
TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

**A Report**

**On**

# Inter Process Communication (IPC)

**Lab Sheet No: 3**

**Submission Date: January 28, 2022**

**Submitted By**

**Name:** Nischal Shakya

**Roll No:** 075BCT055

**Group:** BCT-C

**Submitted To**

Department of Electronics

and Computer Engineering

## Theory

### Inter Process Communication

Inter Process communication (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 interfaces which allows 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 communicating with each other. Each IPC protocol approach has its own advantages and limitations, so it is not unusual for a single program to use all of the IPC methods.

### Need for Inter Process Communication

There are several reasons for the communication. Some of them are:

1. **Information sharing**

   There may be a possibility that processes may need the same piece of information at the same time for their execution. For example, Copying and pasting.

2. **Communication speed**

   It is very important to have high speed in computation. If a task is subdivided into tasks which are executed parallely, it enhances the speed and throughput of the system.

3. **Modularity**

   Modularity means dividing the functions of the operating system into separate processes called threads. To achieve improved throughput.

**Pipes**

A pipe is a technique used for inter process communication. A pipe is a mechanism by which the output of one process is directed into the input of another process. Thus it provides one way flow of data between two related processes.

Although the pipe can be accessed like an ordinary file, the system actually manages it as a FIFO queue. A pipe file is created using the pipe system call. A pipe has an input end and an output end. One can write into a pipe from the input end and read from the output end. A pipe descriptor, has an array that stores two pointers, one pointer is for its input end and the other pointer is for its output end.

Suppose two processes, Process A and Process B, need to communicate. In such a case, it is important that the process which writes, closes its read end of the pipe and the process which reads, closes its write end of a pipe. Essentially, for a communication from Process A to Process B the following should happen.

1. Process A should keep its write end open and close the read end of the pipe.

2. Process B should always keep its read end open and close its write end. When a pipe is created, it is given a fixed size in bytes.

**Program 1**

**Output**

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog1
press a keyo
```

This will not work because the parent and child process has its own variables and they are not intercommunicating with each other. Hence as a result the value of exflag set by the child process is not seen in the parent process. As a result, the parent is stuck in an infinite loop because exflag is still equal to 0 for the parent process.

**Program 1 Solution**

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int exflag = 0;
void main()
{
    char c;
    int pfd[2];
    if (pipe(pfd) > 0) {printf("error");}
    if (!fork()) {
        printf("press a key");
        scanf("%c", &c);
        exflag = 1;
        write(pfd[1], &exflag, 1);
        exit(0);
    }
    else {
        read(pfd[0], &exflag, 1);
        while (!exflag);

        printf("I got the character");
        exit(0);
    }
}
```

**Output**

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog1_sol
press a keyt
I got the characternshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$
```

**Program 2**

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog2
I'm a child
Press any key to exit.....
o
child exiting
I'm parent
received o from child
parent exiting....
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$
```

In this program, pipe is used to transfer a character from the child process to the parent process. The pipe enables intercommunication between the parent and the child process. Also, in the above output we observe that the parent process finished execution earlier so the child process becomes orphan and its last print statement occurs at the new command line.

**Program 3**

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog3
hello one
hello two
hello three
```

In the given program, both ends of the pipe are used by the same process first to write some characters and then to read those written characters.

**Program 3 Solution (Read from parent and write from child)**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
const int msgsz = 16;
void main()
{
   char *msg1 = "hello one";
   char *msg2 = "hello two";
   char *msg3 = "hello three";
   char inbuf[msgsz];
   int p[2];
   pipe(p);
```

```
    if (!fork())
    {
        printf("\nMessage sent from child: %s", msg1);
        fflush(stdout);
        write(p[1], msg1, msgsz);
        printf("\nMessage sent from child: %s", msg2);
        fflush(stdout);
        write(p[1], msg2, msgsz);
        printf("\nMessage sent from child: %s", msg3);
        fflush(stdout);
        write(p[1], msg3, msgsz);
    }
    else
    {
        char inbuf[msgsz];
        for (int i = 0; i < 3; i++)
        {
            read(p[0], inbuf, msgsz);
            printf("\nMessage received in parent: %s", inbuf);
            fflush(stdout);
        }
    }
}
```

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog3_first

Message sent from child: hello one
Message sent from child: hello two
Message sent from child: hello three
Message received in parent: hello one
Message received in parent: hello two
```

**Program 4**

It is observed that the values of file descriptors are the same for both parent and child process as the child process was forked after assigning values to the file descriptor (since child and parent process are identical at the fork system call) and further change to that value doesn't occur in either of process.

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog4
in the parent p[0] is 3 p[1] is 4
in the child p[0] is 3 p[1] is 4
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
void main()
{
    int pfd[2];
    char data;
    if (fork())
    {
        printf("I’m parent\n");
        printf("press any key to exit……\n");
        scanf("%c", &data);
        write(pfd[1], &data, 1);
        wait(NULL);
        // anchor
        printf("Parent exiting\n");
    }
    else
    {
        read(pfd[0], &data, 1);
        printf("I’m child\n");
        printf("received %c from parent\n", data);
        printf("child exiting……\n");
        exit(0);
    }
}
```

**Output**

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog4_sol
I’m parent
press any key to exit……
I’m child
received  from parent
child exiting……
```

**Program 5**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
const int msgsz = 16;
void main()
{
    char *msg = "hello1";
    char inbuf[msgsz];
    int p[2], pid, j;
    pipe(p);
    printf("\nInitially errno: %d", errno);
    fflush(stdout);
    pid = fork();
    if (pid > 0)
    {
        close(p[0]);
        write(p[1], msg, msgsz);
        errno = 0;
        j = read(p[0], inbuf, msgsz);
        if (j == -1)
            printf("\nRead returned error in Parent, errno: %d", errno);
        fflush(stdout);
        wait(NULL);
    }
    else if (pid == 0)
    {
        close(p[1]);
        read(p[0], inbuf, msgsz);
        printf("\nRecieved message in child: %s", inbuf);
        errno = 0;
        j = write(p[1], msg, msgsz);
        if (j == -1)
            printf("\nWrite returned error in Child, errno: %d", errno);
        fflush(stdout);
    }
    else
    {
        printf("Error creating process");
    }
}
```

**Output**

```
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$ ./prog5

Initially errno: 0
Read returned error in Parent, errno: 9
Recieved message in child: hello1
nshakya@nshakya-Inspiron-13-5310:~/Documents/Os Lab/Lab3$
```

The read and write calls return the number of bytes received or sent but -1 in case of error. In the above program we since close was called for read file descriptor of pipe in parent process, it is no longer valid afterwards and similarly for write file descriptor of pipe in child side.

## Discussion

In this lab section, we learned about the intercommunication process that happens between various processes using the pipe. The pipe acts as a bridge between the two processes connecting them both at the read and write end. This allows for passing the data between the processes and hence allows for interprocess communication. Also if a process terminated but another process is still waiting for a message to be sent by the terminated process then it may wait indefinitely if the waiting process didn't close its writing file descriptor.

## Conclusion

Hence, inter-process communication between two independently running child and parent processes can be achieved by using read and write function calls intermediated through use of message pipe via pipe system call.