**Collage : Vishwakarma Institute of Technology**

**Course Name : Operating System**

**Name : Vedika Vikas Sontakke**

**Roll no : 37**

**PRN NO 12220206**

**Assignment No – 5**

**Synchronization Problems**

* **Solve below problems using semaphore and mutex**

1. Reader Writer Problem

Program :

#include<stdio.h>

#include<pthread.h>

#include<semaphore.h>

sem\_t mutex,writeblock;

int data = 0,rcount = 0;

void \*reader(void \*arg)

{

  int f;

  f = ((int)arg);

  sem\_wait(&mutex);

  rcount = rcount + 1;

  if(rcount==1)

   sem\_wait(&writeblock);

  sem\_post(&mutex);

  printf("Data read by the reader%d is %d\n",f,data);

  sleep(1);

  sem\_wait(&mutex);

  rcount = rcount - 1;

  if(rcount==0)

   sem\_post(&writeblock);

  sem\_post(&mutex);

}

void \*writer(void \*arg)

{

  int f;

  f = ((int) arg);

  sem\_wait(&writeblock);

  data++;

  printf("Data writen by the writer%d is %d\n",f,data);

  sleep(1);

  sem\_post(&writeblock);

}

int main()

{

  int i,b;

  pthread\_t rtid[5],wtid[5];

  sem\_init(&mutex,0,1);

  sem\_init(&writeblock,0,1);

  for(i=0;i<=2;i++)

  {

    pthread\_create(&wtid[i],NULL,writer,(void \*)i);

    pthread\_create(&rtid[i],NULL,reader,(void \*)i);

  }

  for(i=0;i<=2;i++)

  {

    pthread\_join(wtid[i],NULL);

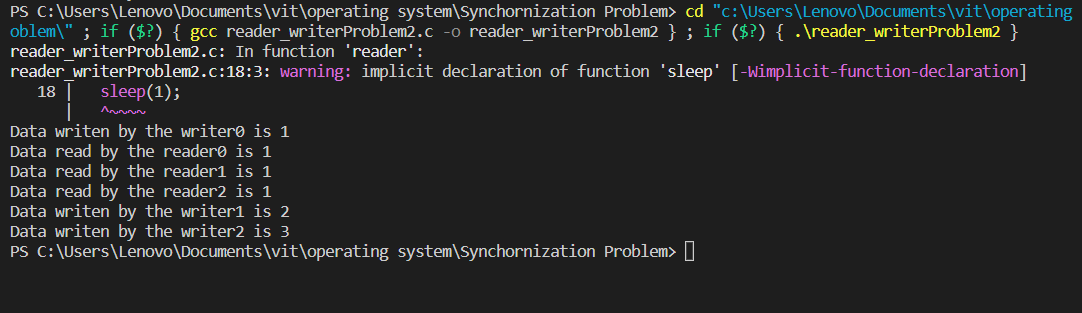
    pthread\_join(rtid[i],NULL);

  }

  return 0;

}

Output :



1. Producer–Consumer Problem

Program :

// C program for the above approach

#include <stdio.h>

#include <stdlib.h>

int mutex = 1;                            // Initialize a mutex to 1

int full = 0;                           // Number of full slots as 0

int empty = 10, x = 0;                 // Number of empty slots as size of buffer

// Function to produce an item and add it to the buffer

void producer()

{

    --mutex;         // Decrease mutex value by 1

    ++full;         // Increase the number of full slots by 1

    --empty;       // Decrease the number of empty slots by 1

    x++;          // Item produced

    printf("\nProducer produces item %d ",x);

    ++mutex;      // Increase mutex value by 1

}

// Function to consume an item and remove it from buffer

void consumer()

{

    --mutex;            // Decrease mutex value by 1

    --full;            // Decrease the number of full slots by 1

    ++empty;          // Increase the number of empty slots by 1

    printf("\nConsumer consumes item %d ",x);

    x--;

    ++mutex;        // Increase mutex value by 1

}

// Driver Code

int main()

{

    int n, i;

    printf("\n1. Press 1 for Producer"

        "\n2. Press 2 for Consumer"

        "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for can give wrong value due to synchronization issues.

// 'critical' specifies that code is executed by only one thread at a time i.e.,

// only one thread enters the critical section at a given time

#pragma omp critical

    for (i = 1; i > 0; i++) {

        printf("\nEnter your choice: ");

        scanf("%d", &n);

        switch (n) {

        case 1:

            // If mutex is 1 and empty is non-zero, then it is possible to produce

            if ((mutex == 1) && (empty != 0)) producer();

            // Otherwise, print buffer is full

            else printf("Buffer is full!");

            break;

        case 2:

            // If mutex is 1 and full is non-zero, then it is possible to consume

            if ((mutex == 1) && (full != 0)) consumer();

            // Otherwise, print Buffer is empty

            else printf("Buffer is empty!");

            break;

        // Exit Condition

        case 3:

            exit(0);

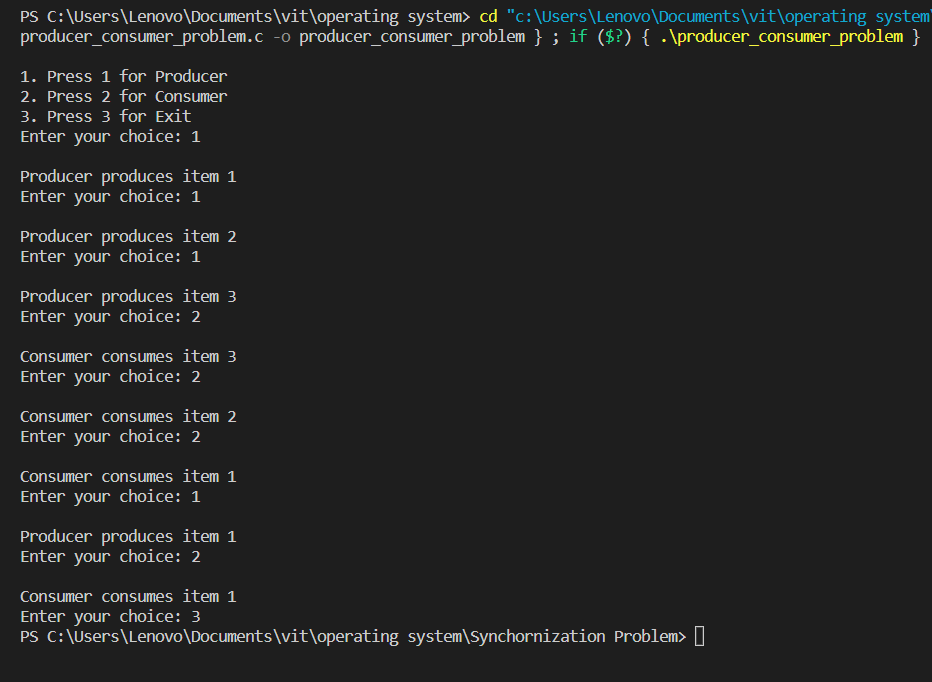
            break;

        }

    }

}

Output :



1. The Dining Philosophers Problem.

Program :

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

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

sem\_t mutex;

sem\_t S[N];

void test(int phnum)

{

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

        state[phnum] = EATING;                  // state that eating

        sleep(2);

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

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

        // sem\_post(&S[phnum]) has no effect during takefork used to wake up hungry philosophers during putfork

        sem\_post(&S[phnum]);

    }

}

// take up chopsticks

void take\_fork(int phnum)

{

    sem\_wait(&mutex);

    state[phnum] = HUNGRY;              // state that hungry

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

    test(phnum);                        // eat if neighbours are not eating

    sem\_post(&mutex);

    sem\_wait(&S[phnum]);                // if unable to eat wait to be signalled

    sleep(1);

}

// put down chopsticks

void put\_fork(int phnum)

{

    sem\_wait(&mutex);

    state[phnum] = THINKING;            // state that thinking

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

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

    test(LEFT);

    test(RIGHT);

    sem\_post(&mutex);

}

void\* philosopher(void\* num)

{

    while (1) {

        int\* i = num;

        sleep(1);

        take\_fork(\*i);

        sleep(0);

        put\_fork(\*i);

    }

}

int main()

{

    int i;

    pthread\_t thread\_id[N];

    sem\_init(&mutex, 0, 1);             // initialize the semaphores

    for (i = 0; i < N; i++)

          sem\_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {

        pthread\_create(&thread\_id[i], NULL, philosopher, &phil[i]);             // create philosopher processes

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

    }

    for (i = 0; i < N; i++)

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

}

Output :

