Operating Systems

Experiment 6 – Producer Consumer

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

To study the concept of semaphore and solve producer consumer problem using semaphore.

Theory -

Process Synchronization

Process Synchronization is a way to coordinate processes that use shared data. It occurs in an operating system among cooperating processes. Cooperating processes are processes that share resources. While executing many concurrent processes, process synchronization helps to maintain shared data consistency and cooperating process execution. Processes have to be scheduled to ensure that concurrent access to shared data does not create inconsistencies. Data inconsistency can result in what is called a race condition. A race condition occurs when two or more operations are executed at the same time, not scheduled in the proper sequence, and not exited in the critical section correctly

Semaphore

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization. The definitions of wait and signal are as follows –

- Wait

  The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
  while (S<=0);

  S--;
}
```

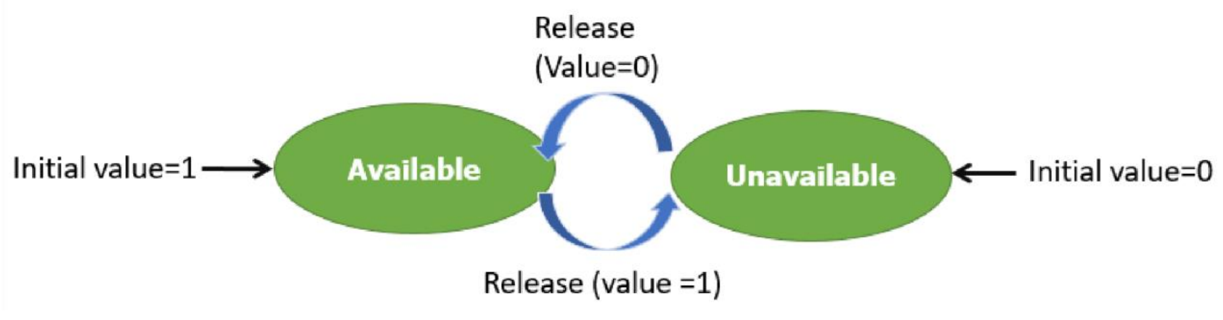
- Signal

  The signal operation increments the value of its argument S.

```
signal(S)
{
  S++;
}
```

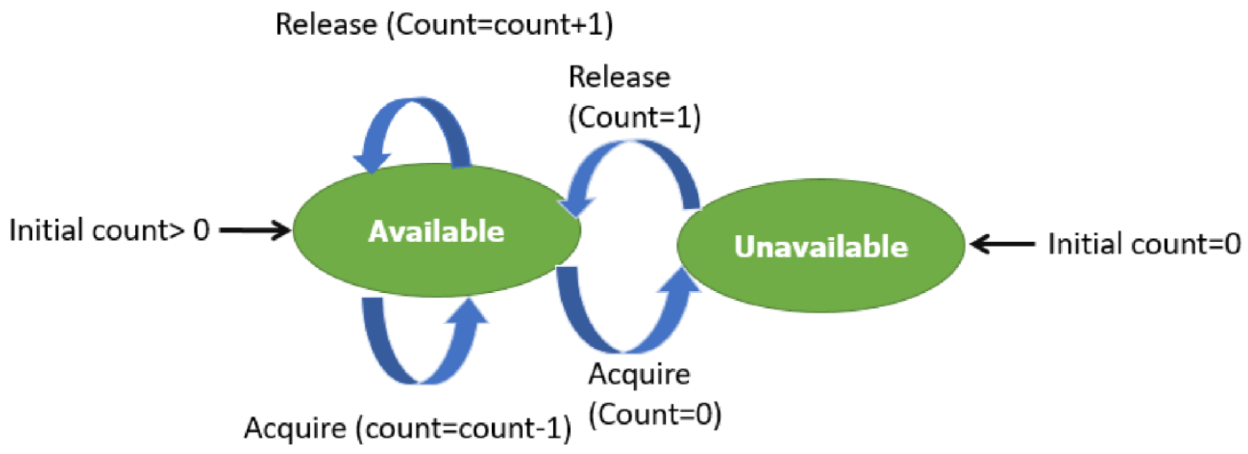
Semaphores are of two types:

1. **Binary Semaphore –**
   This is also known as mutex lock. 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 problems 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.



Advantages of Semaphores:

Some of the advantages of semaphores are as follows –

1. Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
2. There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
3. Semaphores are implemented in the machine independent code of the microkernel. So, they are machine independent.

Disadvantages of Semaphores:

Some of the disadvantages of semaphores are as follows –

1. Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
2. Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
3. Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

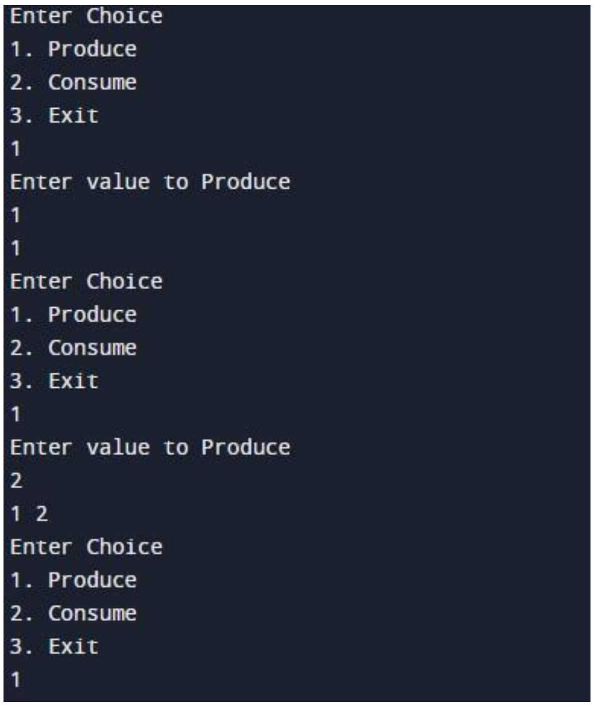
Counting Semaphore vs. Binary Semaphore

Counting Semaphore	**Binary Semaphore**
No mutual exclusion	Mutual exclusion
Any integer value	Value only 0 and 1
More than one slot	Only one slot
Provide a set of Processes	It has a mutual exclusion mechanism.

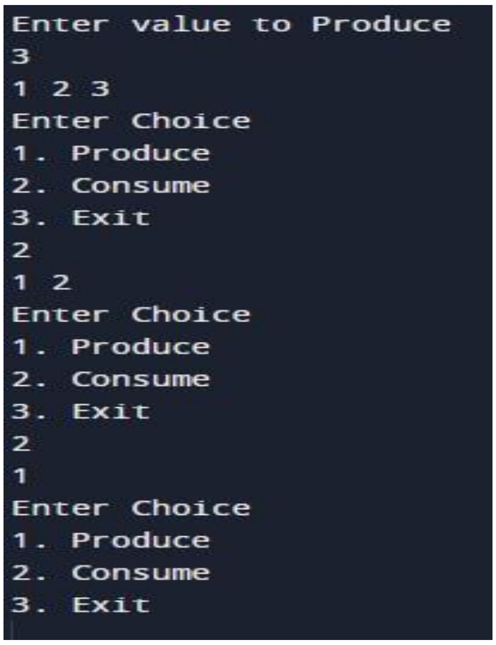
Code –

```
#include <stdio.h>
int s = 0, n = 0, e = 4, b[4];
int Wait(int *s)
{
    return (--*s);
}
int Signal(int *s)
{
    return (++*s);
}
void producer()
{
    int a;
    printf("Enter value to Produce\n");
    scanf("%d", &a);
    Wait(&e);
    Wait(&s);
    b[n] = a;
    Signal(&s);
    Signal(&n);
    for (int i = 0; i < n; i++)
    {
        printf("%d ", b[i]);
    }
    printf("\n");
}
void consumer()
{
    Wait(&n);
    Wait(&s);
    for (int i = 1; i < n; i++)
    {
        b[i - 1] = b[i];
    }
    Signal(&s);
    Signal(&e);
    for (int i = 0; i < n; ++i)
    {
        printf("%d ", b[i]);
    }
    printf("\n");
}
int main()
{
    int c;
    printf("Hello\n");
    do
    {
        printf("Enter Choice\n");
        printf("1. Produce \n2. Consume \n3. Exit\n");
        scanf("%d", &c);
        switch (c)
        {
        case 1:
            if (e == 0)
            {
                printf("Buffer is full\n");
            }
            else
            {
                producer();
            }
            break;
        case 2:
            if (e == 4)
            {
                printf("Buffer is empty\n");
            }
            else
            {
                consumer();
            }
            break;
        case 3:
            break;
        default:
            printf("Invalid Choice\n");
        }
    } while (c != 3);
    return 0;
}
```

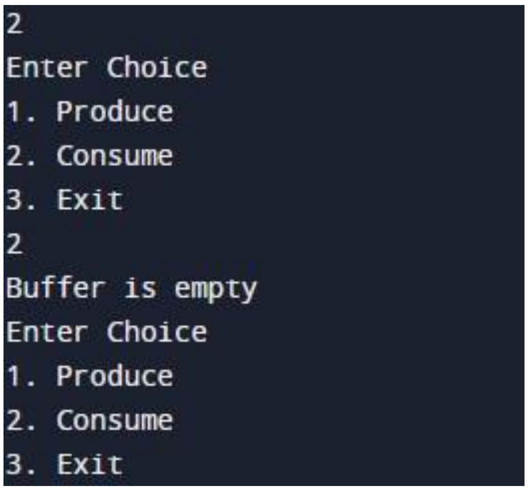
Output -



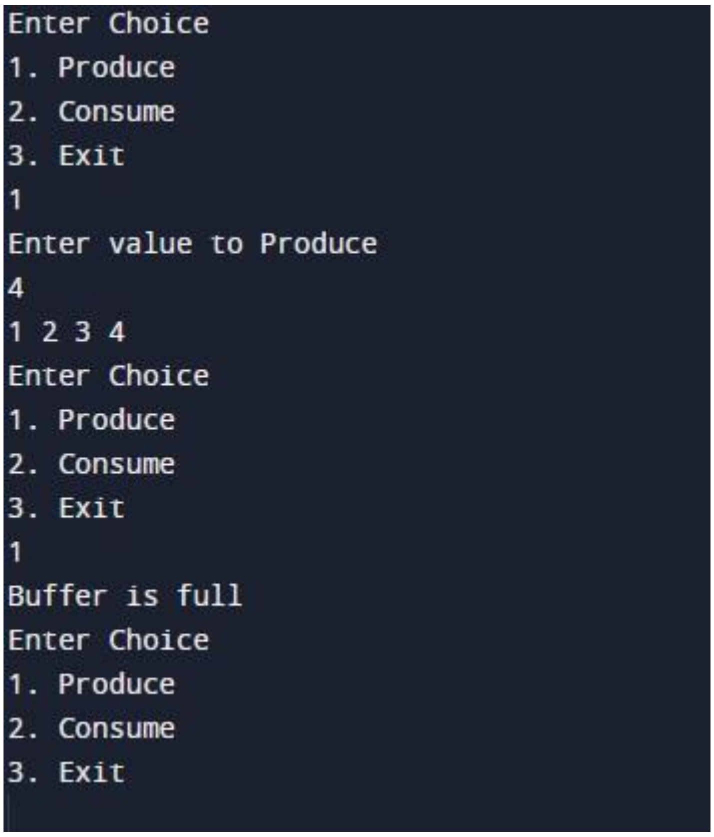
```
Enter Choice
1. Produce
2. Consume
3. Exit
1
Enter value to Produce
1
1
Enter Choice
1. Produce
2. Consume
3. Exit
1
Enter value to Produce
2
1 2
Enter Choice
1. Produce
2. Consume
3. Exit
1
```



```
Enter value to Produce
3
1 2 3
Enter Choice
1. Produce
2. Consume
3. Exit
2
1 2
Enter Choice
1. Produce
2. Consume
3. Exit
2
1
Enter Choice
1. Produce
2. Consume
3. Exit
```



```
2
Enter Choice
1. Produce
2. Consume
3. Exit
2
Buffer is empty
Enter Choice
1. Produce
2. Consume
3. Exit
```



```
Enter Choice
1. Produce
2. Consume
3. Exit
1
Enter value to Produce
4
1 2 3 4
Enter Choice
1. Produce
2. Consume
3. Exit
1
Buffer is full
Enter Choice
1. Produce
2. Consume
3. Exit
```

Conclusion -

Learnt and understood the concept of semaphore and solved producer consumer problem using semaphore.