TRANSPORT AND
TELECOMMUNICATION
INSTITUTE

Operating Systems and System Programming

Topic: Inter-Process Communication (IPC) Demonstration Report

Student’s name: Artjoms
Student’s surname: Voroncovs
St. code: st83663

RIGA

TABLE OF CONTENTS

Task Description

The purpose of this assignment is to demonstrate Inter-Process Communication (IPC) in Linux using two independent processes that exchange data via a named pipe (FIFO).
The first process (sender) transmits data to the second process (receiver), which processes the data and returns the result through another named pipe. Both processes should display detailed information about their execution — process IDs, parameters, and communication steps.

The IPC mechanism must be implemented using named pipes (FIFOs) created with the mkfifo() system call. This solution uses unnamed pipes and demonstrates a reusable, filesystem-based communication channel.

Reflection of the Solution

1. Description of Program Operation

Two applications were developed:

- processA.c – The main process that sends a number to another process through a named FIFO (fifo1), then waits for the processed result through another FIFO (fifo2).
- processB.c– The secondary process that reads the data from fifo1, performs a simple processing operation (e.g., reverses the string or doubles a number), and sends the result back via fifo2.

The data exchange can be fully duplex - the communication path works both ways through two distinct FIFOs. In that case we use communication only in one direction.
Each process reports its PID, actions performed, and the data being sent or received.

2. Justification of Chosen Mechanism

Named pipes were chosen for the following reasons:

- They allow communication between unrelated processes, unlike unnamed pipes which only work between parent-child relationships.
- FIFOs are persistent in the filesystem (created using mkfifo()), which simplifies testing and debugging.
- They are simple and efficient for linear message-passing communication (text or binary).
- They rely solely on Linux system calls and do not require message queues, sockets, or shared memory setup.

System calls used:

- mkfifo() – create named pipe.
- open(), read(), write() – exchange data between processes.
- getpid() – display process IDs.
- perror() – handle system-level errors gracefully.

3. Algorithm Description

Sender Process Algorithm (processA.c)

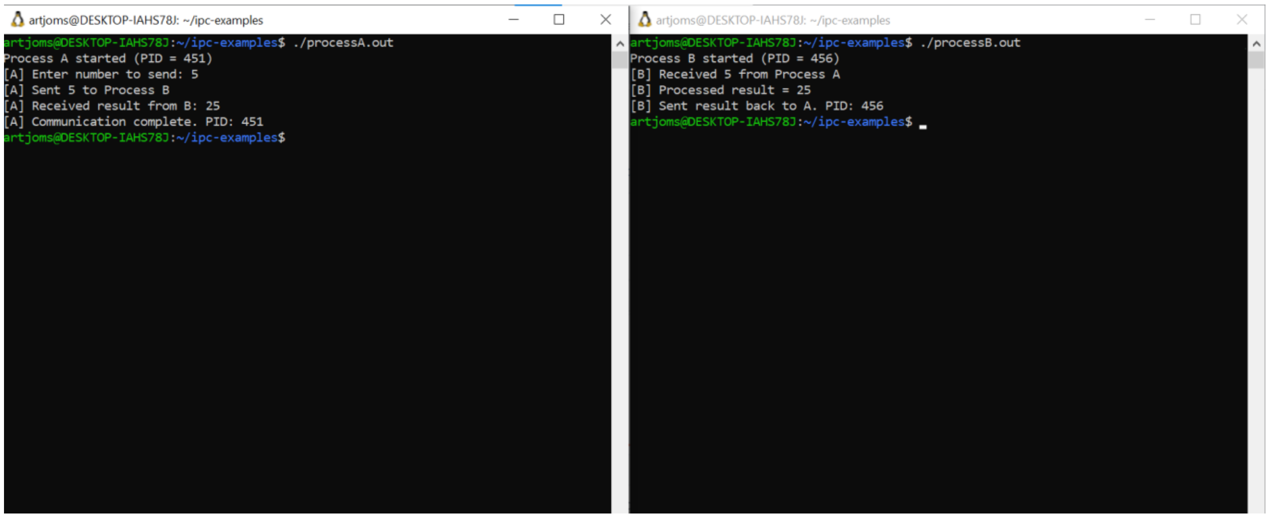
1. Create two FIFOs (fifo1 for sending data, fifo2 for receiving data).
2. Display PID and initialization messages.
3. Open fifo1 for writing and fifo2 for reading.
4. Send input data to the receiver.
5. Wait and read the response from fifo2.
6. Print the result and exit.

Receiver Process Algorithm (processB.c)

1. Open fifo1 for reading and fifo2 for writing.
2. Display PID and communication status.
3. Read the data received from fifo1.
4. Process the data.
5. Send processed result back through fifo2.
6. Close both FIFOs and exit.

4. Proof of Program Operation

Example run, processB.c will square number provided by processA.c.



artjoms@DESKTOP-IAHS78J: ~/ipc-examples

```
artjoms@DESKTOP-IAHS78J:~/ipc-examples$ ./processA.out
Process A started (PID = 451)
[A] Enter number to send: 5
[A] Sent 5 to Process B
[A] Received result from B: 25
[A] Communication complete. PID: 451
artjoms@DESKTOP-IAHS78J:~/ipc-examples$
```

artjoms@DESKTOP-IAHS78J: ~/ipc-examples

```
artjoms@DESKTOP-IAHS78J:~/ipc-examples$ ./processB.out
Process B started (PID = 456)
[B] Received 5 from Process A
[B] Processed result = 25
[B] Sent result back to A. PID: 456
artjoms@DESKTOP-IAHS78J:~/ipc-examples$
```

Both processes successfully exchange information and display detailed process information, proving the correctness of IPC through named FIFOs.

5. Proof of Completion

- Two independent processes developed.
- Communication established using named pipes.
- Data exchange is bidirectional.
- Detailed runtime output with PIDs and messages.
- Full code is compiled and tested on Ubuntu Linux.

Source Code

processA.c

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>


#define FIFO1 "fifo1"

#define FIFO2 "fifo2"


int main() {

    int num, result;

    pid_t pid = getpid();


    printf("Process A started (PID = %d)\n", pid);


    // Create FIFOs if not exist

    mkfifo(FIFO1, 0666);

    mkfifo(FIFO2, 0666);


    printf("[A] Enter number to send: ");

    scanf("%d", &num);


    int fd1 = open(FIFO1, O_WRONLY);

    write(fd1, &num, sizeof(num));

    close(fd1);
    printf("[A] Sent %d to Process B\n", num);


    int fd2 = open(FIFO2, O_RDONLY);
    read(fd2, &result, sizeof(result));
    close(fd2);
    printf("[A] Received result from B: %d\n", result);


    printf("[A] Communication complete. PID: %d\n", pid);
    unlink(FIFO1);
    unlink(FIFO2);
    return 0;
}
```

processB.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


#define FIFO1 "fifo1"
#define FIFO2 "fifo2"


int main() {
    int num, result;
    pid_t pid = getpid();


    printf("Process B started (PID = %d)\n", pid);
    mkfifo(FIFO1, 0666);
    mkfifo(FIFO2, 0666);

    int fd1 = open(FIFO1, O_RDONLY);
    read(fd1, &num, sizeof(num));
    close(fd1);
    printf("[B] Received %d from Process A\n", num);

    result = num * num;
    printf("[B] Processed result = %d\n", result);

    int fd2 = open(FIFO2, O_WRONLY);
    write(fd2, &result, sizeof(result));
    close(fd2);

    printf("[B] Sent result back to A. PID: %d\n", pid);
    return 0;
}
```

Conclusion

This assignment successfully demonstrates Inter-Process Communication using named pipes on Linux. The system allows for flexible and independent process interaction without shared ancestry.
All tasks are completed, and the mechanism is justified, reliable, and clearly observable through detailed console output.

Appendix

- https://github.com/ArtyomVorontsov/ipc-examples