A Mini Project Report

on

Producer – Consumer Problem In Operating System

In Subject: **Operating System**

by

Yash Akotkar (22010814)
Rajdeep Chaurasia (22010948)
Netal Daga (22010244)
Samarth Ghule (22010468)

Department of Artificial Intelligence and Data Science

VIIT

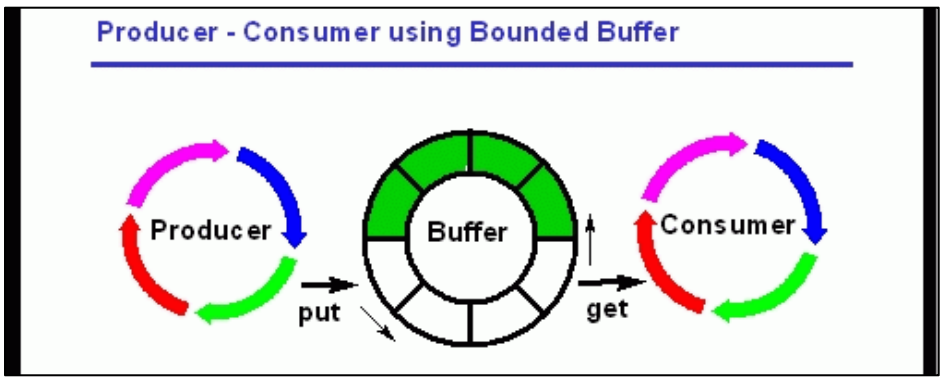
2021-2022

Contents

1.1 Introduction

- In this PBL we have tried to build a project using concepts producer & consumer in operating system.
- We have created a 'Producer-consumer problem's solution' program which uses arrays, pointers, semaphores (signal & wait function) for outputs.
- The complete project is performed using C language on online gbd compiler.
- This is very primitive idea which will help us to understand the theoretical concepts and implement them practically.
- Producer-consumer problem is one classical example of bounder buffer synchronization problems. The synchronization is needed in order to ensure that the producer stops producing when the buffer is full and the consumer stops removing items from the buffer if it is empty.
- The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e., synchronization between more than one processes. Also known as ***bounded-buffer problem.*** 3 main terms:

  - Producer thread: The producer's job is to generate data (a number) , put it into the buffer (array) , and repeat the same.
  - Consumer thread: At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.
  - Mutual Exclusion: The producer and consumer should not access the buffer at the same time.
- Both Producer and Consumer share a common memory buffer. This buffer is a space of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.



PRODUCER-CONSUMER MODELS

This section presents the different models of the producer-consumer problem that can be simulated by the user.

1.) Simple Producer-Consumer Problem:

- This problem describes one producer and one consumer sharing the same finite buffer.
- Producer produces items and place items in the buffer while consumer consumes items from the buffer.

2.) Producer-Consumer Problem in Single Processor Environment:

- This model is the advanced form of the first model. In the first model, producer produces and places items in the buffer at random basis. Consumer simply consumes items from the buffer without acknowledging whether it is meant for them or not.
- But in this model, producer is producing items for the specific consumer and only that consumer is given access to that item. When that consumer is free, it will consume the items from the buffer. This module of the simulator works for single processor environment.

3.) Producer-Consumer Problem in Multi - Processor Environment:

- Parallel processing technique has been employed in this module to enable N number of producers and consumers executing at the same time. This is to ensure that resources are fully optimized.
- In this model, producers can produce items/data for specific consumers and only that consumer can consume those items/data.

1.2 Requirements

- Basic knowledge of programming & producer-consumer problem.
- In our case, sound in technical language like C/C++.
- Understanding of arrays, function creation, pointers, semaphores, mutex, and thread function.
- Software like VS Code or good online compiler.

1.3 Problem Statement

What is the Producer-Consumer Problems?

In producer-consumer problem, there are two types of process involved: producer and consumer who are sharing a fixed size buffer. The role of producer is to put the item one a time in the buffer and the role of consumer is to retrieve the item from the buffer. The problem is to make sure that the producer will not try to add the item once the buffer is full and that the consumer will not try to remove the item from an empty buffer.
The normal behaviour of this problem is:
(i) Random arrival of petitions to put item in the buffer;
(ii) One consumer that immediately after getting an information from the buffer intends to get another one;
(iii) Buffer of defined and finite size;
(iv) Consumer waits when the buffer is empty;
(v) Producer waits when the buffer is full;
(vi) Second and subsequent productions wait when the customer is writing or held up.

1.4 Proposed Work

```
#include<stdio.h>
#include<semaphore.h>
#include<sys/types.h>
#include<pthread.h>
#include<unistd.h>
#include<stdlib.h>

#define BUFFER_SIZE 10

// Prototypes..
void *producer();
void *consumer();
void insert_item(int);
int remove_item();

// Declaring mutex & semaphore
pthread_mutex_t mutex;
sem_t empty, full;

// Buffer is shared by both producer & consumer
```

```c
int buffer[BUFFER_SIZE];

// Counter is the global & shared variable
int counter;
pthread_t thread;

void initialize()
{
    printf("\n");
    pthread_mutex_init(&mutex,NULL);
    sem_init(&full,0,0);
    sem_init(&empty,0,BUFFER_SIZE);
}

void *producer()
{
    int item,wait_time;
    wait_time=rand()%5;
    sleep(wait_time)%5;
    item=rand()%10;
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    // Produce / create item
    printf("Producer produced: %d \n",item);

    // Inserting item into buffer
    insert_item(item);
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
}

void *consumer()
{
    int item,wait_time;
    wait_time=rand()%5;
    sleep(wait_time);
    sem_wait(&full);
    pthread_mutex_lock(&mutex);

    // Removing item from buffer for further processing
    item=remove_item();
    printf("Consumer consumed: %d\n",item);
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
```

```
}

// Insert item
void insert_item(int item)
{
    buffer[counter++]=item;
}

// Remove item
int remove_item()
{
    return buffer[--counter];
}

int main()
{
    int n1,n2;
    int i;
    printf("Enter number of Producers: ");
    scanf("%d",&n1);
    printf("\nEnter number of Consumers: ");
    scanf("%d",&n2);
    initialize();

    // create threads for all producers & consumers
    for(i=0;i<n1;i++)
    pthread_create(&thread,NULL,producer,NULL);
    for(i=0;i<n2;i++)
    pthread_create(&thread,NULL,consumer,NULL);
    sleep(5);
    exit(0);
}
```

2.1 Approach

Solution For Producer Consumer Problem- To solve the Producer-Consumer problem 3 semaphores variable are used :

- Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.

- 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.
- Signal(): The signal function increases the semaphore value by 1.
- Wait(): The wait operation decreases the semaphore value by 1.

Model used:

Simple Producer-Consumer Problem:

- This problem describes one producer and one consumer sharing the same finite buffer.
- Producer produces items and place items in the buffer while consumer consumes items from the buffer.
- Producer produces and places items in the buffer at random basis. Consumer simply consumes items from the buffer without acknowledging whether it is meant for them or not.

Algorithm's:

Producer:

- It first checks if buffer is empty.
- If buffer is full, producer will either go to sleep & keep waiting for buffer to be empty.
- Waiting forever is not the solution.
- Next time consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.

Consumer:

- It checks if buffer is empty.
- If it finds buffer empty, consumer will go to sleep & keeps waiting for it to be full.

- The next time the producer puts the data into the buffer, it wakes up the sleeping consumer.

Pseudo code:

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

Terms/functions used in producer 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.

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

Terms/functions used in consumer 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.

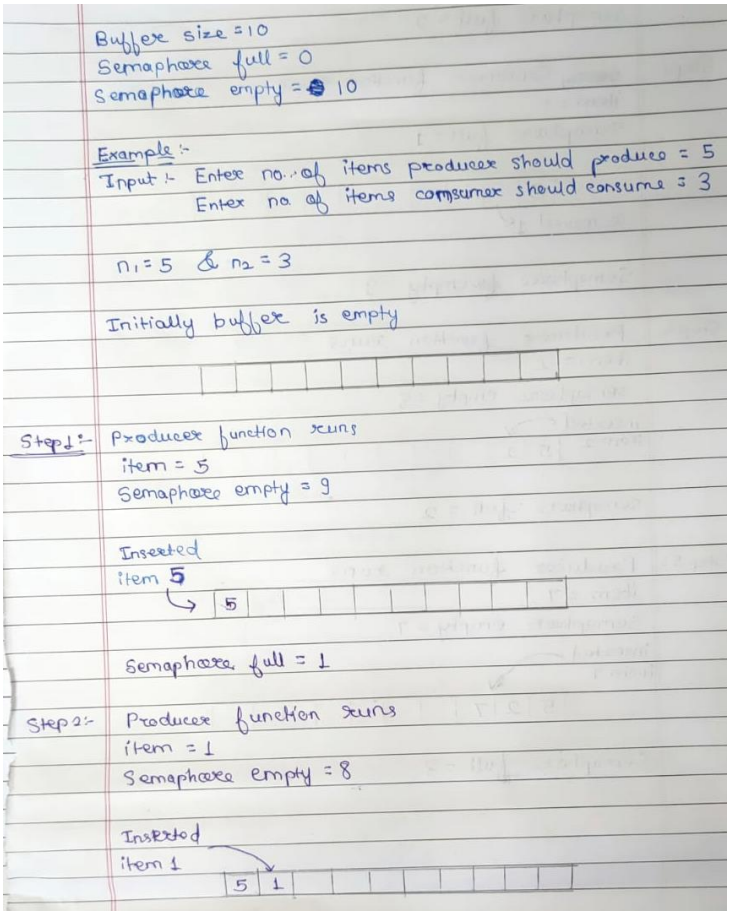
2.2 Platform & Technology

Platform used is Online gbd compiler and technology used is C language.

2.3 Outcomes & Working

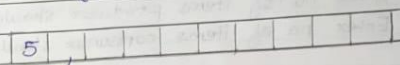
In this project we have taken input of number of producers & consumers from the user.

```
Enter number of Producers: 5

Enter number of Consumers: 3

Producer produced: 5
Producer produced: 1
Consumer consumed: 1
Producer produced: 2
Producer produced: 7
Producer produced: 0
Consumer consumed: 0
Consumer consumed: 7


...Program finished with exit code 0
Press ENTER to exit console.
```



Semaphore full = 2

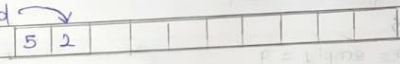
Step3:- ~~Semaphore~~ Consumer function runs
item = 1
Semaphore full = 1



removed 1

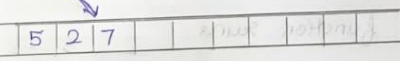
Semaphore ~~full~~ empty = 9

Step4:- Producer function runs
item = 2
semaphore empty = 8
inserted
item 2



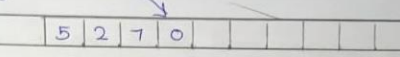
semaphore full = 2

Step5:- Producer function runs
item = 7
Semaphore empty = 7
inserted
item 7



Semaphore full = 3

Step6:- Producer function runs
item = 0
Semaphore empty = 6
inserted
item 0



Semaphore full = 4

Step7:- Consumer function runs
item = 0
Semaphore full = 3
removed
item 0

Semaphore empty = 7

Step8:- Consumer function runs
item = 7
Semaphore full = 2
removed
item 7

Semaphore empty = 8

2.4 Future Work

This system can be improved by adding the function to display the buffer size, number of empty slots, number of full slots.

2.5 Reviews on Research Paper

1.) Processing Models: It have clearly explained the 3 different models used in solving this problem.

2.) Visualization of processes: Processes are easily understandable while referring to those helpful illustration through graphs.

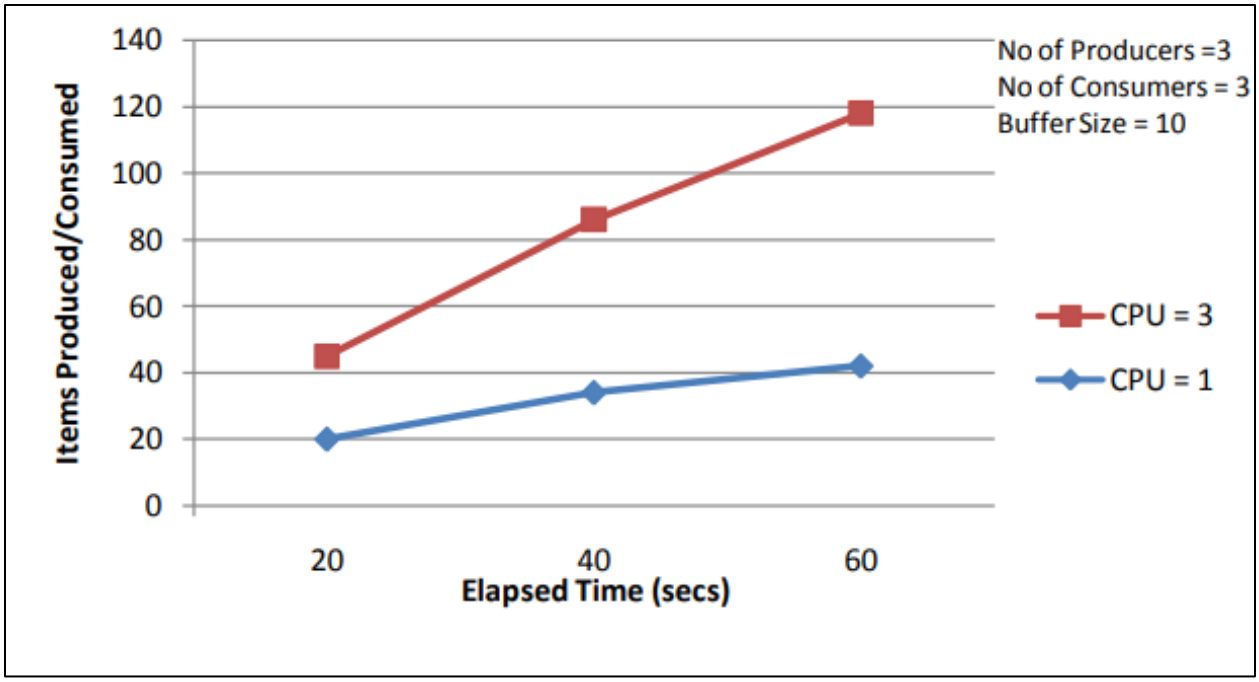
3.) Execution of steps: Easily explained how to build that particular simulation.

4.) Problem explanation: Author has clearly explained what is exact producer-consumer problem & it's different types.

5.) Helpful & on – point: Sources were legitimate. Author was sticked to topic in overall paper & provided credible solution.

Conclusion:

- Full, Empty and mutex semaphore help to solve Producer-consumer problem.
- Full semaphore checks for the number of filled space in the buffer by the producer process.
- Empty semaphore checks for the number of empty spaces in the buffer.
- Mutex checks for the mutual exclusion.
- In this single process model, I have used only one CPU and the rest of the parameters such as Buffer Size, Number of Producers and Consumers must be given by the user for simulation.
- Default number of buffer is 10.



Simulation results for relationship between items produced/consumed and elapsed time (20 seconds) for single process environments.

References:

Operating System Notes.

Research Paper:
https://www.researchgate.net/publication/49607455_Implementation_and_Experimentation_of_Producer-Consumer_Synchronization_Problem

THANK YOU!!