**Name : Tanishq Thuse**

**Branch : CS(AI)**

**Year : SY**

**Div : B**

**Roll no. : 60**

**PRN : 12310237**

**Title : OS Assignment-3 Classical Problems of Synchronization**

**Q1) Producer-Consumer Problem**

**Code :**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

// Shared buffer and semaphores

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty, full, mutex;

// Producer thread function

void\* producer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&empty); // Wait for empty slot

sem\_wait(&mutex); // Enter critical section

buffer[in] = i; // Produce an item

printf("Produced: %d\n", i);

in = (in + 1) % BUFFER\_SIZE;

sem\_post(&mutex); // Exit critical section

sem\_post(&full); // Signal that buffer has an item

}

return NULL;

}

// Consumer thread function

void\* consumer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&full); // Wait for an item

sem\_wait(&mutex); // Enter critical section

int item = buffer[out]; // Consume an item

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

out = (out + 1) % BUFFER\_SIZE;

sem\_post(&mutex); // Exit critical section

sem\_post(&empty); // Signal that buffer has an empty slot

}

return NULL;

}

int main() {

pthread\_t prod, cons;

// Initialize semaphores

sem\_init(&empty, 0, BUFFER\_SIZE); // Buffer starts empty

sem\_init(&full, 0, 0); // Buffer starts with no items

sem\_init(&mutex, 0, 1); // Mutex for critical section

// Create producer and consumer threads

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

// Wait for threads to finish

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

// Destroy semaphores

sem\_destroy(&empty);

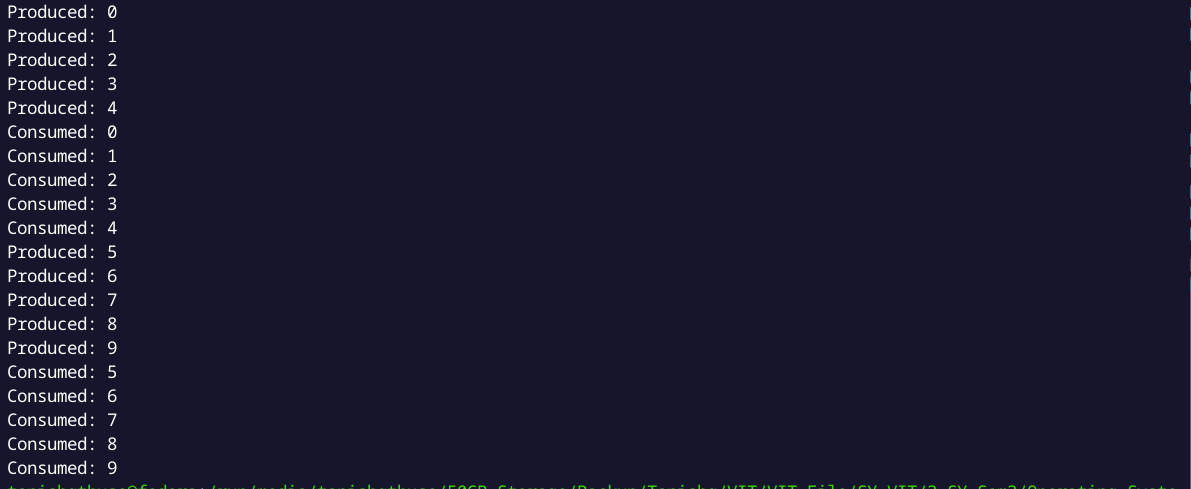
sem\_destroy(&full);

sem\_destroy(&mutex);

return 0;

}

Output :



**Q2) Reader-Writer Problem**

**Code :**

// Reader-Writer problem using monitors

#include <iostream>

#include <pthread.h>

#include <unistd.h>

using namespace std;

class monitor {

private:

// no. of readers

int rcnt;

// no. of writers

int wcnt;

// no. of readers waiting

int waitr;

// no. of writers waiting

int waitw;

// condition variable to check whether reader can read

pthread\_cond\_t canread;

// condition variable to check whether writer can write

pthread\_cond\_t canwrite;

// mutex for synchronization

pthread\_mutex\_t condlock;

public:

monitor()

{

rcnt = 0;

wcnt = 0;

waitr = 0;

waitw = 0;

pthread\_cond\_init(&canread, NULL);

pthread\_cond\_init(&canwrite, NULL);

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

}

// mutex provide synchronization so that no other thread

// can change the value of data

void beginread(int i)

{

pthread\_mutex\_lock(&condlock);

// if there are active or waiting writers

if (wcnt == 1 || waitw > 0) {

// incrementing waiting readers

waitr++;

// reader suspended

pthread\_cond\_wait(&canread, &condlock);

waitr--;

}

// else reader reads the resource

rcnt++;

cout << "reader " << i << " is reading\n";

pthread\_mutex\_unlock(&condlock);

pthread\_cond\_broadcast(&canread);

}

void endread(int i)

{

// if there are no readers left then writer enters monitor

pthread\_mutex\_lock(&condlock);

if (--rcnt == 0)

pthread\_cond\_signal(&canwrite);

pthread\_mutex\_unlock(&condlock);

}

void beginwrite(int i)

{

pthread\_mutex\_lock(&condlock);

// a writer can enter when there are no active

// or waiting readers or other writer

if (wcnt == 1 || rcnt > 0) {

++waitw;

pthread\_cond\_wait(&canwrite, &condlock);

--waitw;

}

wcnt = 1;

cout << "writer " << i << " is writing\n";

pthread\_mutex\_unlock(&condlock);

}

void endwrite(int i)

{

pthread\_mutex\_lock(&condlock);

wcnt = 0;

// if any readers are waiting, threads are unblocked

if (waitr > 0)

pthread\_cond\_signal(&canread);

else

pthread\_cond\_signal(&canwrite);

pthread\_mutex\_unlock(&condlock);

}

}

// global object of monitor class

M;

void\* reader(void\* id)

{

int c = 0;

int i = \*(int\*)id;

// each reader attempts to read 5 times

while (c < 5) {

usleep(1);

M.beginread(i);

M.endread(i);

c++;

}

}

void\* writer(void\* id)

{

int c = 0;

int i = \*(int\*)id;

// each writer attempts to write 5 times

while (c < 5) {

usleep(1);

M.beginwrite(i);

M.endwrite(i);

c++;

}

}

int main()

{

pthread\_t r[5], w[5];

int id[5];

for (int i = 0; i < 5; i++) {

id[i] = i;

// creating threads which execute reader function

pthread\_create(&r[i], NULL, &reader, &id[i]);

// creating threads which execute writer function

pthread\_create(&w[i], NULL, &writer, &id[i]);

}

for (int i = 0; i < 5; i++) {

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

}

for (int i = 0; i < 5; i++) {

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

}

}

Output :



**Q3) Dining Philosopher Problem**

**Code :**

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

#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 };

std::mutex mutex;

std::condition\_variable S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating

state[phnum] = EATING;

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

std::cout << "Philosopher " << phnum + 1 << " takes fork " << LEFT + 1 << " and " << phnum + 1 << std::endl;

std::cout << "Philosopher " << phnum + 1 << " is Eating" << std::endl;

// notify the philosopher that he can start eating

S[phnum].notify\_all();

}

}

// take up chopsticks

void take\_fork(int phnum)

{

std::unique\_lock<std::mutex> lock(mutex);

// state that hungry

state[phnum] = HUNGRY;

std::cout << "Philosopher " << phnum + 1 << " is Hungry" << std::endl;

// eat if neighbours are not eating

test(phnum);

// if unable to eat wait to be signalled

S[phnum].wait(lock);

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

}

// put down chopsticks

void put\_fork(int phnum)

{

std::unique\_lock<std::mutex> lock(mutex);

// state that thinking

state[phnum] = THINKING;

std::cout << "Philosopher " << phnum + 1 << " putting fork " << LEFT + 1 << " and " << phnum + 1 << " down" << std::endl;

std::cout << "Philosopher " << phnum + 1 << " is thinking" << std::endl;

test(LEFT);

test(RIGHT);

}

void philosopher(int num)

{

while (true) {

take\_fork(num);

put\_fork(num);

}

}

int main()

{

std::thread threads[N];

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

threads[i] = std::thread(philosopher, i);

std::cout << "Philosopher " << i + 1 << " is thinking" << std::endl;

}

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

threads[i].join();

return 0; }

**Output :**

****