02

THEORETICAL PART

CONCEPT/IDEA

The Producer/Consumer pattern is used when there are tasks producing data and tasks consuming that data concurrently. It helps prevent data race conditions and ensures efficient data sharing between producers and consumers. The producers add data to a shared data structure (often a queue) while the consumers remove data from it. This pattern decouples the production and consumption of data, allowing producers and consumers to operate independently and at different rates.

PROBLEM

The main problem the Producer/Consumer pattern addresses is synchronizing the activities of producers and consumers to avoid issues like data loss, inconsistencies, or conflicts. For example, producers should not add data to a full queue and consumers should not remove data from an empty queue. This requires coordination and synchronization between the two types of processes

USE IN OPERATING SYSTEMS

The Producer/Consumer pattern is relevant in operating systems when dealing with concurrency and multithreading. It helps in reducing foreground thread latency by offloading tasks from the foreground thread to background worker threads. It also enables load balancing by distributing tasks among a set of worker threads. Additionally, the pattern facilitates backpressure management by controlling the flow of tasks when the system is overloaded.

04 SEMAPHORES AS A SOLUTION

Semaphores are a thread **synchronization** construct used to control access to a shared resource. They can effectively solve the **Producer/Consumer problem** by regulating the number of available permits for accessing the shared resource.They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes. The process of using Semaphores provides two operations: **wait (P) and signal (V)**. The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation. When the shared buffer is full, the producer threads will wait until consumers consume data and release permits. Similarly, when the buffer is empty, the consumer threads will wait until producers produce data and release permits. This ensures that producers and consumers work harmoniously without conflicts.

05

PRACTICAL PART

06 THE CLASSES NEEDED IN THE PROGRAM

Queue.java

```java
import java.util.concurrent.Semaphore;

class Queue {
    int item;
    static Semaphore semCon = new Semaphore(0);
    static Semaphore semProd = new Semaphore(1);

    void get() {
        try {
            semCon.acquire();
        } catch (InterruptedException e) {
            System.err.println("InterruptedException caught: " + e.getMessage());
        }
        System.out.println("Consumer consumed item: " + item);
        semProd.release();
    }

    void put(int item) {
        try {
            semProd.acquire();
        } catch (InterruptedException e) {
            System.err.println("InterruptedException caught: " + e.getMessage());
        }
        this.item = item;
        System.out.println("Producer produced item: " + item);
        semCon.release();
    }
}
```

Producer.java

```java
class Producer implements Runnable {
    Queue q;

    Producer(Queue q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        for (int i = 0; i < 30; i++) {
            q.put(i);
            try {
                Thread.sleep(500); // Delay for 0.5 seconds
            } catch (InterruptedException e) {
                System.err.println("InterruptedException caught: " + e.getMessage());
            }
        }
    }
}
```

07 THE CLASSES NEEDED IN THE PROGRAM

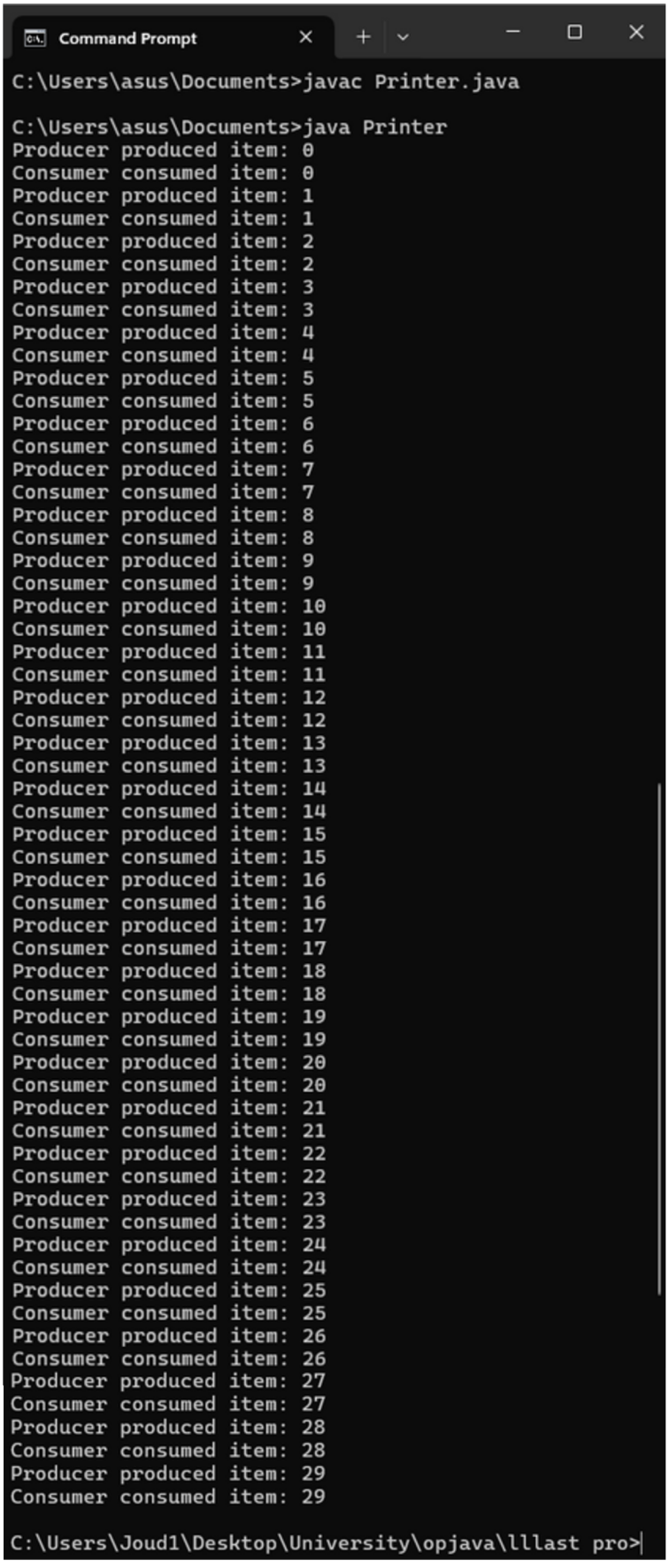
Consumer.java

```java
class Consumer implements Runnable {
    Queue q;

    Consumer(Queue q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        for (int i = 0; i < 30; i++) {
            q.get();
            try {
                Thread.sleep(1000); // Delay for 1 second
            } catch (InterruptedException e) {
                System.err.println("InterruptedException caught: " + e.getMessage());
            }
        }
    }
}
```

Printer.java

```java
class Printer {
    public static void main(String[] args) {
        Queue q = new Queue();
        new Consumer(q);
        new Producer(q);
    }
}
```

08 THE CLASSES NEEDED IN THE PROGRAM



09 THE OPERATIONS ON THE SHARED OBJECT (BUFFER)

In the Producer-Consumer pattern, the shared object, often referred to as the buffer, acts as a central repository for items that are produced and consumed.

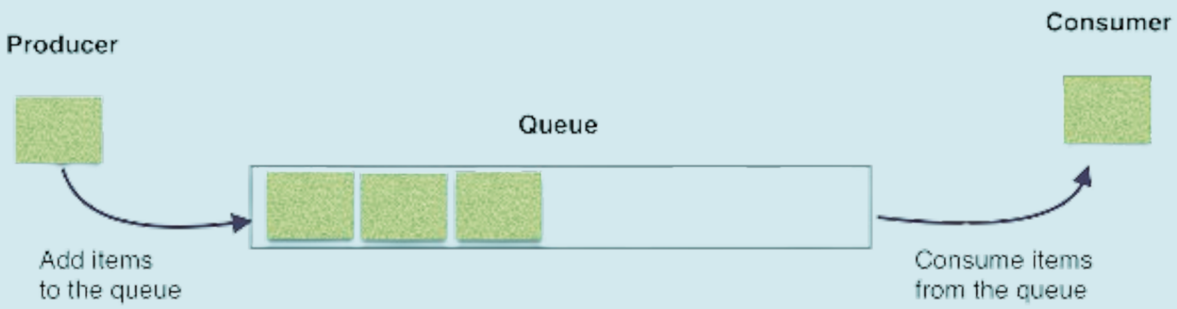
Operations:

Get () method:

- It retrieves an item from the buffer and returns it to the caller.
- Before retrieving an item, the consumer thread acquires a permit from the semCon semaphore. This ensures that only one consumer can remove an item from the buffer at a time.
- After retrieving the item, the consumer releases a permit from the semProd semaphore. This signals to the producer thread that a space is available in the buffer.

Put (item) method:

- It takes an integer item as input and adds it to the buffer.
- Before adding the item, the producer thread acquires a permit from the semProd semaphore. This ensures that only one producer can add an item to the buffer at a time.
- After adding the item, the producer releases a permit from the semCon semaphore. This signals to the consumer thread that an item is available in the buffer.



10 DEALING WITH RACE CONDITION

A **race condition** is a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes. One or more possible outcomes may be undesirable output, resulting in a bug. The code deals with race conditions by using ***semaphores*** for synchronization, specifically in **semProd** and **semCon**. Semaphores are a type of synchronization primitive that allows controling the access to a shared resource. In this case, the shared resource is the buffer (Q object).

11 USING A SEMAPHORE MECHANISM

semaphore mechanism is used to synchronize access to the shared buffer between the producer and consumer threads. A semaphore S is an integer variable that can be accessed only through two standard operations : wait() and signal(). Here we have a **Binary Semaphore.** It can have only two values – 0 and 1. Two semaphores are created: **semCon** and **semProd**. semCon is initialized with 0 permits, and semProd is initialized with 1 permit.
This synchronization mechanism prevents the producer thread from producing items too quickly and overflowing the buffer, and prevents the consumer thread from consuming items too quickly and underflowing the buffer.

The semaphore mechanism ensures that:

- The producer thread can **only produce** an item when there is a available permit in semProd.
- The consumer thread can **only consume** an item when there is a available permit in semCon.
- The producer thread **will block if the consumer thread has not consumed the previous item, and vice versa.**

12 WHERE THE RACE CONDITION IS HAPPENING IN THE PROGRAM AND WHY?

1. Race Condition:
 - The race condition occurs in the `put` and `get` methods of the `Queue` class when multiple threads try to access and modify the `item` variable simultaneously.
 - In the `put` method, multiple `Producer` threads may try to modify the `item` variable concurrently, potentially leading to data corruption or inconsistency.
 - Similarly, in the `get` method, multiple `Consumer` threads may try to read the `item` variable concurrently, leading to unpredictable behavior.

Queue.java

```java
import java.util.concurrent.Semaphore;

class Queue {
    int item;
    static Semaphore semCon = new Semaphore(0);
    static Semaphore semProd = new Semaphore(1);

    void get() {
        try {
            semCon.acquire();
        } catch (InterruptedException e) {
            System.err.println("InterruptedException caught: " + e.getMessage());
        }
        System.out.println("Consumer consumed item: " + item);
        semProd.release();
    }

    void put(int item) {
        try {
            semProd.acquire();
        } catch (InterruptedException e) {
            System.err.println("InterruptedException caught: " + e.getMessage());
        }
        this.item = item;
        System.out.println("Producer produced item: " + item);
        semCon.release();
    }
}
```

14 WHY AND HOW THE DEADLOCK HAPPENED AND HOW IT WAS AVOIDED?

3. Deadlock:
- A deadlock may occur if one thread holds a semaphore and waits for another semaphore that is held by another thread, resulting in a deadlock situation where neither thread can proceed.
- In this scenario, a potential deadlock can occur if a **`Consumer`** thread acquires **`semCon`** and waits for **`semProd`** to release, while a `Producer` thread acquires **`semProd`** and waits for **`semCon`** to release.
- To avoid deadlock, proper semaphore acquisition and release order should be maintained. In this case, **`semCon`** should be acquired before **`semProd`** in the `get` method, and vice versa in the `put` method, ensuring that the semaphores are always released in the reverse order of acquisition.

Queue.java

```java
import java.util.concurrent.Semaphore;

class Queue {
    int item;
    static Semaphore semCon = new Semaphore(0);
    static Semaphore semProd = new Semaphore(1);

    void get() {
        try {
            semCon.acquire();
        } catch (InterruptedException e) {
            System.err.println("InterruptedException caught: " + e.getMessage());
        }
        System.out.println("Consumer consumed item: " + item);
        semProd.release();
    }

    void put(int item) {
        try {
            semProd.acquire();
        } catch (InterruptedException e) {
            System.err.println("InterruptedException caught: " + e.getMessage());
        }
        this.item = item;
        System.out.println("Producer produced item: " + item);
        semCon.release();
    }
}
```

15 SUMMARY

The code tackles a race condition by employing semaphores to regulate access to the shared `item` variable within the `Queue` class. However, to avert potential deadlock, it's vital to guarantee that semaphores are acquired and released in the correct order. The implementation relies on `semCon` and `semProd` semaphores to govern access for consumers and producers respectively, ensuring only one thread accesses the shared resource at a time. Yet, without careful handling, deadlock—a scenario where threads indefinitely wait for resources held by others—could arise. This risk underscores the importance of maintaining a consistent order in acquiring and releasing semaphores across threads. Such meticulous management not only facilitates smooth concurrent execution but also fortifies against the threat of deadlock, thereby enhancing the system's reliability and robustness.