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 <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 10, x = 0;
void producer()
{
   --mutex;
   ++full;
   --empty;
   x++;
   printf("\nProducer produces"
         "item %d",
         x);
   ++mutex;
}
void consumer()
{
   --mutex;
   --full;
   ++empty;
   printf("\nConsumer consumes "
```

```
        "item %d",
        x);
    x--;
    ++mutex;
}
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
        "\n2. Press 2 for Consumer"
        "\n3. Press 3 for Exit");
#pragma omp critical
    for (i = 1; i > 0; i++) {
        printf("\nEnter your choice:");
        scanf("%d", &n);
        switch (n) {
        case 1:
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }
            else {
                printf("Buffer is full!");
```

```
        }
        break;
    case 2:
        if ((mutex == 1)
            && (full != 0)) {
            consumer();}
        else {
            printf("Buffer is empty!");}
        break;
    case 3:
        exit(0);
        break;
    } }}
```

Output:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer producesitem 1
Enter your choice:1

Producer producesitem 2
Enter your choice:1

Producer producesitem 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:
```

Conclusion:

What is Semaphore?

A semaphore is a synchronization primitive used in concurrent programming to control access to shared resources. It is essentially a counter that is used to manage access to a finite set of resources, such as a critical section of code, a shared memory buffer, or a shared data structure. Semaphores can be thought of as signaling mechanisms that allow threads or processes to coordinate their activities and avoid race conditions, where multiple entities attempt to access or modify the same resource simultaneously.

What are different types of Semaphore?

Semaphores are synchronization primitives used in concurrent programming to control access to shared resources. They come in different types, each with its own characteristics and use cases.

1. Binary Semaphores: Also known as mutexes (mutual exclusion semaphores), binary semaphores are the simplest type.

2. Counting Semaphores: Unlike binary semaphores, counting semaphores can have a value greater than 1. They allow multiple threads or processes to access a shared resource simultaneously, up to a specified maximum limit.

3. Named Semaphores: Named semaphores are semaphores that are associated with a unique name within the operating system's namespace.

4. Unnamed Semaphores: Also referred to as anonymous semaphores, unnamed semaphores are not associated with a specific name. They are typically used for synchronization between threads within the same process.

5. Binary vs. Counting Semaphores: While binary and counting semaphores serve similar purposes, they differ in their usage scenarios.