Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Technology

(An Autonomous Institute affiliated to Savitribai Phule Pune University)

**Assignment -3**

|  |  |
| --- | --- |
| **Subject** | Operating System |
| **Name** | Ajinkya Walunj |
| **Class** | CS-A |
| **Roll No.** | 05 |

**Title**: Write a solution on synchronization problems

1. **Producer Consumer (solution using mutex)**

Producer-Consumer problem is a classical synchronization problem in the operating system. With the presence of more than one process and limited resources in the system the synchronization problem arises. If one resource is shared between more than one process at the same time then it can lead to data inconsistency. In the producer-consumer problem, the producer produces an item and the consumer consumes the item produced by the producer.

*Code :-*

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE];

int count = 0; // number of items in buffer

pthread\_mutex\_t mutex; // mutex lock

pthread\_cond\_t empty, full; // condition variables

void \*producer(void \*arg) {

    int item;

    while (1) {

        item = rand(); // produce an item

        pthread\_mutex\_lock(&mutex);

        while (count == BUFFER\_SIZE) // buffer is full

            pthread\_cond\_wait(&empty, &mutex); // wait for a consumer

        buffer[count++] = item;

        printf("Produced item %d\n", item);

        pthread\_cond\_signal(&full); // signal a consumer

        pthread\_mutex\_unlock(&mutex);

    }

}

void \*consumer(void \*arg) {

    int item;

    while (1) {

        pthread\_mutex\_lock(&mutex);

        while (count == 0) // buffer is empty

            pthread\_cond\_wait(&full, &mutex); // wait for a producer

        item = buffer[--count];

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

        pthread\_cond\_signal(&empty); // signal a producer

        pthread\_mutex\_unlock(&mutex);

    }

}

int main() {

    pthread\_t prod, cons;

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

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

    pthread\_cond\_init(&full, NULL);

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

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

    pthread\_join(prod, NULL);

    pthread\_join(cons, NULL);

    pthread\_mutex\_destroy(&mutex);

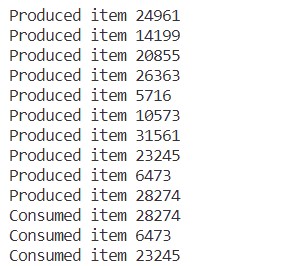
    pthread\_cond\_destroy(&empty);

    pthread\_cond\_destroy(&full);

    return 0;

}

*Output :-*

**

1. **Reader – Writer problem (solution using semaphore)**

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets. The readers-writers problem is used for managing synchronization among various reader and writer process so that there are no problems with the data sets, i.e. no inconsistency is generated.

*Code :-*

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

// #define READ\_COUNT 5

// #define WRITE\_COUNT 5

sem\_t mutex, resource;

int readCount = 0;

void \*reader(void \*arg) {

    int id = \*(int \*)arg;

    sem\_wait(&mutex);

    readCount++;

    if (readCount == 1) {

        sem\_wait(&resource);

    }

    sem\_post(&mutex);

    printf("Reader %d is reading.\n", id);

    sem\_wait(&mutex);

    readCount--;

    if (readCount == 0) {

        sem\_post(&resource);

    }

    sem\_post(&mutex);

    pthread\_exit(NULL);

}

void \*writer(void \*arg) {

    int id = \*(int \*)arg;

    sem\_wait(&resource);

    printf("Writer %d is writing.\n", id);

    sem\_post(&resource);

    pthread\_exit(NULL);

}

int main() {

    int READ\_COUNT,WRITE\_COUNT;

    printf("Enter the number of readers: ");

    scanf("%d",&READ\_COUNT);

    printf("Enter the number of writers: ");

    scanf("%d",&WRITE\_COUNT);

    pthread\_t readers[READ\_COUNT], writers[WRITE\_COUNT];

    int i, readerId[READ\_COUNT], writerId[WRITE\_COUNT];

    sem\_init(&mutex, 0, 1);

    sem\_init(&resource, 0, 1);

    //Following we created readers

    for (i = 0; i < READ\_COUNT; i++) {

        readerId[i] = i;

        pthread\_create(&readers[i], NULL, reader, &readerId[i]);

    }

    //Following we created writers

    for (i = 0; i < WRITE\_COUNT; i++) {

        writerId[i] = i;

        pthread\_create(&writers[i], NULL, writer, &writerId[i]);

    }

    //Following we joined threads

    for (i = 0; i < READ\_COUNT; i++) {

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

    }

    //Following we joined

    for (i = 0; i < WRITE\_COUNT; i++) {

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

    }

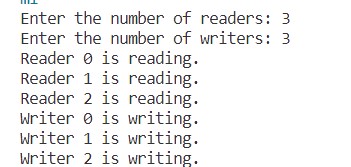
    sem\_destroy(&mutex);

    sem\_destroy(&resource);

    return 0;

}

*Output :-*

**

1. **Producer Consumer (solution using semaphore)**

*Code:-*

*Output:-*

1. **Reader Writer (solution using mutex)**

*Code:-*

*Output:-*

1. **Dinning Philosopher problem using semaphore**

*Code :-*

*Output :-*