National University of Computer and Emerging Sciences

Operating System Lab - 06
Lab Manual

Contents

Objective

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

## Inter-Process Communication (IPC) & its Methodologies

IPC is very vital in any Embedded System. A program may have to feed another process for it to proceed. It is inherent in all the embedded systems. Following are very commonly used IPC mechanisms.

- Pipe
- Named Pipe
- Signals
- Message Queue
- Shared Memory
- Semaphores
- Remote procedure calls
- Sockets

For this lab we will focus on pipes, named pipes, message queue 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 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 here.
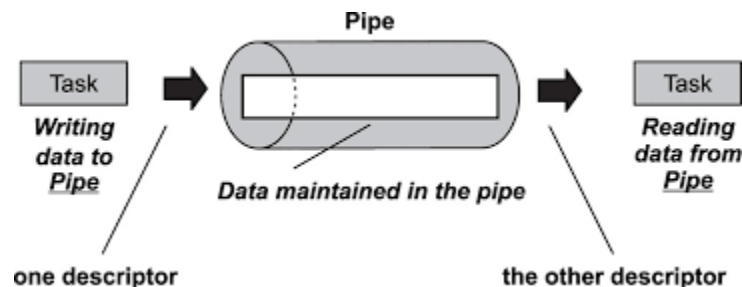
PIPE can be created with pipe() system call and it will return two file descriptors accepting array of integers as an argument. One file descriptor (FD) will be used as Read end file descriptor and second one can be used as Write end file descriptor. File descriptor is an integer allotted by the system 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:


unamed_pipe.c

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. 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. 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.
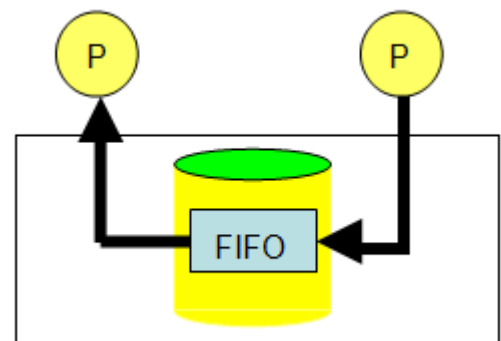
C code for both read and write are presented below, mkfifo has to be specified with the access permissions. Recall from lab manual 02 that A file when created has got permissions associated with it. There are basically three kinds of users available in Linux and three kinds of permissions associated 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.

fifo_read.c   fifo_write.c

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.

Message Queue

Two or more processes can exchange information via access to a common system message queue. The sending process places via some (OS) message-passing module a message onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

Message queues provide an asynchronous way of communication possible, meaning that the sender and receiver of the message need not interact with the message queue at the same time. Message queue has a wide range of applications. Very simple applications can be taken as example here.

1. Taking input from the keyboard
2. To display output on the screen
3. Voltage reading from sensor etc.
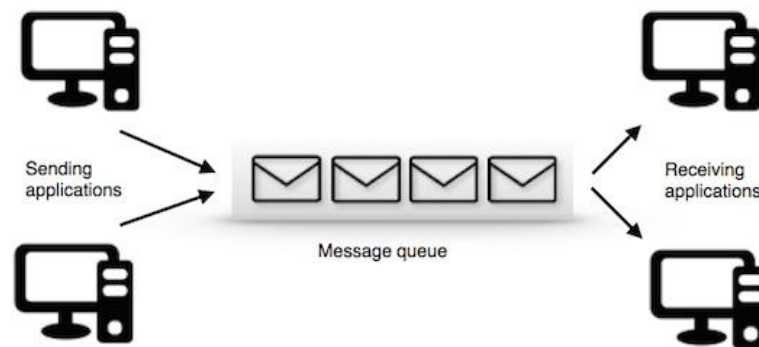
A task which has to send the message can put message in the queue and other tasks. A message queue is a buffer-like object which can receive messages from ISRs (Interrupt Service Routine), tasks and the same can be transferred to other recipients. In short, it is like a pipeline. It can hold the messages sent by sender for a period until receiver reads it. And biggest advantage which someone can have in queue is receiver and sender need not use the queue on same time. Sender can come and post message in queue, receiver can read it whenever needed. Message queue basically composed of few components. A message queue should have a start and it should have an end as well. Starting point of a queue is referred as head of the queue and terminating point is called tail of the queue. Size of the queue has to be decided by the programmer while writing the code. And a queue cannot be read if it is empty. Meanwhile, a queue cannot be written into if it is already full. And a queue can have some empty elements as well.

The message queue can be implemented in Linux machine with available system calls. The basic operations to be carried out in queue are

- Creation/Deletion of queue
- Sending/Receiving of message

Two different files have to be written here: one for sender and another one for receiver. Receiver will wait until the sender writes into the queue. One important advantage with message queue is, it support automatic synchronization between the sender and receiver. Receiver will wait until sender writes. Another advantage is memory can be freed after usage which is very essential in all software system.

Few thing can be taken into consideration before writing code for queue.

1. An Identifier has to be generated (key)
2. msgsnd() -> will initialize the queue.
3. msgrcv() -> will be used to receive the message
4. msgclt() -> Control action can be performed with this call i.e. deletion can be done with msgclt().

Below codes are for demonstrating message queue, you may face administrative privileges if not having while running these codes.

message_send.c   message_receiver.c
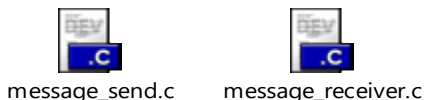
The execution is shown below.

```
student@oslab-vm:~/Desktop/IPC$ ./message_receiver
message_receive [INFO] Message: hellow from ali
message_receive [INFO] Message: how are you?
message_receive [INFO] Message: r u okay?

☒ ⊖ ⊙   student@oslab-vm: ~/Desktop/IPC
student@oslab-vm:~/Desktop/IPC$ ./message_send
message_send [INFO] The message ID is: 32769
message_send [PROMPT] Enter a text: hellow from ali
how are you?
r u okay?
```

So the code should prompt the sender for typing the data to be sent to receiver. In parallel, from another terminal message_rcv would receive all the information that sender types. If receiver compiles and executes first, program will wait until sender drops the message.

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.
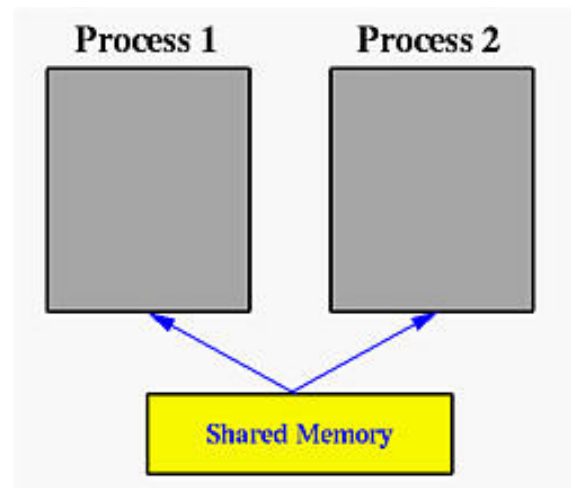
This mechanism is very important and most frequently used. Shared memory can even be used between unrelated processes. By default page memory of 4KB would be allocated as shared memory. Assume process 1 wants to access its shared memory area. It has to get attached to it first. Though its P1's memory 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.

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

shm_server.c   shm_client.c

Lab Activity

1. Write a program that child would send message to parent and parent would print the message on screen. Task of child is to generate a table of 5. Parent will print it out.
2. Run the FIFO example with three read processes and one write process.
3. 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 a code to generate Fibonacci numbers and then pass the numbers to message queue individually. Two readers program will read the alternative numbers of queue.
   For example:
   A: 0,1,3,8,21,55…
   B: 1,2,5,13,34,89…
4. Write two programs that would implement the concept of shared memory, the requirements are as follow:
   a. The first program would create a shared memory and put a number in it.
   b. The second program would store the number (which would come as string) in an integer variable and then writes in the memory "ready".
   c. The term "ready" is then picked up by the first program, it prints this value onto the screen and puts '*' in the memory.
   d. The second program when read '*' will put the table of the number which it stored in part b from 1 – 10. Such that
      i. Assume num is the variable it stored the number came from program 1.
      ii. It will initiate an iterator say int i and assign 1 in it.
      iii. 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.
      iv. This cycle continues till i > 10
5. 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)