2021

16th January

LAB 12

OPERATING SYSTEM

BESE-(9A)

Ali Tariq (241916)

Muhammad Abdullah (241548)

Syed Afraz Hasan Kazmi (241847)

File System Simulation

File System (Server-Client + Threading PART 2)

The objective of this lab is to simulate a filesystem based on an existing directory structure given as an input file and allow multiple clients to manipulate the system residing on a server without inconsistencies. Also implement minimal security and restrictions.

The program accepts the following parameters:

- -f [input file for directory structure]
- -s [Disk Size]
- -b [Block/Sector Size]

Run **server** using the format,

```
python3        filename.py    -f        -s        -b
```

Example: python3 filesystemwolf.py list 65536 512

Note: A prerequisite to this program is installing the `llist` module which is library of Doubly and Singly Linked Lists. To do so, open Linux terminal, cd into the program directory and write the following command 'pip3 install llist'. (Needs to be done only once)

The program will use the input parameters and create a file system structure, against which you will be able to run commands.

Run **client** using the format,

```
python3        filename.py
```

Example: python3 fileSysClient.py

Input file (the -f parameter)

The input files are already provided in this repo but a user can make their own input files using the find Unix command. There are two files namely, 'dir_list.txt' and 'file_list.txt'. The directory list file has **absolute** paths of all directories. The file list file has **size** and **absolute** paths of all files

`find ./ -type d > dir_list.txt` - Generates a list of all directories and subdirectories and outputs it into a txt file.

`find ./ -type f -ls > file_list.txt` - Generates a list of all files with their size and outputs into a txt file. (Note: This file may have to be modified to conform to the required format)

Introducing Client-Server Architecture with Security measures

The server-client architecture achieved in previous lab (lab-11) is still kept intact because the functionality remains the same, only security reforms to apply.

The reader-writer problem had already implicitly been catered in lab-11 as it required multiple users to concurrently access the file system and therefore it was made sure that no user can write a file whilst other is reading it(and vice versa)

The server maintains a list of connected users. Every element of this list holds metadata about the connected client such as username, thread number, IP Address of client, its port number and lastly a count for the times client has accessed the system. This was also implemented in lab 11 but now it has been put to good use.

Every user is allowed 5 accesses to the file system, whether it be of any sort. So basically, every thread can call 5 core functions to manipulate with the filesystem until they are restricted by the server, essentially banned. (Server initiates closing procedure)

Important point to note here is that, a **wrong/misspelled or exit** command **does not** count as an access. The server is smart to distinguish between false commands and not penalize user for it.

When a user voluntarily wishes to exit(client initiates closing procedure), they are removed from the 'dynamic' global array of users maintained at the server. The complexity faced here was, when multiple users try to leave the server at the same time, it had to be made sure that both do not access the global array together, because if they would, there is a chance of nasty index out of bounds errors.

For example, users at index 2 and 3 wish to leave at the same time. Suppose user at index 2 leaves and array resizes(due to being dynamic). The user who **was** at index 3 **is now** on index 2 and if the server attempts to remove the user at index 3, an index out of bound error may occur or worse, some other user may be removed abruptly.

To resolve, this, it was ensured that the threads where synchronized while accessing the global user array(by application of mutex locks).

The interface remains the same as before, both client and server run on the terminal/command-line interface.

The completed tasks are as follows with latest updates at the end,

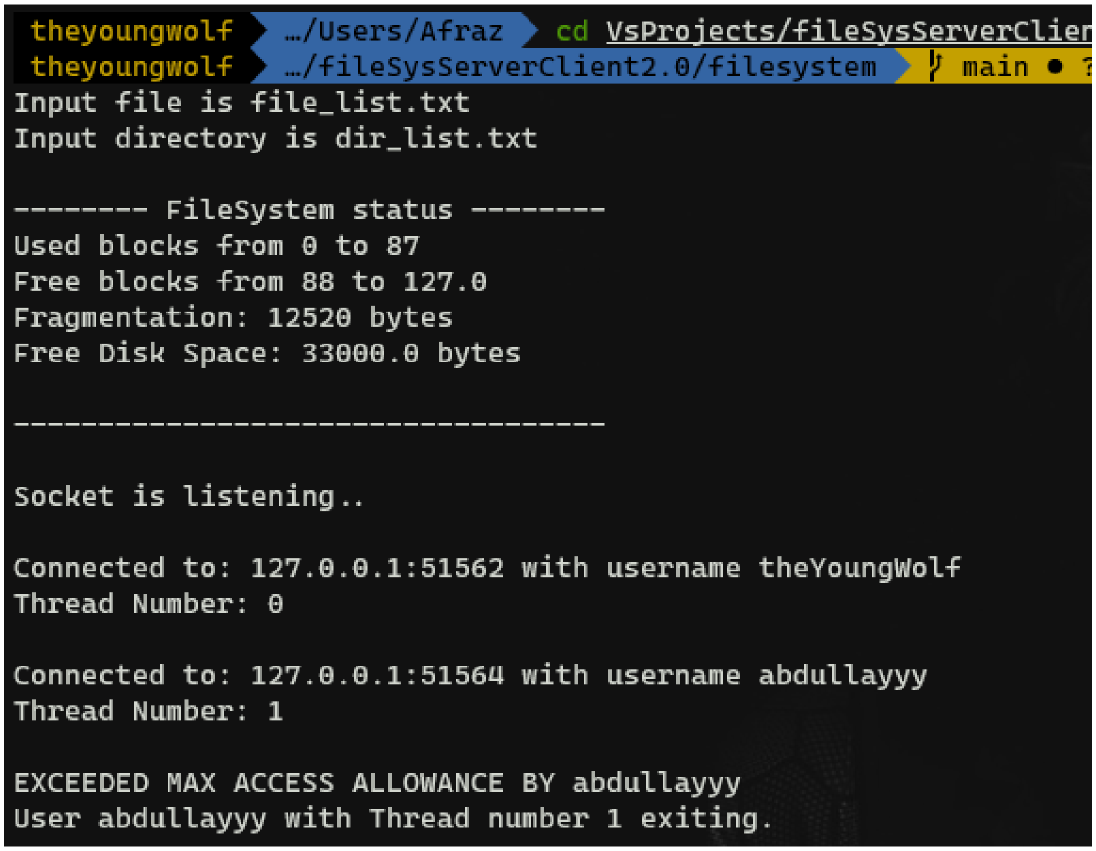
	Tasks
✓	Two programs, server and client
✓	Static IP of server
✓	Client identified by username provided
✓	Interaction: Command line interface
✓	Client waits for set amount of time if server is offline
✓	Response from server is displayed on client side
✓	Server capable of entertaining multiple clients
✓	Server binds to port 21018 (not 95)
✓	Server and client can run on separate machines within local network
✓	Mutual exclusion
✓	Solved Reader-Writers Problem through Synchronization
✓	Restricted every user to 5 File System Accesses with smart servicing

Core functionalities:

- Make directory function
- Remove directory function
- Make file function
- Remove file function
- Append to file function
- Truncate from file function
- Memory map function
- Print disk status
- Print tree hierarchy

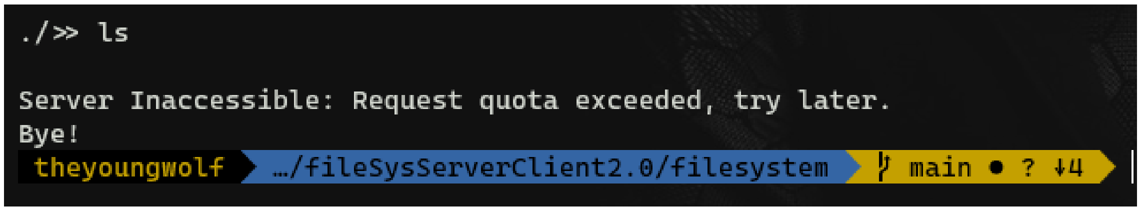
SCREENSHOTS

Server



```
 theyoungwolf  …/Users/Afraz  cd VsProjects/fileSysServerClien
 theyoungwolf  …/fileSysServerClient2.0/filesystem  ᛘ main ● ?
Input file is file_list.txt
Input directory is dir_list.txt

-------- FileSystem status --------
Used blocks from 0 to 87
Free blocks from 88 to 127.0
Fragmentation: 12520 bytes
Free Disk Space: 33000.0 bytes

-----------------------------------


Socket is listening..

Connected to: 127.0.0.1:51562 with username theYoungWolf
Thread Number: 0

Connected to: 127.0.0.1:51564 with username abdullayyy
Thread Number: 1

EXCEEDED MAX ACCESS ALLOWANCE BY abdullayyy
User abdullayyy with Thread number 1 exiting.
```

Client 2



```
./>> ls

Server Inaccessible: Request quota exceeded, try later.
Bye!
 theyoungwolf  …/fileSysServerClient2.0/filesystem  ᛘ main ● ? ↓4 
```