

# CS232L Operating Systems Lab

## Lab 09: Introduction to IPC in Linux

CS Program  
Habib University

Fall 2024

### 1 Introduction

Inter-Process Communication (IPC) is an important part of the Operating System jobs. In this lab we will learn:

1. a brief introduction to Linux IPC
2. use pipe for IPC
3. use named pipes (FIFO) for IPC

### 2 What is Inter-process Communication (IPC)?

The operating system maintains each process image inside the RAM and ensures that each process can access only memory within its own address space i.e., it cannot access memory allotted to other processes. But what if two processes want to communicate with each other. <sup>1</sup> Process A wants to send a value to Process B but it can't do so because the kernel would not let it write into the memory allotted to process B; neither can it read something from Process B memory. So that is the solution?

The operating system provides multiple facilities which let two processes communicate with each other. Thus using these OS services, a process can send data to another process and receive data from it.

Figure 1 on page 2 gives an overview of the facilities supported by the Linux kernel.

### 3 Pipes

A pipe is a very simple way of communicating between two processes. One relevant real time example will be of watering plants in garden. To water the plants available at garden the tube will be connected to a water tank and another end of the pipe will be used to water the plants. Same is the scenario. When process A has to transfer data to process B it can use pipe. And most important thing is pipe here is unidirectional i.e. data can be sent in either of the directions at a time. If there needs to be dual communication then 2 pipes have to be used. Another thing to remember that pipes can only be used between related processes. No two different unrelated processes can use pipe. None will water plants in neighbor's house. This is the case shown in Figure 2 on page 2.

PIPE can be created with `pipe()` system call and it will return two file descriptors accepting array of integers as an argument. One file descriptor (FD) will be used as Read end file descriptor

---

<sup>1</sup>here take a minute the reflect on what it means for two processes to communicate?

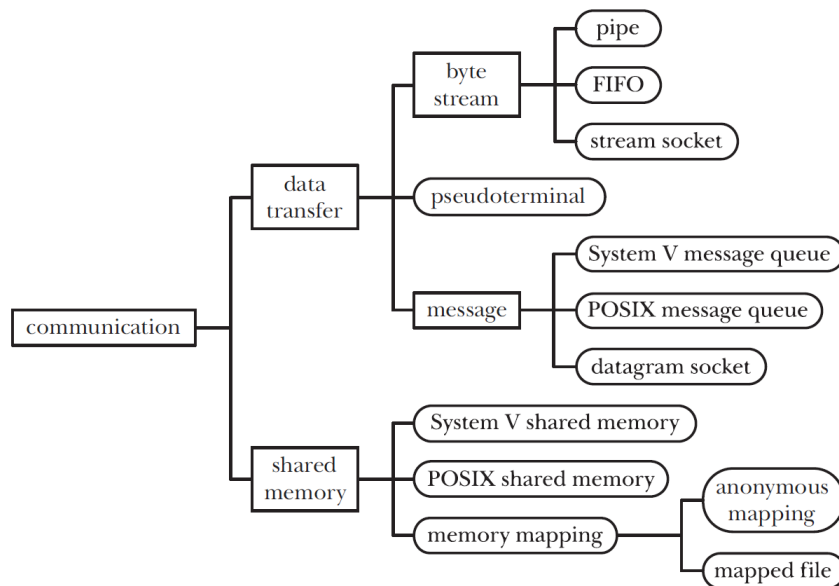


Figure 1: An overview of IPC facilities in Linux

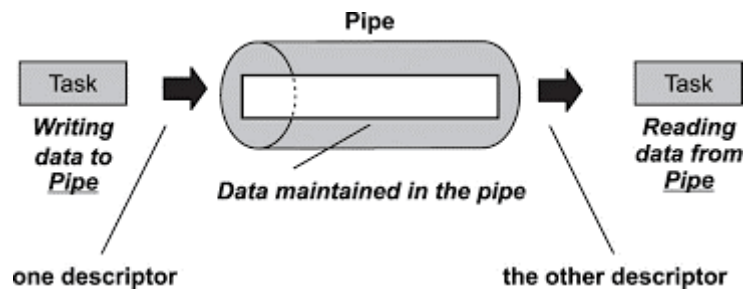


Figure 2: A pipe

and second one can be used as Write end file descriptor. File descriptor is an integer allotted by the system for each file that is created. Most important thing to remember in piped as conveyed earlier is, they can be used only with related processes (A process when has a child for itself then they become related). Keeping the above basic points in mind, one can easily walkthrough the code presented below:

```

1 #include<stdio.h>
2 #include<unistd.h>
3
4 int main(void) {
5     int pipefd[2];
6     int pid;
7     char buffer[15];
8     pipe(pipefd);
9     pid = fork();
10
11     if(pid > 0) {
12         fflush(stdin);
13         printf("unamed_pipe [INFO] Parent Process\n");
14         write(pipefd[1], "Hellow Mr.Linux", 15);
15     }
16     else if(pid == 0) {
17         sleep(5);
18         fflush(stdin);
19         printf("unamed_pipe [INFO] Child Process\n");
  
```

```

20     read(pipefd[0], buffer, sizeof(buffer));
21     write(1, buffer, sizeof(buffer));
22     printf("\n");
23 }
24 else {
25     printf("unnamed_pipe [ERROR] Error in creating child process\n");
26 }
27 if(pid > 0) wait();
28 return 0;
29 }
30

```

Listing 1: unnamed\_pipe.c

The execution of the above code is shown in Figure 3 on page 3.

```

student@OSLAB-VM: ~/Desktop/IPC code Examples
student@OSLAB-VM:~/Desktop/IPC code Examples$ gcc -o unnamed_pipe unnamed_pipe.c
unnamed_pipe.c: In function 'main':
unnamed_pipe.c:27:14: warning: implicit declaration of function 'wait' [-Wimplicit-
function-declaration]
    if(pid > 0) wait();
                   ^
student@OSLAB-VM:~/Desktop/IPC code Examples$ ./unnamed_pipe
unnamed_pipe [INFO] Parent Process
unnamed_pipe [INFO] Child Process
Hellow Mr.Linux
student@OSLAB-VM:~/Desktop/IPC code Examples$

```

Figure 3: Output from unnamed\_pipe.c code

So 'Hellow Mr.Linux' is sent to the child process which the user can see on the screen. One beauty in this mechanism is unless parent writes child cannot read and there exists a synchronization which is very vital. Only disadvantage associated with pipe is, it can be used for only related processes. Now this problem can be overcome by using Named pipe or FIFO.

## 4 Named Pipes

To overcome that we can use 'Named Pipes' which is also known as FIFO (First In First Out). Here the concept is slightly different. Taking a real world example again: suppose a person has to pass a letter to someone. Due to some situations, it cannot be given in person. Simple solution is that find a third person who is familiar to both the people. Now that third person will be able to hand over the paper to destination successfully. Same is the case with named pipe as shown in Figure 4 on page 3.

It can be used for communication between two different processes. The sequence goes like this, Process A will write the data in a common file which Process B can also access. After data has been written by A, B will read the data from that common file. After reading the file can be deleted. The term file has to be refined. It is also called as FIFO in Linux which can be created with available system calls. System call `mkfifo` can be used to create a FIFO. In FIFO two different processes can communicate which is revealed with following given C code, where

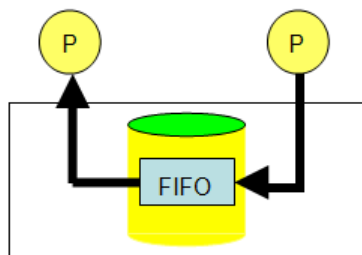


Figure 4: Typical architecture of named pipes.

fifo\_write.c is FIFO write program and fifo\_read.c is read program. Write program has to be executed first then read can be executed. Even if the user executes read program, it will wait for the writer to write the data. So, here exists an auto synchronization which is highly appreciable feature. The C code for both read and write are presented below, mkfifo has to be specified with the access permissions. Recall from lab manual 02 that A file when created has got permissions associated with it. There are basically three kinds of users available in Linux and three kinds of permissions associated with a file. Next question would arise in minds that can the permissions be change? Yes, it can be altered. 'chmod' is the command meant for it.

```

1 #include<stdio.h>
2 #include<sys/stat.h>
3 #include<sys/types.h>
4 #include<fcntl.h>
5 #include<unistd.h>
6
7 int main(void) {
8     int fd, retval;
9     char buffer[] = "TESTDATA";
10
11     fd = open("/tmp/myfifo", O_RDONLY);
12     retval = read(fd, buffer, sizeof(buffer));
13     fflush(stdin);
14     write(1, buffer, sizeof(buffer));
15     printf("\n");
16     close(fd);
17     return 0;
18 }

```

Listing 2: fifo\_read.c

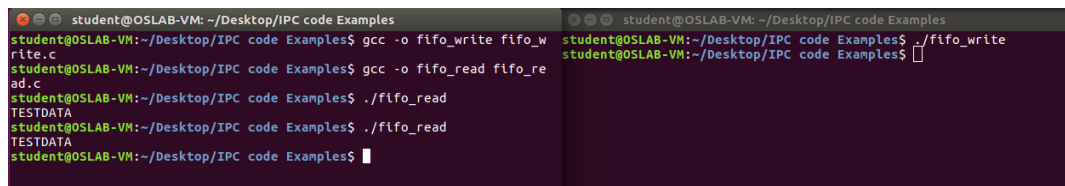
```

1 #include<stdio.h>
2 #include<sys/stat.h>
3 #include<sys/types.h>
4 #include<fcntl.h>
5 #include<unistd.h>
6
7 int main(void) {
8     int fd, retval;
9     char buffer[] = "TESTDATA";
10
11     fflush(stdin);
12     retval = mkfifo("/tmp/myfifo", 0666);
13     fd = open("/tmp/myfifo", O_WRONLY);
14     write(fd, buffer, sizeof(buffer));
15     close(fd);
16     return 0;
17 }

```

Listing 3: fifo\_write.c

The execution of the code given in fifo\_read.c and fifo\_write.c is given in Figure 5 on page 4. If the read is executed first, it will wait until write is executed. Automatic synchronization will be there.



```

student@OSLAB-VM: ~/Desktop/IPC code Examples
student@OSLAB-VM:~/Desktop/IPC code Examples$ gcc -o fifo_write fifo_write.c
student@OSLAB-VM:~/Desktop/IPC code Examples$ gcc -o fifo_read fifo_read.c
student@OSLAB-VM:~/Desktop/IPC code Examples$ ./fifo_read
TESTDATA
student@OSLAB-VM:~/Desktop/IPC code Examples$ ./fifo_read
TESTDATA
student@OSLAB-VM:~/Desktop/IPC code Examples$ █

student@OSLAB-VM: ~/Desktop/IPC code Examples
student@OSLAB-VM:~/Desktop/IPC code Examples$ ./fifo_write
TESTDATA
student@OSLAB-VM:~/Desktop/IPC code Examples$ █

```

Figure 5: Output from fifo\_read.c and fifo\_write.c.

## 5 Exercises

1. Reverse the example in 'unnamed\_pipe.c' so that child would send message to parent and parent would print the message on screen.
2. Run the FIFO example with three read processes and one write process.
3. Used named pipes to create a simple chat application between two processes P1 and P2. The communication can be one way only (half duplex) that is P1 sends messages while P2 simply outputs the received messages. Try to see if you can make the communication two way (full duplex) that is both P1 and P2 are able to send messages to each other without losing any messages.