Week 12. Semaphore

## Building H₂O Molecules

You are given two types of threads: **Oxygen** and **Hydrogen**. Your task is to write a

program that groups these threads to form water molecules (H₂O).

* + Each water molecule requires exactly two Hydrogen (**H**) and one Oxygen (**O**).
  + You need to synchronize the threads so that two Hydrogen and one Oxygen thread bond together to form a water molecule.

## You must ensure that:

* + If an Oxygen thread arrives at the barrier without two Hydrogen threads, it must wait.
  + If a Hydrogen thread arrives at the barrier without one Oxygen thread and another Hydrogen thread, it must wait.
  + The threads should bond in complete sets of two Hydrogen and one Oxygen, ensuring no incomplete groups form before the next group starts.

Your task is to implement the methods:

* + **releaseHydrogen()** – called by Hydrogen threads.
  + **releaseOxygen()** – called by Oxygen threads.

## Input Format

* + You are given a string **water** where each character represents either H (Hydrogen) or O (Oxygen).
    - There will be exactly **2 \* n** Hydrogen (**H**) and exactly **n** Oxygen (**O**).

## Constraints:

* + 3 \* n == water.length
  + 1 <= n <= 20
  + water[i] is either 'H' or 'O'.
  + There are exactly 2 \* n 'H' characters in the string.
  + There are exactly n 'O' characters in the string.

## Output Format

* + The output should be the sequence of water molecules formed, where each molecule consists of exactly two Hydrogen (**H**) and one Oxygen (**O**).

## Example 1

**Input**: water = "HOH"

**Output**: HHO

## Explanation:

HOH, HHO, and OHH are all valid outputs as they represent one water molecule with two hydrogen and one oxygen.

## Example 2

**Input**: water = "OOHHHH"

**Output**: HHOHHO

## Explanation:

The valid outputs include sequences like HHOHHO, HOHHHO, or OHHHHO, where exactly two Hydrogen and one Oxygen form each water molecule.

# Class: H2O

import java.util.concurrent.Semaphore; class H2O {

private final Semaphore hydrogenSemaphore = new Semaphore(2);

private final Semaphore oxygenSemaphore = new Semaphore(1); private int hydrogenCount = 0;

public void releaseHydrogen() throws InterruptedException { hydrogenSemaphore.acquire();

synchronized (this) { hydrogenCount++;

if (hydrogenCount == 2) { hydrogenCount = 0; oxygenSemaphore.release();

}

}

System.out.print("H");

if (hydrogenCount == 0) { oxygenSemaphore.acquire();

}

}

public void releaseOxygen() throws InterruptedException { oxygenSemaphore.acquire();

System.out.print("O"); hydrogenSemaphore.release(2);

}

public static void main(String[] args) { H2O h2o = new H2O();

String water = "OOHHHH";

Thread[] threads = new Thread[water.length()];

for (int i = 0; i < water.length(); i++) { char element = water.charAt(i); threads[i] = new Thread(() -> {

try {

if (element == 'H') { h2o.releaseHydrogen();

} else {

h2o.releaseOxygen();

}

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

}

});

threads[i].start();

}

for (Thread thread : threads) { try {

thread.join();

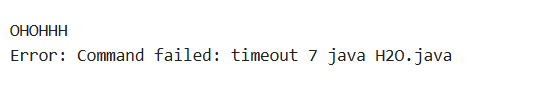
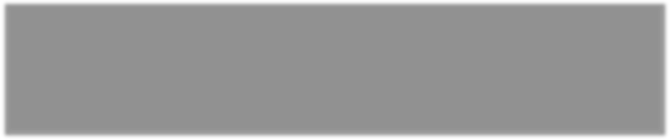
} catch (InterruptedException e) { Thread.currentThread().interrupt();

}

}

}

}



# OUTPUT

## Producer-Consumer Problem with Semaphore-Based Synchronization

You are tasked with solving the classic Producer-Consumer problem using semaphores. A shared buffer of fixed size is used by multiple producer threads to generate data and multiple consumer threads to consume the data. The buffer operates under the following constraints:

* + Producers must wait if the buffer is full.
  + Consumers must wait if the buffer is empty.
  + Only one thread (either a producer or a consumer) can access the buffer at a time.

Implement the synchronization using semaphores:

* + Use two semaphores, empty (to track the number of empty slots) and full (to track the number of filled slots).
  + Use a mutex (binary semaphore) to ensure mutual exclusion when accessing the buffer.

## Function signatures:

class ProducerConsumer {

public void producer() throws InterruptedException; public void consumer() throws InterruptedException;

}

## Input:

* + The buffer size n is provided as input.
  + Producers generate items continuously.
  + Consumers consume items continuously.

## Output:

* + Producers should print Produced: <item> when they successfully produce an item.
  + Consumers should print Consumed: <item> when they successfully consume an item.

## Constraints:

* 1 <= n <= 100 (Buffer size)

## Example: Input:

Buffer size: 5

Output: Produced: 1

Produced: 2

Consumed: 1

Produced: 3

Consumed: 2

Consumed: 3

Produced: 4

Produced: 5

# Class: ProducerConsumer

import java.util.LinkedList; import java.util.Queue;

import java.util.concurrent.Semaphore;

class ProducerConsumer {

private final Queue<Integer> buffer; private final int bufferSize;

private final Semaphore empty; private final Semaphore full; private final Semaphore mutex;

public ProducerConsumer(int size) { this.bufferSize = size;

this.buffer = new LinkedList<>(); this.empty = new Semaphore(size); this.full = new Semaphore(0); this.mutex = new Semaphore(1);

}

public void producer() throws InterruptedException { int item = 0;

while (true) { empty.acquire(); mutex.acquire(); buffer.add(++item);

System.out.println("Produced: " + item); mutex.release();

full.release(); Thread.sleep(100);

}

}

public void consumer() throws InterruptedException { while (true) {

full.acquire(); mutex.acquire();

int item = buffer.poll(); System.out.println("Consumed: " + item); mutex.release();

empty.release(); Thread.sleep(150);

}

}

public static void main(String[] args) { int bufferSize = 5;

ProducerConsumer pc = new ProducerConsumer(bufferSize);

Thread producerThread = new Thread(() -> { try {

pc.producer();

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

}

});

Thread consumerThread = new Thread(() -> { try {

pc.consumer();

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

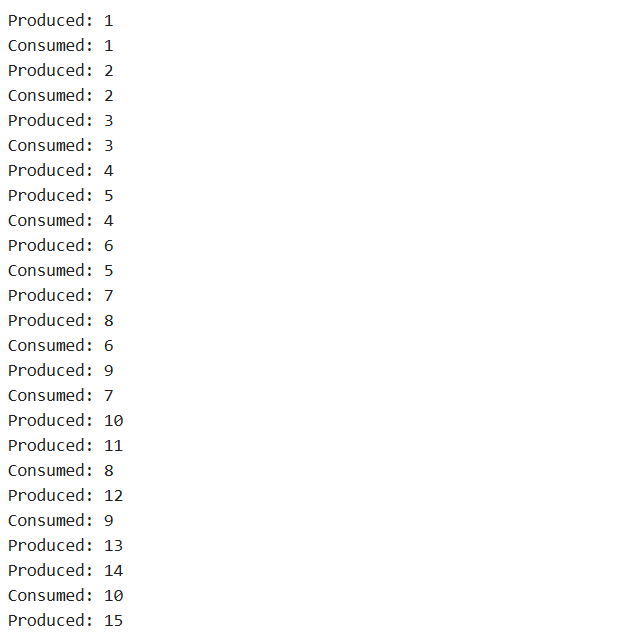
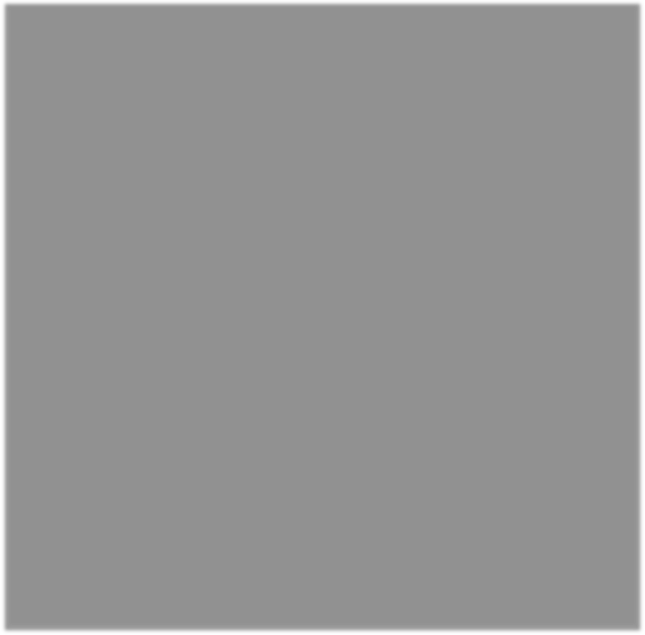
}

});

producerThread.start(); consumerThread.start();

}

}



**OUTPUT**