CLASSICAL SYNCHRONIZATION PROBLEMS

29.Program to use semaphores to solve Reader Writer Problem.

Description:-

In computer science, the readers-writers problems are examples of a common computing problem in concurrency. There are at least three variations of the problems, which deal with situations in which many threads try to access the same shared resource at one time. Some threads may read and some may write, with the constraint that no process may access the share for either reading or writing, while another process is in the act of writing to it. (In particular, it is allowed for two or more readers to access the share at the same time.) A readers-writer lock is a data structure that solves one or more of the readers-writers problems.

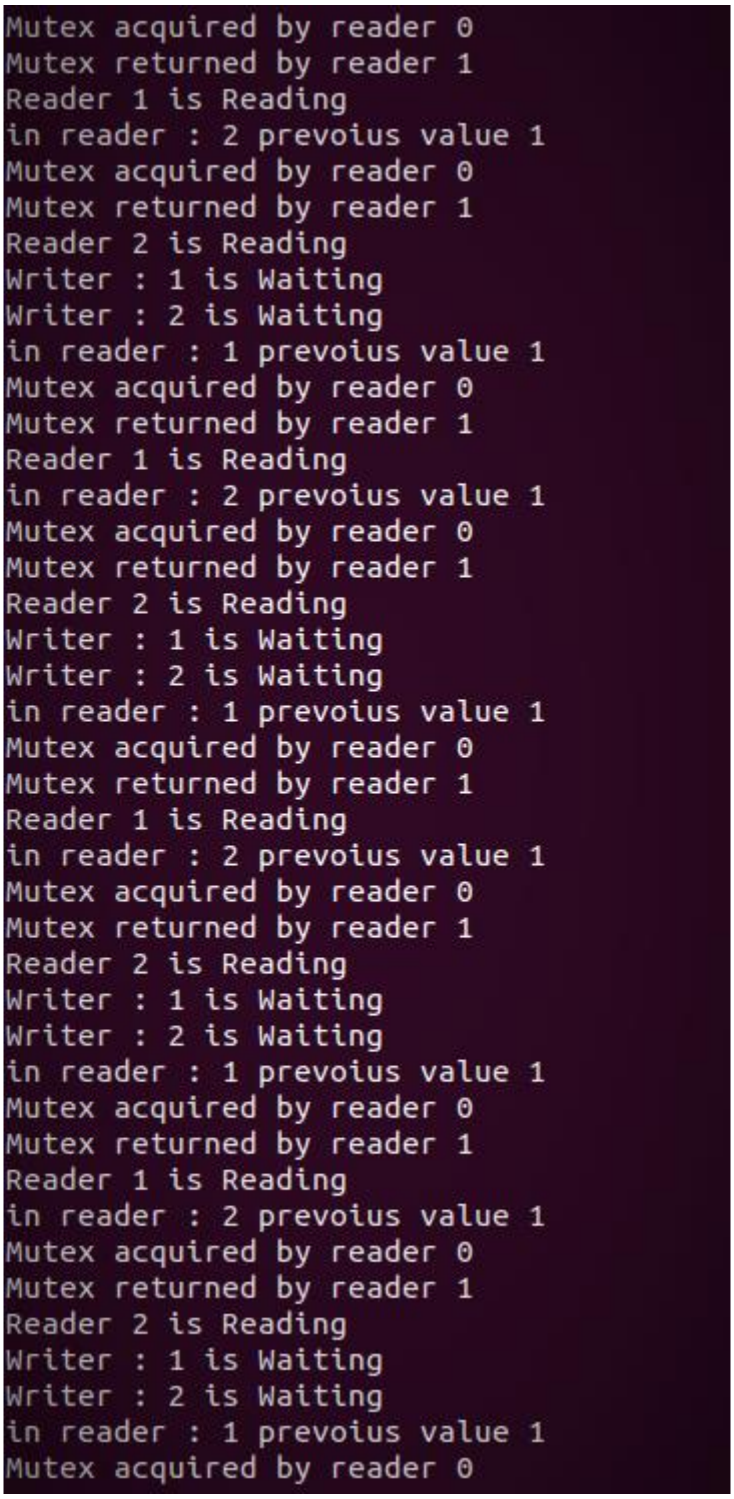
Program:-

```
#include<stdio.h>

#include<semaphore.h>

sem_t mutex;

sem_t db;

int readercount=0;

pthread_t reader1,reader2,writer1,writer2;

void *reader(void *);

void *writer(void *);

void main()

{

sem_init(&mutex,0,1);

sem_init(&db,0,1);

while(1)

{
```

```
pthread_create(&reader1,NULL,reader,"1");
pthread_create(&reader2,NULL,reader,"2");
pthread_create(&writer1,NULL,writer,"1");
pthread_create(&writer2,NULL,writer,"2");
}
}
void *reader(void *p)
{
int temp;
sem_getvalue(&mutex,&temp);
printf("in reader : %s prevoius value %d\n",(char *)p,temp);
sem_wait(&mutex);
sem_getvalue(&mutex,&temp);
printf("Mutex acquired by reader %d\n",temp);
readercount++;
if(readercount==1) sem_wait(&db);
sem_post(&mutex);
sem_getvalue(&mutex,&temp);
printf("Mutex returned by reader %d\n",temp);
printf("Reader %s is Reading\n",(char *)p);
sleep(3);
sem_wait(&mutex);
printf("Reader %s Completed Reading\n",(char *)p);
readercount--;
if(readercount==0) sem_post(&db);
sem_post(&mutex);
}
```

```
void *writer(void *p)

{

printf("Writer : %s is Waiting\n",(char *)p);

sem_wait(&db);

printf("Writer %s is writing\n",(char *)p);

sem_post(&db);

sleep(2);

}
```

Output:-



```
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 1 is Reading
in reader : 2 prevoius value 1
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 2 is Reading
Writer : 1 is Waiting
Writer : 2 is Waiting
in reader : 1 prevoius value 1
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 1 is Reading
in reader : 2 prevoius value 1
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 2 is Reading
Writer : 1 is Waiting
Writer : 2 is Waiting
in reader : 1 prevoius value 1
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 1 is Reading
in reader : 2 prevoius value 1
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 2 is Reading
Writer : 1 is Waiting
Writer : 2 is Waiting
in reader : 1 prevoius value 1
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 1 is Reading
in reader : 2 prevoius value 1
Mutex acquired by reader 0
Mutex returned by reader 1
Reader 2 is Reading
Writer : 1 is Waiting
Writer : 2 is Waiting
in reader : 1 prevoius value 1
Mutex acquired by reader 0
```

30.Program to demonstrate producer consumer problem using semaphores.

Description:

In computing, the producer–consumer problem[1][2] (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

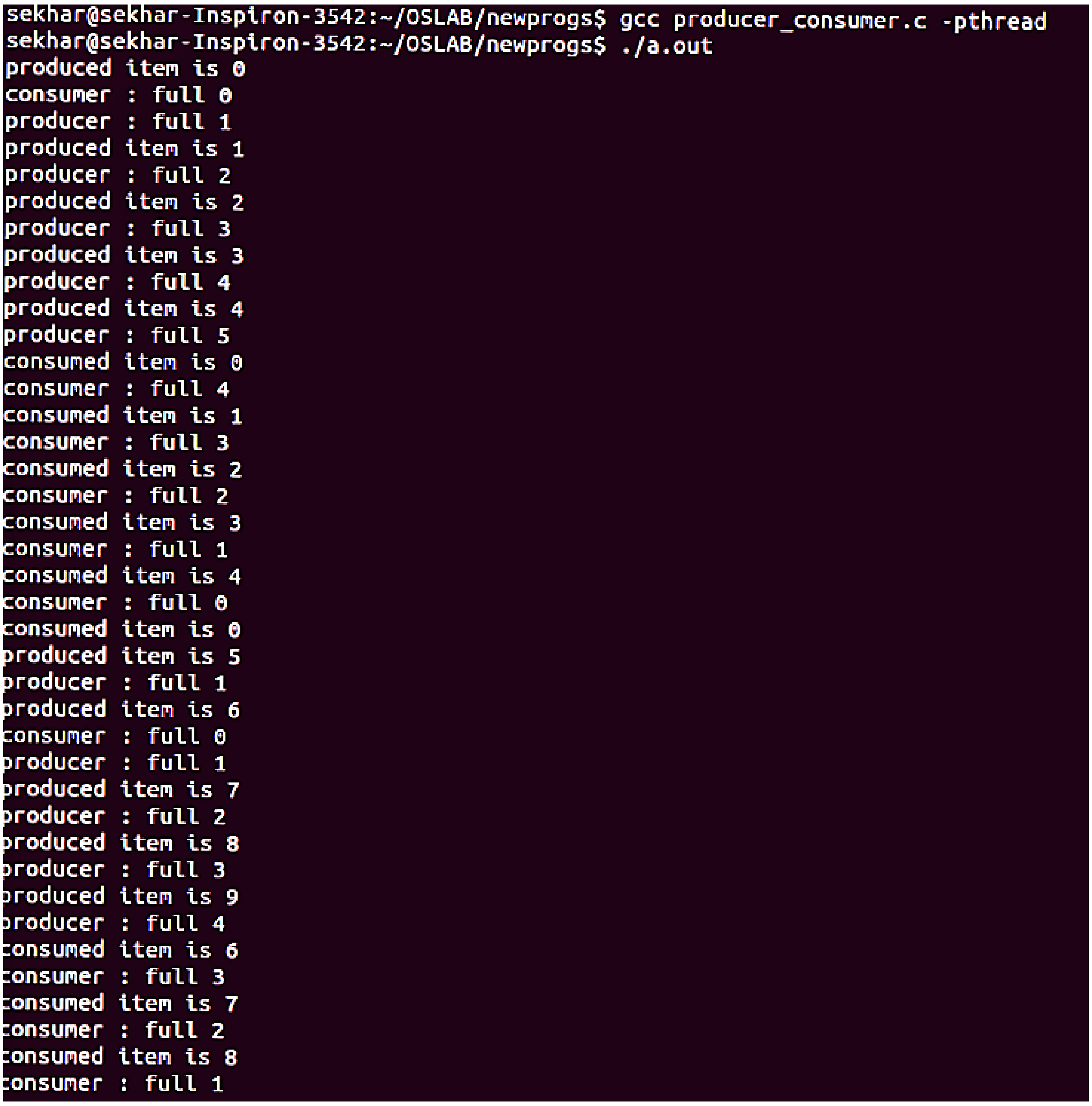
The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.-

Program:-

```
#include<stdio.h>

#include<pthread.h>

#include<semaphore.h>

int buf[5],f,r;

sem_t mutex,full,empty;

void *produce(void *arg)

{

   int i;

   for(i=0;i<10;i++)
   {int temp;
       sem_wait(&empty);
       sem_wait(&mutex);
       printf("produced item is %d\n",i);
       buf[(++r)%5]=i;
       sleep(1);
       sem_post(&mutex);
       sem_post(&full);
           sem_getvalue(&full,&temp);
       printf("producer : full %u\n",temp);
   }
}
void *consume(void *arg)
{
       int item,i;
       for(i=0;i<10;i++)
       {int temp;
             sem_wait(&full);
                     sem_getvalue(&full,&temp);
           printf("consumer : full %u\n",temp);
             sem_wait(&mutex);
             item=buf[(++f)%5];
             printf("consumed item is %d\n",item);
             sleep(1);
             sem_post(&mutex);
             sem_post(&empty);
```

```
        }
}
void main()
{
    pthread_t tid1,tid2;
    sem_init(&mutex,0,1);
    sem_init(&full,0,1);
    sem_init(&empty,0,5);
    pthread_create(&tid1,NULL,produce,NULL);
        pthread_create(&tid2,NULL,consume,NULL);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
}
```

Output:-



```
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ gcc producer_consumer.c -pthread
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ ./a.out
produced item is 0
consumer : full 0
producer : full 1
produced item is 1
producer : full 2
produced item is 2
producer : full 3
produced item is 3
producer : full 4
produced item is 4
producer : full 5
consumed item is 0
consumer : full 4
consumed item is 1
consumer : full 3
consumed item is 2
consumer : full 2
consumed item is 3
consumer : full 1
consumed item is 4
consumer : full 0
consumed item is 0
produced item is 5
producer : full 1
produced item is 6
consumer : full 0
producer : full 1
produced item is 7
producer : full 2
produced item is 8
producer : full 3
produced item is 9
producer : full 4
consumed item is 6
consumer : full 3
consumed item is 7
consumer : full 2
consumed item is 8
consumer : full 1
```

31.Program to demonstrate Dining Philosophers Problem.

Description:-

The Dining Philosophers problems is a classic synchronization problem (Introduced by E. W. Dijkstra) introducing semaphores as a conceptual synchronization mechanism. The problem is discussed in just about every operating systems textbook.

Dining Philosophers. There is a dining room containing a circular table with five chairs. At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti. Near the room are five philosophers who spend most of their time thinking, but who occasionally get hungry and need to eat so they can think some more.

In order to eat, a philosopher must sit at the table, pick up the two chopsticks to the left and right of a plate, then serve and eat the spaghetti on the plate.

Thus, each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
  { THINK;
   PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
   EAT;
   PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

A philosopher may THINK indefinately. Every philosopher who EATs will eventually finish. Philosophers may PICKUP and PUTDOWN their chopsticks in either order, or non-deterministically, but these are atomic actions, and, of course, two philosophers cannot use a single CHOPSTICK at the same time.

The problem is to design a protocol to satisfy the liveness condition: any philosopher who tries to EAT, eventually does.

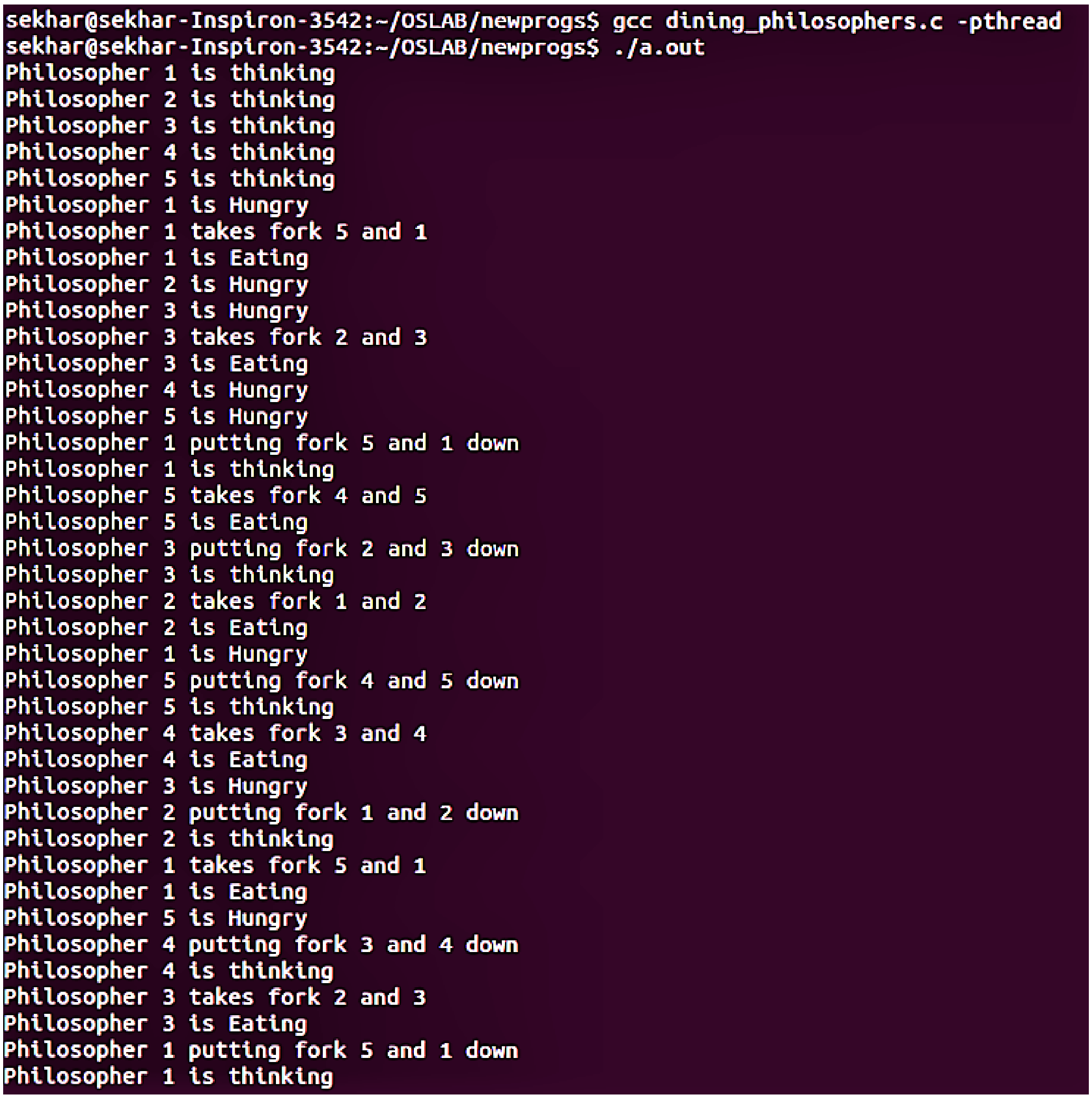
Program:-

```
#include<stdio.h>

#include<semaphore.h>

#include<pthread.h>


#define N 5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

#define LEFT (ph_num+4)%N

#define RIGHT (ph_num+1)%N


sem_t mutex;

sem_t S[N];


void * philospher(void *num);

void take_fork(int);

void put_fork(int);

void test(int);


int state[N];

int phil_num[N]={0,1,2,3,4};


int main()

{
    int i;

    pthread_t thread_id[N];

    sem_init(&mutex,0,1);

    for(i=0;i<N;i++)

        sem_init(&S[i],0,0);

    for(i=0;i<N;i++)

    {

        pthread_create(&thread_id[i],NULL,philospher,&phil_num[i]);

        printf("Philosopher %d is thinking\n",i+1);

    }

    for(i=0;i<N;i++)

        pthread_join(thread_id[i],NULL);

}


void *philospher(void *num)

{

    while(1)

    {

        int *i = num;

        sleep(1);

        take_fork(*i);

        sleep(0);

        put_fork(*i);

    }

}
```

```
void take_fork(int ph_num)

{

   sem_wait(&mutex);

   state[ph_num] = HUNGRY;

   printf("Philosopher %d is Hungry\n",ph_num+1);

   test(ph_num);

   sem_post(&mutex);

   sem_wait(&S[ph_num]);

   sleep(1);

}


void test(int ph_num)

{

   if (state[ph_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)

   {

      state[ph_num] = EATING;

      sleep(2);

      printf("Philosopher %d takes fork %d and %d\n",ph_num+1,LEFT+1,ph_num+1);

      printf("Philosopher %d is Eating\n",ph_num+1);

      sem_post(&S[ph_num]);

   }

}


void put_fork(int ph_num)

{

   sem_wait(&mutex);
```

```
    state[ph_num] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",ph_num+1,LEFT+1,ph_num+1);

    printf("Philosopher %d is thinking\n",ph_num+1);

    test(LEFT);

    test(RIGHT);

    sem_post(&mutex);

}
```

Output:-



```
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ gcc dining_philosophers.c -pthread
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ ./a.out
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 5 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
```

INTER PROCESS COMMUNICATION

32.Program to demonstrate Shared Memory.

Description:-

Shared Memory is a type of IPC where the two processes share same memory chunk and use it for IPC. One process writes into that memory and other reads it.

Program(SERVER):-

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include<stdlib.h>

#define SHMSZ 27


void main()

{

    char c;

    int shmid;

    key_t key;

    char *shm, *s;


    /*

     * We'll name our shared memory segment

     * "5678".

     */

    key = 5678;
```

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

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

/*
 * Now put some things into the memory for the
 * other process to read.
 */
s = shm;

for (c = 'a'; c <= 'z'; c++)
   *s++ = c;
*s = '\0';
```

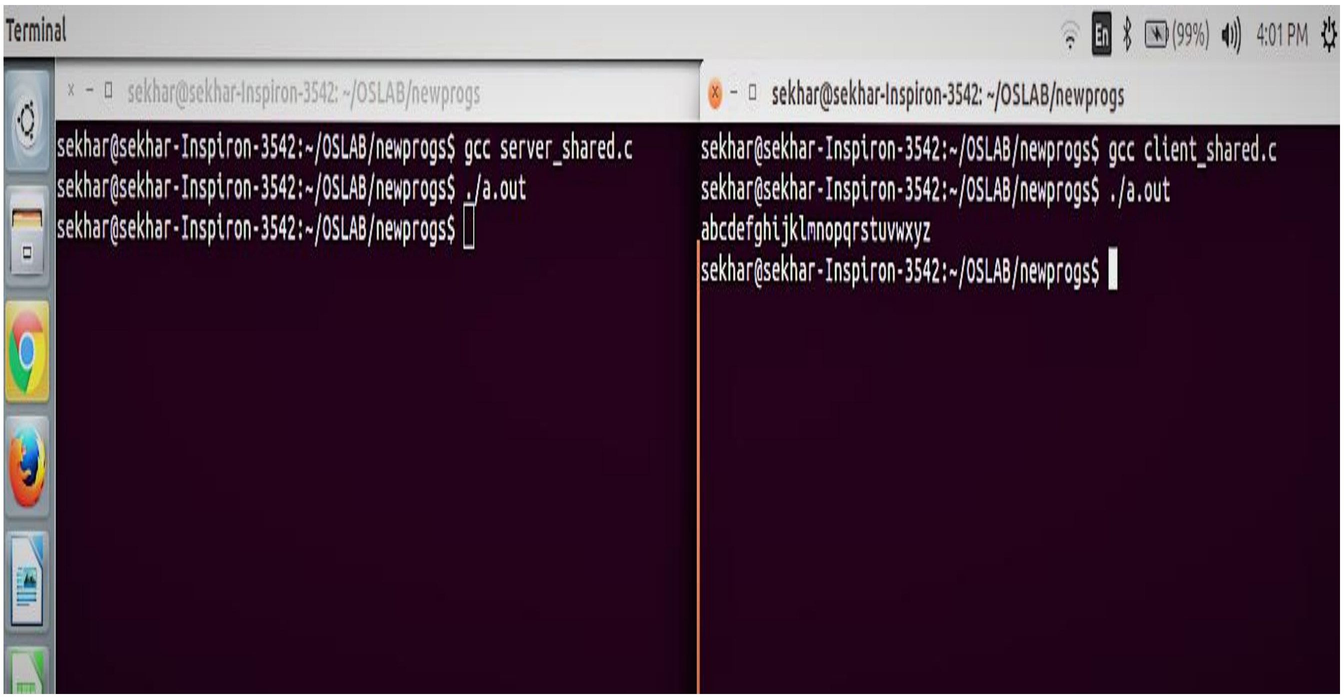
```
    /*
     * Finally, we wait until the other process
     * changes the first character of our memory
     * to '*', indicating that it has read what
     * we put there.
     */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

Program(Client):-

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include<stdlib.h>

#define SHMSZ 27


void main()

{

    int shmid;

    key_t key;

    char *shm, *s;


    /*

     * We need to get the segment named

     * "5678", created by the server.

     */

    key = 5678;


    /*

     * Locate the segment.

     */

    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {

        perror("shmget");
    exit(1);
}

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

/*
 * Now read what the server put in the memory.
 */
for (s = shm; *s != '\0'; s++)
    putchar(*s);
putchar('\n');

/*
 * Finally, change the first character of the
 * segment to '*', indicating we have read
 * the segment.
 */
*shm = '*';

exit(0);
```

}

Output:-


Terminal
4:01 PM
sekhar@sekhar-Inspiron-3542: ~/OSLAB/newprogs
sekhar@sekhar-Inspiron-3542: ~/OSLAB/newprogs

```
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ gcc server_shared.c
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ ./a.out
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$
```

```
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ gcc client_shared.c
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ ./a.out
abcdefghijklmnopqrstuvwxyz
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$
```

33.Program to demonstrate Message Queues.

Description:-

A message queue provide an asynchronous communications protocol, a system that puts a message onto a message queue does not require an immediate response to continue processing. Email is probably the best example of asynchronous messaging. When an email is sent can the sender continue processing other things without an immediate response from the receiver. This way of handling messages decouple the producer from the consumer. The producer and the consumer of the message do not need to interact with the message queue at the same time.

Program(Sender):-

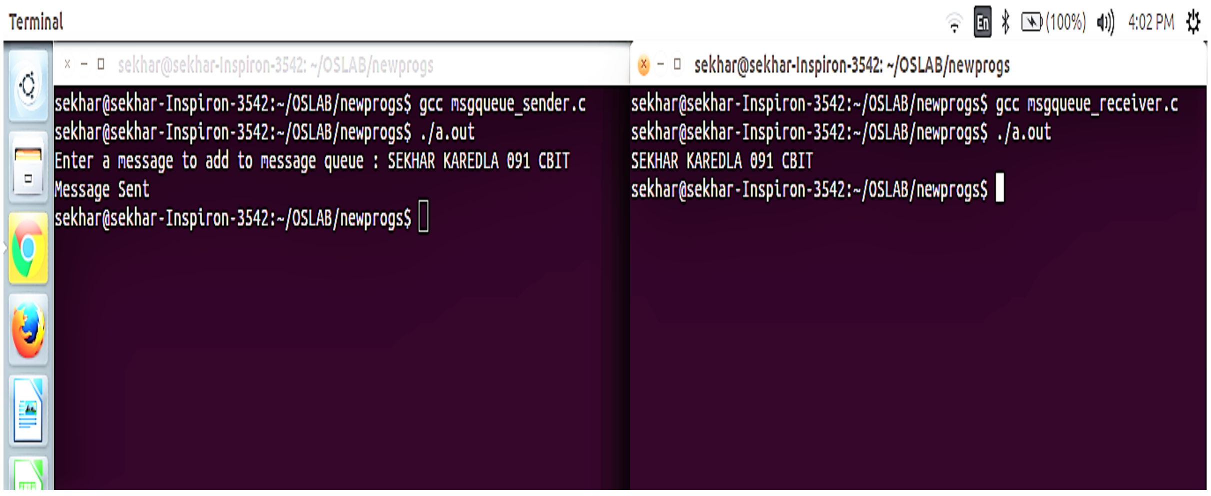
```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXSIZE     128



struct msgbuf

{

   long    mtype;

   char    mtext[MAXSIZE];

};


main()
```

```
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    struct msgbuf sbuf;
    size_t buflen;

    key = 1234;

    if ((msqid = msgget(key, msgflg )) < 0)   //Get the message queue ID for the given key
      {perror("msgget");exit(1);}

    //Message Type
    sbuf.mtype = 1;

    printf("Enter a message to add to message queue : ");
    scanf("%[^\n]",sbuf.mtext);
    getchar();

    buflen = strlen(sbuf.mtext) + 1 ;

    if (msgsnd(msqid, &sbuf, buflen, IPC_NOWAIT) < 0)
    {
        printf ("%d, %ld, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext,(int)buflen);
        perror("msgsnd");
            exit(1);
    }
    else
        printf("Message Sent\n");

    exit(0);
}
```

Program(Receiver):-

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE     128


struct msgbuf

{

    long    mtype;

    char    mtext[MAXSIZE];

};




main()

{

    int msqid;

    key_t key;

    struct msgbuf rcvbuffer;


    key = 1234;


    if ((msqid = msgget(key, 0666)) < 0)

      {perror("msgget()");exit(1);}
     //Receive an answer of message type 1.
    if (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)
      {perror("msgrcv");exit(1);}


    printf("%s\n", rcvbuffer.mtext);
    exit(0);
}
```

Output:-



```
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ gcc msgqueue_sender.c
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ ./a.out
Enter a message to add to message queue : SEKHAR KAREDLA 091 CBIT
Message Sent
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$
```

```
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ gcc msgqueue_receiver.c
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$ ./a.out
SEKHAR KAREDLA 091 CBIT
sekhar@sekhar-Inspiron-3542:~/OSLAB/newprogs$
```