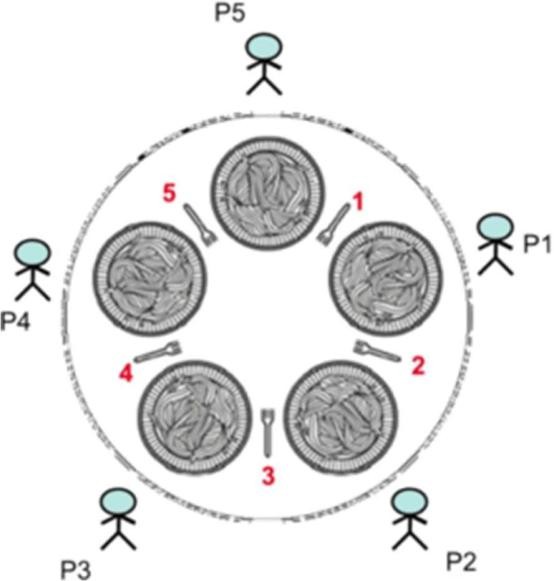
**Week 11. Synchronization, Interthread Communication and Deadlock**

## Dining Philosopher Problem

****

The above figure represents that there are five philosophers (labelled with P1, P2, P3, P4, and P5) sitting around a circular dining table. There are five plates of noodles to eat when philosophers feel hungry. To eat noodles, there are five forks/ chopsticks (labelled with 1 to 5) placed between each philosopher.

You are tasked with simulating the **Dining Philosophers Problem**. There are five philosophers sitting around a circular dining table. Each philosopher alternates between eating and thinking. There are five forks between them, and to eat, a philosopher must pick up both the forks next to them.

The following conditions must be satisfied:

* + A philosopher must pick up both the left and right forks to eat.
  + A philosopher can only pick up a fork if it is not being used by a neighbour.
  + No two adjacent philosophers can eat at the same time.
  + After finishing eating, the philosopher puts down both forks and starts thinking.

You need to implement the simulation using multithreading and synchronization to ensure that no deadlocks occur.

## Input:

There is no input. The task is to simulate the behaviour of philosophers eating and thinking.

## Output:

For each philosopher, print the following:

* + "Philosopher P# is thinking" when the philosopher is thinking.
  + "Philosopher P# is hungry" when the philosopher tries to pick up forks.
  + "Philosopher P# is eating" when the philosopher successfully picks up both forks and eats.
  + "Philosopher P# finished eating" when the philosopher puts down the forks.

## Function Signature:

public class DiningPhilosophers {

public static void main(String[] args) {

// Your code to start the simulation

}

}

class Philosopher implements Runnable {

// Implement the philosopher's behavior here

}

## Constraints:

* + The solution must avoid deadlock.
  + Each philosopher should eat and think alternately without violating the problem conditions.

**Sample Output:** Philosopher P1 is thinking Philosopher P1 is hungry Philosopher P1 is eating

Philosopher P1 finished eating Philosopher P2 is thinking Philosopher P2 is hungry Philosopher P2 is eating Philosopher P2 finished eating

...

# Class: DiningPhilosophers

public class DiningPhilosophers {

public static void main(String[] args) { final int NUM\_PHILOSOPHERS = 5;

Fork[] forks = new Fork[NUM\_PHILOSOPHERS];

Philosopher[] philosophers = new Philosopher[NUM\_PHILOSOPHERS];

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

forks[i] = new Fork();

}

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

philosophers[i] = new Philosopher(i, forks[i], forks[(i + 1) % NUM\_PHILOSOPHERS]);

new Thread(philosophers[i]).start();

}

}

}

class Fork {

private boolean isInUse = false;

public synchronized void pickUp() throws InterruptedException { while (isInUse) {

wait();

}

isInUse = true;

}

public synchronized void putDown() { isInUse = false;

notify();

}

}

class Philosopher implements Runnable { private final int id;

private final Fork leftFork; private final Fork rightFork;

public Philosopher(int id, Fork leftFork, Fork rightFork) { this.id = id;

this.leftFork = leftFork; this.rightFork = rightFork;

}

@Override

public void run() { try {

while (true) { think();

hungry(); leftFork.pickUp(); rightFork.pickUp(); eat(); leftFork.putDown(); rightFork.putDown();

}

} catch (InterruptedException e) { Thread.currentThread().interrupt();

}

}

private void think() {

System.out.println("Philosopher P" + id + " is thinking"); sleep();

}

private void hungry() {

System.out.println("Philosopher P" + id + " is hungry");

}

private void eat() {

System.out.println("Philosopher P" + id + " is eating"); sleep();

System.out.println("Philosopher P" + id + " finished eating");

}

private void sleep() { try {

Thread.sleep((long) (Math.random() \* 1000));

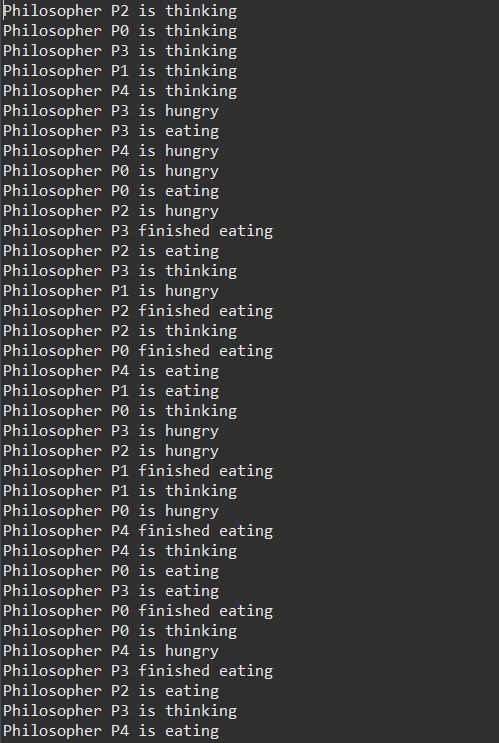
} catch (InterruptedException e) { Thread.currentThread().interrupt();

}

}

}

# OUTPUT

****

1. You are required to implement a **Producer-Consumer system**. The producer thread generates data and puts it into a shared buffer, while the consumer thread processes the data from the buffer. The following conditions must be met:
   * The consumer waits if the buffer is empty until the producer notifies that data is available.
   * The producer notifies the consumer whenever new data is added to the buffer.

## Class Definitions:

Implement two classes:

* + Producer: Produces data and calls notify() to wake up the consumer.
  + Consumer: Waits with wait() when the buffer is empty until the producer calls notify().

## Input Format:

The input consists of:

* **N**: An integer denoting the number of data items to be produced.
* **Buffer Size**: An integer denoting the maximum size of the shared buffer.

## Output Format:

The output should print the following:

* "Produced X" when the producer adds an item to the buffer, where X is the produced data item.
* "Consumed X" when the consumer processes an item from the buffer, where X is the consumed data item.

## Function Signature:

class ProducerConsumer {

public static void main(String[] args) {

Producer producer = new Producer(sharedBuffer, maxBufferSize); Consumer consumer = new Consumer(sharedBuffer);

// Start producer and consumer threads producer.start();

consumer.start();

// Wait for producer and consumer threads to finish producer.join();

consumer.join();

}

}

class Producer {

private final List<Integer> buffer; private final int maxSize;

public Producer(List<Integer> buffer, int maxSize) { this.buffer = buffer;

this.maxSize = maxSize;

}

public void run() {

## // Logic to produce data items

}

}

class Consumer {

private final List<Integer> buffer;

public Consumer(List<Integer> buffer) { this.buffer = buffer;

}

public void run() {

## // Logic to consume data items

}

}

**Input**: 5 3 **Output**: Produced 1

Produced 2

Consumed 1

Produced 3

Consumed 2

Produced 4

Consumed 3

Produced 5

Consumed 4

Consumed 5

## Explanation:

* In the input, the producer generates 5 data items and the buffer has a maximum size of 3.
* The producer adds data until the buffer is full, after which the consumer starts consuming items.
* The process continues until all items are produced and consumed.

Ensure proper synchronization between the producer and consumer threads to avoid race conditions.

# Class: ProducerConsumer

import java.util.ArrayList; import java.util.List;

class ProducerConsumer {

public static void main(String[] args) throws InterruptedException { int maxBufferSize = 3;

List<Integer> sharedBuffer = new ArrayList<>();

Producer producer = new Producer(sharedBuffer, maxBufferSize); Consumer consumer = new Consumer(sharedBuffer);

producer.start(); consumer.start();

producer.join(); consumer.join();

}

}

class Producer extends Thread { private final List<Integer> buffer; private final int maxSize;

public Producer(List<Integer> buffer, int maxSize) { this.buffer = buffer;

this.maxSize = maxSize;

}

public void run() {

for (int i = 1; i <= 5; i++) { try {

synchronized (buffer) {

while (buffer.size() == maxSize) { buffer.wait();

}

buffer.add(i); System.out.println("Produced " + i); buffer.notifyAll();

}

} catch (InterruptedException e) { Thread.currentThread().interrupt();

}

}

}

}

class Consumer extends Thread { private final List<Integer> buffer;

public Consumer(List<Integer> buffer) { this.buffer = buffer;

}

public void run() {

for (int i = 1; i <= 5; i++) { try {

synchronized (buffer) { while (buffer.isEmpty()) {

buffer.wait();

}

int consumedItem = buffer.remove(0); System.out.println("Consumed " + consumedItem); buffer.notifyAll();

}

} catch (InterruptedException e) { Thread.currentThread().interrupt();

}

}

}

}

**OUTPUT**

