|  |  |
| --- | --- |
| **Inter-process communication (IPC)**  107177 – OS Mini Project Report Summers 2021 | **Submitted by:**  Syed Muzzamil Waseem (11067) Sohaib Arshad Siddiqui (11042)  **UNDER SUPERVISION OF:**  Ma’am Saboohi |

**TABLE OF CONTENT**

1. **Acknowledgement………………………………………………………………………. 3**
2. **Introduction…………………………...…………………………………………………… 4**
3. Overview…………………………………………………………………………… 4
4. Motivation………………………………………………………………………….. 4
5. User Interface……………………………………………………………………... 4
6. **Needs and Problems…………………………………………………………………….. 5**
7. **Goals and Objectives…………………………………………………………………… 7**
8. **Implementation…………………………………………………………………………… 8**
9. Pipes (Unnamed Pipes)………………………………………………………….. 8
10. FIFO (Named Pipes)………...……………………………………………….…. 10
11. Message Passing………………………………………………………………… 12
12. **Conclusion……………………………...………………………………………………... 15**
13. **Bibliography……………………………………………………………………………… 16**

**ACKNOWLEDGEMENT**

We have taken efforts in this project. However, it would not have been possible without the kind support and help of our teachers. I would like to extend my sincere thanks to all of them.

We are highly indebted to our teachers, Ma’am Umm-e-Kulsoom and our Lab Prof. Kazim Ali for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

**INTRODUCTION**

**Overview**

Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.

Processes can communicate with each other through both: Pipes and Message Passing. By providing a user with a set of programming interfaces, IPC helps a programmer organize the activities among different processes.

**Motivation to do this project**

The motivation for doing this project was primarily an interest in undertaking a challenging project in an interesting area of research.

**User Interface**

This project was developed, built, and tested on Ubuntu Desktop 20.04.2 LTS. Pipes, FIFO and message-passing will be implemented using the C programming language using GCC. Thus, two interprocess communication methods (pipes, FIFO and message-passing) will be shown.

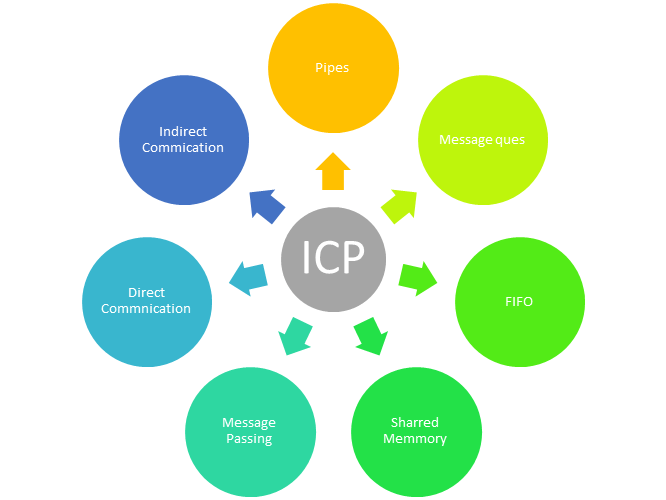
**NEEDS AND PROBLEMS**

**Needs**

**IPC** is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other.

Here, are the reasons for using the interprocess communication protocol for information sharing:

1. It helps to speedup modularity.
2. Computational.
3. Privilege separation.
4. Convenience.
5. Helps operating system to communicate with each other and synchronize their actions.

**Problems**

IPC requires the use of resources, such as memory, which are shared between processes or threads. If special care is not taken to correctly coordinate or synchronize access to shared resources, a number of problems can potentially arise such as:

1. Starvation
2. Deadlock
3. Data Inconsistency
4. Shared Buffer Problem

**GOALS AND OBJECTIVES**

Interprocess communication is the mechanism by which multiple processes can communicate with each other. This is one of the important features provided by the operating system.

1. **Information sharing:** Many processes will be interested in the same piece of information (a shared file or a library).
2. **Computational speedup:** We can break a task into subtasks and let each subtask run on a separate processor. These tasks can then use IPC for exchanging information. This makes the program run fast.
3. **Modularity:** A program can be divided into multiple chunks of code each performing a specific function. Maintaining and debugging the code will then be easy.
4. **Convenience**: Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

**IMPLEMENTATION**

1. **Pipes**

**Setup:**

The setup for the Unix pipes requires a single file called *pipes.c*. The program can be built using GCC 7 using the following command: *gcc pipes.c -o pipes*. The code for *pipes.c*.

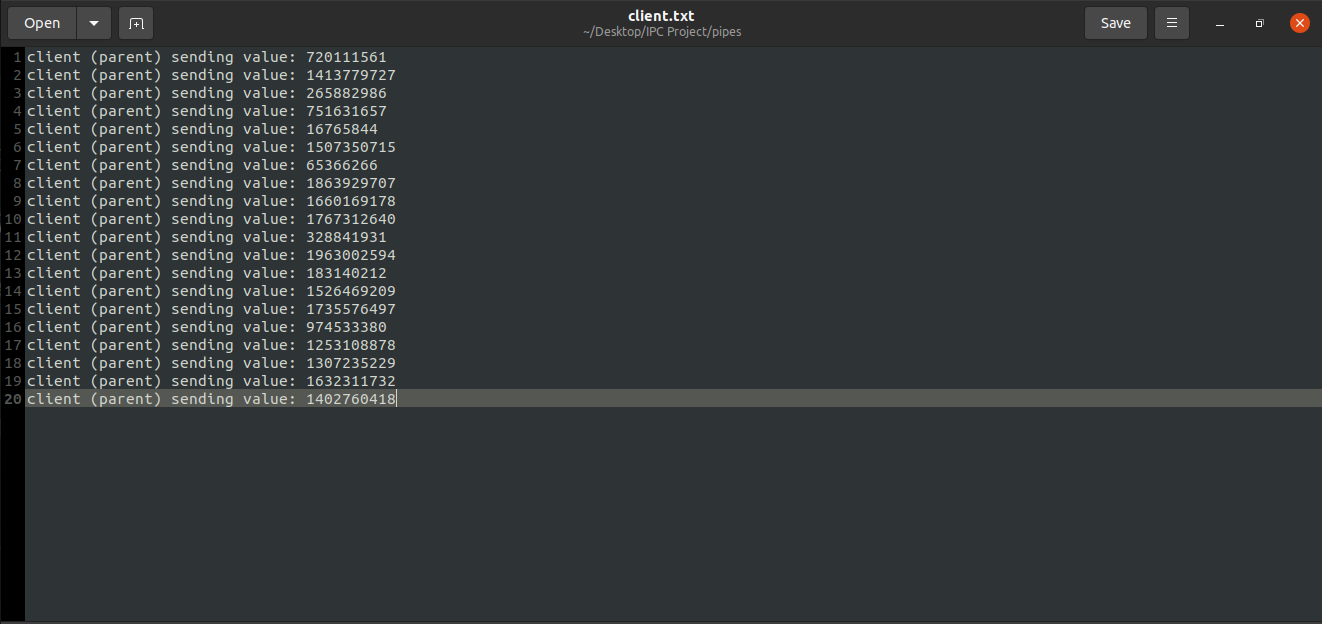
**Design:**

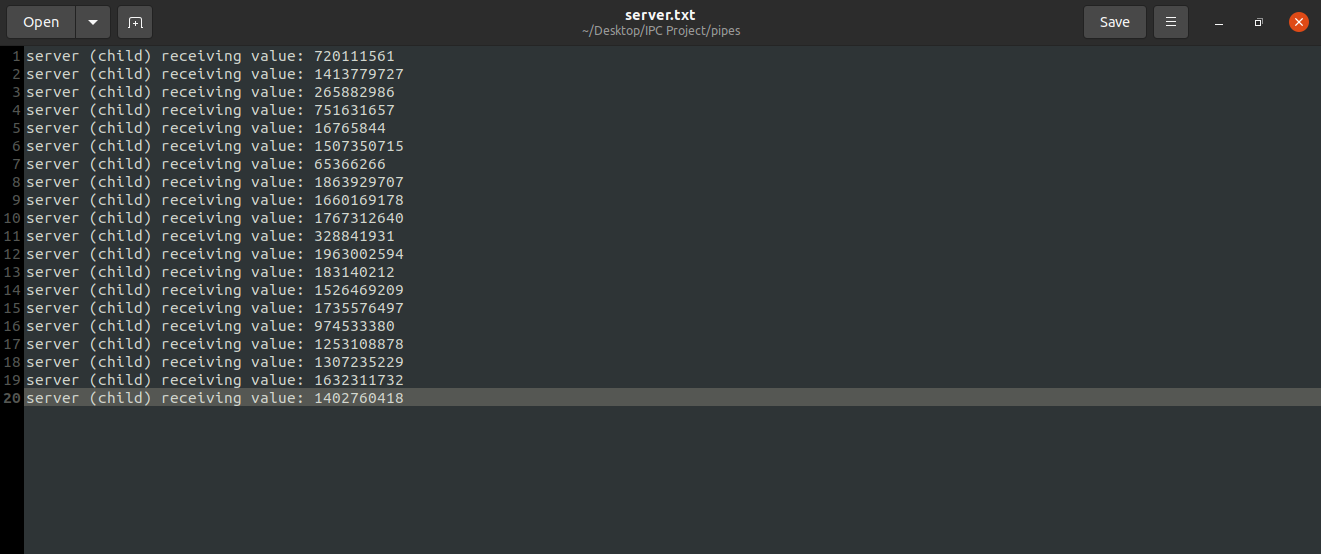
The program begins by initializing the random generator seeded with the current time. The program initializes the pipe that will be used to pass data. Then the program forks into the client and server process.

The parent client process will execute the code. The parent client process will create an integer value called *parent* and a *file* pointer called file. The file *client.txt* will be used to store the values created by the parent client process. Next, the parent producer process closes the read end of the pipe. The for-loop executes the code twenty times. Each time the loop is executed, a new pseudo-random number is generated and stored in *parent*. That value is then written to the pipe. Then the random number is written to the output file using *fprintf*. Once the loop exits, the *client.txt* file is closed using *fclose*. Finally, the write end of the pipe is closed which inserts a *EOF* value into the pipe. This *EOF* value will cause the server to exit its loop. Finally, the parent process waits for the child server process to terminate.

The consumer will execute the code. The server process opens a file for storing the values and read from the pipe. The server also creates an integer variable *child* to store a value from the pipe. Next, the server process closes the write end of the pipe. Then begins a loop that executes until the read is zero. The function read returns zero when the *EOF* value has been read from the pipe. Then we close the server.txt file, and the read-end of the pipe, which concludes its execution.

**Results:**

*client.txt* file for pipes. Sending values of client are stored here.

*server.txt* file for pipes. Receiving values from client are stored here.

All the random numbers received in *server.txt* file matches the numbers sent by *client.txt* file, which means a successful communication through pipes was established.

1. **Named Pipes (FIFO)**

**Setup:**

The setup for the Unix named pipes requires two C files called *client.c* and *server.c*. The program can be built using GCC 7 using the following commands:

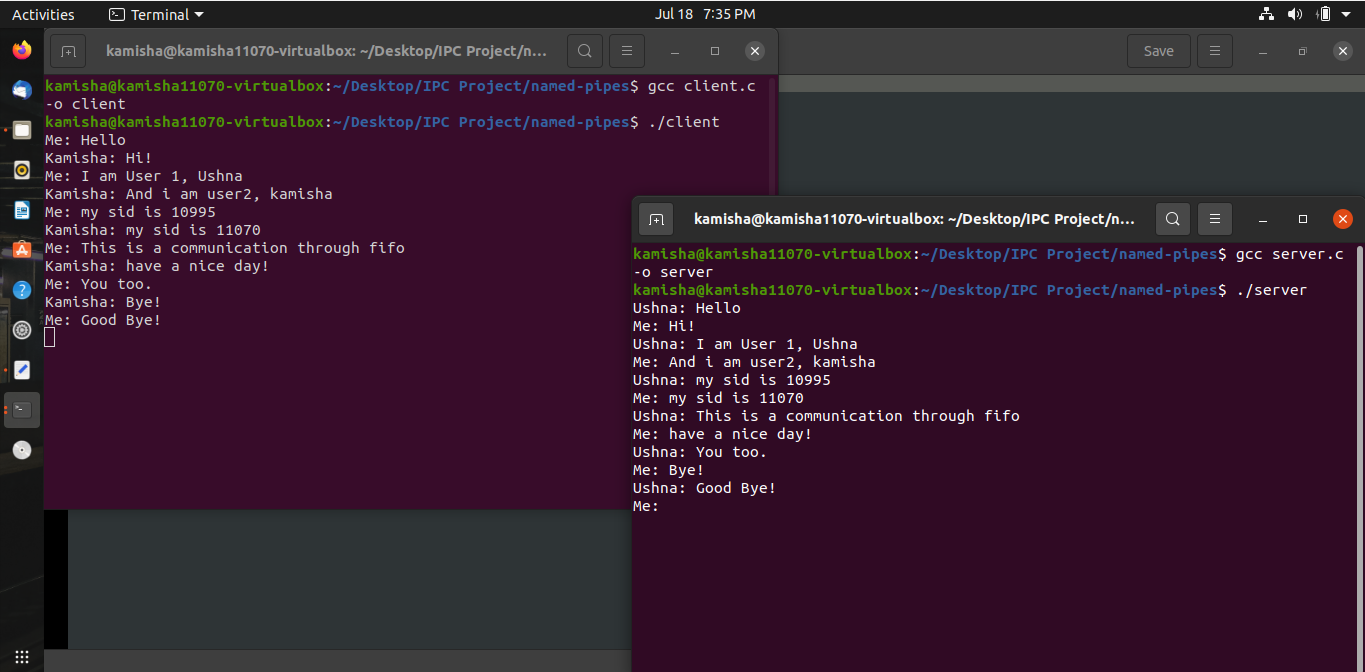
1. *gcc client.c -o client*. The code for *client.c*.
2. *gcc server.c -o server*. The code for *server.c*.

**Design:**

We first give FIFO its file path and then will create a named file FIFO by using *mkfifo,* give its path name and permissions. We then initialize two *char* variables; *arr1* and *arr2*, both can store up to 100 characters. Then we open the FIFO file for write only by using *O\_WRONLY.* By using *fgets* and file descriptor *stdin,* we then take input from the user in *arr2* on FIFO and close it by *close(fd).* Then we open the FIFO file for read only by using *O\_RDONLY*. The program then reads from FIFO and prints the message sent by the server end. Finally, the program closes the read end of the pipe too which concludes the client code.

Now, on server end, we again give FIFO the same file path we gave on client end and will create a named file FIFO, give its path name and permissions. Here, we also initialize two *char* variables; *str1* and *str2,* storing up to 100 characters. Then we open the FIFO file for read only by using *O\_RDONLY* and print the read string and close the pipe. Now we open the write end of the pipe by using *O\_WRONLY*. Then take input *str2* from user on FIFO. Finally, we close the write end of pipe, which concludes the server code.

**Results:**

By executing the two codes for client and server, a chat app like two-way communication is established on two different terminals between the client and the server by using FIFO (named pipes).

1. **Message Passing**

**Setup:**

The setup for message passing requires a single C file: *message-passing.c*. The file *message-passing.c* can be compiled using the following command: *gcc -Wall message-passing.c -lrt -o message-passing*.

**Design:**

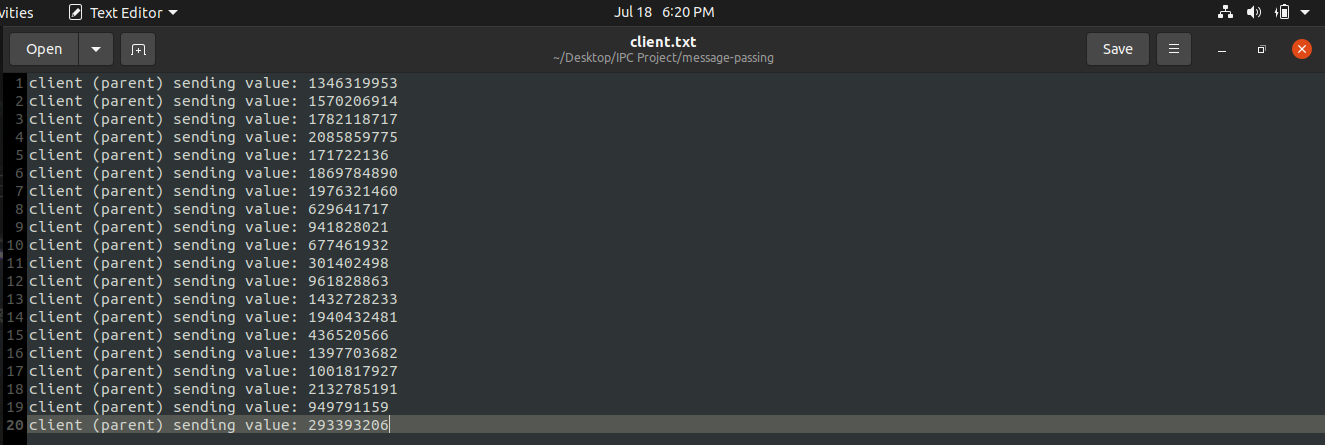
We first define values representing the two message types. Then we defined a struct, *message\_t*, to hold the message type and the integer value. After, we defined a union, *packet\_t*, used to pass message structs from producer process to consumer process.

We then initialize the message queue. The *attr* variable is used to define the attributes of the message queue. The queue is defined as having a size of 10 messages using *mq\_maxmsg*, and each message is defined as having the size of one *message\_t* data type using *mq\_msgsize*. Next, the message queue is opened using *mq\_open*. Then, the pseudo random number generator is seeded with the current time. Finally, the program uses the function *fork()* to create the parent (client) process and child (server) process.

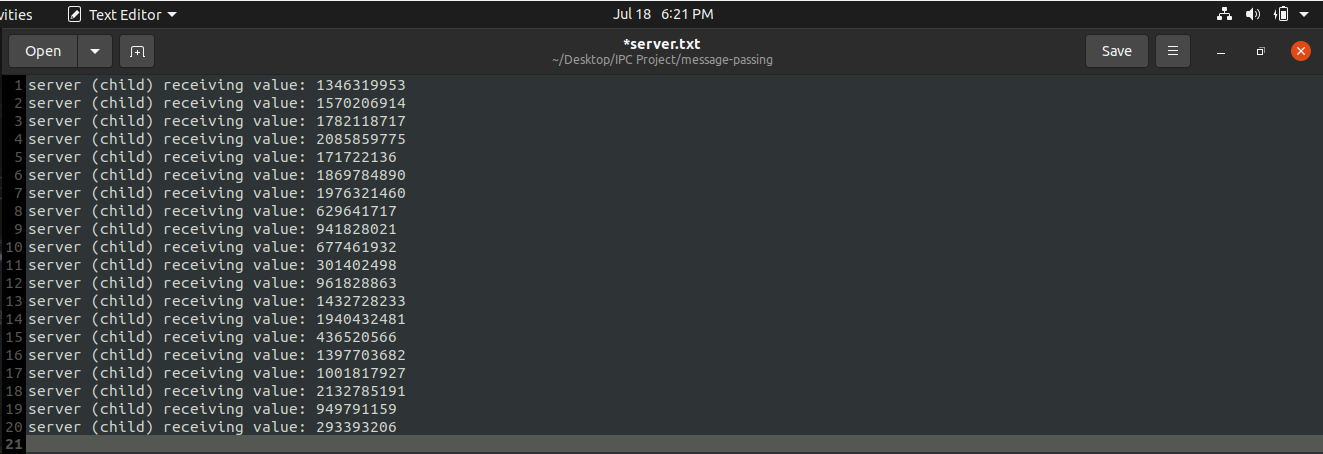
The parent client process executes the code contained in its function. It defines a packet struct to pass a message to the child (server) process. Then it uses *fopen* to create the file *client.txt*. Then the for loop executes the code twenty times. Each time the loop is executed, the message type of the packet is set to *MSG\_NUMBER*, and the message value is set to a randomly generated number by *rand*. Then the pack is sent using the *mq\_send* system call, and the value generated is written to *client.txt* using *fprintf*. Once the loop has completed, the client sends a final packet of *MSG\_EMPTY*. It then closes its message queue using *mq\_close* and closes the file *client.txt* with *fclose.* Finally, it uses the *wait* system call to pause execution until the child (server) process finishes.

The child server process, once created, will execute its function. The child server process defines a variable called packet, and then it opens a text file called *server.txt*. Next the child process reads a message from the queue using the *mq\_receive* communication system call. The loop will run until the message received from the queue is of type *MSG\_EMPTY.* Each time the loop is run, it first stores the received value to *server.txt*; then, it reads from the queue again. After the loop exits, the child (server) process closes its connection to the queue using *mq\_close* and closes its text file using *fclose*. Finally, the message queue is destroyed using *mq\_unlink.*

**Results:**

*client.txt* file for message passing. Sending values of client are stored here.

*server.txt* file for message passing. Receiving values from client are stored here.



All the random numbers received in *server.txt* file matches the numbers sent by *client.txt* file, which means a successful message passing through queues was established.

**CONCLUSION**

In computer science, inter-process communication refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests. Many applications are both clients and servers, as commonly seen in distributed computing.

**BIBLIOGRAPHY**

GeeksforGeeks. (n.d.). *Inter Process Communication (IPC)* [Online]. Available: <https://www.geeksforgeeks.org/inter-process-communication-ipc/>

TutorialsPoint. (n.d.). *What is Interprocess Communication?* [Online]. Available: <https://www.tutorialspoint.com/what-is-interprocess-communication>

Guru99. (n.d.). *Inter Process Communication (IPC)* [Online]. Available: <https://www.guru99.com/inter-process-communication-ipc.html>

W3schools. (n.d.). *Inter Process Communication (IPC)* [Online]. Available: <https://www.w3schools.in/operating-system-tutorial/interprocess-communication-ipc/>