Operating System Lab No 5
Inter Process Communication (PIPES)

CLO: 2

Rubrics for Lab:

Task	0	1	2	3
As mentioned in the task	Student don't know about the concept of pipes and could not write the code	Student know the concept of pipe and do code some of the portion of the task	Student performed the task partially	Student performed the task completely

Topics to be Covered

- Inter process Communication
- Pipes in C
- Pipes Syntax
    - Read
    - Write
- Communication between multiple files

Objectives

Students are able to understand the Inter process

Communication IPC's and able to do communication between processes.

Inter-process Communication

As name suggests, sending messages or useful data between two processes. Shortly say IPC. There are different mechanisms available for IPC. This lab will focus on Process pipes mechanism.

Process Pipes

Piping is a process where the output of one process is made the input of another. LINUX users have seen examples of this from the UNIX command line using pipe sign '|'. For your experience just do a following quick activity.

Exercise 1:

Step1: Following command will simply show you the contents of directories and subdirectory's contents of current working directory. Try it out.

find .

Step2: Above command will show you whole output in one go. If you want to examine its output like file contents. Send output of Step2 to another command "more". You are already familiar with it in lab2.

Step3: Try following command.

find. | more

Pipe sign in above command is serving as "IPC pipe" taking output of find and sending it to more as input

Pipes in C

We will now see how we do this from C programs. We will have two forked processes and will communicate between them.

We must first open a pipe. UNIX allows two ways of opening a pipe.

i. Formatted Piping - popen()
ii. Low level Piping – pipe() We will use second one in this lab.

Note: Learn more about both in chapter 12, Beginning LINUX Programming by WROX – 2nd Edition or Google Pipe ()

Step 1:

man pipe

read how pipe work, get the necessary information before code.

pipe()

It's a low-level function to create a pipe. This provides a means of passing data between two programs, without any overhead. It also gives us more control over the reading and writing of data.

The pipe function has the prototype

```
#include <unistd.h>
int pipe(int file_descriptor[2]);
```

pipe() is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors

pipe() returns a zero on success. On failure, it returns −1 and sets errno to indicate the reason for failure. See man pages for error descriptions.

The two file descriptors returned are connected in a special way. Any data written to a first in, first out basis, usually abbreviated to FIFO. This means that if you write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2,

1. This is different from a stack, which operates on a last in, first out basis, usually abbreviated to LIFO.

Important Note: It's important to realize that these are file descriptors, not file streams, so we must use the lower−level read and write calls to access the data, rather than fread and fwrite.

Let's do following lab activity to understand it practically!

Exercise 2

Step1: create a file with name "pipe1.c". Type in the following code.

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

int main (int argc , char ** argv)
{
    int data_processed=0;
    int file_descripter[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    memset(buffer,'\0',sizeof(buffer));

    if(pipe(file_descripter)==0){
        data_processed = write(); // see man for write parameter
        printf("Wrote %d bytes:",data_processed);
        data_processed = read(); // see man for read parameter
        printf("Read %d, byte %s", data_processed, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);

}
```

Exercise:

1. Compile and run it.

2. Reverse the order of the write () and read () and run the program again.

3. Briefly describe what happened with the reversed order and why the program behaved that way.

4. Modify pipe1.c in the following ways:

   a. Dynamically allocate buffer so it's exactly the right size for some_data.

   b. Copy the string from some_data into buffer.

Graded Task : **Multiple pipes across a fork/exec**

In this task, you will write a program that has two-way communication between parent and child. Write a c program in which parent receive array of integer from command line argument and pass it to child using pipe, child need to sort that array (using bubble sort) and return that array to parent using pipe and parent need to display that sorted array.