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 solution of Producer consumer problem through Semaphore

Objective:

Solve the producer consumer problem based on semaphore

Theory:

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores.

A **semaphore** S is an integer variable that can be accessed only through two standard operations : wait() and signal().
The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

```
wait(S){

while(S<=0);   // busy waiting

S--;

}

signal(S){

S++;

}
```

To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Initialization of semaphores –
mutex = 1
Full = 0// Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

Solution for Producer –

```
do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)
```

When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of "full" is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Solution for Consumer –

```
do{

wait(full);
```

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

Program:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
sem_t empty, full, mutex;

int in = 0, out = 0;

void *producer(void *arg) {
    int item = 1;
    while (1) {
        sleep(1); // Simulate producing time

        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        printf("Producer produced item %d at position %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;
```

```
        item++;

        sem_post(&mutex);
        sem_post(&full);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    while (1) {
        sleep(2); // Simulate consuming time

        sem_wait(&full);
        sem_wait(&mutex);

        int item = buffer[out];
        printf("Consumer consumed item %d from position %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);
```

```
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

Output:

```
Producer produced item 1 at position 0
Consumer consumed item 1 from position 0
Producer produced item 2 at position 1
Producer produced item 3 at position 2
Consumer consumed item 2 from position 1
Producer produced item 4 at position 3
Producer produced item 5 at position 4
Consumer consumed item 3 from position 2
Producer produced item 6 at position 0
Producer produced item 7 at position 1
Consumer consumed item 4 from position 3
Producer produced item 8 at position 2
Producer produced item 9 at position 3
Consumer consumed item 5 from position 4
Producer produced item 10 at position 4
Consumer consumed item 6 from position 0
Producer produced item 11 at position 0
Consumer consumed item 7 from position 1
Producer produced item 12 at position 1
```

Conclusion:

What is Semaphore?

A semaphore is a synchronization primitive used in concurrent programming to control access to shared resources by multiple threads or processes. It is essentially a variable or abstract data type that acts as a signaling mechanism, allowing threads or processes to coordinate and control their access to shared resources in a mutually exclusive manner.

What are different types of Semaphore?

Binary Semaphore:
Also known as mutex (mutual exclusion) semaphore.
Has only two states: 0 and 1.
Used for controlling access to a single resource, ensuring that only one thread or process can access the resource at a time.
Typically used for implementing critical sections to prevent race conditions.
Counting Semaphore:
Can have multiple states, with a count greater than or equal to zero.
Used for controlling access to multiple instances of a resource.
Allows a specified number of threads or processes to access a resource simultaneously.
Useful for scenarios where multiple instances of a resource are available, such as a pool of connections or a fixed-size buffer.
Used for tasks like process synchronization, producer-consumer problems, and resource allocation.