****



**TECNOLÓGICO NACIONAL DE MÉXICO**

**Instituto Tecnológico de León**

**INGENIERIA EN SISTEMAS COMPUTACIONALES**

**TOPICOS AVANZADOS DE PROGRAMACION**

**REPORTE – PRODUCER/CONSUMER**

**ALUMNO:**

LEONEL ALEJANDRO AGUIRRE SERRANO

**PROFESOR**

ING. LUIS EDUARDO GUTIERREZ AYALA

**LEÓN, GUANAJUATO A 20 DE MAYO DEL 2020**

**REDACCION DEL PROBLEMA:**

El problema presentado en este reporte consiste en la creación de un programa que necesite de la implementación de subprocesos sincronizados, de igual manera que mostrar la solución correspondiente.

**CODIGO FUENTE:**

**Clase ProducerConsumerGUI**

package com.milkyblue;

import java.awt.*BorderLayout*;

import java.awt.event.*ActionEvent*;

import java.awt.event.*ActionListener*;

import java.util.concurrent.*ExecutorService*;

import java.util.concurrent.*Executors*;

import javax.swing.*JButton*;

import javax.swing.*JCheckBox*;

import javax.swing.*JFrame*;

import javax.swing.*JPanel*;

import com.github.tomaslanger.chalk.*Chalk*;

// ProduceerConsumerGUI class. Models the GUI.

public class *ProducerConsumerGUI* {

  private *JFrame* mainFrame;

  private *JPanel* mainPanel, topPanel, bottomPanel;

  private *JCheckBox* chkSync;

  private *JButton* btnExecute;

  // Class constructor.

  public ProducerConsumerGUI() {

    // Enables color on terminal outputs.

*Chalk*.setColorEnabled(true);

    mainFrame = **new** JFrame("Producer Consumer");

    mainPanel = **new** JPanel(**new** BorderLayout());

    topPanel = **new** JPanel();

    bottomPanel = **new** JPanel();

    chkSync = **new** JCheckBox("Enable Synchronization");

    btnExecute = **new** JButton("Execute");

    // Main methods are called.

    addAttributes();

    addListeners();

    build();

    launch();

  }

  // Adds attributes to elements in the class.

  private *void* addAttributes() {

    mainFrame.setResizable(false);

    mainFrame.setDefaultCloseOperation(*JFrame*.EXIT\_ON\_CLOSE);

  }

  // Sets listeners to elements in GUI.

  private *void* addListeners() {

    // Creates a Buffer depending on the chkSync state, then with an ExecutorService

    // instance, executes a Producer and a Consumer that will be accessing to the

    // Buffer.

    btnExecute.addActionListener(**new** ActionListener() {

      public *void* actionPerformed(*ActionEvent* *e*) {

*boolean* isSync = chkSync.isSelected();

*ExecutorService* executor = *Executors*.newCachedThreadPool();

        // Buffer created based on chkSync state.

*Buffer* sharedBuffer = (isSync) ? **new** BlockingBuffer() : **new** UnSyncBuffer();

*System*.out.print("\033[H\033[2J");

*System*.out.flush();

        if (!isSync) {

*System*.out.println(*Chalk*.on("Action").bgMagenta() + "\t\t\t" + *Chalk*.on("Value").bgMagenta() + "\t"

              + *Chalk*.on("Produced sum").bgMagenta() + "\t" + *Chalk*.on("Consumed sum").bgMagenta());

*System*.out.println("------\t\t\t-----\t------------\t------------");

        }

        // Thread execution.

        executor.execute(**new** Producer(sharedBuffer, isSync));

        executor.execute(**new** Consumer(sharedBuffer, isSync));

        executor.shutdown();

      }

    });

  }

  // Builds the GUI.

  private *void* build() {

    topPanel.add(chkSync);

    bottomPanel.add(btnExecute);

    mainPanel.add(topPanel, *BorderLayout*.NORTH);

    mainPanel.add(bottomPanel, *BorderLayout*.SOUTH);

    mainFrame.add(mainPanel);

  }

  // Launches the GUI by setting to true the visible attribute of the frame.

  private *void* launch() {

    mainFrame.setVisible(true);

    mainFrame.pack();

    mainFrame.setLocationRelativeTo(null);

  }

}

**Clase Producer**

package com.milkyblue;

import java.util.*Random*;

import com.github.tomaslanger.chalk.*Chalk*;

// Producer class. Models a Producer object that will be storing values into a passed Buffer.

public class *Producer* implements *Runnable* {

  private final static *Random* generator = **new** Random();

  private final *Buffer* sharedBuffer;

  private final *boolean* isSync;

  // Class constructor. Takes a Buffer object where the producer will be working

  // on. Also get a boolean value to specify if the Buffer is working with

  // synchronized approach.

  public Producer(*Buffer* *sharedBuffer*, *boolean* *isSync*) {

*this*.sharedBuffer = sharedBuffer;

*this*.isSync = isSync;

  }

  // run method is called when a thread based in an instance of Producer is

  // executed by an ExecutorService.

  public *void* run() {

*int* sum = 0;

    for (*int* count = 1; count <= 10; count++) {

      try {

        // Waits up to 3 seconds before each iteration.

*Thread*.sleep(generator.nextInt(3000));

        // Adds the new value into the Buffer.

        sharedBuffer.put(count);

        sum += count;

        if (!isSync)

*System*.out.println("\t" + *Chalk*.on(*Integer*.toString(sum)).cyan());

      } catch (*InterruptedException* *exception*) {

        exception.printStackTrace();

      }

    }

*System*.out.println("\n[" + *Chalk*.on("Producer").cyan() + "] stopped producing. Terminating...\n");

  }

}

**Clase Consumer**

package com.milkyblue;

import java.util.*Random*;

import com.github.tomaslanger.chalk.*Chalk*;

// Consumer class. Models a Consumer object that reads data from a passed Buffer.

public class *Consumer* implements *Runnable* {

  private final static *Random* generator = **new** Random();

  private final *Buffer* sharedBuffer;

  private final *boolean* isSync;

  // Class constructor. Takes a Buffer object where the Consumer will be working

  // on. Also a boolean value is passed to keep track of the CheckBox state.

  public Consumer(*Buffer* *sharedBuffer*, *boolean* *isSync*) {

*this*.sharedBuffer = sharedBuffer;

*this*.isSync = isSync;

  }

  // run method is called when a Thread based on an instance of Consumer is

  // executed by an ExecutorService.

  public *void* run() {

*int* sum = 0;

    for (*int* count = 1; count <= 10; count++) {

      try {

        // Waits up to 3 seconds before reading the Buffer.

*Thread*.sleep(generator.nextInt(3000));

        // Takes a value from the Buffer.

        sum += sharedBuffer.take();

        if (!isSync)

*System*.out.println("\t\t\t" + *Chalk*.on(*Integer*.toString(sum)).yellow());

      } catch (*InterruptedException* *exception*) {

        exception.printStackTrace();

      }

    }

*System*.out.println("\n[" + *Chalk*.on("Consumer").yellow() + "] read values, total: "

        + *Chalk*.on(*Integer*.toString(sum)).bgGreen().black() + ". Terminating...\n");

  }

}

**Interfaz Buffer**

package com.milkyblue;

// Buffer interface. Models a Buffer object with two actions, put and take.

public interface *Buffer* {

  public *void* put(*int* *value*) throws *InterruptedException*;

  public *int* take() throws *InterruptedException*;

}

**Clase UnSyncBuffer**

package com.milkyblue;

import com.github.tomaslanger.chalk.*Chalk*;

// UnSyncBuffer class. Models a non-syncrhonized Buffer based object.

public class *UnSyncBuffer* implements *Buffer* {

  private *int* buffer = -1;

  // put method is overwritten, adds a passed value to buffer.

  public *void* put(*int* *value*) throws *InterruptedException* {

*System*.out.print("[" + *Chalk*.on("Producer").cyan() + "] writes\t" + *Chalk*.on(*Integer*.toString(value)).green());

    buffer = value;

  }

  // take method is overwritten, takes a value from buffer.

  public *int* take() throws *InterruptedException* {

*System*.out.print("[" + *Chalk*.on("Consumer").yellow() + "] reads\t" + *Chalk*.on(*Integer*.toString(buffer)).green());

    return buffer;

  }

}

**Clase BlockingBuffer**

package com.milkyblue;

import java.util.concurrent.*ArrayBlockingQueue*;

import com.github.tomaslanger.chalk.*Chalk*;

// BlockingBuffer class. Models a syncrhonized Buffer based object that works over an

// ArrayBlockingQueue instance.

public class *BlockingBuffer* implements *Buffer* {

  private final *ArrayBlockingQueue*<*Integer*> buffer;

  // Class constructor.

  public BlockingBuffer() {

    buffer = **new** *ArrayBlockingQueue*<*Integer*>(1);

  }

  // put method is overwritten, adds a passed value to buffer.

  public *void* put(*int* *value*) throws *InterruptedException* {

    buffer.put(value);

*System*.out.print("[" + *Chalk*.on("Producer").cyan() + "] writes:\t" + *Chalk*.on(*Integer*.toString(value)).green()

        + "\tCells taken: " + *Chalk*.on(*Integer*.toString(buffer.size())).green() + "\n");

  }

  // take method is overwritten, takes a value from buffer.

  public *int* take() throws *InterruptedException* {

*int* readValue = buffer.take();

*System*.out.print("[" + *Chalk*.on("Consumer").yellow() + "] reads:\t" + *Chalk*.on(*Integer*.toString(readValue)).green()

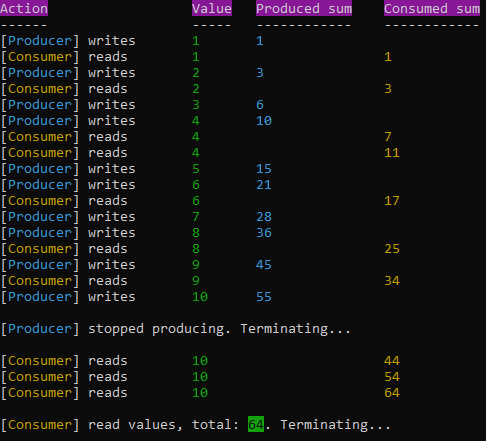
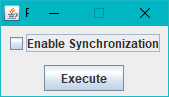
        + "\tCells taken: " + *Chalk*.on(*Integer*.toString(buffer.size())).green() + "\n");

    return readValue;

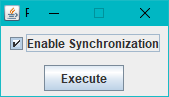
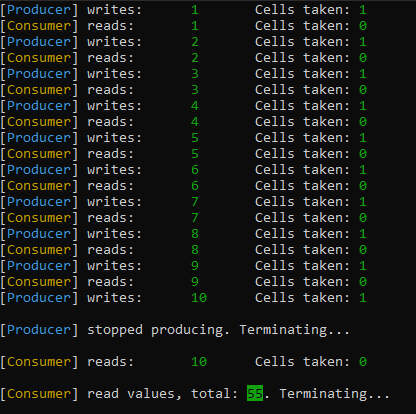
  }

}

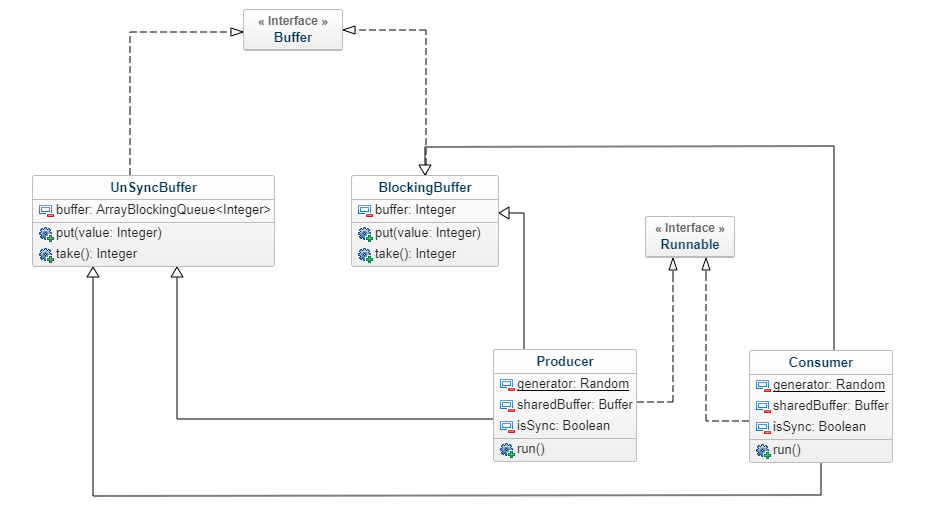
**CAPTURAS:**



**ESTADO INICIAL DE LA INTERFAZ Y RESULTADOS PRODUCIDOS CUANDO SE EJECUTA SIN HABILITAR LA SINCRONIZACION DE SUBPROCESOS.**



**PROGRAMA EJECUTADO CON LA SINCRONIZACION DE SUBPROCESOS HABILITADA Y SUS RESULTADOS CORRESPONDIENTES.**

**DIAGRAMA UML:**

**PREGUNTAS:**

1. **¿Se pueden crear 2 hilos Productores y un solo Consumidor?**

Se puede, pero conseguiremos valores aún más alejados del esperado, ya que cada uno de los valores del 1 al 10 se agregarán dos veces, una por cada Productor, por lo que se repetirán valores, provocando que tanto en el método no sincronizado como en el sincronizado se obtenga un valor distinto al esperado.

1. **¿Se pueden crear 2 hilos Consumidores y un solo Productor?**

Si, pero ahora el buffer llegara a un punto en el que ya no obtendrá nuevos valores, y por lo tanto ambos consumidores leerán valores repetidos.

1. **Crea varios productores y varios consumidores, da una conclusión.**

A pesar de que durante la ejecución no obtendremos ningún error, por cada productor obtendremos un grupo más de valores del 1 al 10 que se repetirán en el buffer, al igual que por cada consumidor se intentaran leer 10 valores del mismo buffer, provocando nuevamente que cada consumidor lea datos repetidos.

**CONCLUSION:**

La implementación de subprocesos sincronizados en nuestras aplicaciones es una herramienta más que hace más flexible y útil la funcionalidad de estas mismas, es algo que siempre se debe de tener en cuenta para mejorar procesos y que estos se realicen de una manera óptima, en este caso en especifico cuando sabemos que vamos a utilizar un mismo recurso por distintos subprocesos.

**NOTAS:**

* Puede encontrar el repositorio de este proyecto en mi cuenta de github en el siguiente enlace: <https://github.com/NoisyApple/AdTopics-16.ProducerConsumer>