client)
{

    /*
    lists all the connections and send it back to the user
    */
    char buffer[BUF_SIZE];
    memset(buffer, 0, BUF_SIZE);
    rwlock_acquire_readlock(&mutex);
    Node * temp = head;
    while (temp != NULL)
    {
        strcat((char *)&buffer, "CONNECTION␣NAME:␣");
        strcat((char *)&buffer, temp->name);
        strcat((char *)&buffer, "\n");
        temp = temp->next;
    }
    rwlock_release_readlock(&mutex);
    write_to_client((char *)&buffer, client);
    return;
```

```
}

void send_msg(int sender, char * recv, char * message)
{
    /* sends  messeage from client having socket ID sender to a client having nam
    */
    int recievr = 0;
    printf("sending␣message␣to␣%s\n", recv);
    char buffer[BUF_SIZE];
    memset(buffer, 0, BUF_SIZE);
    rwlock_acquire_readlock(&mutex);
    Node * temp = head;
    while (temp != NULL) //extracting the socket from the client name.
    {
        if (strcmp(temp->name, recv) == 0) //client with name found.
        {
            recievr = temp->FD;
        }
        temp = temp->next;
    }
    rwlock_release_readlock(&mutex);
    if (recievr == 0) //failture
    {
        sprintf(buffer, "Couldnt␣find␣client␣%s␣\n", recv);
        write_to_client((char *)&buffer, sender);
        return;
    }
    printf("connection␣%s,␣FD:␣%d␣\n", recv, recievr);
    sprintf(buffer, "MESSAGE␣RECIEVED:␣%s:␣%s", recv, message);
    write_to_client((char *)&buffer, recievr);
}

int quit_connection(int client)
{
    close(client);
    removeFD(client); //removing from linkedlist
    printf("Client␣Closed␣With␣socket␣%d\n", client);
    pthread_exit(NULL); //exiting the thread
}

int execute_command(char * command, int client)
{
    /*
    takes the command which is either /list, /msg, /quit,
    returns 1 if command is structued else return 0.
    */
    char * command1 = "/list\n";
    char * command2 = "/msg";
    char * command3 = "/quit";

    if (strcmp(command, command1) == 0)
    {
        list_connections(client);
        return 1;
    }
```

```
    else if (strncmp(command, command2, strlen(command2)) == 0)
    {
        char recv[BUF_SIZE];
        char msg[BUF_SIZE];
        int chars;
        int n = sscanf(command, "/msg␣%s␣%n", recv, &chars);
        strcpy(msg,command + chars);
        if (n > 0)
            send_msg(client, recv, msg);
        return 1;
    }
    else if (strncmp(command, command3, strlen(command3)) == 0)
    {
        quit_connection(client);
        return 1;
    }
    return 0;
}


void * connection(void * ptr)
{
    Node * client = (Node *) ptr;
    int client_socket = client->FD; //the socket to keep checking
    char response[BUF_SIZE];
    int n; //to hold number of characters read
    while (1)
    {

        n = read(client_socket, response, BUF_SIZE-1);

        if(n <= 0)
        { //closed or error on socket
            quit_connection(client_socket);    //close client sockt
            return 0;
        }
        else
        { //client sent a message

            response[n] = '\0'; //NULL terminate

            //echo messget to client
            if (client->name == NULL) //not specified the name yet
            {
                rwlock_acquire_readlock(&mutex);
                Node * temp = head;
                while (temp != NULL)
                {
                    if (strcmp(temp->name, response) == 0) //client was a same na
                    {
                        write_to_client("ERROR:␣Client␣already␣exists␣with␣a␣same
                        rwlock_release_readlock(&mutex);
                        quit_connection(client_socket);
                        return 0;
                    }
```

```
                    temp = temp->next;
                }
                rwlock_release_readlock(&mutex);
                client->name = (char * )malloc(strlen(response));
                strcpy(client->name, response); //store the clients name in the r
                printf("connection: %s, FD: %d \n", client->name, client->FD);
                write_to_client("Accepted", client_socket);
                add(client); //adding to the linkedlist
            }
            else
            {
                //the client sent a command
                printf("Client %s, sent a command %s", client->name, response);
                int status = execute_command((char *)&response, client_socket);
                if (status == 0)
                {
                    char buffer[BUF_SIZE];
                    memset(buffer, 0, BUF_SIZE);
                    sprintf(buffer, "ERROR: THE SERVER DOES NOT UNDERSTAND %s \n'
                    printf("Recieved from client: %s", response);
                    write_to_client((char *)&buffer, client_socket);
                }
            }
        }
    }
}

int main(int argc, char * argv[]){

    char hostname[]="127.0.0.1";   //localhost ip address to bind to
    if (argc == 1)
    {
        printf("Port number not passed\n");
        return 0;
    }
    short port=atoi(argv[1]);
    struct sockaddr_in saddr_in;  //socket interent address of server
    struct sockaddr_in client_saddr_in;  //socket interent address of client
    pthread_t thread;


    socklen_t saddr_len = sizeof(struct sockaddr_in); //length of address

    int server_sock, client_sock;         //socket file descriptor
    char response[BUF_SIZE];           //what to send to the client

    rwlock_init(&mutex);
    //set up the address information
    saddr_in.sin_family = AF_INET;
    inet_aton(hostname, &saddr_in.sin_addr);
    saddr_in.sin_port = htons(port);

    //open a socket
    if( (server_sock = socket(AF_INET, SOCK_STREAM, 0))  < 0){
        perror("socket");
```

```
        exit(1);
    }

    //bind the socket
    if(bind(server_sock, (struct sockaddr *) &saddr_in, saddr_len) < 0){
        perror("bind");
        exit(1);
    }

    if(listen(server_sock, 15)  < 0){
        perror("listen");
        exit(1);
    }

    saddr_len = sizeof(struct sockaddr_in); //length of address
    printf("Listening␣On:␣%s:%d\n", inet_ntoa(saddr_in.sin_addr), ntohs(saddr_in.

    while(1){ //loop
        //update the set of selectable file descriptors
        //accept incoming connections = NON BLOCKING
        client_sock = accept(server_sock, (struct sockaddr *) &client_saddr_in, &
        if (client_sock < 0) //failure
        {
            printf("Error␣Connection\n");
            continue;
        }

        printf("Connection␣From:␣%s:%d␣(%d)\n", inet_ntoa(client_saddr_in.sin_add
                ntohs(client_saddr_in.sin_port), client_sock);

        Node * new_connection = (Node *) malloc(sizeof(Node));
        new_connection->FD = client_sock;
        new_connection->port = client_saddr_in.sin_port;
        new_connection->name = NULL;
        if (pthread_create(&thread, NULL, connection, new_connection) != 0)//crea
        {
            printf("The␣server␣could␣not␣accomodate␣this␣connection\n");
            free(new_connection);
        }

    }

}
```

3 Make file

Code from Make file

```
all:
        gcc  -o server gp04_server.c -pthread
        gcc -o client gp04_client.c  -pthread
```

4 Comments

The assignment was not very lengthy if we had followed the tutorial, but I did not enjoy this assignment as much as I enjoyed the other assignments.

References

[1] Code taken from the book

[2] Lec 26: Socket Addressing and Client Socket Programming

[3] Lec 27: Server Sockets