Experiment No.6
Process Management: Synchronization a. Write a C program to implement the solution of the Producer consumer problem through Semaphore.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Write a C program to implement the solution of the Producer consumer problem through Semaphore.

Objective:

Producer consumer problem through Semaphore.

Theory:

The Producer-Consumer problem is a classic synchronization problem where there are two types of processes, producers and consumers, that share a common, fixed-size buffer. Producers put items into the buffer, and consumers take items out of the buffer. The problem is to ensure that the producers do not produce items into a full buffer and that the consumers do not consume items from an empty buffer.

Here's a C program that implements the Producer-Consumer problem using semaphores for synchronization:

Code :

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define BUFFER_SIZE 5


// Define the buffer and semaphores

int buffer[BUFFER_SIZE];

sem_t empty, full, mutex;

int in = 0, out = 0;
```

```
// Producer function
void *producer(void *arg) {
    int item;
    for (int i = 0; i < BUFFER_SIZE * 2; i++) {
        item = rand() % 100; // Produce a random item

        sem_wait(&empty); // Wait if buffer is full
        sem_wait(&mutex); // Acquire the mutex

        buffer[in] = item;
        printf("Produced item: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex); // Release the mutex
        sem_post(&full); // Signal that buffer is no longer empty

        sleep(1); // Sleep for some time
    }
    pthread_exit(NULL);
}

// Consumer function
void *consumer(void *arg) {
    int item;
```

```
    for (int i = 0; i < BUFFER_SIZE * 2; i++) {
        sem_wait(&full); // Wait if buffer is empty
        sem_wait(&mutex); // Acquire the mutex

        item = buffer[out];
        printf("Consumed item: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex); // Release the mutex
        sem_post(&empty); // Signal that buffer is no longer full

        sleep(2); // Sleep for some time
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);
```

```
    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

In this program:

We define a buffer of fixed size BUFFER_SIZE, along with three semaphores: empty (initialized to BUFFER_SIZE), full (initialized to 0), and mutex (initialized to 1).

The producer function produces items and puts them into the buffer. It waits if the buffer is full and acquires the mutex before accessing the buffer.

The consumer function consumes items from the buffer. It waits if the buffer is empty and acquires the mutex before accessing the buffer.

In the main function, we create two threads for the producer and consumer functions using pthread_create.

We wait for the threads to finish using pthread_join.

Finally, we destroy the semaphores using sem_destroy.

Compile this program using:

gcc -o producer_consumer producer_consumer.c -lpthread

Then run it:

./producer_consumer

You should see the producer producing items and the consumer consuming them, synchronized by the semaphores.

Output :

```
Produced item: 22
Produced item: 79
Produced item: 85
Produced item: 10
Produced item: 7
Consumed item: 22
Consumed item: 79
Consumed item: 85
Consumed item: 10
Consumed item: 7
Produced item: 44
Produced item: 93
Produced item: 17
Produced item: 91
Produced item: 60
Consumed item: 44
Consumed item: 93
Consumed item: 17
Consumed item: 91
Consumed item: 60
```

Conclusion:

In conclusion, the implementation of the Producer-Consumer problem using semaphores in C showcases the power of synchronization mechanisms in concurrent programming. By employing semaphores to control access to shared resources, we ensure that producers and consumers operate in a coordinated manner, preventing issues like race conditions or deadlock.

Through this exercise, we have demonstrated how semaphores can effectively regulate the flow of data between threads, maintaining integrity and avoiding conflicts. This solution not only addresses the fundamental challenge of synchronizing multiple processes but also highlights the importance of careful resource management in concurrent systems.

Overall, the program provides a robust framework for managing the interactions between producers and consumers, serving as a practical example of how semaphores can facilitate efficient communication and coordination in multi-threaded environments.