**OPERATING SYSTEMS**

LAB EXPERIMENT - 7

Submitted To: Prashant Giridhar Shambharkar Sir.

Submitted By: Manav Agrawal (2K19/CO/216)

---------------------------------------------------------------------

Aim:

Write a C program to simulate Producer-Consumer problem using semaphores.

Introduction:

### Producer-Consumer Problem:

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

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 which 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 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.

1. Counting Semaphore :

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

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++;*

*}*

Producer-Consumer Problem:

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.

Semaphores Initialization:

*mutex = 1*

*Full = 0 // Initially, all slots are empty. Thus full slots are 0*

*Empty = n // All slots are empty initially*

Producer Solution:

*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 beacuse the task of producer has been completed and consumer can access the buffer.

Consumer solution:

*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.

Algorithm:

Algorithm for Producer-

do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)

Algorithm for Consumer-

do{

wait(full);

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

Implementation:

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()

{

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.Producer\n2.Consumer\n3.Exit");

while(1)

{

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;

}

}

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",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

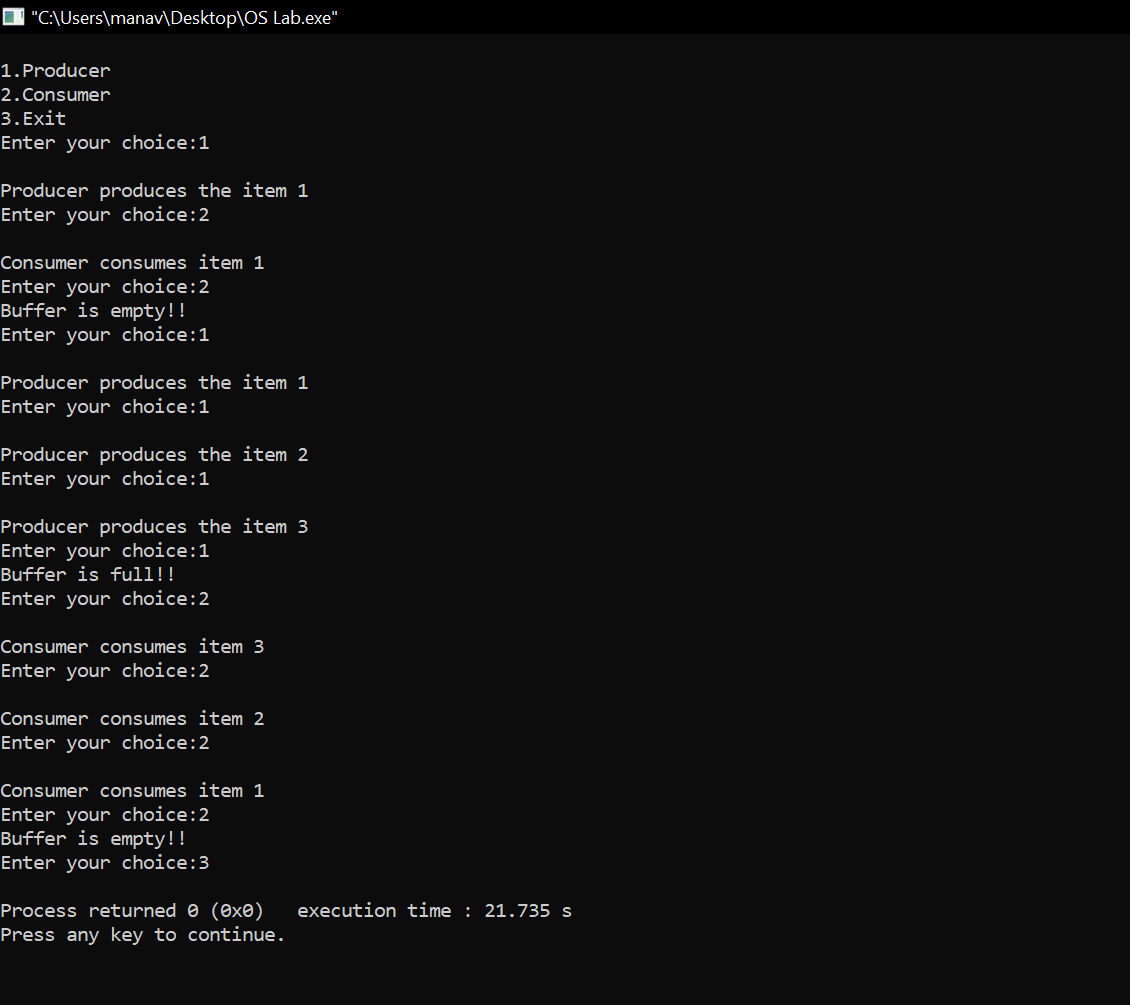
printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}

Output:

****

Learning From The Experiment:

Semaphores has many advantages which are that they do not allow more than one process to enter the critical section. In this way, mutual exclusion is achieved and thus they are extremely efficient than other techniques for synchronization. Due to busy waiting in semaphore, there is no wastage of process time and resources. This is because the processes are only allowed to enter the critical section after satisfying a certain condition.

Having so many advantages of it, there are always some disadvantages which is priority inversion in semaphores. The operating system has to keep track of all calls to wait and to signal the semaphore.

***THANK YOU!***