Experiment No. 5

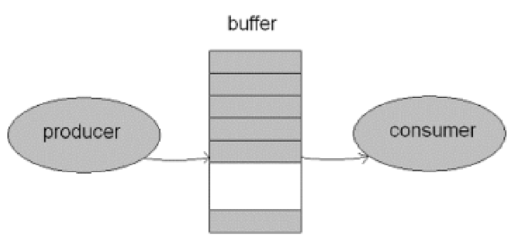
Title: Implementation of Producer-Consumer Problem Using Semaphores in C.
Aim: Write a C program to implement solution of Producer Consumer Problem through Semaphore.

Theory:

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.

A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

The producer should go to sleep when buffer is full. Next time when consumer removes data it notifies the producer and producer starts producing data again. The consumer should go to sleep when buffer is empty. Next time when producer add data it notifies the consumer and consumer starts consuming data. This solution can be achieved 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:

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.

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

Program:

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1, full = 0, empty = 3, x = 0;
void producer();
void consumer();
int wait(int);
int signal(int);
int main() {
   int n;
   printf("\n1. Producer\n2. Consumer\n3. Exit\n");
   while (1) {
      printf("\nEnter your choice: ");
      scanf("%d", &n);
      switch (n) {
         case 1:
            if ((mutex == 1) && (empty != 0)) {
               producer();
```

```
            } else {
                printf("Buffer is full!!\n");
            }
            break;
        case 2:
            if ((mutex == 1) && (full != 0)) {
                consumer();
            } else {
                printf("Buffer is empty!!\n");
            }
            break;
        case 3:
            exit(0);
            break;
        default:
            printf("Invalid choice! Please try again.\n");
      }
   }
   return 0;
}
int wait(int s) {
   return (--s);
}
int signal(int s) {
   return (++s);
}
```

```
void producer() {
   mutex = wait(mutex);
   full = signal(full);
   empty = wait(empty);
   x++;
   printf("\nProducer produces the item %d\n", x);
   mutex = signal(mutex);
}

void consumer() {
   mutex = wait(mutex);
   full = wait(full);
   empty = signal(empty);
   printf("\nConsumer consumes item %d\n", x);
   x--;
   mutex = signal(mutex);
}
```

Output:

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac4
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ gcc prog.c -o prog
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ ./prog

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
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:3
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$
```

Outcome:

Implement and analyse concepts of synchronization and deadlocks.