CS232L Operating Systems Lab
Lab 09: More IPC in Linux
(Message Queue and Shared Memory)

CS Program
Habib University

Fall 2021

1 Introduction

In this lab, we will learn about two more inter-process communication (IPC) mechanisms. Specifically, we will learn to:

1. use message queue for IPC

2. use shared memory 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. [1] 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 mulitple 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 overivew of the facilities supported by the Linux kernel.

3 Message Queues

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 as shown in Figure 2 on page 2. 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

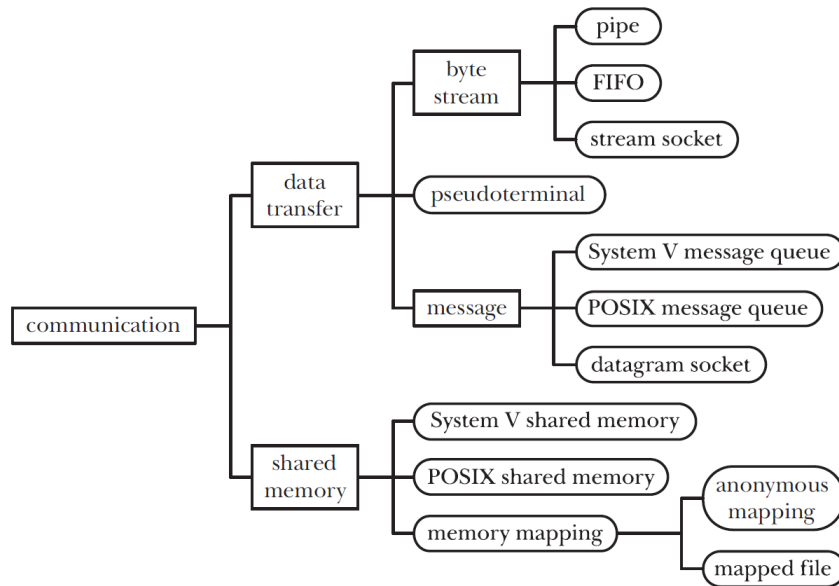3. Voltage reading from sensor etc.

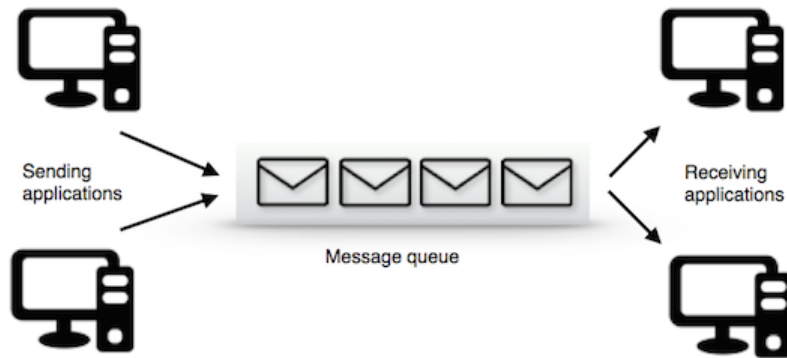Figure 1: An overview of IPC facilities in Linux

Figure 2: Typical architecture of Message Queue.

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

---

[1]here take a minute the reflect on what it means for two processes to communicate?

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. The execution is shown in Figure 3 on page 4. 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.

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/types.h>
#include<sys/msg.h>

struct msgbuf {
  long mtype;
  char msgtxt[200];
};

int main(void) {
  struct msgbuf msg;
  int msgid;
  key_t key;

  if((key = ftok("message_send.c",'b')) == -1 ) {
    perror("key");
    exit(1);
  }

  if((msgid=msgget(key,0644|IPC_CREAT)) == -1 ) {
    perror("key");
    exit(1);
  }
  printf("message_send [INFO] The message ID is: %d\n", msgid);
  printf("message_send [PROMPT] Enter a text: ");
  msg.mtype = 1;
  while(gets(msg.msgtxt),!feof(stdin)) {
    if(msgsnd(msgid,&msg,sizeof(msg),0) == -1) {
      perror("msgsnd");
      exit(1);
    }
  }

  if(msgctl(msgid,IPC_RMID,NULL) == -1) {
    perror("msgctl");
    exit(1);
  }
  return 0;
}

//to delete the msgid through command line with can use the following command
// ipcrm -q <msgid>

```
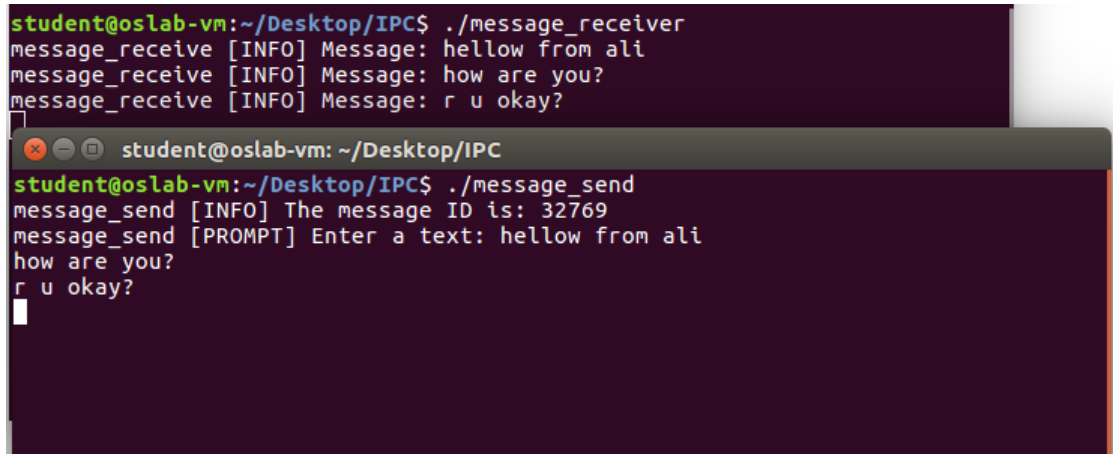
Listing 1: message_send.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/types.h>
#include<sys/msg.h>

struct msgbuf {
  long mtype;
  char msgtxt[200];
};

int main(void) {
  struct msgbuf msg;
  int msgid;
  key_t key;

  if((key = ftok("message_send.c",'b')) == -1 ) {
    perror("key");
    exit(1);
  }

  if((msgid=msgget(key,0644)) == -1 ) {
    perror("msgid");
    exit(1);
  }
  for(;;) {
    if(msgrcv(msgid,&msg,sizeof(msg),1,0) == -1) {
      perror("msgrcv");
      exit(1);
    }
    printf("message_receive [INFO] Message: %s\n",msg.msgtxt);
  }

  return 0;
}


```

Listing 2: message_receiver.c

Figure 3: Output from message_send.c and message_receiver.c.

4   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 shown in Figure 4 on page 5. 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.
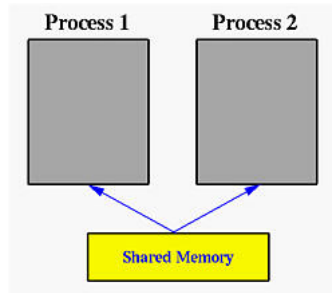

Figure 4: Typical architecture of shared memory.

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. Initialize

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,

4.1   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().

4.2   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 are two separate programs for read and write as presented here.

```c
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<stdlib.h>

#define MAXSIZE 27

void die(char *str) {
  perror(str);
  exit(1);
}

int main(void) {
  char c;
  int shmid;
  key_t key;
  char *shm, *s;

  key = 2211;

  if((shmid = shmget(key, MAXSIZE, IPC_CREAT | 0666)) < 0)
    die("shmget");
  if((shm = shmat(shmid, NULL, 0)) == (char*) -1)
    die("shmat");
  s = shm;
  for(c = 'a'; c <= 'z'; c++)
    *s++ = c;

  while(*shm != '*')
    sleep(1);

  exit(0);
}
```

Listing 3: shm_server.c

```c
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<stdlib.h>

#define MAXSIZE 27

```

```c
9   void die(char *str) {
10      perror(str);
11      exit(1);
12  }
13
14  int main(void) {
15      int shmid;
16      key_t key;
17      char *shm, *s;
18
19      key = 2211;
20      fflush(stdin);
21      if((shmid = shmget(key, MAXSIZE, IPC_CREAT | 0666)) < 0)
22          die("shmget");
23      if((shm = shmat(shmid, NULL, 0)) == (char*) -1)
24          die("shmat");
25      for(s = shm; *s != '\0'; s++)
26          putchar(*s);
27
28      *shm = '*';
29      printf("\n");
30      exit(0);
31  }
```
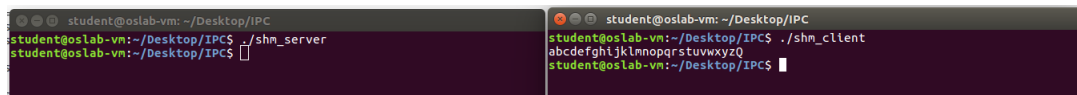
Listing 4: shm_client.c

Figure 5: Output from shm_server.c and shm_client.c programs.

5 Lab Tasks

1. 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)

2. Write a program that implements shared memory concept. The first program makes a shared memory variable and assign a random num between 0 - 100 every 5 seconds and second program also makes a shared memory variable which assigns a random double value between 999.00 - 10000.00 every 2 seconds and the third program would read these both memories in every 10 seconds and find the remainder of value from second program to the value from first program in double form with 2 decimal places.

7