Producer Consumer Problem using Semaphores

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++;

}
```

Semaphores are of two types:

1. **Binary Semaphore –** This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.
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);


// consume item from buffer


signal(mutex);

signal(empty);




}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.