In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem
To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution
The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
An inadequate solution could result in a deadlock where both processes are waiting to be awakened.
Recommended Reading- Multithreading in JAVA, Synchronized in JAVA, Inter-thread Communication

Implementation of Producer Consumer Class

- A **LinkedList list** – to store list of jobs in queue.
- **A Variable Capacity** – to check for if the list is full or not
- A mechanism to control the insertion and extraction from this list so that we do not insert into list if it is full or remove from it if it is empty.

Note: It is recommended to test the below program on a offline IDE as infinite loops and sleep method may lead to it time out on any online IDE

Java

```java
// Java program to implement solution of producer
// consumer problem.

import java.util.LinkedList;

public class Threadexample {
    public static void main(String[] args)
        throws InterruptedException
    {
        // Object of a class that has both produce()
        // and consume() methods
        final PC pc = new PC();
```

```
        // Create producer thread
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                try {
                    pc.produce();
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Create consumer thread
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                try {
                    pc.consume();
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // t1 finishes before t2
        t1.join();
        t2.join();
    }

    // This class has a list, producer (adds items to list
    // and consumer (removes items).
    public static class PC {

        // Create a list shared by producer and consumer
        // Size of list is 2.
        LinkedList<Integer> list = new LinkedList<>();
        int capacity = 2;

        // Function called by producer thread
        public void produce() throws InterruptedException
        {
            int value = 0;
            while (true) {
                synchronized (this)
                {
                    // producer thread waits while list
                    // is full
```

```
                    while (list.size() == capacity)
                        wait();

                    System.out.println("Producer produced-"
                                       + value);

                    // to insert the jobs in the list
                    list.add(value++);

                    // notifies the consumer thread that
                    // now it can start consuming
                    notify();

                    // makes the working of program easier
                    // to  understand
                    Thread.sleep(1000);
                }
            }
        }

        // Function called by consumer thread
        public void consume() throws InterruptedException
        {
            while (true) {
                synchronized (this)
                {
                    // consumer thread waits while list
                    // is empty
                    while (list.size() == 0)
                        wait();

                    // to retrieve the first job in the list
                    int val = list.removeFirst();

                    System.out.println("Consumer consumed-"
                                       + val);

                    // Wake up producer thread
                    notify();

                    // and sleep
                    Thread.sleep(1000);
                }
            }
        }
    }
}
```

Output:

```
Producer produced-0
Producer produced-1
Consumer consumed-0
```

```
Consumer consumed-1
Producer produced-2
```

Important Points

- In **PC class** (A class that has both produce and consume methods), a linked list of jobs and a capacity of the list is added to check that producer does not produce if the list is full.
- In **Producer class**, the value is initialized as 0.
  - Also, we have an infinite outer loop to insert values in the list. Inside this loop, we have a synchronized block so that only a producer or a consumer thread runs at a time.
  - An inner loop is there before adding the jobs to list that checks if the job list is full, the producer thread gives up the intrinsic lock on PC and goes on the waiting state.
  - If the list is empty, the control passes to below the loop and it adds a value in the list.
- In the **Consumer class**, we again have an infinite loop to extract a value from the list.
  - Inside, we also have an inner loop which checks if the list is empty.
  - If it is empty then we make the consumer thread give up the lock on PC and passes the control to producer thread for producing more jobs.
  - If the list is not empty, we go round the loop and removes an item from the list.
- In both the methods, we use notify at the end of all statements. The reason is simple, once you have something in list, you can have the consumer thread consume it, or if you have consumed something, you can have the producer produce something.
- sleep() at the end of both methods just make the output of program run in step wise manner and not display everything all at once so that you can see what actually is happening in the program.