GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING FACULTY

CSE 344 SYSTEM PROGRAMMING

MIDTERM REPORT

ENES AYSU
1901042671

Program Description

The Concurrent File Access System is a server-client architecture designed to enable multiple clients to connect, access, modify, and archive files stored in a specific directory on the server side. The system comprises two main components: the server-side program, neHosServer, and the client-side program, neHosClient.

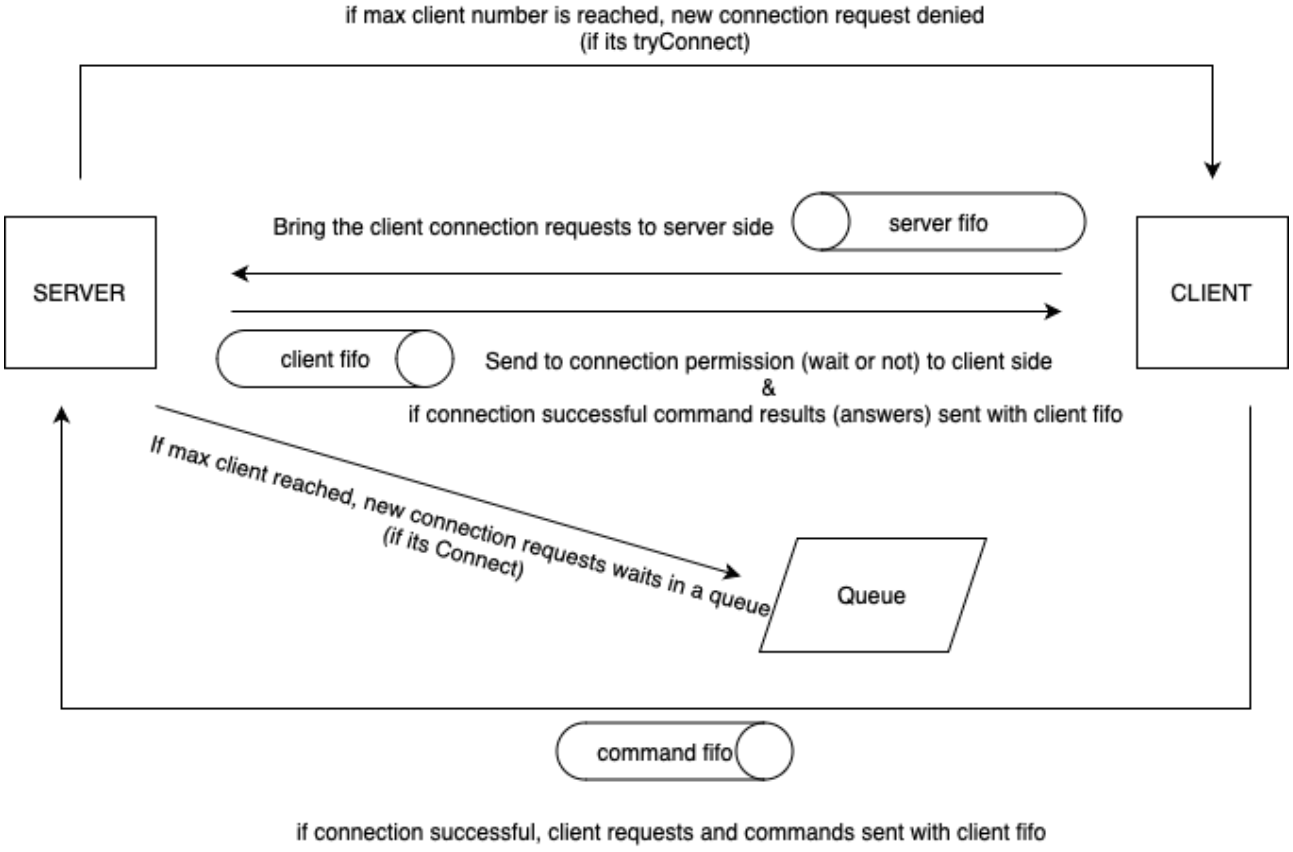
On the server side, neHosServer is responsible for managing client connections, creating a log file for client activities, and handling client requests. It supports concurrent access by forking processes for each connected client, ensuring mutual exclusion to prevent race conditions and data corruption. The server monitors client activity, displays relevant information, and gracefully handles termination signals.

On the client side, neHosClient facilitates client-server communication by requesting connection spots from the server and executing various commands to interact with files on the server. Clients can list files, read specific lines or entire contents of files, write to files, upload/download files, archive server-side files, send kill signals to the server, and gracefully quit while logging activities.

The system ensures data consistency and integrity by enforcing mutual exclusion during file access and synchronization, allowing for concurrent read and write operations. It supports large and diverse file formats, providing robust file I/O capabilities. Signals are handled efficiently on both client and server sides to maintain system stability and responsiveness.

The project implements a comprehensive solution for concurrent file access, addressing requirements such as handling multiple processes, enforcing mutual exclusion, ensuring data consistency, and supporting various file operations. Through rigorous testing, the system demonstrates its ability to handle concurrent access scenarios while maintaining data integrity and reliability.

System Design



Function and Processes (neHosServer.c)

1. `cleanup()`:

- This function is crucial for cleaning up any created FIFO files before exiting the program. It's called in case of an error or upon program termination.

2. `error_exit(char *msg)`:

- This function is used to handle errors gracefully by printing an error message `msg` and then exiting the program with a failure status.

3. `writeToLogFile(const char *log_entry, int log)`:

- It writes log entries (`log_entry`) to the log file specified by the file descriptor `log`.
- It ensures that log entries are properly written to the log file to maintain a record of server-client interactions.

4. `handle_disconnect(int client_pid)`:

- This function handles the disconnect message from a client identified by `client_pid`.
- It notifies the server when a client disconnects and updates any necessary data structures.

5. `handle_signal(int signal)`:

- Signal handler function responsible for handling termination signals like SIGINT (Ctrl+C).
- It ensures that upon receiving a termination signal, the server cleans up resources and exits gracefully.

6. `helpRequest(const char *arg1, const char *arg2)`:

- This function generates help messages based on the arguments provided (`arg1` and `arg2`).
- It assists clients by providing information on available commands and their usage.

7. `listRequest(char *dirname)`:

- It generates a list of files present in the server's directory specified by `dirname`.
- This function facilitates clients in exploring the files available on the server.

8. `readFRequest(const char* filename, int line_number, char* dirname)`:

- This function reads the content of a file (`filename`) or a specific line (`line_number`) from the server's directory (`dirname`).
- It provides clients with the ability to retrieve file contents or specific lines from files.

9. `writeTRequest(const char* filename, int line_number, const char* content, char* dirname)`:

- It writes content (`content`) to a file (`filename`) either at a specific line (`line_number`) or appends it to the end.
- This function enables clients to modify or add content to files on the server.

10. `downloadRequest(char *filename, char *dirname)`:

- This function handles download requests by transferring a file (`filename`) from the server's directory (`dirname`) to the client.
- It ensures seamless file transfer between the server and the client.

11. `uploadRequest(char *filename, char *dirname)`:

- It handles upload requests by transferring a file (`filename`) from the client to the server's directory (`dirname`).
- This function enables clients to upload files to the server for storage or processing.

12. `archServer(const char* filename, char* dirname)`:

- This function creates a tar archive (`filename`) of all files present in the server's directory (`dirname`).
- It utilizes fork and exec to create a child process that executes the necessary commands to create the tar archive.

13. `doConnectedClientRequests(int client_pid, Node *head, char *dirname)`:

- This function processes requests from connected clients identified by `client_pid`.
- It handles various client requests concurrently and communicates with clients using FIFOs.
- It maintains a log of client-server interactions for tracking and auditing purposes.

14. `main(int argc, char *argv[])`:

- The main function orchestrates the server's operations.
- It initializes data structures, sets up signal handlers, and parses command-line arguments.
- It manages client connections, processes client requests, and ensures proper cleanup before exiting.

Function and Processes (neHosClient.c)

Header Files: The necessary header files are included for various system calls and functionalities.

Constants: Three constant strings FIFO_PATH, FIFO_PATH1, and FIFO_PATH2 are defined to hold paths for FIFO files.

Global Variables:

- connected_flag: Indicates whether the client is connected to the server or not.
- server_fd: File descriptor for the server FIFO.
-

Function Prototypes:

- cleanup(): Cleans up FIFO files before program termination.
- error_exit(): Prints an error message and exits the program.
- handle_signal(): Signal handler function to handle termination signals.
- main(): The main function where the client's logic is implemented.

cleanup(): Removes FIFO files (FIFO_PATH1 and FIFO_PATH2) using unlink().

error_exit(): Prints an error message using perror() and exits the program after cleanup.

handle_signal(): Handles termination signals, specifically SIGINT (Ctrl+C). It notifies the server if the client is connected before exiting.

main() Function:

- Parses command-line arguments to determine connection options and server PID.
- Sets up signal handling for SIGINT.
- Creates FIFOs (FIFO_PATH1 and FIFO_PATH2) for client-server communication.
- Connects to the server based on the provided connection option (Connect or tryConnect).
- Sends connection requests to the server, waits for a response, and establishes communication.
- Reads user input, sends it to the server through FIFOs, and displays server responses.
- Handles termination conditions such as user input of "quit" or "killServer".

Execution Steps of the Program

Server Execution Steps

Server Initialization:

- The server program starts executing.
- It creates a named pipe (FIFO) for receiving client connection requests.

Waiting for Client Connection:

- The server enters a loop where it continuously waits for incoming connection requests from clients.
- It listens on its named pipe (FIFO) for any new connection requests.

Client Connection Request Handling:

- When a client sends a connection request, the server accepts it.
- It establishes communication with the client.

Handling Client Requests:

- Once the connection is established, the server waits for commands or requests from the client.
- It reads incoming requests from the client through the FIFO.

Processing Client Requests:

- The server processes the received requests and executes the corresponding actions.
- It may perform various operations based on the received commands.

Sending Responses to Client:

- After processing the client's request, the server sends back the response to the client.
- It writes the response to the client's designated FIFO.

Continued Operation:

- The server continues to handle incoming requests from connected clients until either the client disconnects or the server is terminated.

Client Execution Steps:

Client Initialization:

- The client program starts executing.
- It creates two named pipes (FIFOs) for communication with the server.

(One for communicate with server, one for handling client requests)

Sending Connection Request to Server:

- The client sends a connection request to the server.
- It writes a message indicating its desire to connect to the server through the designated FIFO.

Waiting for Server Response:

- After sending the connection request, the client waits for a response from the server.
- It listens on its designated FIFO for a response from the server.

Connection Established:

- If the server accepts the client's connection request, the client receives an acknowledgment.
- It sets a flag indicating that it is connected to the server.

User Interaction:

- The client waits for user input or generates commands to send to the server.
- It prompts the user for input or generates commands based on its internal logic.

Sending Requests to Server:

- The client sends requests or commands to the server through its designated FIFO.
- It writes the commands or requests to the FIFO for the server to read.

Receiving and Processing Server Responses:

- After sending a request, the client waits for responses from the server.
- It listens on its designated FIFO for responses from the server.
- Upon receiving a response, it processes and displays the response to the user.

Continued Operation:

• The client continues to interact with the user, sending requests to the server and receiving responses, until either the user initiates termination or the server disconnects.

Tests

For Compilation

$ make

$ ./neHosServer <directory name> <max client size>

$ ./neHosClient <connect/tryConnect> <server PID>

Starting server side…

```
enesaysu@192 midterm % make
gcc -Wall -Wextra -g -c neHosServer.c -o neHosServer.o
gcc -Wall -Wextra -g -c guestQueue.c -o guestQueue.o
gcc -Wall -Wextra -g  neHosServer.o guestQueue.o -o neHosServer
gcc -Wall -Wextra -g -c neHosClient.c -o neHosClient.o
gcc -Wall -Wextra -g -c clientList.c -o clientList.o
gcc -Wall -Wextra -g  neHosClient.o clientList.o -o neHosClient
enesaysu@192 midterm % ./neHosServer midterm 2
>> Server Started PID 66080...
>> waiting for clients...
```

Starting client side then clien connects to the server

```
enesaysu@192 midterm % make
gcc -Wall -Wextra -g -c neHosServer.c -o neHosServer.o
gcc -Wall -Wextra -g -c guestQueue.c -o guestQueue.o
gcc -Wall -Wextra -g  neHosServer.o guestQueue.o -o neHosServer
gcc -Wall -Wextra -g -c neHosClient.c -o neHosClient.o
gcc -Wall -Wextra -g -c clientList.c -o clientList.o
gcc -Wall -Wextra -g  neHosClient.o clientList.o -o neHosClient
enesaysu@192 midterm % ./neHosServer midterm 2
>> Server Started PID 66616...
>> waiting for clients...
>> Client PID 66644 connected as 'client01'
```

```
enesaysu@192 midterm % ./neHosClient Connect 66616
> NehosClient connect 66616...
    >> Waiting for Que.. Connection established:
    >> Enter comment :
```

“help” command

```
enesaysu@192 midterm % ./neHosClient Connect 66616
> NehosClient connect 66616...
    >> Waiting for Que.. Connection established:
    >> Enter comment : help

Available comments are :

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

“list” command

```
enesaysu@192 midterm % ./neHosClient Connect 66742
> NehosClient connect 66742...
    >> Waiting for Que.. Connection established:
    >> Enter comment : list

.DS_Store
enes.txt
ahmet.txt
asude.txt
eren.txt
```

“readF” command

```
    >> Enter comment : readF enes.txt 1
"In the heart of the bustling city, where the neon lights never sleep and
 the streets pulse with life, there exists a world within a world. It is
a place where dreams are made and broken, where fortunes are won and lost
 with the turn of a card or the roll of the dice.
```

“writeF” command

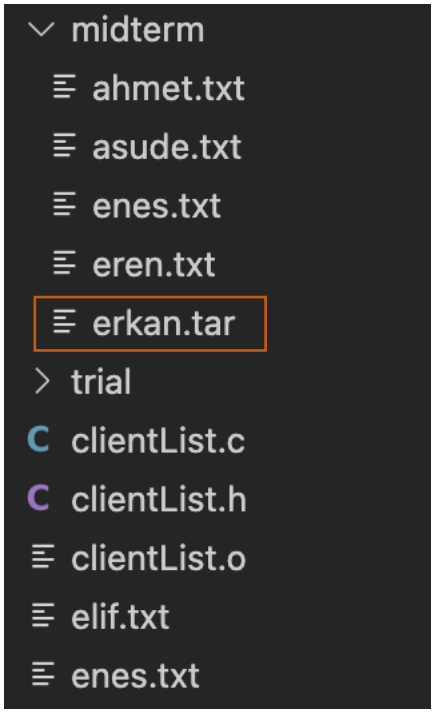
```
enesaysu@192 midterm % ./neHosClient Connect 67173
> NehosClient connect 67173...
    >> Waiting for Que.. Connection established:
    >> Enter comment : writeT ahmet.txt 1 hava
Content "hava" written to the 1th line of the file "ahmet.txt"
    >> Enter comment : writeT ahmet.txt 2 nasıl
Content "nasıl" written to the 2th line of the file "ahmet.txt"
```

```
1  hava
2  nasıl
3
```

“download” command

“upload” command

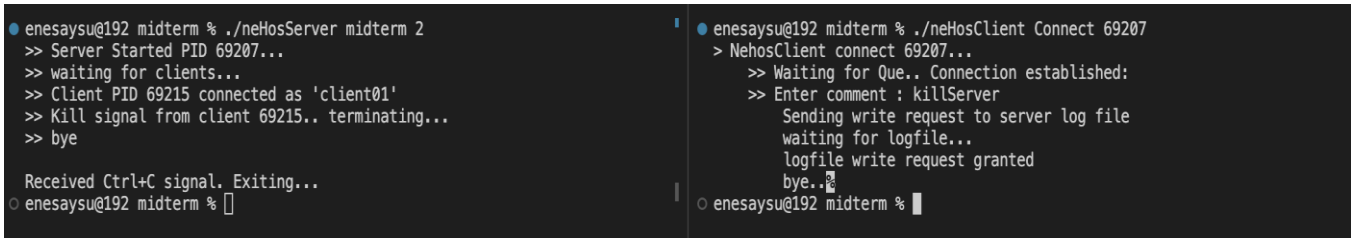
“archServer” command



“quit” command

```
enesaysu@192 midterm % ./neHosClient Connect 67909
> NehosClient connect 67909...
    >> Waiting for Que.. Connection established:
    >> Enter comment : quit
        Sending write request to server log file
        waiting for logfile...
        logfile write request granted
        bye..%
enesaysu@192 midterm %
```

“killServer” command



log file

```
1   Client PID 69715 connected as 'client01'
2   Client PID 69715 requested 'help'
3   Client PID 69715 requested 'list'
4   Client PID 69715 requested 'readF'
5   Client PID 69715 requested 'download'
6   Client PID 69715 requested 'upload'
7   Client PID 69715 requested 'quit'
```

Summary

This project presents a client-server communication system implemented in C using named pipes (FIFOs). The server component awaits connection requests from clients, which are then handled accordingly. Clients, upon initiation, send connection requests to the server and wait for confirmation before proceeding. Once connected, clients can send various commands or queries to the server, initiating interactions. The server processes these requests, executes commands, and generates responses, establishing a dynamic exchange of information between clients and the server. Robust error-handling mechanisms are incorporated to manage potential issues, ensuring smooth execution even in adverse conditions. Additionally, the program includes signal handling to gracefully terminate processes upon receiving termination signals. This project lays the groundwork for understanding fundamental client-server communication concepts and serves as a starting point for developing more sophisticated networked applications with enhanced scalability and functionality.