Taller 3 – Manejo de Semáforos

El propósito de este taller es entender la forma de utilizar semáforos para sincronizar secciones críticas en Java y aplicar los conceptos vistos en clase acerca de sincronización de threads sin objetos auxiliares.

Parte 1: Semáforos

Para desarrollar 1: Implementar un semáforo en Java

Implemente una clase llamada **`Semaforo`**. En esta clase deberá tener un contador como atributo de tipo entero, el cual será inicializado en el método constructor. Además, esta clase debe incluir los métodos públicos sincronizados **`p()`** y **`v()`**. Tome como referencia las diapositivas usadas en clase por el profesor.

1. Clase **`Semaforo`**. Atributos.

2. Clase **`Semaforo`**. Método constructor.

3. Clase **`Semaforo`**. Método **`p()`**.

4. Clase **`Semaforo`**. Método **`v()`**.

Parte 2: Exclusión mutua

El objetivo de esta parte es entender cómo funcionan los semáforos como herramienta para tratar controlar el acceso concurrente a una sección crítica.

a. Semáforo binario (mutex)

Un semáforo binario se utiliza para implementar la exclusión mutua. La exclusión mutua consiste en evitar que más de un thread ejecute la sección crítica al mismo tiempo.

Utilizando la clase **`Semaforo`** creada en el punto anterior, implemente un algoritmo en el cual se tienen dos threads que modifican una variable compartida inicializada en 0:

(1) un thread **`A`** que ejecuta un método **`a()`**, y
(2) un thread **`B`** que ejecuta un método **`b()`**.

Se pretende que los métodos **`a()`** y **`b()`** no se puedan ejecutar al mismo tiempo.

Escriba los dos métodos no sincronizados: **`a()`** que **`incrementa en 1000`** el valor de la variable compartida, y **`b()`** que **`aumenta en 1`** el valor de la variable compartida.

En la ejecución, el thread **`A`** llama al método **`a()`** y el thread **`B`** llama al método **`b()`** un número determinado de veces.

Al finalizar la ejecución del método **`a()`** o el método **`b()`** según sea el caso, puede hacer una pausa aleatoria usando el siguiente código:

```
private void esperaAleatoria() {
	try {
		Thread.sleep((long) (Math.random() * 1000));
	} catch (InterruptedException e) {
		e.printStackTrace();
	}
}
```

Para desarrollar 2:

Sincronice la ejecución de **`a()`** y **`b()`** usando semáforos de manera que no se puedan ejecutar al mismo tiempo. Recuerde que el contador de un semáforo binario inicia en 1.

Vamos a desarrollar dos clases:

(1) La clase **`Mutex`** y
(2) la clase **`MutexThread`**

La clase **`MutexThread`** implementa los threads. En la clase **`MutexThread`** hay un atributo que define el tipo (el cual puede ser **`A`** y **`B`**).

La clase **`Mutex`** tendrá los métodos **`a()`** y **`b()`**, y puede ser utilizada como clase principal para lanzar la aplicación, o si lo desea, puede lanzar la aplicación desde una clase aparte.

En el método **`main ()`** puede crear los threads usando una de las dos formas siguientes:

extends Thread	implements Runnable
`MutexThread a = new MutexThread('A');` `MutexThread b = new MutexThread('B');`	`Thread a = new Thread(new MutexThread('A'));` `Thread b = new Thread(new MutexThread('B'));`

En la ejecución de este programa, puede inicializar primero el thread **a** y luego el thread **b**, o viceversa. También puede considerar que el orden entre los threads sea determinado al azar usando el siguiente código:

```
int orden = (int) (Math.random() * 100) % 2;

if (orden == 0) {
	System.out.println("Inicia a");
	a.start();
	b.start();
} else {
	System.out.println("Inicia b");
	b.start();
	a.start();
}
```

Clase **Mutex**.

1. Clase **Mutex**. Método **a()**.

2. Clase **Mutex**. Método **b()**.

3. Clase **Mutex**. Método **main()**.

Clase **MutexThread**.

Clase **MutexThread**. Atributos. Un atributo al momento de crear el thread, determina si es de tipo **A** o de tipo **B**.

4. Clase **MutexThread**. Método constructor.

5. Clase **`MutexThread`**. Método **`run()`**.

```
```

Nota: Para probar sus programas puede hacer que en las secciones críticas impriman el estado del programa; también puede usar el método **`sleep()`** para hacer que los threads se demoren en las secciones críticas (incrementando la probabilidad de que haya colisiones).

Parte 3: Sincronización sin objetos auxiliares

Ejemplo: Productor – Consumidor usando objetos auxiliares

Este ejemplo corresponde al patrón productor – consumidor. En este caso se dispone de dos threads, uno que genera datos, el productor, y otro que los procesa, el consumidor, y un buffer de tamaño limitado para almacenar datos de manera temporal. El productor almacena uno a uno los elementos. Cuando el buffer está lleno, debe esperar a que el consumidor retire algún mensaje y libere espacio de almacenamiento. El consumidor retira uno a uno los elementos. Cuando el buffer está vacío, debe esperar a que el productor almacene algún mensaje. Este programa se compone de cuatro clases: **`Productor.java`**, **`Consumidor.java`**, **`Buffer.java`** y **`Main.java`**.

`Productor.java`

```
public class Productor extends Thread {
	private Buffer buffer;
	private int mensajesPorAlmacenar;

	public Productor (int mensajes, Buffer buffer) {
		this.mensajesPorAlmacenar = mensajes;
		this.buffer = buffer;
	}

	public void run() {
		int m=0;
		while(mensajesPorAlmacenar>0) {
			// cada mensaje es un entero numerado desde 0.
			buffer.almacenar(m);
			mensajesPorAlmacenar--;
			m++;
		}
	}
}
```

`Consumidor.java`

```
public class Consumidor extends Thread {
	private Buffer buffer;
	private int mensajesPorRetirar;

	public Consumidor(int mensajes, Buffer buffer) {
		this.mensajesPorRetirar = mensajes;
		this.buffer = buffer;
	}
```

```
	public void run() {
		while (mensajesPorRetirar > 0) {
			Integer mensaje = buffer.retirar();
			System.out.println("Retirando del buffer el mensaje " + mensaje);
			mensajesPorRetirar--;
		}
	}
}
```

Buffer.java

```
import java.util.ArrayList;
public class Buffer {
	private ArrayList<Integer> buff;
	private int n;
	Object lleno;
	Object vacio;

	public Buffer(int n) {
		this.n = n;
		buff = new ArrayList<Integer>();
		lleno = new Object();
		vacio = new Object();
	}

	public void almacenar(Integer i) {
		synchronized (lleno) {
			while (buff.size() == n) {
				try {
					System.out.println("Buffer lleno");
					lleno.wait();
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		}
		synchronized (this) {
			buff.add(i);
		}
		synchronized (vacio) {
			vacio.notify();
		}
	}

	public Integer retirar() {
		synchronized (vacio) {
			while (buff.size() == 0) {
				try {
					System.out.println("Buffer vacio");
					vacio.wait();
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		}
		Integer i;
```

```
            synchronized (this) {
                  i = buff.remove(0);
            }
            synchronized (lleno) {
                  lleno.notify();
            }
            return i;
      }
}
```

Main.java

```
public class Main {
      public static void main(String[] args) {
            Buffer buffer = new Buffer(5);

            int mensajeParaRetirar = 20;
            int mensajesParaAlmacenar = 20;

            Productor p = new Productor(mensajesParaAlmacenar,  buffer);
            Consumidor c = new Consumidor(mensajeParaRetirar,  buffer);

            p.start();
            c.start();
      }
}
```

En este ejemplo, el buffer tiene una capacidad de almacenar 5 mensajes, y tanto el productor como el consumidor, tienen 20 elementos para almacenar y retirar, respectivamente.

Para desarrollar 3:

Retome el ejercicio del productor-consumidor. Pero, en esta ocasión, no use objetos auxiliares; haga uso del método **`wait ()`** y **`notify()`** directamente sobre el buffer mismo. Escriba el código de los métodos **`almacenar ()`** y **`retirar ()`** de la clase **`Buffer`**. Tome como referencia las diapositivas usadas en clase por el profesor.

`almacenar ( )`

```

```

`retirar ( )`

```

```

Para desarrollar en casa:

b. Semáforo multiple (multiplex)

Generalice la solución anterior para que permita que un número limitado de threads ejecuten la sección crítica al mismo tiempo. Se tiene un arreglo de threads con identificador. Cada thread ejecuta un método **`a()`**. Se pretende que los métodos **`a()`** que son llamados por cada thread se puedan ejecutar al mismo tiempo sin sobrepasar la cantidad determinada. Sincronice la ejecución de los threads usando semáforos. En este caso, el contador del semáforo tiene un valor inicial mayor que 1. El número de threads del arreglo debe ser mayor que el valor con el que se inicia el contador del semáforo.

Vamos a desarrollar dos clases: (1) La clase **`Multiplex`** y (2) la clase **`MultiplexThread`**. La clase **`MultiplexThread`** implementa los threads. La clase **`Multiplex`** tendrá el método **`a()`**, y puede ser utilizada como clase principal para lanzar la aplicación, o si lo desea, puede lanzar la aplicación desde una clase aparte.

Clase **`MultiplexThread`**

1. Clase **`MultiplexThread`**. Atributos.

2. Clase **`MultiplexThread`**. Método constructor.

3. Clase **`MultiplexThread`**. Método **`run()`**.

Clase **`Multiplex`**.

4. Clase **`Multiplex`**. Método **`a()`**.

5. Clase **`Multiplex`**. Método **`main()`**.