Assignment 2: Design and Implementation of a Multi-Threaded Asynchronus Client-Server Application

Important Note!

Your Client/Server implementation **must follow** the structure and functionality of the instructor's sample Client/Server applications. *The client and server are two independent applications with no parent-child relationship between them.* It is essential to experiment with the instructor-provided Client/Server sample applications to thoroughly understand the assignment requirements and gain insights for initiating your implementation.

1. To facilitate testing with the provided sample server executable, multiple client instances can be created using the sample client executable. This can be accomplished by duplicating the sample client application and assigning unique identifiers, such as *client1*, *client2*, etc.
2. You may select any **Inter-Process Communication (IPC)** mechanism from the following options: **Message Queue**, **Shared Memory**, or **Pipe**. However, the use of **SOCKETS is strictly prohibited**, as they are beyond the scope of this course.
3. To prevent permission-related issues when creating **POSIX message queues**, particularly on the school server, **it is strongly recommended to prefix message queue names with your Panther ID (PID)**. This practice ensures uniqueness, avoiding conflicts with existing queue names. Some Linux distributions impose restrictions on common queue names such as `/posix_server_queue`, `/server`, or `/mq_server`, as these may already be in use by system services or other users. By appending your PID to the queue name (e.g., `"/1234567_server_message_queue"` for PID `1234567`), you can ensure that the queue is uniquely associated with your process. Additionally, before creating a queue, it is advisable to invoke `mq_unlink()` on the queue name to remove any lingering queues from previous executions, thereby preventing unintended interference with your program.
4. If you do not have execute permissions for the instructor-provided sample executables, you can explicitly grant **read (r), write (w), and execute (x) permissions** using the following command on a Linux computer:

   ```
   chmod 777 client server sysinfo
   ```

Overview

This assignment focuses on developing a ***multi-threaded client-server application*** in C that handles ***asynchronous*** (non-blocking) communication between multiple clients and a server. The server acts as a Linux shell that processes various commands sent by clients and responds to the clients appropriately. The assignment aims to deepen the understanding of multithreading, process management, inter-process communication, and graceful termination of programs.

Asynchronous refers to a mode of operation where tasks or processes execute independently, without waiting for other tasks to complete. In an asynchronous system, a program can continue performing other operations while waiting for the result of a task, thereby improving efficiency and responsiveness. **Threads** are a common mechanism for achieving asynchrony, as they enable tasks to run concurrently, improving responsiveness and efficiency.

Learning Objectives

By completing this assignment, you will:

1. Understand and implement multithreaded Client/Server communication.
2. Demonstrate proper usage of threads, processes, and IPC (Inter-Process Communication) mechanisms.
3. Differentiate between synchronous and asynchronous IPC operations.
4. Manage Client/Server resources efficiently by ensuring graceful termination using proper signal handling.
5. Validate user commands and handle incorrect or unexpected input gracefully.
6. Analyze the scalability and limitations of different IPC mechanisms in real-world scenarios.

Shell Commands and User-defined Commands

Valid Shell Commands

A valid shell command is a command that is syntactically correct and can be executed successfully by the shell. These commands either correspond to built-in shell utilities, executable files, or properly formatted instructions that the shell can interpret and execute without errors.

Examples of Valid Shell Commands

1. `exit` – Terminates the process abruptly without performing any cleanup.
2. `ls` – Lists files and directories in the current working directory.
3. `pwd` – Prints the current working directory.
4. `echo "Hello, World!"` – Outputs the text "Hello, World!" to the terminal.
5. `./executable` – Executes a compiled binary file named `executable` in the current directory.
6. `cat file.txt` – Displays the contents of `file.txt` (if it exists).
7. `sleep 100000` – Pauses the process for 100,000 seconds.
8. `grep "pattern" file.txt` – Searches for the text "pattern" in `file.txt`.

Invalid Shell Commands

An invalid shell command is a command that the shell cannot execute successfully due to reasons such as syntax errors, incorrect formatting, missing arguments, or unrecognized commands. These commands usually result in an error message from the shell.

Examples of Invalid Shell Commands

1. `ls-l` – Invalid because there is no space between `ls` and the `-l` flag.
2. `abc` – Invalid because it is not a recognized command or executable.
3. `echohelloworld` – Invalid because there is no space between `echo` and the string to be printed.
4. `catfile.txt` – Invalid because there is no space between `cat` and the file name.
5. `mkdirnew_folder` – Invalid because there is no space between `mkdir` and the directory name.
6. `rm` – Invalid if no file or directory is specified.
7. `grep patternfile.txt` – Invalid because there is no space between the pattern and the file name.

Valid User-Defined Commands

For the context of this assignment, Valid and Invalid user defined commands are defined. These are not the regular shell commands. A valid user-defined (non-Shell) commands are syntactically correct instructions recognized by our Client-Server application. These commands adhere to the specified format and trigger the corresponding actions as defined in the application logic.

Valid User-Defined Commands

Note: To fully understand the required behavior of the Client/Server for the valid user-defined commands across all possible expected scenarios, thoroughly experiment with the instructor-provided sample Client and Server applications.

1. `CHPT new_prompt`
   Client: Changes the client's input prompt to `new_prompt`.
   Server: Not involved in handling this command.
2. `EXIT`
   Client: Gracefully disconnects the client, releases resources, and terminates the session.
   Server: Removes the client from the registered list and logs the disconnection.
3. `LIST`
   Client: Requests the server to provide a list of all registered and visible clients.
   Server: Responds with the list of registered clients, excluding those marked as hidden.
4. `HIDE`
   Client: Requests the server to hide the client from the visible client list.
   Server: Marks the client as hidden and excludes it from future `LIST` responses.
5. `UNHIDE`
   Client: Requests the server to make the client visible again in the `LIST` response.
   Server: Marks the client as visible and includes it in the list of registered clients. Any subsequent `LIST` command from the clients will list the unhidden client.
6. `exit` **(lowercase)**
   Client: Sends it to the server. It should be Ignored by the server to prevent accidental termination of the shell session.
   Server: Logs and replies to the client: "Ignored 'exit' command as it may Exit the Shell Session..."

7. `SHUTDOWN`
   Client: Not a valid command for the client to send directly.
   Server: Broadcasts the `SHUTDOWN` message to all connected clients, instructing them to terminate their sessions and clean up resources.

Invalid User-Defined Commands

Invalid user-defined commands refer to those that either do not adhere to the correct syntax or are not recognized by our Client/Server applications. Such commands may include errors, misspellings, unsupported formats, or improper arguments, rendering them invalid. Since the server is not designed to recognize these invalid user-defined commands, they are passed to the shell instead. The shell, in turn, treats them as invalid shell commands.

***Examples of Invalid User-Defined Commands ***

1. `chpt new_prompt`

   Invalid because commands are case-sensitive and should be uppercase as `CHPT`.

2. `CHPT`

   Invalid because the `CHPT` command requires an argument (e.g., `CHPT new_prompt`), and no argument is provided.

3. `EXIT NOW`

   Invalid because the `EXIT` command does not accept any arguments.

4. `LIST all`

   Invalid because the `LIST` command does not support additional arguments such as `all`.

5. `HIDE client`

   Invalid because the `HIDE` command does not take any arguments such as `client`.

6. `UNHIDE user`

   Invalid because the `UNHIDE` command does not accept arguments like `user`.

7. `SHUTDOWN` (when sent by the client)

   Invalid because `SHUTDOWN` is a server-initiated broadcast command and cannot be sent by the client.

Requirements

Inter-Process Communication (IPC) Mechanisms Usage

For this assignment, you must ***use only the following IPC mechanisms of your choice***. You may combine more than one mechanism if required to meet the assignment objectives:

- **Message Queues**
- **Shared Memory**
- **Pipes**

Additionally, the ***server and clients must run on the same machine***. Supporting execution on different machines can be considered as a future enhancement.

In your `README.txt`, ***clearly justify your choice of IPC mechanism(s)*** and explain how they fit the design and requirements of this assignment.

Shell Server Implementation

The asynchronous server is designed to emulate a Linux shell and must be implemented as a multi-threaded system, with each client command being processed in a separate thread.

1. ***Thread and Process Management:***
    - Each client command is handled in a separate child thread. **Hint:** Explore the differences between `pthread_join()` and `pthread_detach()`, and adopt the one that is more suitable for your implementation while adhering to the asynchronous Client/Server model.
    - **For both valid and invalid** `Shell commands`**, spawn a new process** using the `execlp` system call. **Hint :** To handle shell commands in a new process, you can use the following pattern in your code:

      ```
      execlp("/bin/bash", "bash", "-c", command, NULL);
      ```

      Alternatively, you may explore other options (e.g., `execvp`, `execv`) if they are more appropriate for your implementation.
    - The server should monitor the execution time of any newly spawned process and terminate it forcefully **if the execution exceeds 3 seconds**. This ensures that no command runs indefinitely (e.g., `sleep 10000`). To achieve this, the server can use the `kill(child_pid, SIGKILL)` system call to terminate the child process.
    - The `User-defined commands` (non shell commands) are processed directly by the child thread. There is no need to spawn a new child process for executing them. *However, if your design approach requires it, you may opt to spawn a new child process to execute these commands. This choice depends entirely on your preferred implementation strategy.*
    - Your terminal output should display detailed information about the processes and threads to clearly illustrate the creation and utilization of multiple processes and threads, as outlined in the instructor's server implementation.
2. ***Commands to be supported:***
    - Shell Commands
        - The server receives shell commands—whether valid or invalid—from the client, executes these commands within a separate process in its shell environment, and returns the corresponding output or error message to the client.
        - Support the execution of user-defined executable files, such as `./sysinfo` or `../sysinfo`. The `sysinfo` executable, included in the downloadable resources, provides system-specific information. You are also encouraged to test by running your own executable files to ensure compatibility and functionality.
    - User-defined Commands

      User-defined commands sent from the client can be **processed directly by the child thread,** *eliminating the need to spawn a new process for their execution*.
        - `REGISTER`: Registers a new client.

- `EXIT`: Upon receiving this command from the client, server gracefully disconnect the client and terminates its session by releasaing all the resources dedicated for managing that client.
- `LIST`: Send a list of all registered visible (unhidden) clients to the requesting client. If no clients are visible (hidden), server sends a message `All Clients Are Hidden...`
- `HIDE`: Hide the requesting client from the visible list and send a message `You Are Now Hidden...` to that client. By default all the clients must be visible unless they wish to hide themself from other clients. If the client is already hidden, server sends a message `You Are Already Hidden...`
- `UNHIDE`: Unhide the requesting client and make it visibleand send a message `You Are Now Visible Again...` to that client. If the client is not hidden, server sends a message `You Are Not Hidden At All...`
- `exit`: Ignore `exit` (lowercase) to prevent accidental shell logout/termination. Send a message `Ignored 'exit' command as it may Exit ther Shell Session...`

3. **Client *Responses:***

   - Ensure that client responses are handled with accurate and appropriate messages based on the received commands. Refer to the instructor's sample Client/Server applications to understand the correct flow of communication between the client and server. Below are examples of potential client responses:

     - `Command Timeout...`

       The specified command has exceeded the allowable execution time of **3 seconds**.

     - `Ignored 'exit' command as it may Exit ther Shell Session...`

       The 'exit' command was received but has been disregarded.

     - `You Are Now Visible Again...`

       The client's visibility status has been updated to visible.

     - `You Are Not Hidden At All...`

       The client was already visible and not in a hidden state.

     - `You Are Already Hidden...`

       The client is already in a hidden state; no changes were applied.

     - `You Are Now Hidden...`
       The client's visibility status has been updated to hidden.

     - `All Clients Are Hidden...`
       All clients are now set to a hidden state and, hence, cannot list the registered clients.

4. ***Graceful Termination***:

   - The server broadcasts a `SHUTDOWN` message to all connected clients before its termination (such as through a `Ctrl+C` command).

- The server must ensure that all allocated resources—including threads, memory, IPC mechanisms, file descriptors, and other system resources—are properly released before exiting. **Hint:** Register appropriate signal handlers to catch and handle termination signal `Ctrl+C` (SIGINT).

Client Implementation

The multi-threaded asynchronous client sends both valid and invalid shell commands, as well as user-defined commands, to the server and receives appropriate responses. The client must gracefully terminate upon receiving termination signals, such as `EXIT`, `SHUTDOWN`, or `Ctrl+C`.

1. ***Thread and Process Management:***
   - The client creates a child thread to monitor and receive the server's `SHUTDOWN` broadcast message. **Both shell commands and user-defined commands can be executed within the main thread.** *However, you may choose to implement them in a separate child thread, depending on your preferred design approach.*
   - **There is no need to spawn any child processes in the client for any purpose.** *However, if your design approach requires it, you may choose to implement a multi-process client. This choice depends entirely on your preferred implementation strategy.*
   - The terminal output should display detailed information about the processes and threads to clearly illustrate the creation and utilization of the main thread and child thread, as outlined in the instructor's client implementation.
2. ***Commands to be supported:***
   - Clients send shell commands, whether valid or invalid, to the server. The server processes these commands and returns responses, which are then displayed on the client terminal.
   - `EXIT`: Gracefully disconnects the client and terminates its session. Sends the `EXIT` command to the server, allowing it to gracefully disconnect the client and release any associated resources.
   - `LIST`: Provides a list of all registered and visible (unhidden) clients.
   - `HIDE`: Hides the client from the visible client list.
   - `UNHIDE`: Unhides the client if it is hidden, making it visible again.
   - `exit`: Server Ignores the `exit` command (lowercase) to prevent accidental shell logout or termination and send the appropriate message to the client .

   IMPORTANT: Ensure that the client displays accurate and appropriate messages based on the commands entered by the user. Refer to the section `*Client Responses*` under `Shell Server Implementation` to understand the responses received by the client from the server. Experiment with the instructor's sample client/server applications to observe and understand the correct flow of communication between the client and the server.
3. **Command Handling**:
   - Main Thread:
     - The **main thread** is responsible for processing user commands entered by the client. These commands can include both shell commands (e.g., `ls`, `pwd`, `./sysinfo`) and user-defined commands (e.g., `EXIT`, `LIST`, `HIDE`).

        - Based on the command entered, the main thread sends the command to the server and waits for an appropriate response.
        - The main thread should handle invalid, empty, or improperly formatted commands gracefully by displaying relevant feedback to the user.
    - Child Thread:
        - A **child thread** is created to monitor the server's `SHUTDOWN` broadcast message.
        - This thread operates independently of the main thread, ensuring it asynchronously monitors the `SHUTDOWN` message from the server without interrupting the main thread's operations.
        - Upon receiving a `SHUTDOWN` message from the server, the child thread triggers the client to terminate gracefully by coordinating with the main thread.
4. **Graceful Termination**:
    - The client terminates gracefully when either:
        1. The user issues the `EXIT` command.
        2. The server sends a `SHUTDOWN` broadcast message.
        3. The user interrupts the client using `Ctrl+C` (or other signals).
    - The client must ensure that all allocated resources—including threads, memory, IPC mechanisms, file descriptors, and other system resources—are properly released before exiting. **Hint :** You may register appropriate signal handlers to intercept and manage termination signals (e.g., `EXIT`, `SHUTDOWN`, or `Ctrl+C`).
    - When the user wishes to terminate the client, the client should send the `EXIT` command to the server. This allows the server to remove the client's entry and release the associated resources before the client terminates gracefully.

Testing Scenarios

You must thoroughly test their implementation using the following scenarios. The list below includes sample test cases, but it is not comprehensive. Please refer to the instructor's sample Client/Server applications to fully understand all testing requirements and to identify additional scenarios.

1. Graceful Termination

- Test the following methods for terminating the client:
    - `CTRL+C`.
    - Entering the `EXIT` command.
    - Server-sent `SHUTDOWN` broadcast message.
- Verify that the client releases all resources (e.g., threads, memory, IPC mechanisms) before terminating.
- Test graceful termination of the server by pressing `CTRL+C`, ensuring that all active threads and resources are cleaned up and a `SHUTDOWN` broadcast message is sent to all the registered clients.

2. Command Validation

- **Valid Commands**:
  - Use `CHPT` to change the client prompt and validate the updated prompt display.
  - Use `LIST` to display registered clients. Test with both a single client and multiple clients connected.
  - Use `HIDE` to hide a client and verify its effect with the `LIST` command. Attempt to hide the same client again and observe the server's response.
  - Use `UNHIDE` to make a hidden client visible again.
- **Invalid Commands**:
  - Enter incorrect commands (e.g., `CHPT123`, `LIST all`, `HIDE client`) and validate that the server sends appropriate error responses.

3. Thread Management

- Run multiple clients concurrently and send command both **Shell** and **User-defined** commands to verify the proper request-response of Clients and the Server. Observe the server's behavior by checking the assigned `thread_id` values to ensure each client request is handled by a separate thread and that thread management is functioning correctly.
- Test if the server can handle and respond to multiple concurrent client requests efficiently without crashing, hanging, or causing any deadlocks.
- Verify the server's behavior when multiple clients send long-running commands (e.g., `sleep 10000`). Ensure that the server detects commands exceeding the **3-second duration** and sends a `Command Timeout...` message to the respective clients while continuing to process other client requests concurrently.

4. Execution of User-Defined Files

- Test running user-defined executables like `./sysinfo` in the current directory and `../sysinfo` in the parent directory.

5. Edge Cases

- **Empty Input**: Press `Enter` without typing any command to ensure that the server and client do not crash. The client should handle this gracefully by displaying the error message: `Invalid input. Please enter a valid command.`
- **Commands with Only Spaces**: Test inputs such as `" "` (commands containing only spaces) to ensure that the server and client do not crash. The client should handle this gracefully by displaying the error message: `Invalid input. Please enter a valid command.`
- **Invalid Command Syntax**: Enter malformed commands (e.g., `LS`, `register`, or `HIDD`) to verify that the server sends proper error responses and that these are logged on both the server and client sides.
- **Improper `CHPT` Command Inputs**: Test the `CHPT` command with invalid inputs (e.g., `CHPT`, `CHPT`) to ensure the client displays the error message `Invalid command. Usage: CHPT <new_prompt>`, and with valid prompts followed by multiple spaces (e.g., `CHPT $`) to verify spaces are trimmed and the prompt is updated successfully.

- **Experiment Thoroughly**: Use the instructor's sample Client/Server applications to identify all possible edge cases and understand the expected error messages for proper validation and testing.

6. Server Responses

- Test various combinations of user-defined commands to confirm correct server responses.

Deliverables

Submit a single `Deliverables.zip` file containing the following:

1. A `CODE` Folder

The `CODE` folder must include:

- Source Code

  All `.c` files and any optional `.h` header files used in your project.
- Statically-Linked Executable
  - Provide statically-linked executables `client` and `server` for your Client and Server applications that are ready to be run for testing purposes.
  - If you are using **non-Linux computer**, explore ways to generate a statically-linked executable by compiling your final source files on a Linux computer. For example:

    ```
    gcc -static Server.c -o server -lpthread -lrt
    gcc -static Client.c -o client -lpthread -lrt
    ```

  - Alternatively, explore online C compilers capable of generating statically-linked executables.
- Make file
  - Provide a Makefile that supports the following commands:
    - `make`: Compiles the project and generates two executable files named `server` and `client` for the Server and Client implementations, respectively.
    - `make clean`: Removes all object files, executables, and any temporary files generated during the build process.

2. A `README.txt` File

The `README.txt` file must include:

- Team Member Information
  - Section Name, Full Name, PID, FIU email of all team members.
- Compilation and Execution Instructions
  - Include any specific guidelines, notes, or considerations about the project implementation.
- ***Clearly justify your choice of IPC mechanism(s)*** (e.g., pipes, message queues, or shared memory)

- Briefly explain how your selected IPC mechanism(s) align with the design and requirements of this assignment, and why they are the most appropriate choice for your implementation.

****** *Please refer to the course syllabus for additional assignment submission requirements and guidelines.*

Deliverables Folder Structure

You are required to strictly adhere to the following structure for the `A2_Deliverables.zip` file when submitting the assignment.

A2_Deliverables.zip
|
|-- **CODE/**
| |-- *.c # All .c source files
| |-- *.h # Any optional header files
| |-- Makefile # Makefile with commands: make and make clean
| |-- executable # *Statically linked executable files* 'client' and 'server' of your implementation. Try running it on different computers to check its portability
|-- **README.txt** # Includes team details, IPC usage justification, any specific compilation instructions

Grading Rubric

Criteria	Marks
No Makefile	**0**
No IPC	**0**
Cannot Generate/Run Executables	**0**
No Multithreading, Multiprocessing	**Max 70**
Partial Implementation *(Based on completeness)*	**0 - 70**

Detailed Breakdown of 100 Marks

Category	Functionality	Marks
(1). Server Functionalities (3 Marks Each)	Client Registration	3
	LIST	3
	HIDE	3
	UNHIDE	3
	EXIT	3
	Timeout	3
	Valid Shell Commands	3

Category	Functionality	Marks
	Invalid Shell Commands	3
	Invalid User-defined Commands	3
	SHUTDOWN	3
	Total of (1)	**30**
(2). Client Functionalities (5 Marks Each)	CHPT	5
	Edge Cases Validation	5
	Total of (2)	**10**
(3). Graceful Termination & Resource Cleanup	Server	10
	Client	10
	Total of (3)	**20**
(4). Multithreading & Multiprocessing	Server	20
	Client	10
	Total of (4)	**30**
(5). Code Quality, Documentation, README	**Total of (5)**	**10**

Hints and Notes

- Refer to the **instructor-provided sample Client/Server applications** to understand the required implementation details.
- Use debugging tools like `gdb` to identify and troubleshoot issues effectively.
- Consult **man pages** (`man <command>`) for system calls and library functions to gain additional insights and usage details.

Additional Resources

1. Linux Programming (System Calls & Libraries)

- **Linux Programming (System Calls & Libraries)** – Comprehensive documentation of system calls and C library functions.

2. POSIX Threads (pthread)

- **POSIX Threads Programming Tutorial** – Covers thread creation, termination, joining, and synchronization.
- **POSIX Threads Programming (LLNL)** – In-depth explanations and examples of Pthreads.
- **POSIX Threads Tutorial (UCLA)** – Detailed tutorial with sample programs.

3. Inter-Process Communication (IPC)

- **POSIX IPC (Oracle Docs)** – Covers shared memory and message queues in POSIX IPC
- **A Quick Guide to Pipes and FIFOs**: This guide provides an overview of POSIX pipes, including how to create and use both unnamed and named pipes, with practical examples.