Computer System and Programming

Winter Exam Session

January 2026 - February 2026 Exam Session

This document does not apply to you if you are not willing to take the exam in the winter session.

A.A. 2025-2026

Intro

- Develop a Linux-based C-based client-server system that manages virtual home directories for users created "on the fly". The server maintains a storage space organized in directory by user and allows file management operations and concurrent interaction between multiple clients.

General Architecture

- The system consists of:
  - **Server:** maintains users, credentials, and the virtual filesystem.
  - **Client:** allows users to authenticate and send commands.
- The server must support multiple concurrent clients through forking or multiple processes.
- Each client communicates with the server via a network interface (sockets).

Server

- The server is a process that manages a virtual file system based on a root directory provided at startup. All operations sent by the clients are performed exclusively within this root, ensuring isolation and control.

- The server receives commands from a client. Communication between server and client is done through the network interface (**socket**)

Server

- When the server is launched, it must receive as a mandatory parameter: **./Server <root_directory> <IP> <port>.**
- **<IP>** and **<port>** they are the IP and the port where the server listens to receive requests.
    - Default values: **127.0.0.1** e **8080**
- **<root_directory>** is the root directory within which the server will keep:
    - **Users' home**
    - **Their files**
    - **Any metadata (if any)**
- If the directory does not exist, the server must: **create it automatically.**

Server: User Management

- **User management:** The server must support the dynamic creation of users (command **create_user <username> <permissions (in octal)>**)
    - Create the username directory
    - Set the logical owner of the home
    - Set the home permissions to **<permissions (in octal)>**. Note: All users must have the same **GROUP**.
    - Creation take place without a password.
    - **HINT:** Since the user will be the owner of his files, it is useful to create a real user in the system. This can be done via the command-line tool **adduser**. For example **sudo adduser --disabled-password username** (exectable via **exec**) creates a username username.
- If necessary the server can be run with sudo. In this case, manage euid well (**euid = root** must be maintained the bare minimum)

Client

- When the client is launched, it must receive as a mandatory parameter: **./Client <IP> <port>.**
- **<IP>** and **<port> t**hey are the IP and the server port
  - **Default values: 127.0.0.1** e **8080**

Server/Client: Session Management

- **Session management**: The server allows you to log in to the client. The client can send the command **login <username>**
- The server authenticates the user (up to you how to do it). The password is not required.
- If the login takes place, all subsequent operations of the client are referred to the home of that user and can be attributed to that user.
- The login does not use a password: it only serves to identify "the owner" of the commands.

Server/Client: File Operations

- Once logged in, the user can perform:
  - **create <path> <permissions (in octal)>**: creates an empty file in the location **<path>** with permissions **<permissions>**. The **-d** option creates a directory.
  - **chmod <path> <permissions (in octal)>**: Set the **<path>** file permissions to **<permissions>**
  - **move <path1> <path2>**: moves a file identified by **<path1>** to **<path2>**.
  - **upload <client path> <server path>**: The client sends the **<client path>** file via network (socket) by uploading it to the server in **<server path>**. With the **-b** option the upload must take place in the background (the client can continue to send new commands). When the background upload is completed, the client must notify the user by pressing **“[Background] Command: upload <server path> <client path> concluded”**.
  - **download <server path> <client path>**: The server sends the **<server path>** file via network (socket). The client will save it to **<server path>.** With the **-b** option, the download must take place in the background (the client can continue to send new commands). When the background download is completed, the client must notify the user by printing **“[Background] Command: download <server path> <client path> concluded”**.

Server/Client: File Operations

- **cd <path>:** Change the **current working directory** using **<path>.**

- **list <path>**: lists the files/directories contained in **<path>** also printing their **permissions** and **logical size**. If a directory is not specified, it lists the **current working directory**. Unlike other commands, you can also list the directories of other **users** (**others' home**).

- **read <path>**: Sends the content of **<path>** to the client who will print it in **stdout**. It is possible to specify the **-offset=<num>** option which will force the sending from the **<num>** byte of the file. Example: **read -offset=10 <path>:** Send the file starting from the offset byte **10.**

- **write <path>**: The client will read from **stdin** and send the input to the server that will write the content inside **<path>** (if the file does not exist it is created with permission **0700**). It is possible to specify the **-offset=<num>** option which will force the server to write from the **<num>** byte of the file. Example: **read -offset=10 <path>:** Send the file starting from the offset byte **10.**

- **delete <path>**: the server deletes the **<path>** file.

Server/Client: Exit

- Server termination.
  - **exit**: The server can receive the **exit** command from the user. This will terminate the server immediately.
- Client termination.
  - **exit**: The client can receive the **exit** command from the user. Termination is performed immediately if there are no background operations (e.g., no **upload -b** or **download -b** pending). If, on the other hand, there are processes in background, this is notified to the user and the termination is interrupted (the client returns to being ready to receive the next command).

Server/Client: Considerations

- All users will have the same **GROUP**. So, to allow you to list a directory of another user, just set the **GROUP** permissions appropriately (**OTHER** is never used).

- All operations must be **sandboxed**: a user can never access outside the server's **root directory** and the user's **home directory**. Only **list** can go outside the user's **home** (but not outside the server's **root directory**). **list** runs correctly only if the permissions of the other users' **files/directories** allow it.
  HINT: Remember that all users have the same **GROUP**. **OTHER** permissions are never used.

- **Clients can run in parallel.** Multiple clients can perform operations on a file. It is your responsibility to manage the competition in the best possible way. Example: If a client is reading a file, no one can write to it. Writings cannot be parallel. Handle the delete operation as a write.

- Print error messages and operation outcomes to the clients' **stdout**. **Do not make the user (me) work blindly.**

Server/Client: Considerations

- **IMPORTANT:** All operations use **<path>** to identify locations. All operations should accept both relative and absolute paths. Also, they must accept **.. (parent dir)** and **. (current working dir)**

- **IMPORTANT:** The server must be able to handle multiple clients in parallel without delay. More than one user can log in in parallel, the same user can start multiple clients and log in multiple times.

- **IMPORTANT:** Clients can perform operations in parallel without getting in the way of each other (see previous discussion on parallel read/write on the same file).

- **IMPORTANT:** Although the server can run with **sudo**, it is your responsibility to release the permissions as soon as possible. **The less you are root, the better.**

Error Handling

- It's your responsibility:
  - Manage I/O errors:
  - Handle unexpected client (or server) terminations.
  - Eliminate/collect zombie processes.
  - Prevent or manage race conditions.
  - Commands with missing parameters, options (notify the user).
- Remember that not all errors have to end in termination. Some can be handled by continuing the server/client to work normally. It is your responsibility to manage them in the best way.

Modularity

- The project should **NOT** consist of **server.c** and **client.c** alone. Division into several modules is required. Try to minimize the code as much as possible. Don't copy-paste the same code here and there (but modularize it).
- Example (it's just an example. Each project will require a different modularity):
    - Network.c / network.h
    - Session.c / session.h
    - Transfer.c / transfer.h
    - Utilis.c /utils.h
    - ...

How to get **30** (with honors)

- Everything described so far will allow you to get a vote of **25** (out of 30). The vote will be decided based on how many functionalities you have implemented, how, and based on the oral.

- To get a vote between **26** and **30 (with honors)** it is necessary to implement the following additional feature **(see next slide)**.

Server/Client: Transfer Request

- The system must support the **transfer_request <file> <dest_user>** operation
- The client executes the **transfer_request <file> <dest_user>** command:
    - **<file>** a file belonging to the client user (decided at login).
    - **<dest_user>** a recipient user who will receive the file.
    - The **transfer_request** should have an identification **<ID>** generated by the system (that is required later to accept it).
- The server receives the request and notifies (sending also the **<ID>**) the target user in real time (must be online). Up to you how you want to notify the target user.
- The recipient (client who logged in with **<dest_user>**) can do on of the following actions:
    - Send the **accept <directory> <ID>** command: the client accept the request identified by an <**ID>** sent by a server. The server will copy the sender's **<file>** file to the **<directory>** directory of the receiver **<dest_user>**
    - Send the **reject** command: The request is rejected and the two users notified
- In practice, this operation allows you to copy a file between two distinct users but requires interaction and acceptance.
- **NOTE**: For simplicity, you can assume that during a **transfer_request** there is only one client logged in as **<dest_user>.** Still, a different pairs of users can execute two independent **transfer_request** commands (there can be multiple **transfer_request** in parallel).

Server/Client: Transfer Request

- If the sender runs **transfer_request <file> <dest_user>** and no client is logged as **<dest_user>**, the sender's client remains blocked waiting for **<dest_user>** to log in.

Concurrency (again!)

- It is your responsibility to properly manage concurrent access to files and parallel requests. You should handle concurrency (between reads, writes, transfer requests, etc.) in the best way possible.
- Make your own design decisions when implementing these steps. Your grade will take this aspect into consideration.

Prohibited System calls

- For security reasons, you cannot use the system call **system()** and **popen()** in your code. However, you can implemented it and simulate it manually by using **fork()+exec()** and **pipe()+dup()** (or **dup2()**).
    - Any code using **system()** and **popen()** will be rejected.
- Also, you **cannot** implement any of the functionalities required by the project by **exec-ing command-line tools** (**cat, ls, mv, rm, etc**) except for those tools mentioned in the slides (e.g., **adduser**). You must implement these features using the system calls we covered in class.
- If you are unsure which system calls or command-line tools you are allowed to use, feel free to email me.

Submission and Oral

- The delivery must take place at least/about one week before the exam. The precise date of submission has not yet been decided and will be communicated through google groups.

- **NEW RULE**: You can do the project in pairs. However, it is your responsibility to know the code completely as the oral will be done individually.

- **The oral will consist of questions about the code (e.g., discussing how parallelism is implemented). Depending on the situation, it is also possible that you will be required to answer to "theorical" questions about the program (e.g., how the fork works, file descriptors, etc.)**

Submission

- ***Submission is via email. You MUST use your institutional email***
- The subject of the email must be **“[CSAP PROJECT 2026] <student ID number 1> <student ID number 2>”**. If you do the project alone you can put only one **student ID number**.
- **IMPORTANT:** ***use the right email subject. otherwise I won't see your email.***
- In the body of the email also mention your name and surname.

What you need to submit?

- What to submit (email attachment): A **.zip** file containing the following files.
  - README (docx, pages, txt, etc.): Document describing the following aspects:
    - How to compile your code.
    - How to start the server
    - How to start the client
    - How to execute the implemented commands and the expected outputs (including command options, e.g., **-offset** of the **read** command)
  - **Project folder:** Containing all the source files of the project.
  - **Script (Makefile, bash, python):** File to be executed that will automatically compile the project producing the **server** and **client** executables.

Other info

- The project will be compiled using **Ubuntu 24.04**. It is your responsibility to ensure that it compiles and runs correctly on this OS.
  Tip: Use a virtual machine with **Ubuntu 24.04** for code compilation and testing.

- **IMPORTANT:** If the code does not compile, you will not be admitted to the oral exam. As a conseguence, you need to wait for the next exam session.
  Example: If you submit on **January** and your code does not compile, you must wait for the **February** session to submit again.
  THIS IS A STRICT RULE! NO EXCEPTIONS!

- You can collaborate between groups but not copy. The submitted code will be checked by **anti-plagiarism** software. If you are **"caught"** there is a high chance that you will get **0** (both who had copied and who copied).
  This should not discourage you from collaborating. If you collaborate honestly you don't run any risks.

- ***The code must constitute an original creation, therefore it is not possible to share parts of it or copy significant portions of contents from other sources. No external libraries are permitted. Use the system calls/functions we have seen in class.***

Other info

- Once you get a grade **>= 18** (after project + exam), you have two options:
  - **Refuse the vote.** If you refuse it, you **CANNOT** use the same project at the next exam. You will be asked to do a different **project** or an **oral exam** (more in detail about the whole course).
  - **Accept the vote. You are good to go! :)**
- If you **fail** (after the project + oral) or if the code does not compile, you can return to the **next exam** with the same project.
- **TO KEEP IN MIND:** If you are willing to do the exam after the **winter session (after February)**, the project will change!