CSE 344 FINAL

BIBAKBOX

FILE SHARING SOFTWARE SYSTEM

Çağrı Çaycı

1901042629

OVERVIEW

The main goal of this project is to provide file sharing system through the network by socket connections. A file sharing system allows users to distribute or exchange files. It enables individuals or groups to share various types of files such as documents, images, videos, software programs, and more. The most critical part of the file sharing system is ensuring this system simultaneously and continuously until the client leaves.

To successfully implement a file sharing system in the C language, there are several key concepts that must be known. These are socket connections, synchronization mechanisms and threads.

WORKING PRINCIPLES

At the start of the program, the server establishes a socket connection and enters a blocking state, waiting for a client to connect. Once a client successfully connects to the server, the server initiates the process of copying all the files from its own folder to the client's folder. Following the copying operation, synchronization mechanisms come into play. Their primary responsibility is to ensure consistency between the server's folder and all the connected clients' folders. To achieve this, any changes made in the clients' folders or the server's folder must be promptly notified to the relevant parties. To implement these synchronization mechanisms, a dedicated thread is utilized in both the server and the clients. This thread constantly monitors the corresponding folders for any modifications. Whenever a change is detected, such as a new file being added, an existing file being modified, or a file being deleted, the thread notifies the relevant clients or the server. By employing this constant monitoring and notification mechanism, the file sharing system maintains synchronization between the server and the clients' folders, ensuring that any changes made in either location are promptly reflected in the other. This approach guarantees that all connected clients have access to the latest files and that the server remains updated with any modifications made by the clients.

CONCEPTS

SOCKET CONNECTIONS

Firstly, a socket is created with IPv4 address family and TCP type. Then, the socket is set to accept connections on any available network interface, also port number is assigned to server. Socket is associated with specific IP address and port number by bind function. Socket is marked as a listening socket by listen function. At the end, the socket is set to accept an incoming connection.

```
struct sockaddr_in serverAddress;

serverfd = socket(AF_INET, SOCK_STREAM, 0);
if(serverfd == -1){
    errExit("socket");
}

serverAddress.sin_addr.s_addr = INADDR_ANY;
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(portNumber);

if(bind(serverfd, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == -1){
    errExit("bind");
}

if(listen(serverfd, threadPoolSize) == -1){
    errExit("listen");
}

while(1){
    struct sockaddr_in clientAddress;
    socklen_t clientAddressSize = sizeof(clientAddress);
    int clientfd = accept(serverfd, (struct sockaddr*) &clientAddress, &clientAddressSize);
    if(clientfd == -1){
        errExit("connect");
    }

    printf("A CLIENT IS CONNECTED TO THE SERVER!\n");
```

(CREATING SOCKET IN SERVER CODE)

Firstly, a socket is created with IPv4 address family and TCP type. IP address is converted to binary by inet_pton method. At the end, connection is established by connect method.

```
struct sockaddr_in serverAddress;

clientfd = socket(AF_INET, SOCK_STREAM, 0);
if(clientfd == -1){
    errExit("socket");
}

serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(portNumber);
if(inet_pton(AF_INET, "127.0.0.1", &serverAddress.sin_addr) == -1){
    errExit("inet_pton");
}

if(connect(clientfd, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == -1){
    errExit("connect");
}

printf("THE CLIENT IS CONNECTED TO THE SERVER!\n");
```

(CONNECTING SOCKET IN CLIENT CODE)

PTHREADS

In this project, a thread pool is created so that each thread in the thread pool deals with only one client until the client leaves. This is provided with the following structures.

```
while(1){
    struct sockaddr_in clientAddress;
    socklen_t clientAddressSize = sizeof(clientAddress);
    int clientfd = accept(serverfd, (struct sockaddr*) &clientAddress, &clientAddressSize);
    if(clientfd == -1){
        errExit("connect");
    }

    printf("A CLIENT IS CONNECTED TO THE SERVER!\n");

    pthread_mutex_lock(&mutex);

    int * temp = malloc((number_of_clients + 1) * sizeof(int));

    for(int i = 0; i < number_of_clients; i++){
        temp[i] = clients[i];
    }
    temp[number_of_clients] = clientfd;

    free(clients);

    clients = temp;

    number_of_clients++;

    pthread_mutex_unlock(&mutex);

    pthread_cond_signal(&condition);
}
```

(Client socket file descriptor is put an array.)

```
while(1){
    pthread_mutex_lock(&mutex);
    while(number_of_clients == 0){
        pthread_cond_wait(&condition, &mutex);
    }

    int clientfd = clients[0];
    int * temp = malloc((number_of_clients - 1) * sizeof(int));
    for(int i = 0; i < number_of_clients - 1; i++){
        temp[i] = clients[i+1];
    }

    number_of_clients--;
    free(clients);
    clients = temp;

    pthread_mutex_unlock(&mutex);
```

(First client in the array is get and it removed from the queue.)

SYCNHRONIZATION MECHANISMS

As mentioned in the working principles section, there are unique threads for clients and server. These threads check the corresponding folders constantly and notify the changes to corresponding clients or server.

```
void * handleServer(){
    while(1){
        check_files(directory);
        for(int i = 0; i < number_of_files; i++){
            if(files[i].isChecked == 0){
                printf("%s is deleted.\n", files[i].filename);

                char message[1024];
                if(files[i].isDirectory){
                    sprintf(message, "DELETEDIR#0000%s", &files[i].filename[strlen(directory) + 1]);
                }else
                    sprintf(message, "DELETEFILE#0000%s", &files[i].filename[strlen(directory) + 1]);

                for(int i = 0; i < threadPoolSize; i++){
                    if(clientfds[i] != 0){
                        write(clientfds[i], message, strlen(message));

                        sem_wait(semaphores[i]);
                    }
                }

                removeFileFromFiles(i);
            }
            else
                files[i].isChecked = 0;
        }
        sleep(2);
    }
    pthread_exit(NULL);
}
```

(check_files method checks all the files in the server folder with first version of it, and if there is any updated/added file/folder, it does what necessary to be done such as updating older version, sending necessary information to clients. After the check_files function completes its task, the files/folders which are deleted are checked.)

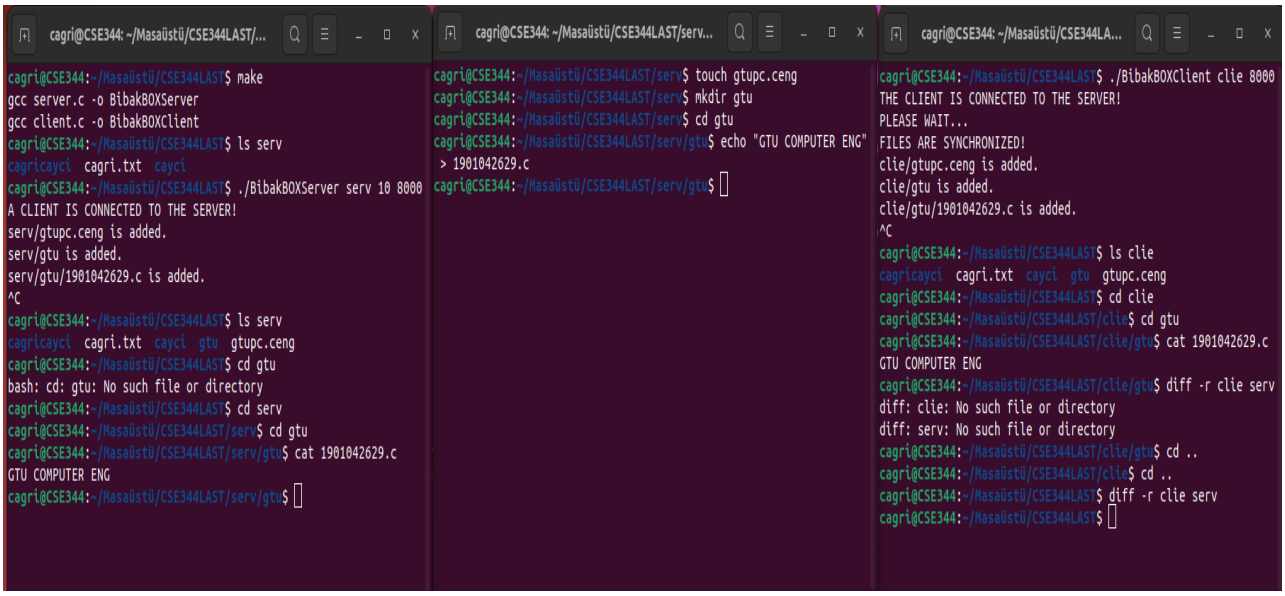
SENDING FILES OVER SOCKET

When sending a file over a socket, two delimiters are needed to handle received string. Firstly “FILE#000” and path of the file is sent. Receiving site of the socket compared the filename with the filename which sent previously. If the file is different from the previous file, a new file pointer is opened and writes the file content from the beginning. If the filename is same as the previous file, the content is continuous to be written to the same file. Also, a semaphore is used to provide synchronization while sending and reading file content.

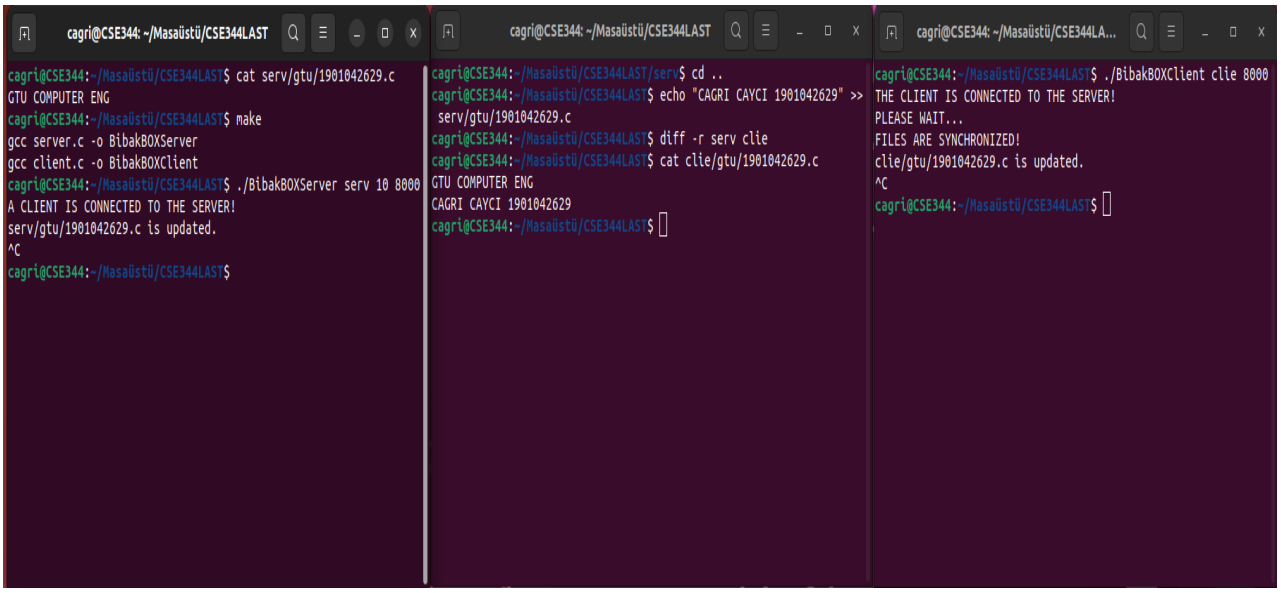
```c
void send_file(int clientfd, char * filename, sem_t * semaphore){
    FILE * file = fopen(filename, "r");
    if(file == NULL){
        errExit("fopen");
    }
    char message[10240], buffer[9600];

    int bytesRead;

    sprintf(message, "FILE#0000%s", &filename[strlen(dirname) + 1]);

    write(clientfd, message, sizeof(message));

    memset(message, 0, sizeof(message));

    sem_wait(semaphore);

    while((bytesRead = fread(buffer, sizeof(char), sizeof(buffer), file)) > 0){
        sprintf(message, "FILE#0000%s %.*s", &filename[strlen(dirname) + 1], bytesRead, buffer);

        write(clientfd, message, strlen(message));

        memset(message, 0, sizeof(message));

        sem_wait(semaphore);
    }

    sprintf(message, "FILE#9999");

    write(clientfd, message, sizeof(message));

    sem_wait(semaphore);

    fclose(file);
}
```

```c
void readSocket(int clientfd, sem_t * semaphore){
    int bytesRead;

    char buffer[10240];

    char prevPath[512] = "", currentPath[512] = "";

    FILE * file = NULL;

    while((bytesRead = read(clientfd, buffer, sizeof(buffer))) > 0){
        if(strncmp(buffer, "DELETEFILE#0000", 15) == 0){
            char filename[512];

            sprintf(filename, "%s/%.*s", directory, 496, &buffer[15]);

            remove(filename);

        }else if(strncmp(buffer, "DELETEDIR#0000", 14) == 0){
            printf("REMOVING FOLDERS ARE NOT CURRENTLY SUPPORTED!\n");
        }else if(strncmp(buffer, "DIRECTORY#0000", 14) == 0 && strcmp(&buffer[14], "") != 0){
            char path[512];
            sprintf(path, "%s/%.*s", directory, 350, &buffer[14]);
            mkdir(path, 0777);
        }else if(strncmp(buffer, "FILE#0000", 9) == 0){
            int length = 0;
            for(int i = 0; i < 10240; i++){
                if(buffer[i] == ' '){
                    length = i;
                    break;
                }
            }

            sprintf(currentPath, "%s/%.*s", directory, length - 9, &buffer[9]);

            if(strcmp(prevPath, currentPath) != 0){
                file = fopen(currentPath, "w");
                if(file == NULL){
                    errExit("fopen");
                }

                strcpy(prevPath, currentPath);
            }

            fwrite(&buffer[length + 1], sizeof(char), strlen(buffer) - length - 1, file);

            fflush(file);

        }else if(strncmp(buffer, "FILE#9999", 9) == 0){
            fclose(file);

            memset(prevPath, 0, sizeof(prevPath));

        }

        memset(buffer, 0, sizeof(buffer));

        sem_post(semaphore);
    }
}
```

TESTS



(ADDING FILE AND DIRECTORY)



(UPDATING FILE)

For some simultaneous tests;
https://clipchamp.com/watch/OXUBMKAX3QW

ADDITIONAL EXPLANATIONS

My program does not create log files because it gives information to the terminal already.

My program does not support removing directories, it is a complex task to handle.

My program does not take IP address as parameter like pdf says. The purpose of “server and client are running machine” state is to avoid sending path instead of files’ content. So, my program already sends files’ content.

My program is very strict about command line arguments. So, be loyal to the format.

./BibakBOXServer foldername_which_must_be_exist_in_current_directory thread_pool_size port_number

./BibakBOXClient foldername port_number