LAB # 07

## Statement Purpose:

This lab will explain to you the inter process communication through shared memory and message passing in Linux.

## Activity Outcomes:

This lab describes

- How two processes communicate via shared memory
- How two processes communicate via message passing

## Instructor Note:

As pre-lab activity, introduce students to the concept of inter-process communication

## 1) Stage J (Journey)

A process can be of two type:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

1. Shared Memory
2. Message passing

The Figure below shows a basic structure of communication between processes via shared memory method and via message passing.

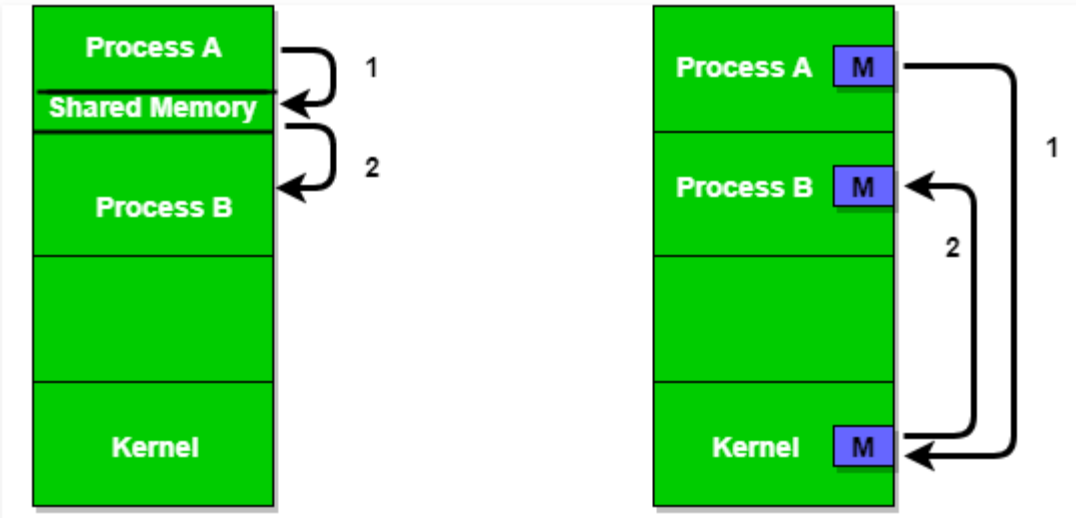



**Figure 1 -** Shared Memory and Message Passing

An operating system can implement both methods of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process 2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

Let's discuss an example of communication between processes using shared memory method.

i. **Shared Memory Method**

**Ex: Producer-Consumer problem**

There are two processes: Producer and Consumer. Producer produces some item and Consumer

consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it. The pseudo code are given below:

**Shared Data between the two Processes**

```
#define buff_max 25
#define mod %

    struct item{

        // diffrent member of the produced data
        // or consumed data
        ---------
    }

    // An array is needed for holding the items.
    // This is the shared place which will be
    // access by both process
    // item shared_buff [ buff_max ];

    // Two variables which will keep track of
    // the indexes of the items produced by producer
    // and consumer The free index points to
    // the next free index. The full index points to
    // the first full index.
    int free_index = 0;
    int full_index = 0;
```

**Producer Process Code**

```
item nextProduced;

    while(1){

        // check if there is no space
        // for production.
        // if so keep waiting.
        while((free_index+1) mod buff_max == full_index);

        shared_buff[free_index] = nextProduced;
        free_index = (free_index + 1) mod buff_max;
    }
```

**Consumer Process Code**

```
item nextConsumed;

    while(1){

        // check if there is an available
        // item  for consumption.
        // if not keep on waiting for
        // get them produced.
        while((free_index == full_index);

        nextConsumed = shared_buff[full_index];
        full_index = (full_index + 1) mod buff_max;
    }
```

In the above code, The producer will start producing again when the (free_index+1) mod buff max will be free because if it it not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index points to the same index, this implies that there are no item to consume.

### ii) Messaging Passing Method

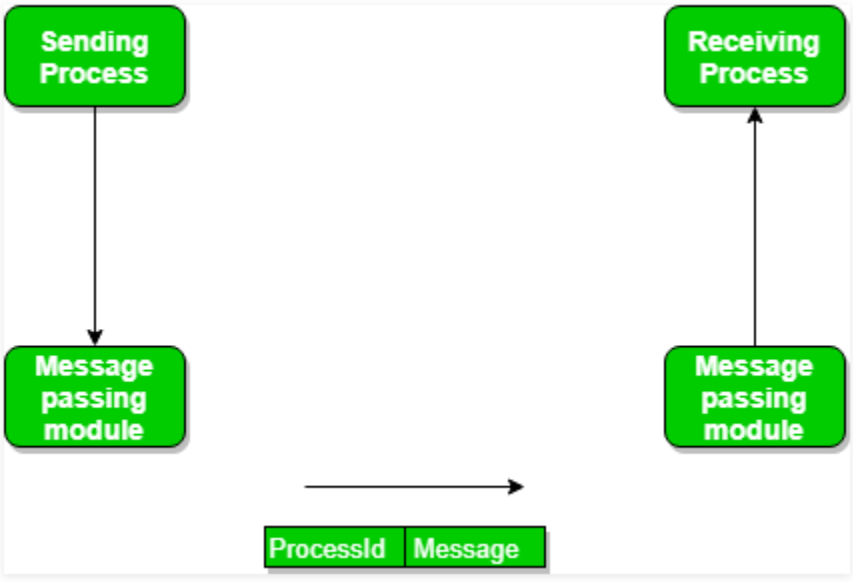
Now,We will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.

  We need at least two primitives:

  – **send**(message, destinaion) or **send**(message)

  – **receive**(message, host) or **receive**(message)




The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for

programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

### A. IPC through shared memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.
The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

***ftok()****: is use to generate a unique key.*

***shmget()****: int shmget(key_t,size_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.*

***shmat()****: Before you can use a shared memory segment, you have to attach yourself to it using shmat(). void *shmat(int shmid ,void *shmaddr ,int shmflg);*
*shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.*

***shmdt()****: When you're done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void *shmaddr);*

***shmctl()****: when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used. shmctl(int shmid,IPC_RMID,NULL);*

**SHARED MEMORY FOR WRITER PROCESS**

```cpp
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

**SHARED MEMORY FOR READER PROCESS**

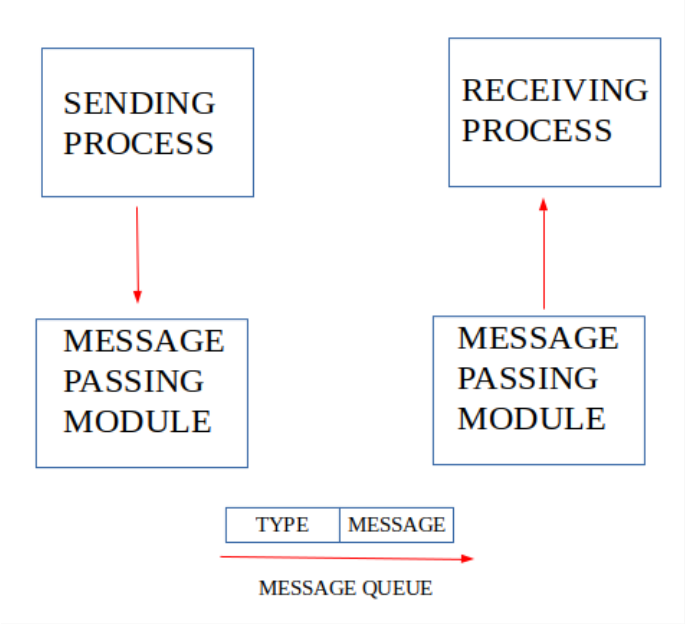
```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

### B. IPC using Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**. New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) 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.




System calls used for message queues:

**ftok()**: is use to generate a unique key.

**msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.

**msgsnd()**: Data is placed on to a message queue by calling msgsnd().

**msgrcv()**: messages are retrieved from a queue.

**msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue.

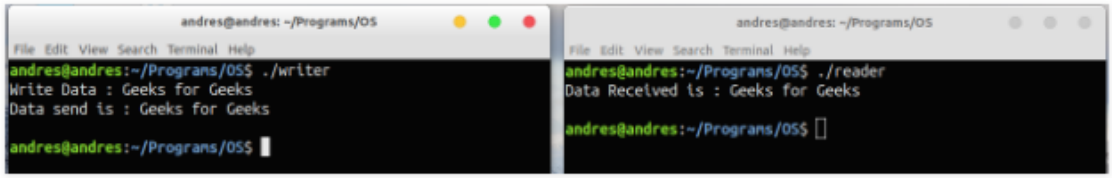
**MESSAGE QUEUE FOR WRITER PROCESS**

```c
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data : ");
    gets(message.mesg_text);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);

    // display the message
    printf("Data send is : %s \n", message.mesg_text);

    return 0;
}
```

**MESSAGE QUEUE FOR READER PROCESS**

```c
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

     // to destroy the message queue
     msgctl(msgid, IPC_RMID, NULL);

     return 0;
 }
```

Output:



## 2) Stage a1 (apply)

### Lab Activities:

1. Implement Merge sort in shared memory

**Solution:**

Given a number 'n' and a n numbers, sort the numbers using **Concurrent** Merge Sort. (Hint: Try to use shmget, shmat system calls).

**Part1: The algorithm (HOW?)**

Recursively make two child processes, one for the left half, one of the right half. If the number of elements in the array for a process is less than 5, perform a Insertion Sort. The parent of the two children then merges the result and returns back to the parent and so on. But how do you make it concurrent?

**Part2: The logical (WHY?)**

The important part of the solution to this problem is not algorithmic, but to explain concepts of Operating System and kernel.

To achieve concurrent sorting, we need a way to make two processes to work on the same array at the same time. To make things easier Linux provides a lot of system calls via simple API endpoints. Two of them are, **shmget()** (for shared memory allocation) and **shmat()** (for shared

memory operations). We create a shared memory space between the child process that we fork. Each segment is split into left and right child which is sorted, the interesting part being they are working concurrently! The shmget() requests the kernel to allocate a shared page for both the processes.

```c
// C program to implement concurrent merge sort
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void insertionSort(int arr[], int n);
void merge(int a[], int l1, int h1, int h2);

void mergeSort(int a[], int l, int h)
{
    int i, len=(h-l+1);

    // Using insertion sort for small sized array
    if (len<=5)
    {
        insertionSort(a+l, len);
        return;
    }

    pid_t lpid,rpid;
    lpid = fork();
    if (lpid<0)
    {
        // Lchild proc not created
        perror("Left Child Proc. not created\n");
        _exit(-1);
    }
```

```
else if (lpid==0)
{
    mergeSort(a,l,l+len/2-1);
    _exit(0);
}
else
{
    rpid = fork();
    if (rpid<0)
    {
        // Rchild proc not created
        perror("Right Child Proc. not created\n");
        _exit(-1);
    }
    else if(rpid==0)
    {
        mergeSort(a,l+len/2,h);
        _exit(0);
    }
}

int status;

// Wait for child processes to finish
waitpid(lpid, &status, 0);
waitpid(rpid, &status, 0);
```

```
    // Merge the sorted subarrays
    merge(a, l, l+len/2-1, h);
}

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;

       /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
           j = j-1;
       }
       arr[j+1] = key;
   }
}
```

```
// Method to merge sorted subarrays
void merge(int a[], int l1, int h1, int h2)
{
    // We can directly copy  the sorted elements
    // in the final array, no need for a temporary
    // sorted array.
    int count=h2-l1+1;
    int sorted[count];
    int i=l1, k=h1+1, m=0;
    while (i<=h1 && k<=h2)
    {
        if (a[i]<a[k])
            sorted[m++]=a[i++];
        else if (a[k]<a[i])
            sorted[m++]=a[k++];
        else if (a[i]==a[k])
        {
            sorted[m++]=a[i++];
            sorted[m++]=a[k++];
        }
    }

    while (i<=h1)
        sorted[m++]=a[i++];

    while (k<=h2)
        sorted[m++]=a[k++];
```

\

```
    int arr_count = l1;
    for (i=0; i<count; i++,l1++)
        a[l1] = sorted[i];
}

// To check if array is actually sorted or not
void isSorted(int arr[], int len)
{
    if (len==1)
    {
        printf("Sorting Done Successfully\n");
        return;
    }

    int i;
    for (i=1; i<len; i++)
    {
        if (arr[i]<arr[i-1])
        {
            printf("Sorting Not Done\n");
            return;
        }
    }
    printf("Sorting Done Successfully\n");
    return;
}
```

```c
// To fill randome values in array for testing
// purpise
void fillData(int a[], int len)
{
    // Create random arrays
    int i;
    for (i=0; i<len; i++)
        a[i] = rand();
    return;
}

// Driver code
int main()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;


    // Using fixed size array.  We can uncomment
    // below lines to take size from user
    int length = 128;

    /* printf("Enter No of elements of Array:");
    scanf("%d",&length); */

    // Calculate segment length
    size_t SHM_SIZE = sizeof(int)*length;
```

```c
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        _exit(1);
    }

    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *) -1)
    {
        perror("shmat");
        _exit(1);
    }

    // Create a random array of given length
    srand(time(NULL));
    fillData(shm_array, length);

    // Sort the created array
    mergeSort(shm_array, 0, length-1);

    // Check if array is sorted or not
    isSorted(shm_array, length);

      // Check if array is sorted or not
      isSorted(shm_array, length);

      /* Detach from the shared memory now that we are
         done using it. */
      if (shmdt(shm_array) == -1)
      {
          perror("shmdt");
          _exit(1);
      }

      /* Delete the shared memory segment. */
      if (shmctl(shmid, IPC_RMID, NULL) == -1)
      {
          perror("shmctl");
          _exit(1);
      }

      return 0;
}
```

Output:

```
Sorting Done Successfully
```

Task:

2. Implement Bubble sort in shared memory

Bonus Task:

3. Implement chat application using message queue

## 1. Stage V (verify)

### Home Activities:

1. Practice the Linux commands