**National University of Computer and Emerging Sciences**  
Operating System Lab - 05  
*Lab Manual*

Contents

[Objective 2](#_Toc495305158)

[Inter-Process Communication (IPC) & its Methodologies 2](#_Toc495305159)

[Pipe 2](#_Toc495305160)

[Named Pipe 3](#_Toc495305161)

[Shared Memory 6](#_Toc495305163)

# Objective

The purpose of this lab is to introduce you with IPC (Inter-Process Communication) and their methodologies

# Inter-Process Communication (IPC) & its Methodologies

**Inter-Process Communication (IPC)** is a mechanism that allows processes to communicate with each other and synchronize their actions. IPC is essential in operating systems that allow multitasking and multi-user environments, enabling processes to share data, signals, or other information effectively.

**Importance of IPC:**

* **Data Sharing**: Processes often need to share data, such as configuration settings, state information, or results of computations.
* **Synchronization**: IPC helps coordinate actions between processes to avoid race conditions and ensure consistency.
* **Resource Management**: It allows processes to work collaboratively while managing shared resources, such as memory or devices.

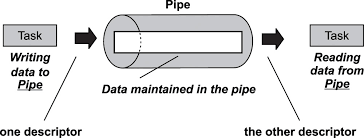
Following are very commonly used IPC mechanisms.

* Pipe
* Named Pipe
* Shared Memory

For this lab we will focus on pipes, named pipes and shared memory. Signals and semaphores will be cover in later labs

# Pipe

A pipe is 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 must 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 here.



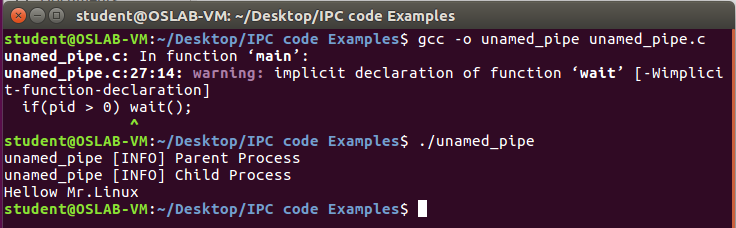
PIPE can be created with pipe() system call and it will return two file descriptors accepting array of integers as an argument. One file descriptor (FD) will be used as Read end file descriptor and second one can be used as Write end file descriptor. File descriptor is an integer allotted by the system 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:

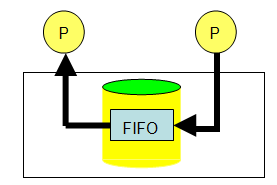


The execution of the above code is shown below:



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.

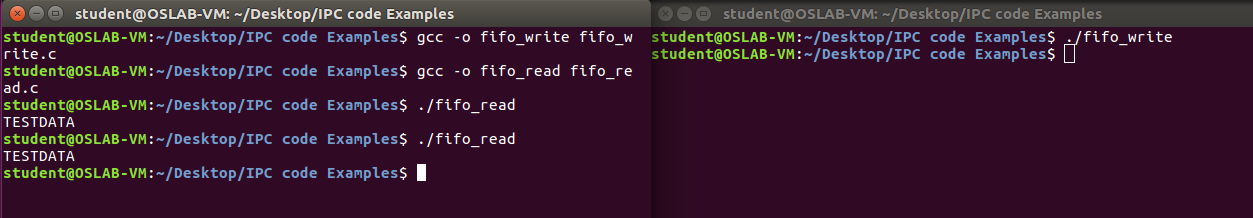
# Named Pipe

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. The solution is to 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. 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 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.

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.



The execution of the above code is given below



The execution of fifo\_write and fifo\_read is shown. If the read is executed first, it will wait until write is executed. Automatic synchronization will be there.

# Shared Memory

In the discussion of the fork ( ) system call, we mentioned that a parent and its children have separate address spaces. While this would provide a more secured way of executing parent and children processes (because they will not interfere each other), they shared nothing and have no way to communicate with each other. A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address space is "extended" by attaching this shared memory.

This mechanism is very important and most frequently used. Shared memory can even be used between unrelated processes. By default page memory of 4KB would be allocated as shared memory. Assume process 1 wants to access its shared memory area. It has to get attached to it first. Though its P1’s memory area, it cannot get access as such. Only after attaching it can gain access. A process creates a shared memory segment using shmget(). The original owner of a shared memory segment can assign ownership to another user with shmclt(). It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using shmctl(). Once created a shared memory segment can be attached to a process address space using shmcat(). It can be detached using shmdt(). The attaching process must be appropriate permissions for shmat(). Once attached, the process can read and write segment, as allowed by the permission requested in the attach operation. A shared memory segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structure and prototypes can be found in <sys/shm.h>. There are three steps:

1. Initialization
2. Attach
3. Detach

The client server scenario would be perfect to demonstrate shared memory, the general scheme of using shared memory is the following

**For Server**

1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call shmget().
2. Attach this shared memory to the server’s address space with system call shmat().
3. Initialize the shared memory, if necessary.
4. Do something and wait for all clients’ completion.
5. Detach the shared memory with system call shmdt().
6. Remove the shared memory with system call shmclt().

**For Client**

1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
2. Attach this shared memory to the client’s address space
3. Use the memory
4. Detach all shared memory segments, if necessary
5. Exit.

**Implementation:**

In UNIX-like operating systems, shared memory can be created and used through system calls. Below are the common steps involved in using shared memory:

**1. Create a Shared Memory Segment:**

To create a shared memory segment, the shmget() system call is used. This call allocates a shared memory segment and returns an identifier (shmid).

#include <sys/ipc.h>

#include <sys/shm.h>

key\_t key = ftok("path/to/some/file", 'R'); // Generate a unique key

int shmid = shmget(key, size, IPC\_CREAT | 0666); // Create shared memory

* **Parameters**:
  + key: A unique identifier for the shared memory segment.
  + size: The size of the memory segment in bytes.
  + IPC\_CREAT: A flag to create the segment if it does not exist.
  + 0666: Permissions for the segment (read/write for user, group, and others).

**2. Attach to the Shared Memory Segment:**

Once the shared memory segment is created, a process must attach to it using the shmat() system call. This call maps the shared memory segment into the process's address space.

#include <sys/shm.h>

char \*shm\_ptr = shmat(shmid, NULL, 0); // Attach to the shared memory

* **Parameters**:
  + shmid: The identifier returned by shmget().
  + NULL: The address at which the memory should be attached (0 lets the system choose).
  + 0: Flags; typically 0 is used.

**3. Use the Shared Memory:**

Once attached, the process can read from and write to the shared memory as if it were a regular array.

strcpy(shm\_ptr, "Hello from Shared Memory"); // Write to shared memory

printf("%s\n", shm\_ptr); // Read from shared memory

**4. Detach from the Shared Memory Segment:**

After the process is done using the shared memory, it should detach from it using shmdt().

shmdt(shm\_ptr); // Detach from shared memory

**5. Remove the Shared Memory Segment:**

If a process no longer needs the shared memory segment, it can remove it using the shmctl() system call.

shmctl(shmid, IPC\_RMID, NULL); // Mark the segment to be destroyed

Example Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

#define SHM\_SIZE 1024 // Define the size of the shared memory

int main() {

int shmid;

key\_t key = ftok("shmfile", 65); // Generate a unique key

char \*str;

// Create a shared memory segment

shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666);

if (shmid < 0) {

perror("shmget");

exit(1);

}

// Attach to the shared memory segment

str = (char\*) shmat(shmid, NULL, 0);

if (str == (char \*) -1) {

perror("shmat");

exit(1);

}

// Write to the shared memory

printf("Write to shared memory: ");

fgets(str, SHM\_SIZE, stdin); // Read input from user

printf("Data in shared memory: %s\n", str); // Read from shared memory

// Detach from the shared memory segment

shmdt(str);

// Remove the shared memory segment

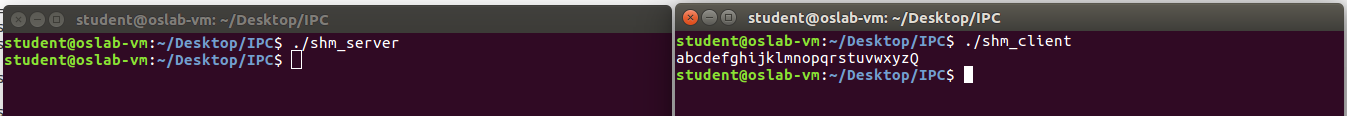
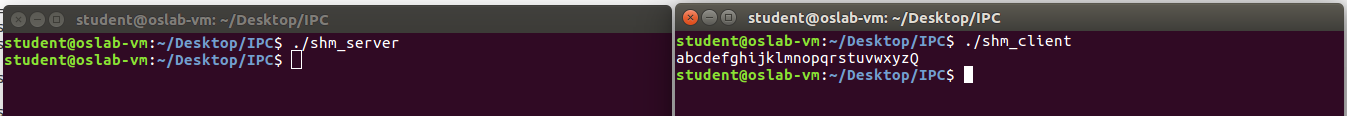
shmctl(shmid, IPC\_RMID, NULL);

return 0;

}

Below is the two separate program for read and write presented here.





Note: In the code IPC\_CREAT|0666 basically performs bitwise or of the two to set the flag in shmget, and both of them simply perform their usual functions. 0666 sets the access permissions of the memory segment while IPC\_CREAT tells the system to create a new memory segment for the shared memory.

# Lab Activity

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. Write two programs that would implement the concept of shared memory, the requirements are as follow:
   1. The first program would create a shared memory and put a number in it.
   2. The second program would store the number (which would come as string) in an integer variable and then writes in the memory “ready”.
   3. The term “ready” is then picked up by the first program, it prints this value onto the screen and puts ‘\*’ in the memory.
   4. The second program when read ‘\*’ will put the table of the number which it stored in part b from 1 – 10. Such that
      1. Assume num is the variable it stored the number came from program 1.
      2. It will initiate an iterator say int i and assign 1 in it.
      3. Whenever it sees ‘\*’ in the shared memory it will put the value calculated from the equation: i\*n in the shared memory. Which then picked up by the first program and again it will read the value, print it on the screen and put ‘\*’ again.
      4. This cycle continues till i > 10
4. In mathematics, the Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones:   
     
    0 ,1 ,1 ,2 ,3 ,5 ,8 ,13 ,21 ,34 ,55 ,89 ,144,….

Write three programs two writers (say A, B) and one reader (say C). Initially A and B will have a shared memory (A = 0 and B = 1) and C would attach these shared memories and would generate Fibonacci series. Given below is a general algorithm

* C: Read memory of A
* C: Read memory of B
* C: Add A+B
* C: Assign memory of B to memory of A
* C: Assign value of A+B to memory of B
* The above iteration is done n times (where n can be any value from one – hundred)