Operating Systems (CSL 3030)

Readme file

Lab Assignment	**02**
Roll No.	**B20CS018**
Name	**Harshita Gupta**

PART I:

To run the driver.c file you have to open the terminal and give the below commands –

1. gcc driver.c -o driver (to compile the code)
2. ./driver
3. Enter the file path

This will open the given file by the user and will create a second file in which output will be stored.

Output –

The output at various stages (passing from parent to child, receiving from a parent, passing from child to parent, receiving at parent finally).

In the final output, the characters will be swapped and the count of non-alphabetic will be printed, which implies that the code is working fine.

To implement the pipe we have used pipe() function, and fork() function is used for process creation. The pipe function gives the negative value if it does not creates a pipe, and the fork function also does the same if it is not able to create processes.

Pipe[0] is the reading end and pipe[1] is the writing end. If the value from the fork function is 0 it means, it is a child process and we can write the commands for the child's work in this section of code, and if the value from the **fork function is > 0**, it means a parent process and we can write the commands for parent's work in this section of code. This whole thing is used in the code for the implementation of pipe and process creation.

The close() system command is called two times in code –

1. In the child part of the code (pid==0), the close command is called at the end to ensure closing the writing end of the pipe (mypipe[1]) because till now everything

that has to be written in the pipe has been done and only receiving is left at the parent's end and hence,

2. In the parent part of the code (pid>0), the close command is called at the end to ensure the closing of the receiving end of the pipe (mypipe[0]) because till now everything that needs to be read is also completed.

If we have to redirect the file descriptor to any of the 0, 1, 2 (input, output, stderr) or any other file descriptor, then we could have used the dup() command as per the need.

For the given ques, the code's flow of control is like this –

Main function → pipe() creation → fork() creation → parent process → write to pipe → wait() for a child process to execute → child process → read from pipe → make changes to the string read → write to pipe → close the writing end → parent process → read from pipe → close reading end → return 0 (function end…)

PART II:

The question is implemented by the approach of Inter-process communication (IPC) with use of Linux message queue and threads. The project consists of two programs that run independently: the server and the client.

By using the mechanism of the message queue, the task is implemented with the ability to support multiple clients at the same time. Used one queue in the solution messages. Used appropriate message labeling to distinguish between the queue data for the server and data for individual clients.

The server creates a message queue and waits for messages from clients. The server removes the queue after receiving a user-defined signal (eg SIGINT).

The client is a multi-threaded application. Thread 1 is responsible for sending messages and thread 2 for receiving messages from the server. Threads work asynchronously, for example: thread 1 can send several messages before thread 2 receives any feedback from the server.

There is also a need to handle a queue overflow and a message overflow error. For system functions - implement error handling based on the perror() function and the errno variable.

The server and client programs need to be compiled with the -lrt option. First, the server is run. Then one or more clients can be run for playing the game. Below is an example of how the program will execute.

Running the server,

```
$ # server
$ gcc server.c -o server -lrt
$ gcc client.c -o client -lrt
$ ./server

Enter max number: 10
Enter min number: 5

Server: Received token R1
Server: Received token R2
Server: Received token R3

Round 1
Range 5 - 10
R1 - 7, R2 - 9, R3 - 8,....
Score - 0, 5, 0, …
```

Running the client,

```
$ ./client

Client 1: Range received from the server is 10-5
Client 1: Token sent to the server, R1

Client 2: Range received from the server is 10-5
Client 2: Token sent to the server, R2

Client 3: Range received from the server is 10-5
Client 3: Token sent to the server, R3
…
```