Implementing Solution To Producer Consumer Problem Using Semaphore

Kashish Sharma, Harpreet Kour, Shivam Bhardwaj, Yatharth Choudhary

2021a1r058@mietjammu.in
2021a1r042@mietjammu.in
2021a1r060@mietjammu.in
2021a1r040@mietjammu.in

MODEL INSTITUTE OF ENGINEERING AND TECHNOLOGY, CSE DEPARTMENT
KOT BHALWAL, JAMMU ,JAMMU & KASHMIR

Abstract-

This paper represents the working of the two processes namely producer and consumer Working without and without Semaphores .The main contribution of our work is the facility to run the simulation in a multi-processors environment. This has been deemed appropriate since most of the current applications are distributed in nature and run in multi-processing environment.The report describes the usage of Semaphores in solving the critical section problems that occur during the producer consumer functioning.

Keywords-

Bounded Buffer problem,producer,consumer Synchronization ,Semaphores,multiprocessing .

Introduction-

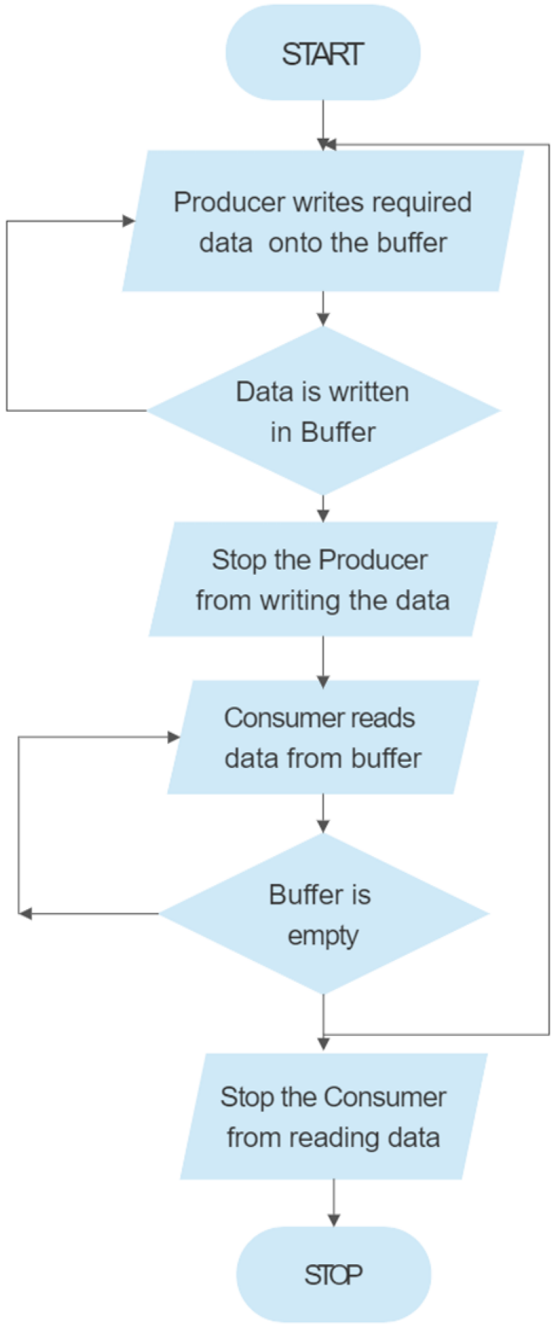
The Producer –Consumer problem is a family of problems described by Edsger W. Dijkstra since 1965. The Producer-Consumer problem is a classic problem this is used for multi-process synchronization,i.e.synchronization between more than one process. Process Synchronization can be defined as the coordination between two process that have access to common materials such as a common section of code, resources or data etc.In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. a producer and a consumer coupled via a buffer with unbounded capacity .

Producer-consumer Problem

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 Producer produces an 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 The producer is a cyclic process and each time it goes through its cycle it produces a certain portion of information, that has to be processed by the consumer. The consumer is also a cyclic process and each time it goes through its cycle, it can process the
next portion of information, as has been produced by the producer .We assume the two processes to be connected for a purpose via a buffer with unbounded capacity.The consumer waits when the buffer is empty; Producer waits when the buffer is full.The below flowchart describes the functioning of the whole process.

Methodology

Flowchart-



Working–

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items .The working of Producer is explained in the following code--

.

```
int count=0;
void producer(void)
{
int itemP;
while(1)
{
Producer_item(itemP);
while(count==n);//buffer is full
buffer[in]=itemP;
in=(in+1)mod n;
count=count+1;
}
}
```

In this code, Rp is a register which keeps the value of m[count].

-Rp is incremented (As element has been added to buffer)

-Then, the Incremented value of Rp is stored back to m[count]

0	A
1	B
2	C
3	D
4	E
5	
6	
7	

As we can see from the above figure, Buffer has total of 8 spaces out of which the first 5 are filled, in = 5(pointing next empty position) and out = 0(pointing first filled position).

The producer here ,who wanted to produce an item F according to code it will enter into the producer() function, while(1) will always be true, itemP = F will be tried to insert into the buffer, before that while(count == n); will evaluate to be False.

Buffer[in] = itemP → Buffer[5] = F. (F is inserted now)

in = (in + 1) mod n → (5 + 1)mod 8→ 6, therefore in = 6; (next empty buffer)

After insertion of F, Buffer looks like this-

0	A
1	B
2	C
3	D
4	E
5	F
6	
7	

Since count = count + 1; is divided into three parts:

Load Rp, m[count] → will copy count value which is 5 to register Rp.

Increment Rp → will increment Rp to 6.

Suppose just after Increment and before the execution of third line (store m[count], Rp), Context Switching occurs and code jumps to consumer code.

CONSUMER -

The task of the Consumer is to consume the item from the memory buffer.

Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer. Accessing memory buffer should not be allowed to producer and consumer at the same time.

The Consumer code is given below-

```
Void consumer(void)
{
int itemc;

 while(1)
 {
 while(count==0);
 itemc=Buffer(out);
 out=(out+1)mod n;

 count=count+1;
 Process_item(itemc);
 }
 }
```

In the above code,

Rc is a register which keeps the value of m[count]

- Rc is decremented (As element has been removed out of buffer)
- The decremented value of Rc is stored back to m[count].

The Consumer who wanted to consume the first element , according to code it will enter into the consumer() function, while(1) will always be true, while(count == 0); will evaluate to be False(since the count is still 5, which is not equal to 0)

itemC = Buffer[out]→ itemC = A (since out is 0)

out = (out + 1) mod n → (0 + 1)mod 8→ 1, therefore out = 1(first filled position)

So, A is removed now. After removal of A, Buffer look like this

0	
1	B
2	C
3	D
4	E
5	F
6	
7	

Since count = count - 1; is divided into three parts:

Load Rc, m[count] → will copy count value which is 5 to register Rp.

Decrement Rc → will decrement Rc to 4.

store m[count], Rc → count = 4.

Now the current value of count is 4

Suppose after this, Context Switch occurs back to the leftover part of producer code.

Since context switch at producer code was occurred after Increment and before the execution of the third line (store m[count], Rp).So we resume from here since Rp holds 6 as incremented value

Hence store m[count], Rp → count = 6

Now the current value of count is 6, which is wrong as Buffer has only 5 elements, Since context switch at producer code was occurred after Increment and before the execution of the third line (store m[count], Rp)

This condition is known as Race Condition and Problem is Producer-Consumer Problem**.**

Problems In Producer-Consumer Working

When the producer wants to put a new item in the buffer,and it is already full.So here the solution for the producer is to go to sleep and to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up. This

approach sounds simple enough, but it leads to race conditions.So these problems need to be solved.

Solution-

The producer should produce data only when the buffer is not full. If the buffer is full, then the producer should not be allowed to put any data into the buffer.

The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer should not be allowed to take any data from the buffer.

The producer and consumer should not access the buffer at the same time.

The problems in Producer –Consumer working can

be solved with the help of Semaphores.

Semaphores-

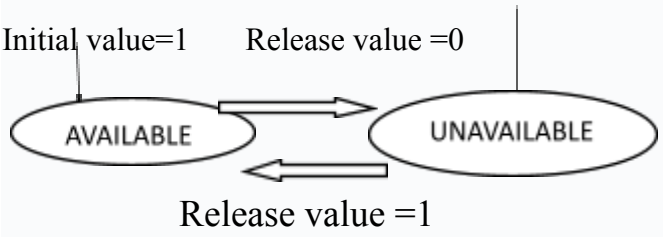
Semaphores are integer variables that are used to solve the critical section problem Semaphore cannot be implemented in the user mode because race condition may always arise when two or more processes try to access the variable simultaneously. It always needs support from the operating system to be implemented.

According to the demand of the situation, Semaphore can be divided into two categories.

1. Counting Semaphore
2. Binary Semaphore or Mutex

Binary Semaphores

In these type of Semaphores the integer value of the semaphore can only be either 0 or 1. If the value of the Semaphore is 1, it means that the process can proceed to the critical section (the common section that the processes need to access). However, if the value of binary semaphore is **0**, then the process cannot continue to the critical section **of** the code. When a process is using the critical section of the code, we change the Semaphore value to 0, and when a process is not using it, or we can allow a process to access the critical section, we change the value of semaphore to 1. Binary semaphore is also called mutex lock.



Counting Semaphores

Counting semaphores are signaling integers that can take on any integer value. Using these Semaphores we can coordinate access to resources and here the Semaphore count is the number of resources available. If the value of the Semaphore is anywhere above 0, processes can access the critical section, or the shared resources. The number of processes that can access the resources / code is the value of the semaphore. However, if the value is 0, it means that there aren't any resources that are available or the critical section is already being accessed by a number of processes and cannot be accessed by more processes. Counting semaphores are generally used when the number of instances of a resource are more than **1**, and multiple processes can access the resource**.**

The Semaphores used in solution to Producer-Consumer problem are-

Full()-

The full variable is used to track the space filled in the buffer by the Producer process. It is initialized to 0 initially as initially no space is filled by the Producer process.

Empty()-

The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialized to the buffer size as initially, the whole buffer is empty.

Mutex()-

Mutex is used to achieve mutual exclusion. Mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.Mutex is a binary semaphore variable that has a value of 0 or 1.

We will use the Signal() and wait() operation in the above-mentioned semaphores to arrive at a solution to the Producer-Consumer problem.

Signal() - The signal function increases the semaphore value by 1.

Wait() - The wait operation decreases the semaphore value by 1.

Mutex is used to solve the producer-consumer problem as mutex helps in mutual exclusion. It prevents more than one process to enter the critical section. As mutexes have binary values i.e 0 and 1. So whenever any process tries to enter the critical section code it first checks for the mutex value by using the wait operation.

wait(mutex) decreases the value of mutex by 1. So, Suppose a process P1 tries to enter the critical section when mutex value is 1. P1 executes wait(mutex) and decreases the value of mutex. Now, the value of mutex becomes 0 when P1 enters the critical section of the code.

Now, suppose Process P2 tries to enter the critical section then it will again try to decrease the value of mutex .But the mutex value is already 0. So, wait(mutex) will not execute, and P2 will now keep waiting for P1 to come out of the critical section. Now, P2 comes out of the critical section by executing signal(mutex).

signal(mutex) increases the value of mutex by 1. mutex value again becomes 1. Now, the process P2 which was in a busy-waiting state will be able to enter the critical section by executing wait(mutex).

So, mutex helps in the mutual exclusion of the processes.

The Producer Code using semaphore-

```
void Producer(){
   while(true){
      // producer produces an item/data
      wait(Empty);
      wait(mutex);
      add();
      signal(mutex);
      signal(Full);
   }
}
```

Understanding the above Producer process code :

wait(Empty) - Before producing items, the producer process checks for the empty space in the buffer. If the buffer is full producer process waits for the consumer process to consume items from the buffer. so, the producer process executes wait(Empty) before producing any item.

wait(mutex) - Only one process can access the buffer at a time. So, once the producer process enters into the critical section of the code it decreases the value of mutex by executing wait(mutex) so that no other process can access the buffer at the same time.

add() - This method adds the item to the buffer produced by the Producer process. once the Producer process reaches add function in the code, it is guaranteed that no other process will be able to access the shared buffer concurrently which helps in data consistency.

signal(mutex) - Now, once the Producer process added the item into the buffer it increases the mutex value by 1 so that other processes which were in a busy-waiting state can access the critical section.

signal(Full) - When the producer process adds an item into the buffer spaces is filled by one item so it increases the Full semaphore so that it indicates the filled spaces in the buffer correctly.

The Consumer code using Semaphore is -

```
void Consumer() {
   while(true){
   // consumer consumes an item
      wait(Full);
      wait(mutex);
      consume();
      signal(mutex);
      signal(Empty);
   }}
```

In the above Consumer process code :

wait(Full) - Before the consumer process starts consuming any item from the buffer it checks if the buffer is empty or has some item in it. So, the consumer process creates one more empty space in the buffer and this is indicated by the full variable. The value of the full variable decreases by one when the wait(Full) is executed. If the Full variable is already zero i.e the buffer is empty then the

consumer process cannot consume any item from the buffer and it goes in the busy-waiting state.

wait(mutex) - It does the same as explained in the producer process. It decreases the mutex by 1 and restricts another process to enter the critical section until the consumer process increases the value of mutex by 1.

consume() - This function consumes an item from the buffer. when code reaches the consuming () function it will not allow any other process to access the critical section which maintains the data consistency.

signal(mutex) - After consuming the item it increases the mutex value by 1 so that other processes which are in a busy-waiting state can access the critical section now.

signal(Empty) - when a consumer process consumes an item it increases the value of the Empty variable indicating that the empty space in the buffer is increased by 1.

In the above section in both the Producer process code and consumer process code, we have the wait and signal operation on mutex which helps in mutual exclusion and solves the problem of the Producer consumer process.

CONCLUSION

This simulator provides a facility for the user to experiment with three different contexts such as simple producer-consumer problem, producer-consumer problem in a single processor system and producer-consumer problem in multi-processing system. This has been deemed appropriate since most of the current applications are distributed in nature and run in multi-processing environment. In addition, users can carry out comparative evaluation among different contexts in a repeatable and controllable environment as to gain insight

on the best model to use in certain situations. All developed modules of the simulator guarantee synchronization of processes and satisfy necessary requirements to provide solution to the critical section problem .

This simulator has been designed to run self-driven simulations which are intrinsically limited in accuracy. It is because the input data on which the simulation runs is generated artificially to model the target system. As such, we plan to incorporate the trace-driven simulation in the existing simulator for more accurate results. Trace-driven uses as input, a trace of actual events collected and recorded on a real system. Capability of running a trace-driven simulation can be incorporated in the existing system by developing a software module that can translate the trace of events recorded on a real system, into the format of input data file. The input data file so generated then can be used to run the simulation.

References-

https://www.scaler.com

https://www.geeksforgeeks.org

https://afteracademy.com

https://www.javatpoint.com

Github Repository links-

https://github.com/Kashish9086/2021a1r058__COM-312

https://github.com/HKour2003/2021a1r042_COM-312