**OS - Practical – 7**

**Name: Rishabh Jain**

**Branch & Sem: CSE - IV**

**Sec: A**

**Roll no.:54**

**AIM:** Write C programs to implement threads and semaphores for process synchronization.

**7A  
THREADS**

**Program 1:**

A simple C program to demonstrate use of pthread basic functions and to implement multiple threads with global and static variables.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

// Let us create a global variable to change it in threads

int g = 0;

// The function to be executed by all threads

void \*myThreadFun(void \*vargp)

{

    // Store the value argument passed to this thread

    int myid = (int)vargp;

    // Let us create a static variable to observe its changes

    static int s = 0;

    // Change static and global variables

    ++s;

    ++g;

    // Print the argument, static and global variables

    printf("Thread ID : %d,Static : %d, Global : %d\n", myid, ++s, ++g);

}

int main()

{

    int i;

    pthread\_t tid;

    // Let us create 3 Threads

    for (i = 0; i < 3; i++)

        pthread\_create(&tid, NULL, myThreadFun, (void \*)i);

    pthread\_exit(NULL);

    return 0;

}

**OUTPUT:**

****

**Program 2:**

To demonstrate thread system calls

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

pthread\_t tid[2];

void \*doSomeThing(void \*arg)

{

    unsigned long i = 0;

    pthread\_t id = pthread\_self();

    if (pthread\_equal(id, tid[0]))

        printf("\nFirst Thread Processing \n");

    else

        printf("\nSecond Thread Processing \n");

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

        ;

    return NULL;

}

int main(void)

{

    int i = 0;

    int err;

    while (i < 2)

    {

        err = pthread\_create(&tid[i], NULL, &doSomeThing, NULL);

        if (err != 0)

            printf("\nCan't create Thread : [%s]", strerror(err));

        else

            printf("\n Thread created successfully\n");

        i++;

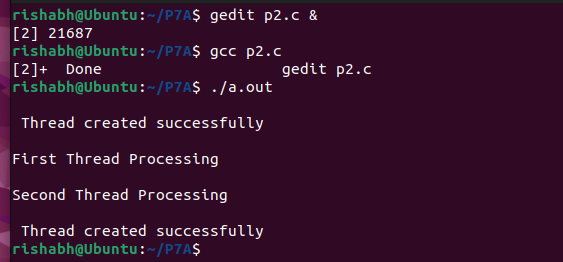
    }

    sleep(5);

    return 0;

}

OUTPUT:



**Program 3:**

Matrix Multiplication using Threads

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#define size 3

int A[size][size] = {{1, 0, 0}, {1, 2, 0}, {1, 2, 3}};

int B[size][size] = {{1, 2, 3}, {4, 0, 6}, {0, 8, 9}};

int C[size][size];

typedef struct

{

    int row;

    int col;

} m;

void \*mult(void \*args)

{

    m \*index = (m \*)args;

    int r = index->row;

    int c = index->col;

    for (int i = 0; i < size; i++)

    {

        C[r][c] += A[r][i] \* B[i][c];

    }

}

int main()

{

    pthread\_t t[size][size];

    printf("A matrix :\n");

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            printf("%d ", A[i][j]);

        }

        printf("\n");

    }

    printf("B matrix :\n");

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            printf("%d ", B[i][j]);

        }

        printf("\n");

    }

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            m \*index = (m \*)malloc(sizeof(m));

            index->row = i;

            index->col = j;

            pthread\_create(&t[i][j], NULL, mult, (void \*)index);

        }

    }

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            pthread\_join(t[i][j], NULL);

        }

    }

    printf("Answer of Multiplication :\n");

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            printf("%d ", C[i][j]);

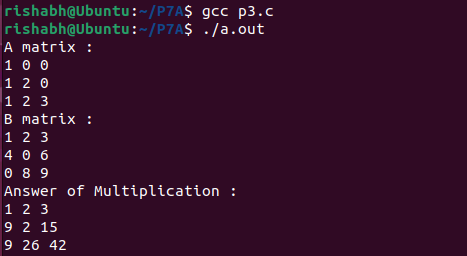
        }

        printf("\n");

    }

}

**OUTPUT:**

****

**Program 4:**

Linear search using Multi-threading (use n number of threads)

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define max 20

#define thread\_max 4

int a[max] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,

              19, 20};

int key = 15;

int flag = 0, i;

int current\_thread = 0;

void \*LinearSearch(void \*args)

{

    int num = current\_thread++;

    for (num \*(max / 4); i < ((num + 1) \* (max / 4)); i++)

    {

        if (a[i] == key)

            flag = 1;

    }

}

int main()

{

    pthread\_t thread[thread\_max];

    for (int i = 0; i < thread\_max; i++)

        pthread\_create(&thread[i], NULL, LinearSearch, (void \*)NULL);

    for (int i = 0; i < thread\_max; i++)

        pthread\_join(thread[i], NULL);

    if (flag == 1)

        printf("Element found (%d)\n", key);

    else

        printf("Element not present(%d)\n", key);

}

**OUTPUT:**

****

**Program 5:**

To find the maximum and minimum element in an array using Multithreading (for 100 to 200 numbers or more and create 10 or more threads)

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define size 100

#define thread\_max 10

int a[size];

int max = -9999;

int min = 9999;

void \*find(void \*args)

{

    int tid = (int \*)args;

    int subarray\_size = size / thread\_max;

    int s\_index = tid \* subarray\_size;

    int e\_index = s\_index + subarray\_size;

    int MAX = -9999;

    int MIN = 9999;

    for (int i = s\_index; i < e\_index; i++)

    {

        if (a[i] > MAX)

            MAX = a[i];

        else if (a[i] < MIN)

            MIN = a[i];

    }

    if (MAX > max)

        max = MAX;

    if (MIN < min)

        min = MIN;

    pthread\_exit(NULL);

}

int main()

{

    pthread\_t t[thread\_max];

    for (int i = 0; i < size; i++)

    {

        a[i] = rand() % 1000;

    }

    for (int i = 0; i < thread\_max; i++)

    {

        pthread\_create(&t[i], NULL, find, i);

    }

    for (int i = 0; i < thread\_max; i++)

    {

        pthread\_join(t[i], NULL);

    }

    printf("Maximum Number of the array is %d\n", max);

    printf("Minimum Number of the array is %d\n", min);

}

**OUTPUT:**

****

**Program 6:**

Example without synchronization FOR PRODUCER CONSUMER PROBLEM.

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

void \*producer(); // the thread

void \*consumer(); // the thread

int main()

{

    pthread\_t ptid, ctid; // Thread Id for Producer &

    Consumer

        pthread\_create(&ptid, NULL, producer, NULL); // Producer

    pthread\_create(&ctid, NULL, consumer, NULL);     // Consumer

    pthread\_join(ptid, NULL);

    pthread\_join(ctid, NULL);

}

// The thread will begin control in this function

void \*producer(void \*param)

{

    do

    {

        printf("I am Producer\n");

    } while (1);

    pthread\_exit(0);

}

// The thread will begin control in this function

void \*consumer(void \*param)

{

    do

    {

        printf("I am Consumer\n");

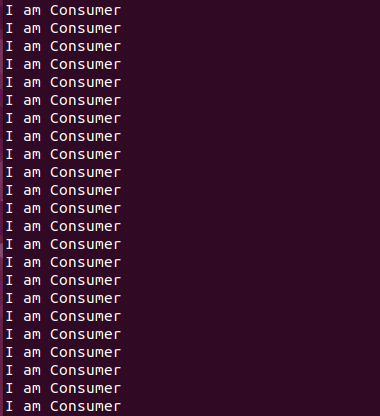
    } while (1);

    pthread\_exit(0);

}

**OUTPUT:**

****

****

Here “I am Producer” and “I am Consumer ” will be printed infinitely. No synchronization between Producer and Consumer.

**Program 7:**

Example with synchronization : using mutex & condition variable FOR PRODUCER CONSUMER PROBLEM

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define BufferSize 10

void \*Producer();

void \*Consumer();

int BufferIndex = -1;

char BUFFER[10];

pthread\_cond\_t Buffer\_Empty = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t Buffer\_Full = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t mVar = PTHREAD\_MUTEX\_INITIALIZER;

int main()

{

    pthread\_t ptid, ctid;

    pthread\_create(&ptid, NULL, Producer, NULL);

    pthread\_create(&ctid, NULL, Consumer, NULL);

    pthread\_join(ptid, NULL);

    pthread\_join(ctid, NULL);

    return 0;

}

void \*Producer()

{

    int i;

    for (i = 0; i < 15; i++)

    {

        pthread\_mutex\_lock(&mVar);

        if (BufferIndex == BufferSize - 1)

            pthread\_cond\_wait(&Buffer\_Empty, &mVar);

        BUFFER[++BufferIndex] = '#';

        printf("Produce : %d \n", BufferIndex);

        pthread\_mutex\_unlock(&mVar);

        pthread\_cond\_signal(&Buffer\_Full);

    }

}

void \*Consumer()

{

    int i;

    for (i = 0; i < 15; i++)

    {

        pthread\_mutex\_lock(&mVar);

        if (BufferIndex == -1)

        {

            pthread\_cond\_wait(&Buffer\_Full, &mVar);

        }

        printf("Consume : %d \n", BufferIndex--);

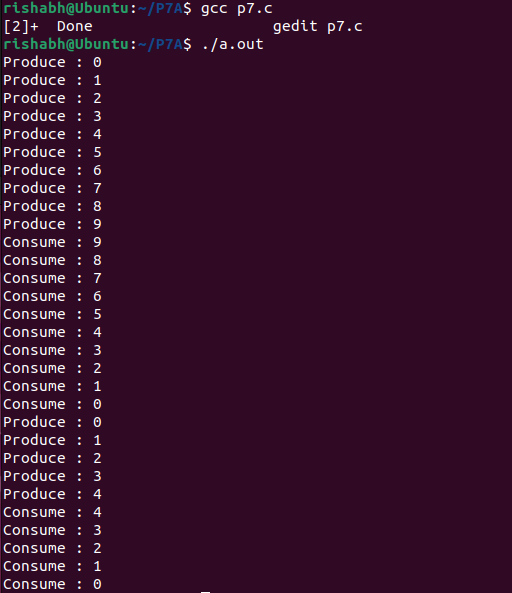
        pthread\_mutex\_unlock(&mVar);

        pthread\_cond\_signal(&Buffer\_Empty);

    }

}

**OUTPUT:**

****

**Program 8:**

Readers Writers Problem solved with mutex and pthread.

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define readers 5

#define writers 2

int sharedResource = 0;

pthread\_mutex\_t resourceLock = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t readerCountLock = PTHREAD\_MUTEX\_INITIALIZER;

int readerCount = 0;

void \*reader(void \*arg)

{

    int readerId = \*(int \*)arg;

    pthread\_mutex\_lock(&readerCountLock);

    readerCount++;

    if (readerCount == 1)

    {

        pthread\_mutex\_lock(&resourceLock);

    }

    pthread\_mutex\_unlock(&readerCountLock);

    printf("Reader %d reads shared resource: %d\n", readerId, sharedResource);

    pthread\_mutex\_lock(&readerCountLock);

    readerCount--;

    if (readerCount == 0)

    {

        pthread\_mutex\_unlock(&resourceLock);

    }

    pthread\_mutex\_unlock(&readerCountLock);

    pthread\_exit(NULL);

}

void \*writer(void \*arg)

{

    int writerId = \*(int \*)arg;

    pthread\_mutex\_lock(&resourceLock);

    sharedResource = writerId;

    printf("Writer %d writes shared resource: %d\n", writerId,

           sharedResource);

    pthread\_mutex\_unlock(&resourceLock);

    pthread\_exit(NULL);

}

int main()

{

    pthread\_t readerThreads[readers];

    pthread\_t writerThreads[writers];

    int readerIds[readers];

    int writerIds[writers];

    for (int i = 0; i < writers; i++)

    {

        writerIds[i] = i + 1;

        pthread\_create(&writerThreads[i], NULL, writer, &writerIds[i]);

    }

    for (int i = 0; i < readers; i++)

    {

        readerIds[i] = i + 1;

        pthread\_create(&readerThreads[i], NULL, reader, &readerIds[i]);

    }

    for (int i = 0; i < readers; i++)

    {

        pthread\_join(readerThreads[i], NULL);

    }

    for (int i = 0; i < writers; i++)

    {

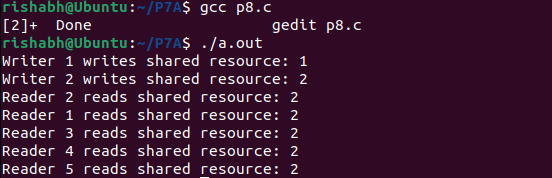
        pthread\_join(writerThreads[i], NULL);

    }

    return 0;

}

**OUTPUT:**

****

**7A  
SEMAPHORE**

**Program 1:**

C PROGRAM FOR PRODUCER CONSUMER PROBLEM with synchronization using semaphores for n producer and n consumer)

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <pthread.h>

#define buffersize 10

pthread\_mutex\_t mutex;

pthread\_t tidP[20], tidC[20];

sem\_t full, empty;

int counter;

int buffer[buffersize];

void intialize()

{

    pthread\_mutex\_init(&mutex, NULL);

    sem\_init(&full, 1, 0);

    sem\_init(&empty, 1, buffersize);

    counter = 0;

}

void write(int item)

{

    buffer[counter++] = item;

}

int read()

{

    return (buffer[--counter]);

}

void \*producer(void \*param)

{

    int waittime, item, i;

    item = rand() % 5;

    waittime = rand() % 5;

    sem\_wait(&empty);

    pthread\_mutex\_lock(&mutex);

    printf("\nProducer has produced item : %d\n", item);

    write(item);

    pthread\_mutex\_unlock(&mutex);

    sem\_post(&full);

}

void \*consumer(void \*param)

{

    int waittime, item;

    waittime = rand() % 5;

    sem\_wait(&full);

    pthread\_mutex\_lock(&mutex);

    item = read();

    printf("\nConsumer has consumed item : %d\n", item);

    pthread\_mutex\_unlock(&mutex);

    sem\_post(&empty);

}

int main()

{

    int n1, n2, i;

    intialize();

    printf("Enter the no of Producers : \n");

    scanf("%d", &n1);

    printf("\nEnter the no of Consumers : \n");

    scanf("%d", &n2);

    for (i = 0; i < n1; i++)

        pthread\_create(&tidP[i], NULL, producer, NULL);

    for (i = 0; i < n2; i++)

        pthread\_create(&tidC[i], NULL, consumer, NULL);

    for (i = 0; i < n1; i++)

        pthread\_join(tidP[i], NULL);

    for (i = 0; i < n2; i++)

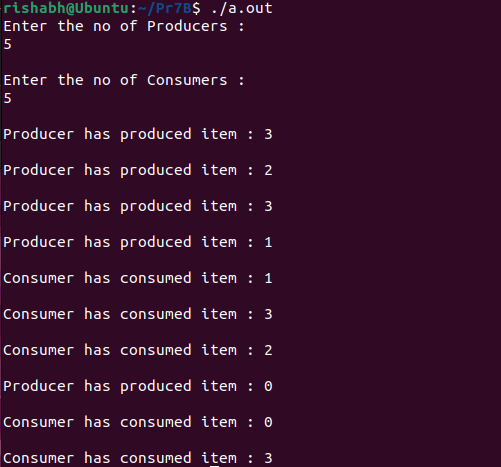
        pthread\_join(tidC[i], NULL);

    // sleep(5);

    exit(0);

}

**OUTPUT:**

****

**Program 2:**

PRODUCER-CONSUMER PROBLEM – using SEMAPHORE (for one producer and one consumer)

#include <stdio.h>

#include <stdlib.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++)

    {

        sem\_wait(&empty);

        sem\_wait(&mutex);

        printf("Producer item is %d\n", i);

        buf[(++r) % 5] = i;

        sleep(1);

        sem\_post(&mutex);

        sem\_post(&full);

    }

}

void \*consume(void \*arg)

{

    int item, i;

    for (i = 0; i < 10; i++)

    {

        sem\_wait(&full);

        printf("Full %u\n", full);

        sem\_wait(&mutex);

        item = buf[(++f) % 5];

        printf("Consumed item is %d\n", item);

        sleep(1);

        sem\_post(&mutex);

        sem\_post(&empty);

    }

}

int main()

{

    pthread\_t tid1, tid2;

    sem\_init(&mutex, 0, 1);

    sem\_init(&empty, 0, 5);

    sem\_init(&full, 0, 0);

    pthread\_create(&tid1, NULL, produce, NULL);

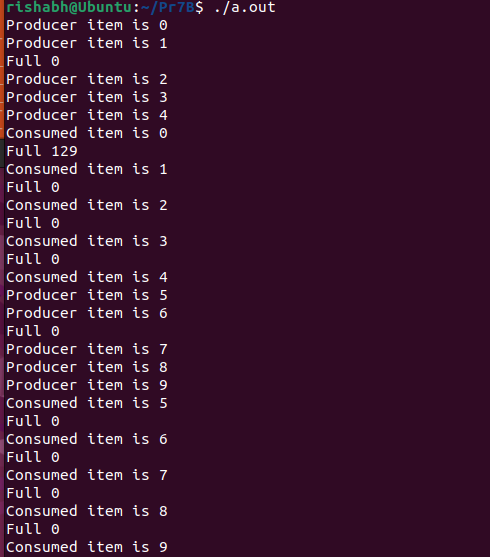
    pthread\_create(&tid2, NULL, consume, NULL);

    pthread\_join(tid1, NULL);

    pthread\_join(tid2, NULL);

}

**OUTPUT:**

****

**Program 3:**

Write a program to create an integer variable using shared memory concept and increment the variable simultaneously by two processes. Use semaphores to avoid race conditions.

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <pthread.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/shm.h>

#include <sys/ipc.h>

int \*a;

sem\_t mutex;

int main()

{

    sem\_init(&mutex, 0, 1);

    int shmid;

    shmid = shmget(IPC\_PRIVATE, 2 \* sizeof(int), 0777 | IPC\_CREAT);

    a = (int \*)shmat(shmid, NULL, 0);

    \*a = 0;

    pid\_t pid;

    pid = fork();

    if (pid == -1)

    {

        printf("Error !!\n");

    }

    else if (pid == 0)

    {

        // Child Process

        printf("This is the Child Process.\n");

        for (int i = 0; i < 10; i++)

        {

            sem\_wait(&mutex);

            printf("Child prints %d\n", (\*a)++);

            sem\_post(&mutex);

        }

    }

    else if (pid > 0)

    {

        // Parent Process

        printf("This is the Parent Process.\n");

        for (int i = 0; i < 10; i++)

        {

            sem\_wait(&mutex);

            printf("Parent prints %d\n", (\*a)++);

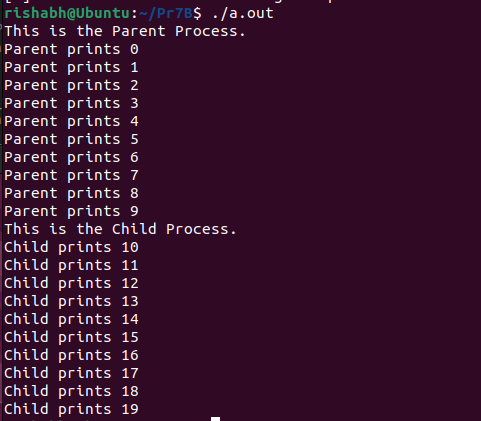
            sem\_post(&mutex);

        }

    }

}

OUTPUT:



Program 4:

Producer - Consumer problem solved with semaphores and shared memory.

header.h

#include <stdio.h>

#include <semaphore.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <fcntl.h>

#include <sys/shm.h>

#define BUFFER\_SIZE 10

#define CONSUMER\_SLEEP\_SEC 3

#define PRODUCER\_SLEEP\_SEC 1

#define KEY 1010

// A structure to store BUFFER and semaphores for synchronization

typedef struct

{

    int buff[BUFFER\_SIZE];

    sem\_t mutex, empty, full;

} MEM;

// Method for shared memory allocation

MEM \*memory()

{

    key\_t key = KEY;

    int shmid;

    shmid = shmget(key, sizeof(MEM), IPC\_CREAT | 0666);

    return (MEM \*)shmat(shmid, NULL, 0);

}

void init()

{

    // Initialize structure pointer with shared memory

    MEM \*M = memory();

    // Initialize semaphores

    sem\_init(&M->mutex, 1, 1);

    sem\_init(&M->empty, 1, BUFFER\_SIZE);

    sem\_init(&M->full, 1, 0);

}

Producer.c

#include "header.h"

void producer()

{

    int i = 0, n;

    MEM \*S = memory();

    while (1)

    {

        i++;

        sem\_wait(&S->empty); // Semaphore down operation

        sem\_wait(&S->mutex);

        sem\_getvalue(&S->full, &n);

        S->buff[n] = i; // Place value to BUFFER

        printf("[PRODUCER] Placed item [%d]\n", i);

        sem\_post(&S->mutex);

        sem\_post(&S->full); // Semaphore up operation

        sleep(PRODUCER\_SLEEP\_SEC);

    }

}

int main()

{

    init();

    producer();

    return 0;

}

Consumer.c

#include "header.h"

void consumer()

{

    int n;

    MEM \*S = memory();

    while (1)

    {

        sem\_wait(&S->full);         // Semaphore down operation

        sem\_wait(&S->mutex);        // Semaphore for mutual exclusion

        sem\_getvalue(&S->full, &n); // Assign value of semaphore full to

        integer n

            printf("[CONSUMER] Removed item [%d]\n", S->buff[n]);

        sem\_post(&S->mutex); // Mutex up operation

        sem\_post(&S->empty); // Semaphore up operation

        sleep(CONSUMER\_SLEEP\_SEC);

    }

}

int main()

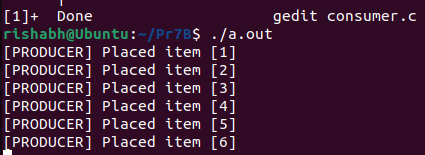
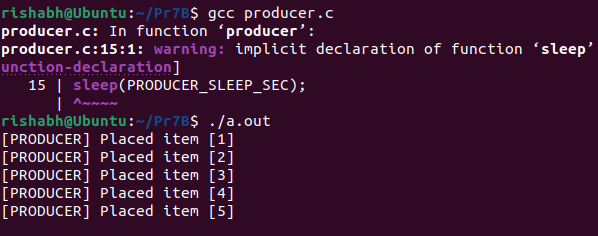
{

    consumer();

    return 0;

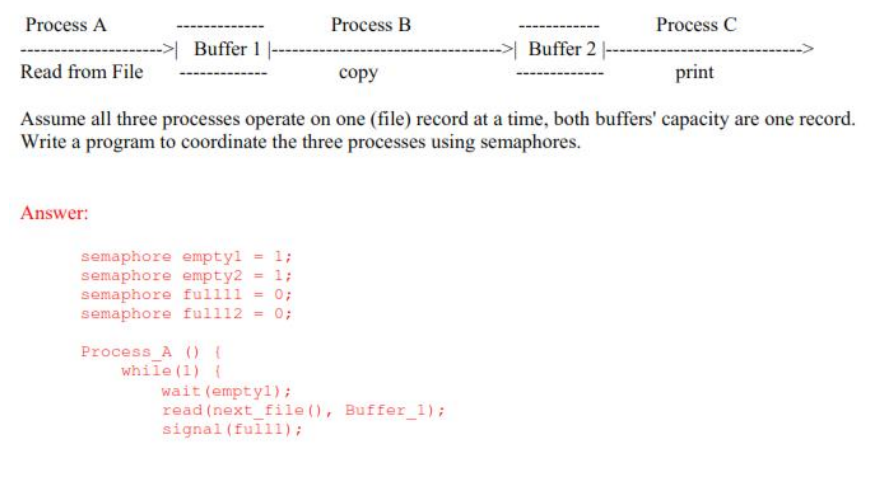
}

**OUTPUT:**

****

**Program 5:**

Implement C program for the processes given below using semaphores and system calls required

****

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/shm.h>

#include <sys/ipc.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <string.h>

sem\_t empty1, empty2, full1, full2;

void \*PA(void \*args)

{

    int \*a = (int \*)args;

    int val = 32;

    for (int i = 0; i < 3; i++)

    {

        sem\_wait(&empty1);

        sem\_wait(&empty2);

        printf("Process A is writing into the buffer 1 : %d\n", val \* i);

        \*a = val \* i;

        sem\_post(&empty2);

        sem\_post(&full1);

    }

}

void \*PB(void \*args)

{

    int \*a = (int \*)args;

    int copy;

    for (int i = 0; i < 3; i++)

    {

        sem\_wait(&full1);

        copy = \*a;

        sem\_post(&empty1);

        sem\_wait(&empty2);

printf("Process B is copting contents from buffer 1 to buffer 2 :

%d\n", copy);

\*a = copy;

sem\_post(&full2);

    }

}

void \*PC(void \*args)

{

    int \*a = (int \*)args;

    for (int i = 0; i < 3; i++)

    {

sem\_wait(&full2);

printf("Process C takes data from buffer 2 : %d\n", \*a);

sem\_post(&empty2);

    }

}

int main()

{

    key\_t key;

    key = 5678;

    pthread\_t TA, TB, TC;

    int \*a; // shared variable

    int shmid = shmget(key, sizeof(int), IPC\_CREAT | 0666);

    a = shmat(shmid, NULL, 0);

    sem\_init(&empty1, 0, 1);

    sem\_init(&empty2, 0, 1);

    sem\_init(&full1, 0, 0);

    sem\_init(&full2, 0, 0);

    pthread\_create(&TA, NULL, PA, a);

    pthread\_create(&TB, NULL, PB, a);

    pthread\_create(&TC, NULL, PC, a);

    pthread\_join(TA, NULL);

    pthread\_join(TB, NULL);

    pthread\_join(TC, NULL);

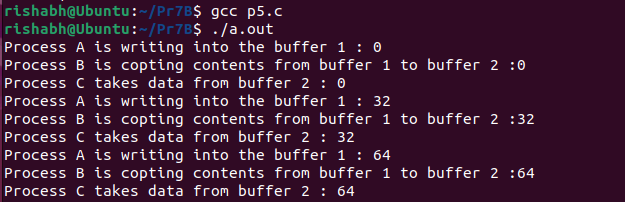
    shmdt(a);

    shmctl(shmid, IPC\_RMID, NULL);

    return 0;

}

**OUTPUT:**

****

**Program 6:**

Readers Writers Problem solved with semaphores and shared memory

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <semaphore.h>

#define readers 5

#define writers 2

typedef struct

{

    int sharedData;

    sem\_t rwMutex;

    sem\_t mutex;

    int readersCount;

} SharedMemory;

void readerProcess(SharedMemory \*sharedMemory, int readerId)

{

    while (1)

    {

        sem\_wait(&(sharedMemory->mutex));

        sharedMemory->readersCount++;

        if (sharedMemory->readersCount == 1)

        {

            sem\_wait(&(sharedMemory->rwMutex));

        }

        sem\_post(&(sharedMemory->mutex));

        printf("Reader %d reads shared data: %d\n", readerId, sharedMemory - > sharedData);

        sem\_wait(&(sharedMemory->mutex));

        sharedMemory->readersCount--;

        if (sharedMemory->readersCount == 0)

        {

            sem\_post(&(sharedMemory->rwMutex));

        }

        sem\_post(&(sharedMemory->mutex));

        usleep(rand() % 1000000);

    }

}

void writerProcess(SharedMemory \*sharedMemory, int writerId)

{

    while (1)

    {

        sem\_wait(&(sharedMemory->rwMutex));

        sharedMemory->sharedData = writerId;

        printf("Writer %d writes shared data: %d\n", writerId, sharedMemory - > sharedData);

        sem\_post(&(sharedMemory->rwMutex));

        usleep(rand() % 1000000);

    }

}

int main()

{

    int shmid;

    SharedMemory \*sharedMemory;

    shmid = shmget(IPC\_PRIVATE, sizeof(SharedMemory), IPC\_CREAT | 0666);

    sharedMemory = (SharedMemory \*)shmat(shmid, NULL, 0);

    sharedMemory->sharedData = 0;

    sem\_init(&(sharedMemory->rwMutex), 1, 1);

    sem\_init(&(sharedMemory->mutex), 1, 1);

    sharedMemory->readersCount = 0;

    for (int i = 0; i < readers; i++)

    {

        pid\_t pid = fork();

        if (pid < 0)

        {

            perror("fork error");

            exit(1);

        }

        else if (pid == 0)

        {

            readerProcess(sharedMemory, i + 1);

            exit(0);

        }

    }

    for (int i = 0; i < writers; i++)

    {

        pid\_t pid = fork();

        if (pid < 0)

        {

            perror("fork error");

            exit(1);

        }

        else if (pid == 0)

        {

            writerProcess(sharedMemory, i + 1);

            exit(0);

        }

    }

    for (int i = 0; i < readers + writers; i++)

    {

        wait(NULL);

    }

    shmdt(sharedMemory);

    shmctl(shmid, IPC\_RMID, NULL);

    return 0;

}

**OUTPUT:**

****

**Program 7:**

Readers Writers Problem solved with semaphores and pthread.

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define readers 5

#define writers 2

int sharedData = 0;

sem\_t rwMutex, mutex;

int readersCount = 0;

void \*reader(void \*arg)

{

    int readerId = \*(int \*)arg;

    while (1)

    {

        sem\_wait(&mutex);

        readersCount++;

        if (readersCount == 1)

        {

            sem\_wait(&rwMutex);

        }

        sem\_post(&mutex);

        printf("Reader %d reads shared data: %d\n", readerId, sharedData);

        sem\_wait(&mutex);

        readersCount--;

        if (readersCount == 0)

        {

            sem\_post(&rwMutex);

        }

        sem\_post(&mutex);

        usleep(rand() % 1000000);

    }

    pthread\_exit(NULL);

}

void \*writer(void \*arg)

{

    int writerId = \*(int \*)arg;

    while (1)

    {

        sem\_wait(&rwMutex);

        sharedData = writerId;

        printf("Writer %d writes shared data: %d\n", writerId, sharedData);

        sem\_post(&rwMutex);

        usleep(rand() % 1000000);

    }

    pthread\_exit(NULL);

}

int main()

{

    pthread\_t readerThreads[readers];

    pthread\_t writerThreads[writers];

    int readerIds[readers];

    int writerIds[writers];

    sem\_init(&rwMutex, 0, 1);

    sem\_init(&mutex, 0, 1);

    for (int i = 0; i < readers; i++)

    {

        readerIds[i] = i + 1;

        pthread\_create(&readerThreads[i], NULL, reader, &readerIds[i]);

    }

    for (int i = 0; i < writers; i++)

    {

        writerIds[i] = i + 1;

        pthread\_create(&writerThreads[i], NULL, writer, &writerIds[i]);

    }

    for (int i = 0; i < readers; i++)

    {

        pthread\_join(readerThreads[i], NULL);

    }

    for (int i = 0; i < writers; i++)

    {

        pthread\_join(writerThreads[i], NULL);

    }

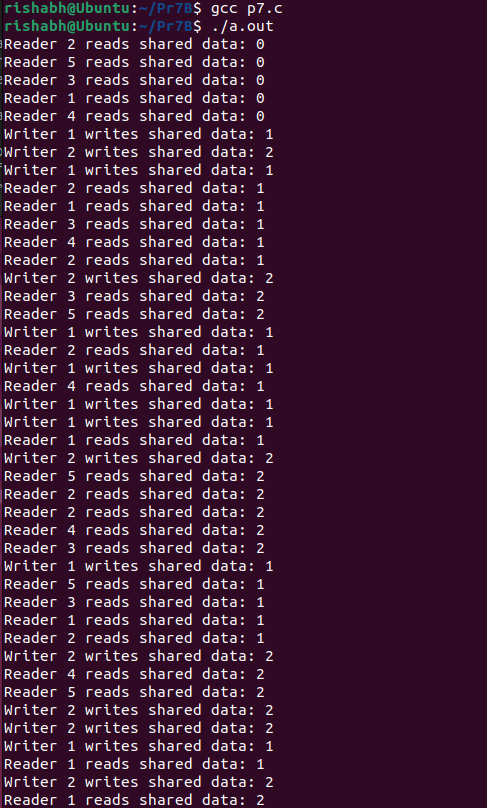
    sem\_destroy(&rwMutex);

    sem\_destroy(&mutex);

    return 0;

}

**OUTPUT:**

****

**Program 8:**

Cook cooks pizza and puts that pizza onto shelf. Waiter picks pizza from the shelf and serves it to customers. The shelf can hold three pizza at most at the same time. When the shelf is full, cook wait until picked up; when there is no pizza on the shelf, waiter waits until made. Hint: We use three semaphores to synchronize cook and waiter. cook.c is a producer program. After cooking one pizza and placing it on shelf, it posts semaphore fill and makes shelf increase by 1. waiter.c is a consumer program. After picking a pizza from the shelf, it posts semaphore avail and makes shelf decrease by 1. The value of shelf is the current number of pizza and processes should access it exclusively. Semaphore mutex is response for the mutual exclusion. Note that shared memory is used in these two programs

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define MAX\_PIZZA 3

int shelf = 0;

sem\_t mutex;

void \*cook(void \*arg)

{

    while (1)

    {

        if (shelf < MAX\_PIZZA)

        {

            sem\_wait(&mutex);

            shelf++;

            printf("Cook: Cooked a pizza, there are %d pizzas now.\n", shelf);

            sem\_post(&mutex);

        }

        sleep(1);

    }

}

void \*waiter(void \*arg)

{

    while (1)

    {

        if (shelf > 0)

        {

            sem\_wait(&mutex);

            printf("Waiter: I picked up a pizza\n");

            shelf--;

            sem\_post(&mutex);

        }

        sleep(2);

    }

}

int main()

{

    pthread\_t Tcook, Twaiter;

    sem\_init(&mutex, 0, 1);

    printf("Cook: I have started cooking pizza.\n");

    pthread\_create(&Tcook, NULL, cook, NULL);

    pthread\_create(&Twaiter, NULL, waiter, NULL);

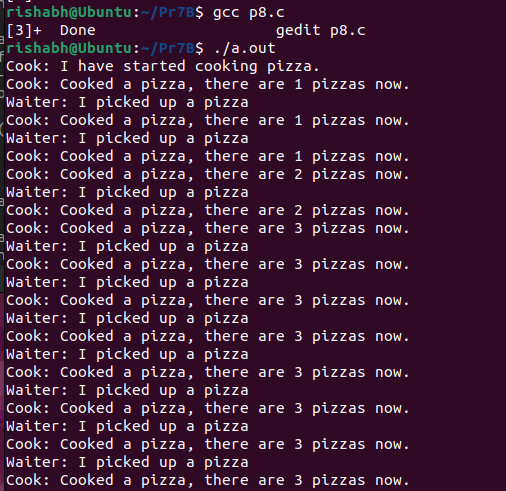
    pthread\_join(Tcook, NULL);

    pthread\_join(Twaiter, NULL);

    return 0;

}

**OUTPUT:**

****

**Result :**

Linux C programs to demonstrate the concept of threads and semaphores for process synchronization has been implemented.