

PRÁCTICA #4

SINCRONIZACION DE PROCESOS

Objetivo.

- Revisar implementaciones con las diferentes herramientas para la sincronización de procesos concurrentes

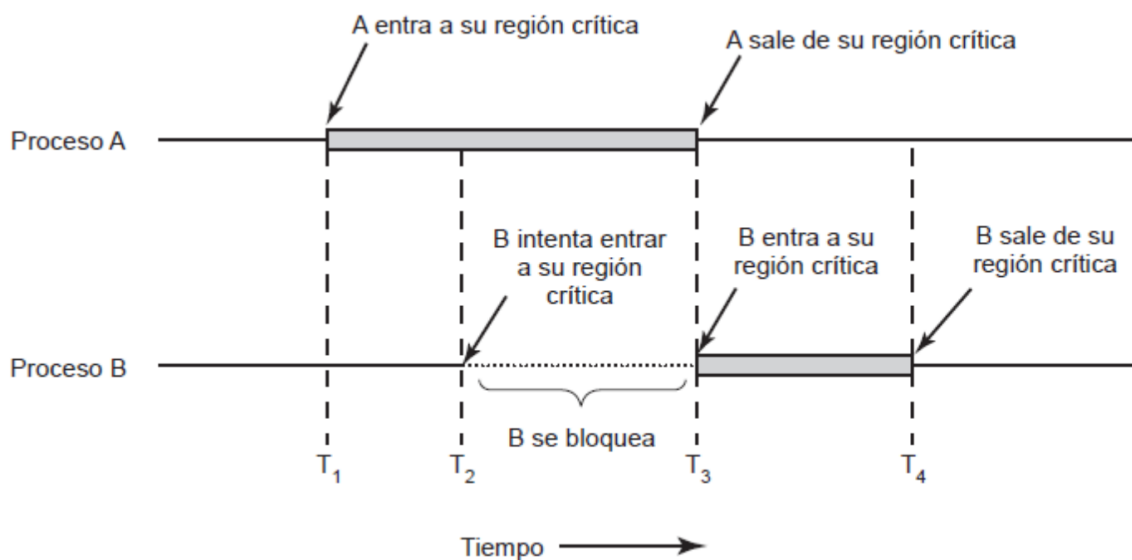
Introducción

En diversos problemas o situaciones dos o más procesos pueden requerir acceder a las mismas variables o datos

Cuando varios procesos deben manejar los mismos datos concurrentemente y el resultado de la ejecución depende del orden concreto en que se produzcan los accesos, se conoce como condición de carrera. En tales situaciones se necesita garantizar que solo un proceso pueda acceder a esas variables o datos.

Esto se realiza mediante la sincronización. La sección crítica de un proceso es un segmento de código en el que se van a encontrar variables que pueden ser accedidas por otros procesos. Cuando un proceso se encuentra dentro de su sección crítica, ningún otro proceso puede ejecutar su respectiva región crítica.

El problema de la sección crítica consiste en diseñar un protocolo para que los procesos puedan trabajar de esta manera



Cualquier solución al problema de la sección crítica deberá satisfacer los siguientes requisitos

- Exclusión mutua
- Progreso
- Espera limitada

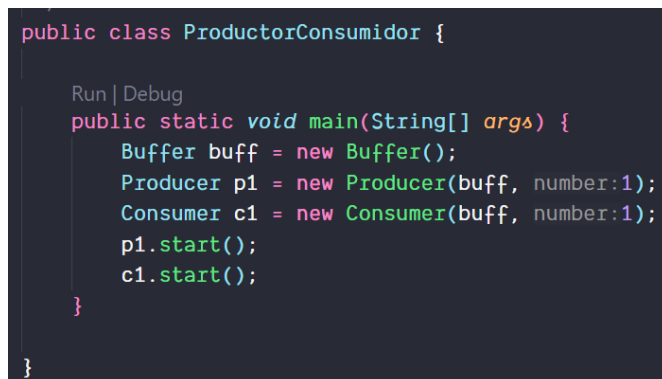
SECCIÓN 1.- PRODUCTOR CONSUMIDOR

1.1.- Explica a grandes rasgos en qué consiste el problema del productor consumidor.

El problema del productor-consumidor se refiere a un escenario donde hay un productor que produce ítems o datos y un consumidor que consume esos ítems. Ambos comparten una memoria o buffer, y hay que garantizar la sincronización adecuada para que el productor no produzca cuando el buffer esté lleno y el consumidor no consuma cuando el buffer esté vacío.

1.2.- Ejecuta el programa de ejemplo proporcionado de Productor consumidor en Java y responde las siguientes preguntas.

a) Explica el funcionamiento del programa



```

public class ProductorConsumidor {

    Run | Debug
    public static void main(String[] args) {
        Buffer buff = new Buffer();
        Producer p1 = new Producer(buff, number:1);
        Consumer c1 = new Consumer(buff, number:1);
        p1.start();
        c1.start();
    }
}

```

Se puede ver claramente que comparten el mismo buffer ya que se les esta pasando a Producer y a Consumer, y después inician ambos procesos llamados p1 y c1.



```

class Producer extends Thread{
    Buffer buff;
    int number;

    public Producer(Buffer buff, int number) {
        this.buff = buff;
        this.number = number;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            buff.put(i);
            System.out.println("Productor #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {}
        }
    }
}

```

Se puede ver que Producer extiende de la clase Thread, lo que hace que tenga todas las propiedades de un hilo y puede ser ejecutado en paralelo o concurrente mente con otros hilos, como en este caso junto con Consumer.

También en la sección de run, se sobrescribe el método run, lo que hace es que ahora el productor ingrese un valor en el buffer, y se imprima que valor puso, también se manda a llamar sleep, para simular un contexto real en el que la producción no es constante y así también dar paso a que el consumidor pueda obtener valores del buffer.

```
class Consumer extends Thread {
    Buffer buff;
    int number;

    public Consumer(Buffer c, int number) {
        buff = c;
        this.number = number;
    }
    @Override
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = buff.get();
            System.out.println("Consumidor" + this.number + " got: " + value);
        }
    }
}
```

Esta clase representa al consumidor

En el consumer cambia, ya que en el run en lugar de usar put, usa get para obtener los valores previamente puestos por el productor, aquí no se usa sleep, en el bucle for, también solo va a tomar 10 elementos del buffer. Y se imprimen en pantalla.

```

class Buffer {

    private int contents;
    private boolean available = false;
    public synchronized int get() {
        while (available == false) {
            System.out.println(x:"no pude tomar");
            try {
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        System.out.println(x:"Pude tomar");
        available = false;
        notifyAll();
        return contents;
    }

    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}

```

Algo muy importante en este es que se usa synchronized, que asegura que solo un hilo pueda ejecutar este método a la vez, lo que previene problemas de concurrencia.

El bucle while verifica si hay valores para consumir y sino imprime no pude tomar y el proceso entre en estado de espera hasta que haya valores disponibles para tomar

Cuando el buffer está vacío, el productor coloca el valor en contents, establece available en true para indicar que hay un valor en el buffer, y luego usa notifyAll() para despertar a todos los hilos que están esperando (en este caso, principalmente el consumidor que podría estar esperando para consumir).

la clase Buffer asegura que:

El productor solo pueda producir cuando el buffer esté vacío.

El consumidor solo pueda consumir cuando haya un valor en el buffer.

Solo un hilo pueda acceder al buffer a la vez, gracias a los métodos synchronized.

```
Puede tomar
Consumidor1 got: 0
no pude tomar
Productor #1 put: 0
Puede tomar
Productor #1 put: 1
Consumidor1 got: 1
no pude tomar
Puede tomar
Productor #1 put: 2
Consumidor1 got: 2
no pude tomar
Puede tomar
Productor #1 put: 3
Consumidor1 got: 3
no pude tomar
Productor #1 put: 4
Puede tomar
Consumidor1 got: 4
no pude tomar
Puede tomar
Productor #1 put: 5
Consumidor1 got: 5
no pude tomar
Puede tomar
Productor #1 put: 6
Consumidor1 got: 6
no pude tomar
Puede tomar
Productor #1 put: 7
Consumidor1 got: 7
no pude tomar
Puede tomar
Productor #1 put: 8
Consumidor1 got: 8
no pude tomar
Puede tomar
Productor #1 put: 9
Consumidor1 got: 9
```

b) Qué elemento del programa es el equivalente de la memoria compartida

El equivalente de la memoria compartida en este programa es la clase Buffer, específicamente la variable contents dentro de la clase Buffer. Esta es la ubicación donde el productor coloca los ítems y el consumidor los toma.

c) Qué elementos del programa son el equivalente de los procesos o hilos

Los elementos equivalentes a los procesos o hilos en el programa son las clases `Producer` y `Consumer`. Ambas clases extienden de la clase `Thread`, lo que significa que actúan como hilos separados que pueden ejecutarse en paralelo.

d) Qué elementos del programa son los que permiten que el productor y el consumidor no accedan al mismo tiempo a la región compartida.

Son varios elementos que trabajan juntos para garantizar esto:

Métodos Sincronizados: Los métodos `put()` y `get()` en la clase `Buffer` están marcados con la palabra clave `synchronized`. Esto asegura que solo un hilo (ya sea un productor o un consumidor) pueda acceder al método a la vez.

Variables de Control: La variable `available` en la clase `Buffer` actúa como un indicador para saber si el buffer está ocupado o no. Esta variable ayuda a determinar si el productor puede producir o el consumidor puede consumir.

Métodos `wait()` y `notifyAll()`: Estos métodos se utilizan dentro de `put()` y `get()` para hacer que un hilo espere si no puede realizar su acción o para notificar a otros hilos que pueden proceder. Por ejemplo, si el buffer está vacío, el consumidor esperará usando `wait()` hasta que el productor haya producido un ítem y lo haya notificado con `notifyAll()`.

e) Modifica el programa para limitar la región compartida (max 50 unidades), que el productor genere varias unidades (no solo una) y el consumidor pueda tomar más de una unidad. Agrega un análisis de los elementos necesarios para la realización de estas modificaciones, incluyendo las complicaciones que enfrentaste y la forma de resolverlas

```
2 package productorconsumidor;
3 import java.util.Arrays;
4
5
6
7 You, hace 9 segundos | 1 author (You)
8 class Producer extends Thread {
9     Buffer buff;
10    int number;
11
12    public Producer(Buffer buff, int number) {
13        this.buff = buff;
14        this.number = number;
15    }
16
17    @Override
18    public void run() {
19        int totalProduced = 0;
20        while (totalProduced < 50) {
21            int quantity = 1 + (int)(Math.random() * 5); // Produciendo entre 1 y 5 unidades
22            int[] values = new int[quantity];
23            for (int i = 0; i < quantity; i++) {
24                values[i] = totalProduced + i;
25            }
26            buff.put(values);
27            System.out.println("Productor #" + this.number + " put: " + Arrays.toString(values));
28            totalProduced += quantity;
29            try {
30                sleep((int)(Math.random() * 100));
31            } catch (InterruptedException e) { }
32        }
33    }
34
35 }
```

Se hicieron varias modificaciones para poder conseguir ese resultado, como se puede ver en el código se creo el totalProduced que ahora tiene que ser menor a 50 para que se cumpla, también se puso que pueda producir entre 1 y 5 unidades, ya que no se especifico la cantidad se decidió hacer así con random, con base en esta cantidad se crea un arreglo de valores que se llena con base en los producidos y se incrementa, también el buffer al momento de usar put se ponen los valores que se produjeron. Y se imprimen en pantalla, y se incrementa el valor de total producidos con base en la cantidad. De igual forma se usa sleep para que puedan consumirse los productos generados.

```
2 package productorconsumidor;
3
4 import java.util.Arrays;
5
6
7 You, hace 4 segundos | 1 author (You)
8 class Consumer extends Thread {
9     Buffer buff;
10     int number;
11
12     public Consumer(Buffer c, int number) {
13         buff = c;
14         this.number = number;
15     }
16
17     @Override
18     public void run() {
19         int totalConsumed = 0;
20         while (totalConsumed < 50) {
21             int quantity = 1 + (int)(Math.random() * 5); // Consumiendo entre 1 y 5 unidades
22             int[] values = buff.get(quantity);
23             System.out.println("Consumidor" + this.number + " got: " + Arrays.toString(values));
24             totalConsumed += values.length;
25         }
26     }
27 }
28 }
```

En consumer se hizo algo similar, por ejemplo el while se dejo en menor a 50 e igual se consume entre 1 y 5 unidades con ayuda de random y se guarda en la variable cantidad, después obtenemos del buffer la cantidad entre 1 y 5 y se agrega a valores, y se imprime en pantalla los valores que se obtuvieron con el consumidor, también se suman los valores consumidos en la variable totalconsumed.

```

5  class Buffer {
6
7      private static final int MAX_SIZE = 50;
8      private int[] contents = new int[MAX_SIZE];
9      private int currentSize = 0; // Tamaño actual del buffer
10     private boolean available = false;
11
12     public synchronized int[] get(int n) {
13         while (currentSize < n) {
14             try {
15                 wait();
16             } catch (InterruptedException e) { }
17         }
18         int[] retrieved = new int[n];
19         for (int i = 0; i < n; i++) {
20             retrieved[i] = contents[--currentSize];
21         }
22         available = currentSize > 0;
23         notifyAll();
24         return retrieved;
25     }
26
27
28     public synchronized void put(int[] values) {
29         while (currentSize + values.length > MAX_SIZE) {
30             try {
31                 wait();
32             } catch (InterruptedException e) { }
33         }
34         for (int value : values) {
35             contents[currentSize++] = value;
36         }
37         available = true;
38         notifyAll();
39     }
40
41 }

```

Aquí es donde hubo mas modificaciones, por ejemplo se sigue usando contents, pero se cambio y creo una nueva variable llamada maxsize que es la que se encarga de tener el valor del total completo, que en este caso es 50, y con base en ese valor se crea contents con ese tamaño, también se inicia currentsize en 0 y available en false, ya que no se han creado o producido unidades.

También los métodos get y put se cambiaron para poder trabajar con arreglos, por ejemplo con base en tamaño actual si es menor a n que es la cantidad que quiere obtener, lo manda a esperar ya que no hay nada disponible, también en retrieved se crea un nuevo arreglo de enteros para poder saber que valores se dieron, por ejemplo si esta disponible la cantidad entramos al ciclo for y de contents, que es el total se resta el currentSize y se guarda en retrieved en su índice de i, y saliendo del ciclo for si current size es mayor a 0 regresa true y se guarda en available, lo que implica que hay todavía recursos para consumir y se notifica a todos y se retornan retrieved que son los valores que se usaron.

En el método put se hace algo similar, pero se reciben la cantidad de valores que se van a poner de producción, también lo que sucede es que mientras el currentsize mas el valor que se recibe sea mayor a el tamaño máximo de 50 no se va a poder producir y se manda a esperar a que haya espacio suficiente para poder producirla, si es que hay espacio suficiente, se agregan los valores al arreglo contents para que puedan ser consumido y se pone como available en true, ya que hay disponibles y se notifica a todos.

```
Productor #1 put: [0]
Productor #1 put: [1, 2, 3, 4, 5]
Consumidor1 got: [5, 4, 3, 2, 1]
Productor #1 put: [6, 7, 8]
Consumidor1 got: [8, 7]
Consumidor1 got: [6]
Consumidor1 got: [0]
Productor #1 put: [9, 10, 11, 12]
Consumidor1 got: [12]
Consumidor1 got: [11, 10, 9]
Productor #1 put: [13, 14, 15, 16, 17]
Consumidor1 got: [17]
Consumidor1 got: [16, 15, 14, 13]
Productor #1 put: [18, 19, 20, 21, 22]
Consumidor1 got: [22, 21, 20]
Consumidor1 got: [19, 18]
Productor #1 put: [23, 24, 25, 26, 27]
Consumidor1 got: [27]
Consumidor1 got: [26, 25]
Consumidor1 got: [24, 23]
Productor #1 put: [28]
Consumidor1 got: [28]
Productor #1 put: [29]
Productor #1 put: [30, 31, 32]
Consumidor1 got: [32, 31, 30]
Productor #1 put: [33, 34]
Consumidor1 got: [34, 33, 29]
Consumidor1 got: [38, 37]
Productor #1 put: [35, 36, 37, 38]
Consumidor1 got: [36, 35]
Productor #1 put: [39]
Productor #1 put: [40, 41, 42, 43, 44]
Consumidor1 got: [44, 43, 42, 41]
Productor #1 put: [45, 46]
Consumidor1 got: [51, 50, 49, 48, 47]
Productor #1 put: [47, 48, 49, 50, 51]
Consumidor1 got: [46, 45, 40]
```

Análisis y Complicaciones:

Manejo de Arreglos: Al pasar de un único valor a un arreglo, nos enfrentamos al desafío de gestionar el índice y garantizar que no superáramos el tamaño máximo del arreglo. Esta transición añadió una capa adicional de complejidad al programa, ya que ahora debemos manejar múltiples elementos en el buffer en lugar de uno solo.

Producción y Consumo Variable: La introducción de una producción y consumo variables en el sistema supuso un reto. No solo se trataba de producir y consumir, sino de decidir cuánto producir y consumir en cada iteración. Esto requiere una lógica adicional para garantizar que el productor no añada más unidades de las permitidas y que el consumidor no intente tomar más unidades de las disponibles.

Gestión del Bucle y Limitación de Unidades: El bucle principal de producción y consumo tenía que considerar el total de unidades procesadas para garantizar que no se excediera el límite de 50 unidades. Esta lógica tuvo que ser robusta para evitar la producción o el consumo excesivo, especialmente dado el comportamiento variable de producción y consumo.

Errores de Compilación y Dependencias Faltantes: La falta inicial de importación de la clase Arrays llevó a errores de compilación. Estos errores subrayan la importancia de revisar las dependencias y asegurarse de que todas las clases y métodos necesarios estén correctamente referenciados.

Resolución:

Para manejar el buffer con múltiples unidades, introdujimos arreglos y utilizamos una variable `currentSize` para rastrear el número de elementos en el buffer en tiempo real.

A través de lógicas condicionales y bucles, controlamos la cantidad de producción y consumo en cada iteración, garantizando que no se excedieran los límites establecidos.

Finalmente, corregimos los errores de compilación importando las clases necesarias y asegurándonos de que el programa se ejecutara sin problemas.

SECCIÓN 2.- MONITORES Y SEMAFOROS

2.1.- Investiga cómo funcionan los monitores en Java y que relación hay con el bloque `synchronized`

Los monitores son mecanismos de sincronización que permiten a los hilos cooperar de manera segura. En Java, cada objeto tiene un monitor asociado, y un hilo puede adquirir el monitor de un objeto usando el bloque `synchronized`.

Bloque `synchronized`:

Para que un hilo acceda a un bloque `synchronized`, primero debe obtener el monitor del objeto. Si otro hilo ya tiene el monitor, el hilo entrante se bloqueará hasta que el hilo original salga del bloque `synchronized`, liberando el monitor.

Ejemplo:

```
public class Ejemplo {  
    private Object recursoCompartido;  
  
    public void metodoSincronizado() {  
        synchronized(this) {  
            // Operaciones sobre el recursoCompartido  
        }  
    }  
}
```

Aquí, el bloque `synchronized` adquiere el monitor del objeto de la clase `Ejemplo`. Si otro hilo intenta acceder al método `metodoSincronizado` mientras un hilo está dentro del bloque, deberá esperar.

2.2.- Investiga la clase Semaphore de java y como se relaciona con las operaciones que vimos en clase y cómo se puede utilizar

La clase `Semaphore` en Java es parte del paquete `java.util.concurrent`. Es un contador que te permite controlar el acceso a uno o más recursos. Funciona sobre la base de permisos; un hilo puede adquirir un permiso (si está disponible) o liberar un permiso.

Operaciones principales:

- `acquire()`: Un hilo intenta adquirir un permiso. Si no hay permisos disponibles, el hilo se bloquea hasta que pueda obtener uno o se interrumpe.
- `release()`: Libera un permiso, aumentando el número de permisos disponibles.

Supongamos que estás desarrollando una nueva característica para "formulae pro" en la que necesitas gestionar el acceso a un recurso compartido, como una conexión a una base de datos, y quieres permitir solo 3 conexiones simultáneas. Podrías utilizar `Semaphore` de la siguiente manera:

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

public class GestorConexiones {

    private final Semaphore semaforo = new Semaphore(3); // 3 permisos

    public void obtenerConexion() throws InterruptedException {

        semaforo.acquire();

        // Conexión obtenida

    }

    public void liberarConexion() {

        semaforo.release();

        // Conexión liberada

    }

}
```

Este enfoque garantiza que solo tres hilos pueden obtener una conexión al mismo tiempo. Otros hilos deberán esperar hasta que una conexión se libere.

2.-3 Para cada uno de los programas que se proporcionan (monitores y semáforos) realiza lo siguiente

a) Ejecuta el programa e incluye evidencias de su ejecución

corriendo semaphore demo.java la salida es la siguiente

```
5
10
15
20
25
100
200
300
400
500
```

Al correr semaphoreTest.java

```
Total available Semaphore permits : 4
C : acquiring lock...
B : acquiring lock...
D : acquiring lock...
E : acquiring lock...
A : acquiring lock...
F : acquiring lock...
D : available Semaphore permits now: 4
B : available Semaphore permits now: 4
F : available Semaphore permits now: 4
F : got the permit!
E : available Semaphore permits now: 4
E : got the permit!
D : got the permit!
B : got the permit!
A : available Semaphore permits now: 4
C : available Semaphore permits now: 4
F : is performing operation 1, available Semaphore permits : 1
B : is performing operation 1, available Semaphore permits : 0
E : is performing operation 1, available Semaphore permits : 0
D : is performing operation 1, available Semaphore permits : 0
D : is performing operation 2, available Semaphore permits : 0
F : is performing operation 2, available Semaphore permits : 0
B : is performing operation 2, available Semaphore permits : 0
E : is performing operation 2, available Semaphore permits : 0
D : is performing operation 3, available Semaphore permits : 0
B : is performing operation 3, available Semaphore permits : 0
F : is performing operation 3, available Semaphore permits : 0
E : is performing operation 3, available Semaphore permits : 0
D : is performing operation 4, available Semaphore permits : 0
B : is performing operation 4, available Semaphore permits : 0
F : is performing operation 4, available Semaphore permits : 0
E : is performing operation 4, available Semaphore permits : 0
D : is performing operation 5, available Semaphore permits : 0
B : is performing operation 5, available Semaphore permits : 0
F : is performing operation 5, available Semaphore permits : 0
E : is performing operation 5, available Semaphore permits : 0
D : releasing lock...
C : got the permit!
D : available Semaphore permits now: 1
C : is performing operation 1, available Semaphore permits : 0
F : releasing lock...
B : releasing lock...
F : available Semaphore permits now: 1
A : got the permit!
B : available Semaphore permits now: 1
A : is performing operation 1, available Semaphore permits : 1
E : releasing lock...
E : available Semaphore permits now: 2
C : is performing operation 2, available Semaphore permits : 2
A : is performing operation 2, available Semaphore permits : 2
C : is performing operation 3, available Semaphore permits : 2
A : is performing operation 3, available Semaphore permits : 2
A : is performing operation 4, available Semaphore permits : 2
C : is performing operation 4, available Semaphore permits : 2
A : is performing operation 5, available Semaphore permits : 2
C : is performing operation 5, available Semaphore permits : 2
C : releasing lock...
C : available Semaphore permits now: 3
A : releasing lock...
A : available Semaphore permits now: 4
```

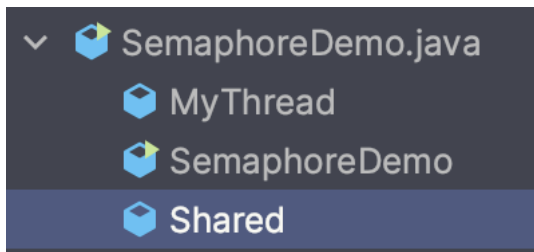
al correr el programa monitor1 esta es la salida

```
Contador decrementado: 1
Contador incrementado: 2
Contador decrementado: 1
Contador incrementado: 2
Contador decrementado: 1
Contador decrementado: 1
Contador incrementado: 2
Contador decrementado: 1
Contador incrementado: 2
Contador decrementado: 1
Contador decrementado: 1
Contador decrementado: 1
Contador incrementado: 2
Contador decrementado: 1
Contador decrementado: 1
Contador incrementado: 2
Contador decrementado: 1
Contador decrementado: 1
Contador incrementado: 2
```

al correr monitor 2 se ve lo siguiente y no para nunca.

```
Starting A
A is waiting for a permit.
A gets a permit.
Starting B
B is waiting for a permit.
A: 1
A: 2
A: 3
A: 4
A: 5
A releases the permit.
B gets a permit.
B: 4
B: 3
B: 2
B: 1
B: 0
B releases the permit.
count: 0
```

SemaphoneDemo.java tiene lo siguiente, todo esta dentro del mismo archivo.



b) Explica lo que ocurre y la solución de sincronización implementada

Para Monitor1.java

En el archivo Monitor1.java se ve la condición de carrera.

Contiene printTable que imprime la tabla de multiplicar de un numero dado.

En el método printtable tomamos un numero como argumento y se imprimen 5 múltiplos de ese numero, en cada impresión se ve un retarde de 400 milisegundos. Se usa synchronized antes de printTable para asegurar que solo un hilo a la vez pueda acceder al método.

La clase monitor1 crea un objeto table y dos hilos que ambos trabajan con el mismo objeto table y se inicializan ambos con start.

Si se quitara la palabra synchronized es posible que los hilos se mezclen mientras se imprimen los valores, pero con synchronized se ve que un hilo completa su ejecución y después el otro hilo sigue, primero múltiplos de 5 y después múltiplos de 100

Se asegura que un solo hilo a la vez ejecute el método printTable sobre el objeto Table

Para Monitor2.java

En la clase MonitorExample contiene count y tres métodos sincronizados para incrementar, decrementar y obtener count.

El método increment() aumenta el valor de count en uno. Sin embargo, si count ya es 2, el método espera (wait()) hasta que count ya no sea 2.

De manera similar, el método decrement() disminuye el valor de count en uno. Si count es 0, el método espera hasta que count ya no sea 0.

El método getCount() simplemente devuelve el valor actual de count.

Clase Incrementer:

Implementa la interfaz Runnable y tiene una referencia al monitor MonitorExample.

En su método run(), incrementa el contador dos veces y luego imprime el valor de count. Esto se repite indefinidamente con un retardo de 100 milisegundos entre cada repetición.

Clase Decrementer:

Similar a Incrementer, pero decrementa el contador una vez en su método run(). La impresión se repite con un retardo de 500 milisegundos.

Clase Monitor2 (clase principal):

Crea una instancia del monitor MonitorExample.

Inicia dos hilos: uno para incrementar y otro para decrementar el contador.

El hilo incrementerThread intentará incrementar el contador dos veces en rápida sucesión. Sin embargo, si el contador alcanza el valor 2, se detendrá y esperará hasta que el hilo decrementerThread lo reduzca.

El hilo decrementerThread disminuye el contador una vez, pero si el contador ya es 0, esperará hasta que el hilo incrementerThread lo incremente.

Este programa es un excelente ejemplo de cómo los monitores pueden ser usados para controlar el acceso concurrente a recursos compartidos y mantener ciertas invariantes (en este caso, que el contador nunca sea menor que 0 ni mayor que 2).

Para SemaphoreDemo.java

Utiliza semáforos para sincronizar el acceso de múltiples hilos a un recurso compartido.

Clase Shared:

Contiene una variable estática count inicializada a 0. Esta variable será el recurso compartido que los hilos intentarán modificar.

Clase MyThread:

Extiende Thread y se le pasa un semáforo y el nombre del hilo en el constructor.

En el método run():

Si el nombre del hilo es "A", se adquiere un permiso del semáforo y se incrementa la variable count 5 veces.

Si el nombre del hilo es diferente de "A" (en este caso "B"), se adquiere un permiso del semáforo y se decrementa la variable count 5 veces.

Después de modificar el count, se libera el permiso del semáforo.

Clase SemaphoreDemo (clase principal):

Se crea un semáforo con un solo permiso.

Se crean dos hilos: uno con el nombre "A" y el otro con el nombre "B".

Se inician ambos hilos y luego se espera a que ambos finalicen.

Al final, se imprime el valor de Shared.count.

Al principio, el contador es 0.

El hilo "A" intentará adquirir un permiso del semáforo. Si lo consigue, incrementará el contador 5 veces. Cada vez que lo hace, duerme por 10 milisegundos para permitir un cambio de contexto.

De manera similar, el hilo "B" intentará adquirir un permiso del semáforo. Si lo consigue, decrementará el contador 5 veces.

Dado que solo hay un permiso disponible en el semáforo, solo uno de los hilos puede modificar el contador a la vez. Esto asegura que las operaciones de incremento y decremento no se entremezclen.

Al final del programa, el valor de Shared.count siempre será 0, porque un hilo incrementó el contador 5 veces y el otro lo decrementó 5 veces.

ParaSemaphoreTest.java

También usa semáforos para su sincronización

Clase SemaphoreTest:

Declara un semáforo estático con 4 permisos, lo que significa que hasta 4 hilos pueden adquirir el semáforo simultáneamente.

Clase interna PruebaHilos:

Extiende Thread y tiene una variable name que representa el nombre del hilo.

En el método run(), el hilo intenta adquirir un permiso del semáforo.

Si lo consigue, imprime algunas declaraciones y duerme durante 1 segundo. Esto se repite 5 veces.

Después de terminar sus operaciones, libera el permiso del semáforo.

Método main de la clase SemaphoreTest:

Inicialmente, imprime el número total de permisos disponibles en el semáforo.

Crea e inicia 6 instancias de PruebaHilos con nombres diferentes.

Al principio, el semáforo tiene 4 permisos disponibles.

Se inician 6 hilos. Los primeros 4 hilos que intenten adquirir un permiso del semáforo lo conseguirán y ejecutarán su código.

Los otros 2 hilos tendrán que esperar hasta que al menos uno de los primeros 4 hilos libere un permiso.

Cada hilo que adquiere un permiso duerme un total de 5 segundos (1 segundo por cada iteración).

Después de que un hilo libera un permiso, otro hilo en espera puede adquirirlo.

Al final, todos los hilos habrán adquirido el semáforo, ejecutado su código y liberado el semáforo.

c) Modifica algunas líneas del programa para entender mejor su funcionamiento y explica las modificaciones realizadas

```

public class SemaphoreDemo
{
    Run | Debug
    public static void main(String args[]) throws InterruptedException
    {
        // creating a Semaphore object
        // with number of permits 1
        Semaphore sem = new Semaphore(permits:2);
        // creating two threads with name A and B
    }
}

```

Una modificación que se hizo fue la siguiente que ahora en lugar de tener un permiso, tiene 2, lo que implica que se permite ahora que 2 hilos modifiquen el recurso al mismo tiempo.

Esto significa que los hilos "A" y "B" podrían incrementar y decrementar el contador al mismo tiempo, lo que podría resultar en un entrelazado de sus operaciones. Sin embargo, el semáforo garantizará que no más de dos hilos accedan al recurso compartido al mismo tiempo.

```

// sleep
Thread.sleep(millis:500);

```

You, hace 16 segundos | 1 author (You)

```

public class SemaphoreTest {

    // max 4 people
    static Semaphore semaphore = new Semaphore(permits:3);
}

```

Haciendo estos cambios se hacen menos los permisos, de ser 4 ahora son 3, y se reduce el tiempo de sleep de los hilos de 1000 a 500

Al reducir los permisos a 3, solo tres hilos pueden adquirir el semáforo simultáneamente, dejando a los otros tres hilos esperando. Al ajustar el tiempo de sueño, los hilos liberarán el semáforo más rápidamente, permitiendo a otros hilos adquirirlo en un intervalo más

corto. Esto resultará en una ejecución más rápida y en un entrelazado más frecuente de las operaciones de los hilos.

d) Explica en qué solución real se podría aplicar ese programa

Dependiendo de los programas, por ejemplo en

Table y Monitor1 (Control de Acceso con synchronized):

En este programa, se enfrenta un problema que puede compararse a cuando diferentes transacciones intentan escribir en una base de datos única. Para garantizar la integridad de los datos, se ha implementado un mecanismo que asegura que solo una transacción pueda escribir en la base de datos en un momento dado. Es similar a cuando se quiere evitar que dos personas modifiquen un documento al mismo tiempo.

Monitor2 (Uso de wait y notifyAll):

Este código simula el control de tráfico de trenes en un tramo único. Si un tren está ocupando un tramo, automáticamente, otros trenes deben esperar hasta que esté libre. Se ha configurado el sistema para asegurar que el tramo solo pueda ser ocupado por un número determinado de trenes simultáneamente. Podría compararse a la gestión de carriles en una autopista durante horas pico.

semaphoreDemo.java (Semáforos con permisos únicos):

Aquí, se aborda una situación que podría encontrarse en sistemas donde se requiere acceso exclusivo a un recurso. Podría ser, por ejemplo, un archivo específico en un servidor que solo puede ser accedido por un usuario a la vez. Otro escenario sería una operación de actualización que necesita ser llevada a cabo sin interferencias.

semaphoreTest.java (Semáforos con múltiples permisos):

Este programa representa un escenario donde hay un número limitado de recursos disponibles para ser utilizados simultáneamente. Un buen ejemplo sería una aplicación de carpooling con un número fijo de autos disponibles. Cuando un cliente obtiene un auto, lo utiliza y, una vez finalizado, lo devuelve para que otros clientes lo utilicen. Otro ejemplo práctico sería un servidor que soporta un número limitado de conexiones simultáneas. Cuando todas las conexiones están ocupadas, cualquier cliente adicional que intente conectarse debe esperar.

CONCLUSIONES

Tras realizar esta práctica, me ha quedado claro la importancia de la sincronización en la programación concurrente. Al enfrentarme a los ejercicios, pude observar de manera tangible cómo los hilos pueden interferir entre sí al acceder a recursos compartidos, lo que reafirmó mi comprensión de la teoría detrás de las llamadas al sistema y la sincronización. La aplicación directa de conceptos teóricos, como el uso de `synchronized`, `wait`, `notifyAll` y semáforos, en los problemas prácticos fue particularmente ilustrativa. Me pareció desafiante al principio, especialmente cuando se trataba de identificar los problemas de concurrencia y pensar en las soluciones adecuadas. Sin embargo, a medida que avanzaba, empecé a apreciar la elegancia y eficacia de estas herramientas. Estoy satisfecho con lo que he aprendido y, aunque hubo momentos en los que me sentí desafiado, me siento motivado para seguir explorando y dominando estos conceptos en futuras prácticas y aplicaciones.