System Programming CSE344
Midterm Assignment Report

R.Hilal Saygin
200104004111

Usage:

make in server and client directory
./neHosServer <dirname> max#ofclients
    To run the server, then server listens incoming request.
./neHosClient connect <serverPID>
    Connection established if there is available space for a new client. Now the client can enter a request command. Enter ‘help’ to display available commands

System Design:
A Concurrent File Access System

FIFOs

FIFOs are used for inter-process communication between the server and multiple client processes. For each client connected to server, its own FIFOs for communication with the server are created. These FIFOs used for bidirectional communication channels between server-client. The server creates a FIFO for receiving requests from clients, enabling asynchronous handling of multiple client connections.

Server FIFO (Named Pipe):

A server FIFO ,in the format "/tmp/send_fifo_client.%d", is created using mkfifo() to handle connection requests from clients.
When a client wants to connect, it send the connect/tryConnect a request to this FIFO indicating its process ID and the action it wants to perform.
The server reads from this FIFO to receive connection requests and processes them accordingly.
If a client sends <connect> and there is an available place for new client connection, meaning the client count isn’t reached to its upper limit, the server accepts the connection request. A new client process created along with the unique FIFOs ,created with child id, to establish further client commands.

Client FIFOs (Named Pipes):

Two FIFOs are created for each client: one for receiving data from the server to the client (***client_get_fifo)*** and one for sending data ***(client_send_fifo)*** from the client to the connected server .
These FIFOs enable bidirectional communication between the server and each client. They are unique to each client and used for sending commands from the client to the server and receiving responses or data from the server.

Shared Memory Usage

Shared memory is utilized in this project to store a queue data structure that manages client connections. The server and client processes attach to the shared memory segment, allowing them to access and modify the queue contents. This shared memory approach enables efficient communication between the server and clients, as they can exchange information without the overhead of inter-process communication mechanisms like FIFOs. Additionally, shared memory facilitates seamless data sharing and synchronization between multiple processes, enhancing the scalability and performance of the system.

Semaphores

Semaphores are used for synchronisation, specifically for controlling access to shared resources. When a client connects or disconnects, it increments or decrements the semaphore respectively, ensuring that only one process accesses the shared memory at a time. Therefore, the structure prevents race conditions and ensures data integrity when multiple clients interact with the server simultaneously.

Semaphore Initialization:

A semaphore is initialised using sem_open() with a unique name based on the server's process ID. This semaphore is used to control access to critical sections where shared resources are accessed or modified.

Semaphore Usage:

The server will call *sem_wait* before attempting to connect with client. When a client wishes to connect to the server, it will increment the semaphore value, allowing the server to proceed past *sem_wait*. The client will then write its PID to the FIFO, which the server can read since the semaphore value has been incremented.
Clients wait for the semaphore to be posted by the server before proceeding with their requests using sem_wait().
The server posts the semaphore after processing each client request using sem_post() to allow the next client to proceed.

Queue Struct:

A queue data structure is implemented and stored in shared memory to manage client connections. The queue struct consists of an array to hold client PIDs and a counter to track the number of connected clients. When a client connects, its PID is enqueued (added) into the shared queue, and when it disconnects, its PID is dequeued(removed).
This queue structure ensures fair handling of client connections and allows the server to prioritise requests based on arrival order. Since the order and the amount of connected clients is stored, if the max number of allowed client count reached no new incoming client can connect to server. By storing the queue in shared memory, multiple server processes can access and manipulate it concurrently, facilitating efficient client management.

Signal Handling

Signal handling is implemented to manage proper termination of the processes and clean-up of resources.
Specifically, the server process handles SIGINT (Ctrl+C) and SIGTSTP (Ctrl+Z) signals using the sigaction function. When either signal is received, the server process invokes a signal handler function (handle_kill_signal) to perform cleanup tasks before exiting. These tasks include closing open FIFOs, freeing shared memory, and removing semaphores. By handling these signals, the server ensures proper shutdown procedures and resource deallocation, maintaining system integrity and stability.

Implementation Details

Server Side:

First of all, the command-line arguments are parsed to extract the directory name and the maximum number of clients allowed. It checks the validity of the arguments.
If argos are valid, semaphores, shared memory, log files, and the server FIFO are initialised. These initializations are crucial for establishing communication and synchronization between the server and its clients.
Then, the server enters a loop to listen the incoming connection requests from clients.
The server FIFO is continuously monitored for incoming requests. When a client wants to connect, it increments the semaphore value, signaling the server that it is ready to establish a connection. The server then reads the client's PID from the FIFO, allowing it to identify the client.
Once a client is accepted for connection, it is added to a queue. This queue acts as a buffer, allowing the server to manage multiple client connections concurrently while maintaining synchronisation and preventing race conditions. If the queue is full, the server informs the client that it cannot accept further connections at the moment.

After a successful connection, for each client a separate child process is forked to ensure concurrent operation. Within the child process, the client's requests are processed according

to the commands it sends. These commands include listing files in the server directory, reading from and writing to files, uploading and downloading files, and archiving server files.

The server uses inter-process communication mechanisms to facilitate communication between the server and its clients. FIFOs are opened for bidirectional communication between the server and each client, allowing them to exchange messages and data.

Semaphores are employed for synchronization, ensuring that resources are accessed safely and that multiple clients can interact with the server simultaneously without conflicts.

Additionally, shared memory is used to maintain a queue of connected clients. This shared memory region allows the server and its child processes to access and update the client queue efficiently, enabling effective management of client connections.

For the whole program, signal handling is implemented with signal handler function to handle termination requests gracefully. SIGINT, SIGTERM, SIGTSTP, and SIGQUIT signals are handled. If the server receives a termination signal (e.g., SIGINT), it ensures that all child processes are terminated, resources are cleaned up, and the server exits gracefully, ensuring data integrity and proper shutdown.

Client Side:

First the command-line arguments are cotroled to ensure that the user provides the correct input, namely the connection option ('connect' or 'tryConnect') and the server's process ID. If the arguments are invalid, an error message is displayed, and the program exits.

After parsing the command-line arguments, the client initialises signal handlers to capture termination signals (SIGINT, SIGTERM, SIGTSTP, SIGQUIT) gracefully. Therefore it’s ensured proper cleanup of resources and termination of the program upon receiving termination signals.

Next, the client creats FIFOs with unique ids of the child process for bidirectional communication with the server. One FIFO is used for sending messages to the server (client_fifo_name_send). The other FIFO is used for receiving messages from the server (client_fifo_name_get). These FIFOs facilitate communication between the client and the server, enabling them to exchange commands and responses.

The client then initializes a semaphore to synchronize with the server. The semaphore ensures that the server processes incoming client connections in a controlled manner, preventing race conditions and resource conflicts.

After setting up the communication channels and synchronization mechanisms, the client sends its process ID (PID) to the server using the server's FIFO. This step establishes a connection between the client and the server, allowing the server to identify and communicate with the client.

Once the connection is established, the client enters a loop to handle user commands continuously. It prompts the user to enter a command and reads the input from the standard input (stdin). The input is then parsed into individual commands, which are processed based on their type.

For each command, the client sends the command to the server through the send FIFO (*client_fifo_fd_send*). Upon receiving a response from the server through the get FIFO (*client_fifo_fd_get*), the client processes the response accordingly and displays the output to the user.

The commands are handled:
'help' command invokes the 'handle_common()' function, which sends the command to the server and displays the server's response, providing users with assistance and guidance on available commands.
'archServer' command triggers the 'handle_archServer_command()' function, which sends the command to the server and retrieves information about the server's file structure, aiding in server management tasks.
'quit' execute the 'handle_quit_kill_command()' function, gracefully terminating the client-server connection and ensuring clean exit from the program.
'killServer' send kill request to server and on serverside the program terminates and the resources are cleaned.
'list' command, managed by 'handle_common()', requests a list of files from the server, facilitating file navigation and exploration.
'readF', interact with 'handle_readf_command()' allowing users to read file contents from the server.
'upload', 'handle_upload_command()' upload files to the server.
'download' interact with 'handle_download_command()'download files from the server to the client's local system.
Each command handler function ensures proper communication with the server, error handling, and display of relevant information to the user

In case of errors or unexpected server responses, the client displays error messages or notifies the user of any issues encountered during command execution.

Finally, the client cleans up resources, closes FIFOs, and unlinks FIFO files before exiting the program. This ensures proper cleanup and prevents resource leaks.

Test Results

```
>> Enter comment: help

Available comments listed as :

help, list, readF, writeT, upload, download, archServer, quit, killServer
```

```
>> Enter comment: help writeT


writeT <file> <line #> <string>
request to write the content of "string" to the #th line the <file>, if the line # is not g
iven writes to the end of file. If the file does not exists in Servers directory creates an
d edits the file at the same time
```

```
>> Enter comment: help archServer


archServer <fileName>.tar
  Archives the files that are available on the Server. Stores them in the <filen
ame>.tar
```

```
(base) hilal@hllLinx:~/Desktop/2024Spring/cse344/midterm/mid$ ./neHosServer Here 3
> neHosServer Here 3

>> Server Started PID 32091...
>> waiting for clients...
serpid: /tmp/neHos_server_fifo_32091
>> Client PID 32121 connected as "client01"
>> Client PID 32160 connected as "client02"
>> Client PID 32196 connected as "client03"
>> Connection request PID 32237... Queue FULL
```

```
>> Enter comment: upload client.txt
File transfer request received. Beginning file transfer:
21 bytes transferred

>> Enter comment:
```

```
>> Enter comment: archServer ttt.tar
Archiving the current contents of the server...
Creating archive directory...
/6/files downloaded...


SUCCESS Server side files are achived in>>/ttt.tar


>> Enter comment: readF new.txt
2121234
434
1112
1212000
777777tttttttt
mmmmmm
enened
endned
newline atend
```

```
(base) hilal@hllLinx:~/Desktop/2024Spring/cse344/midterm/mid/2001
m_code$ ./neHosClient connect 13889
>> Waiting for Queue...Connection established:
>> Enter comment: list
server_log.txt

>> Enter comment: upload client.txt
File transfer request received. Beginning file transfer:
21 bytes transferred

>> Enter comment: dowload client.txt


Invalid command


>> Enter comment: download client.txt
Download Success 0 bytes stored

>> Enter comment: upload client.txt
File transfer request received. Beginning file transfer:
58 bytes transferred

>> Enter comment: download client.txt
```