

Programación de servicios y procesos

Tarea para PSP02.

Ejercicio 1

De igual manera a lo visto en el tema, ahora te proponemos un ejercicio del tipo productor-consumidor que mediante un hilo productor almacene datos (15 caracteres) en un búfer compartido, de donde los debe recoger un hilo consumidor (consume 15 caracteres). La capacidad del búfer ahora es de 6 caracteres, de manera que el consumidor podrá estar cogiendo caracteres del búfer siempre que éste no esté vacío. El productor sólo podrá poner caracteres en el búfer, cuando esté vacío o haya espacio.

Para la resolución de este ejercicio, había realizado previamente algunos de los ejemplos facilitados en el tema. Uno de ellos, el del almacén de cuadros, se adecuaba bastante bien a lo propuesto en la tarea y lo he modificado creando los hilos productor y consumidor y una clase principal, llamada "productorconsumidor", para los métodos, la creación de los hilos y su inicialización.

HILO_PRODUTOR.JAVA

```
/*
 * TAREA PSP02. EJERCICIO 1.
 * Un hilo productor debe almacenar 15 caracteres en un búfer
compartido.
 * Un hilo consumidor, los recoge Pueden depositarse hasta 6
caracteres, de manera
 * que puedan ccnsuimirse siempre que el bufer no esté vacío.
 * El hilo productor solo puede añadir si el bufer está vacío o hay
espacio.
 */
package tareaproductorconsumidor;

/**
 *
 * @author juang <juangmuelas@gmail.com>
 * @since 17/11/2020
 */
public class Hilo_Productor extends Thread {
    /**
     * @param prod_con objeto compartido
     */
    private ProductorConsumidor prod_con;

    /**
     * @param caracteres string que completa el buffer con las letras
de teclado
     */
    private String caracteres = "QWERTYUIOPASDFGHJKLÑZXCVBNM";

    /**
     * Constructor que recoge un parámetro para uso local
     * @param pc copia del objeto
     */
    public Hilo_Productor(ProductorConsumidor pc) {
        prod_con = pc;
    }

    /**
     * método run
     */
}
```

```

@Override
public void run() {
    /**
     * @param letra recoge caracteres desde el bucle
     * @throws InterruptedException
     */
    char letra;

    //Podemos añadir 15 caracteres
    for( int i = 0; i < 15; i++){
        letra =
caracteres.charAt((int) (Math.random()*27)); //contiene Ñ=27
        prod_con.depositar(letra);
        //Si puede añadir un nuevo caracter
        System.out.println("Depositado el caracter " + letra + "
en el buffer.");

        //Debe comprobar si puede añadir otra
        try{
            sleep(100);
        }catch(InterruptedException e){}
    } //Fin del for que controla la ejecución
} //Fin run()
} //Fin clase Hilo_Productor

```

HILO_CONSUMIDOR.JAVA

```

/*
 * TAREA PSP02. EJERCICIO 1.
 * Un hilo productor debe almacenar 15 caracteres en un búfer
compartido.
 * Un hilo consumidor, los recoge Pueden depositarse hasta 6
caracteres, de manera
 * que puedan consumirse siempre que el bufer no esté vacío.
 * El hilo productor solo puede añadir si el bufer está vacío o hay
espacio.
 */
package tareaproductorconsumidor;

/**
 *
 * @author juang <juangmuelas@gmail.com>
 * @since 17/11/2020
 */
public class Hilo_Consumidor extends Thread {
    /**
     * @param prod_con objeto compartido
     */
    private ProductorConsumidor prod_con;

    /**
     * Constructor que recoge un parámetro para uso local
     * @param pc copia del objeto
     */
    public Hilo_Consumidor(ProductorConsumidor pc) {
        prod_con = pc;
    }

    /**
     * método run
     */
    @Override

```

```

public void run() {
    /**
     * @param letra recoge caracteres desde el bucle
     * @throws InterruptedException
     */
    char letra;

    //Podemos añadir 15 caracteres
    for( int i = 0; i < 15; i++){
        letra =prod_con.recoger();
        //Muestra el caracter recogido
        System.out.println("Recogido el caracter " + letra + " en
el buffer.");

        //Debe comprobar si puede añadir otra
        try{
            sleep(100);
        }catch(InterruptedException e){}
    } //Fin del for que controla la ejecución
} //Fin run()
} //Fin clase Hilo_Consumidor

```

PRODUCTORCONSUMIDOR.JAVA

```

/*
 * TAREA PSP02. EJERCICIO 1.
 * Un hilo productor debe almacenar 15 caracteres en un búfer
compartido.
 * Un hilo consumidor, los recoge Pueden depositarse hasta 6
caracteres, de manera
 * que puedan ccnsumirse siempre que el bufer no esté vacío.
 * El hilo productor solo puede añadir si el bufer está vacío o hay
espacio.
 */
package tareaproductorconsumidor;

/**
 *
 * @author juang <juangmuelas@gmail.com>
 * @since 17/11/2020
 */
public class ProductorConsumidor {

    /**
     * @param args the command line arguments
     */

    // pide reducir la entrada a 6, generamos el buffer
    private char buffer[] = new char[6];

    // Controles para el estado del buffer
    /**
     * @param nuevodato integer de llenado
     * @param buffervacio booleano que controla estado buffer. Si es
true permite
     * añadir nuevos caractees al buffer.
     * @param bufferlleno booleano que controla estado buffer
     */
    private int nuevodato = 0;
    private boolean bufferVacio = true;
    private boolean bufferlleno = false;

```

```

/**
 * Método depositar que añade caracteres en el buffer
 * Mientras esté lleno, este espera con wait().
 * Si puede, añade un caracter.
 * Luego comprueba si está lleno (máximo 6 caracteres), para ver
si puede
 * continuar.
 * @param c
 */
public synchronized void depositar( char c ) {
    // Espera hasta que haya sitio para otra letra
    while( bufferlleno == true )
    {
        try {
            wait();
        } catch( InterruptedException e ) {}
    } //Fin while
    buffer[nuevodato] = c;
    nuevodato++;
    if( nuevodato == 6 )
        bufferlleno = true;
        bufferVacio = false;
    notify();
} //Fin método depositar

/**
 * Método recoger para retirar letras del buffer.
 * Mientras esté vacío, este espera con wait().
 * Si hay alguno, decrementa en uno al consumirlo.
 * Luego comprueba si es la última, de ser así cambiaría a vacío y
pueden
 * entrar nuevos valores.
 * @return nuevodato al thread del consumidor
 */
public synchronized char recoger() {
    while( bufferVacio == true ){
        try {
            wait();
        } catch( InterruptedException e ) {}
    } //Fin del while
    nuevodato--;
    // Comprueba si se retiró la última letra
    if( nuevodato == 0 )
        bufferVacio = true;
        bufferlleno = false;
    notify();

    return( buffer[nuevodato] );
} //Fin método recoger

public static void main(String[] args) {
    /**
     * Creamos un nuevo objeto de nuestra clase principal
     */
    ProductorConsumidor prod_con = new ProductorConsumidor();

    //Creamos los hilos
    Hilo_Productor hp = new Hilo_Productor(prod_con);
    Hilo_Consumidor hc = new Hilo_Consumidor(prod_con);

```

```

        //Iniciamos ejecución de los hilos
        hp.start();
        hc.start();
    } //Fin main
} //Fin clase ProductorConsumidor

```

La salida, como pide el enunciado, recoge el último carácter guardado.

```

>
Salida - TareaProductorConsumidor (run) x
run:
Recogido el caracter M en el buffer.
Depositado el caracter M en el buffer.
Depositado el caracter E en el buffer.
Recogido el caracter E en el buffer.
Depositado el caracter Y en el buffer.
Recogido el caracter Y en el buffer.
Depositado el caracter X en el buffer.
Recogido el caracter X en el buffer.
Depositado el caracter I en el buffer.
Recogido el caracter I en el buffer.
Depositado el caracter I en el buffer.
Recogido el caracter I en el buffer.
Depositado el caracter Ñ en el buffer.
Recogido el caracter Ñ en el buffer.
Depositado el caracter F en el buffer.
Recogido el caracter F en el buffer.
Depositado el caracter V en el buffer.
Recogido el caracter V en el buffer.
Depositado el caracter C en el buffer.
Recogido el caracter C en el buffer.
Depositado el caracter Ñ en el buffer.
Recogido el caracter Ñ en el buffer.
Recogido el caracter N en el buffer.
Depositado el caracter N en el buffer.
Depositado el caracter Q en el buffer.
Recogido el caracter Q en el buffer.

```

Ejercicio 2

De igual manera a lo visto en el tema, ahora te proponemos que resuelvas el clásico problema denominado "La cena de los filósofos" utilizando la clase `Semaphore` del paquete `java.util.concurrent`.

El problema es el siguiente: Cinco filósofos se sientan alrededor de una mesa y pasan su vida comiendo y pensando. Cada filósofo tiene un plato de arroz chino y un palillo a la izquierda de su plato. Cuando un filósofo quiere comer arroz, cogerá los dos palillos de cada lado del plato y comerá. El problema es el siguiente: establecer un ritual (algoritmo) que permita comer a los filósofos. El algoritmo debe satisfacer la exclusión mutua (dos filósofos no pueden emplear el mismo palillo a la vez), además de evitar el interbloqueo y la inanición.



Siguiendo los consejos propuestos en la plataforma para la resolución de la tarea, se ha aprovechado la estructura propuesta en [PSP02_Recurso_Tarea.zip](#).

De ahí, generamos dos clases: `Cenafilosofos.java` y `Filosofo.java`

FILOSOFO.JAVA

```
/*
 * TAREA PSP02. EJERCICIO 2.
 * La cena de los filósofos.
 * 5 filósofos se sientan alrededor de una mesa para comer arroz y
pensar.
 * Cada filósofo tiene un plato de arroz chino y un palillo a la
izquierda de
 * su plato. Cuando un filósofo quiere comer arroz, cogerá los dos
palillos de
 * cada lado del plato y comerá. El problema es el siguiente:
establecer un ritual
 * (algoritmo) que permita comer a los filósofos. El algoritmo debe
satisfacer
 * la exclusión mutua (dos filósofos no pueden emplear el mismo
palillo a la
 * vez), además de evitar el interbloqueo y la inanición.
 */
package cenafilosofos;

import java.util.Random;
import java.util.concurrent.Semaphore;

/**
 *
 * @author juang <juangmuelas@gmail.com>
 * @since 18/11/2020
 * @version 1
 */
public class Filosofo extends Thread {

    /**
     * Siguiendo la documentación facilitada en el recurso javadoc de
la tarea,
     * Se crea el hilo Filósofo, con métodos pensar y comer, que se
invocan en un
     * bucle infinito en el método run()
     */

    /**
     * Declaramos las variables siguiendo las indicaciones del javadoc
de la tarea.
     * @param miIndice índice que identifica al filósofo (un entero
del 0 al 4)
     * @param semaforoPalillo vector de semáforos (uno para cada
palillo).
     * @param palilloFilosofo matriz que para cada valor de su primer
índice,
     * la fila, almacena los palillos que necesita el filósofo de ese
índice
     * para comer. Por ejemplo: el filósofo de índice 0 necesita los
palillos de
     * índices {0,4}, el de índice 1 los de índices {1,0}, etc...
Puedes

```

```

    * prescindir de este vector si se te ocurre como calcular en
    tiempo real
    * los índices de los palillos que necesita cada filósofo para
    comer
    * @param palilloDerecho integer extraído del array comentado
    antes.
    * @param palilloIzquierdo integer extraído del array comentado
    antes.
    */
    private int miIndice;
    private final Semaphore[] semaforoPalillo;
    private final int[][] palilloFilosofo;
    private int palilloDerecho;
    private int palilloIzquierdo;

    /**
     * Constructor de tres parámetros, cada uno de los cuales se
    guardará en una
     * variable local para usarla cuando sea necesario
     * @param miIndice
     * @param semaforoPalillo
     * @param palilloFilosofo
     */
    public Filosofo(int miIndice, Semaphore[] semaforoPalillo, int[][]
    palilloFilosofo) {
        this.miIndice = miIndice;
        this.semaforoPalillo = semaforoPalillo;
        this.palilloFilosofo = palilloFilosofo;
        this.palilloDerecho = palilloFilosofo[miIndice][1];
        this.palilloIzquierdo = palilloFilosofo[miIndice][0];
    } // Fin constructor

    /**
     * @param tiempoAleatorio objeto de la clase Random para marcar
    los tiempos
     * en cada proceso
     */

    private final Random tiempoAleatorio = new Random();

    /**
     * método pensar(): mostrará un mensaje en la Salida de que el
    'Filósofo
     * ' N ' está pensando'.
     * @return miIndice indica el filósofo que está pensando
     * @throws InterruptedException si no captura ningún valor
     */
    protected void pensar() {
        System.out.println("Filósofo " + miIndice + " está
    pensando.");
        try {
            // Damos un tiempo para pensar
            Filosofo.sleep(tiempoAleatorio.nextInt(4000) + 500);
        } catch (InterruptedException ex) {
            System.out.println("Error en el método pensar(): " +
            ex.toString());
        }
    } // Fin método pensar

```

```

protected void comer() {
    /**
     * método comer(): mostrará un mensaje en la Salida de que el
'Filósofo
     * ' N ' está hambriento', mientras trata de conseguir los dos
palillos
     * que necesita para comer.
     *
     * El filósofo debe intentar coger el palillo de su izquierda.
Si lo
     * tiene, intenta coger el de la derecha para comer.
     * Para ello, usamos el boolean tryAcquire, que intenta
adquirir un
     * permiso en caso de estar disponible, devolviendo en ese
supuesto un true.
     *
     * Asimismo, facilitamos la lectura en consola mediante el uso
de color
     * en cada tipo de salida.
     *
     * @throws InterruptedException si falla el booleano o el
tiempo.
     */

    if (semaforoPalillo[palilloIzquierdo].tryAcquire()) {
        if (semaforoPalillo[palilloDerecho].tryAcquire()) {
            System.out.println("\033[36mFILÓSOFO " + miIndice + "
ESTÁ COMIENDO.");
            try {
                // Damos un tiempo para comer
                sleep(tiempoAleatorio.nextInt(3000) + 500);
            } catch (InterruptedException ex) {
                System.out.println("Error : " + ex.toString());
            }
            System.out.println("\033[33mFilósofo " + miIndice + "
termina de comer.Libres palillos " + palilloIzquierdo + " , " +
palilloDerecho);
            /**
             * Tras usar Acquire para el permiso, se libera con
release
             */
            semaforoPalillo[palilloDerecho].release();
        }
        /**
         * Tras usar Acquire para el permiso, se libera con
release
         */
        semaforoPalillo[palilloIzquierdo].release();
    } else {
        System.out.println("\033[31mFilósofo " + miIndice + "
hambriento.");
        // si no consigue comer.
    }
} //Fin método comer

/**
 * bucle infinito: llamada al método pensar(), llamada al método
comer()
 */

```



```

@Override
public void run(){
    while (true){
        pensar();
        comer();
    } //fin bucle while
} //fin método run
} //Fin clase Filosofo

```

CLASE CENAFILOSOFOS.JAVA

```

/*
 * TAREA PSP02. EJERCICIO 2.
 * La cena de los filósofos.
 * 5 filósofos se sientan alrdeor de una mes para comer arroz y
pensar.
 * Cada filósofo tiene un plato de arroz chino y un palillo a la
izquierda de
 * su plato. Cuando un filósofo quiere comer arroz, cogerá los dos
palillos de
 * cada lado del plato y comerá. El problema es el siguiente:
establecer un ritual
 * (algoritmo) que permita comer a los filósofos. El algoritmo debe
satisfacer
 * la exclusión mutua (dos filósofos no pueden emplear el mismo
palillo a la
 * vez), además de evitar el interbloqueo y la inanición.
 *
 */
package cenafilosofos;

import java.util.concurrent.Semaphore;

/**
 *
 * @author juang <juangmuelas@gmail.com>
 * @since 18/11/2020
 * @version 1
 */
public class CenaFilosofos {

    /**
     * Siguiendo la documentación facilitada en el recurso javadoc de
la tarea,
     * Define los parámetros necesarios para construir hilos filósofos
(ver el
     * constructor de la clase Filosofo). Se crean los 5 filósofos, y
los inicia.
     * @param args the command line arguments
     */

    final static int numFilosofos = 5;

    final static int[][] palilloFilosofo = {
        {0, 4}, // palillos filosofo 0
        {1, 0}, // palillos filosofo 1
        {2, 1}, // palillos filosofo 2
        {3, 2}, // palillos filosofo 3
        {4, 3} // palillos filosofo 4
    };
}

```

```

        final static Semaphore[] semaforoPalillo = new
Semaphore[numFilosofos];

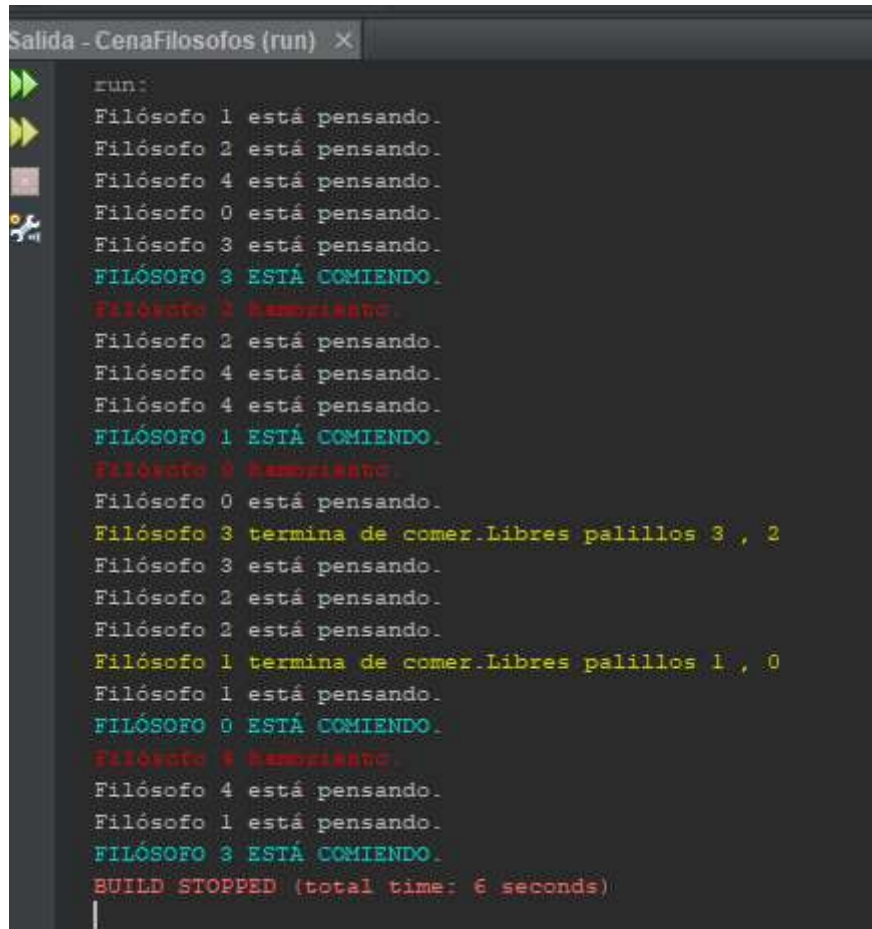
        public static void main(String[] args) {

            /**
             * Control a través del semáforo de los índices de palillos
para cada
             * filósofo. El valor es 1 para dar un acceso por palillo
             */
            for(int i = 0; i < numFilosofos; i++){
                semaforoPalillo[i] = new Semaphore(1);
            } //Fin for

            //Creamos los filósofos y se inician.
            for(int miIndice=0; miIndice < numFilosofos; miIndice++){
                new Filosofo(miIndice, semaforoPalillo,
palilloFilosofo).start();
            }
        } //Fin main
    } //Fin clase CenaFilosofos

```

Para la salida, he probado con distintos rangos de tiempo para ver como ayudar a mejorar la rotación, pero salvo con alguna descoordinación expresa, por descompensación entre los valores de espera y tiempo de comer, no he observado grandes cambios. También, he intentado que se mostrase mejor cada evento mediante distintos colores:



```

run:
Filósofo 1 está pensando.
Filósofo 2 está pensando.
Filósofo 4 está pensando.
Filósofo 0 está pensando.
Filósofo 3 está pensando.
FILÓSOFO 3 ESTÁ COMIENDO.
Filósofo 3 pensando.
Filósofo 2 está pensando.
Filósofo 4 está pensando.
Filósofo 4 está pensando.
FILÓSOFO 1 ESTÁ COMIENDO.
Filósofo 0 pensando.
Filósofo 0 está pensando.
Filósofo 3 termina de comer.Libres palillos 3 , 2
Filósofo 3 está pensando.
Filósofo 2 está pensando.
Filósofo 2 está pensando.
Filósofo 1 termina de comer.Libres palillos 1 , 0
Filósofo 1 está pensando.
FILÓSOFO 0 ESTÁ COMIENDO.
Filósofo 0 pensando.
Filósofo 4 está pensando.
Filósofo 1 está pensando.
FILÓSOFO 3 ESTÁ COMIENDO.
BUILD STOPPED (total time: 6 seconds)

```