Taller 3 – Sincronización con Objetos – Semáforos

Parte 1: Implementación de semáforos

Para desarrollar: Implementar un semáforo en Java

Implemente una clase llamada **`Semaforo`**. A continuación, encuentra los atributos que debe usar. Además, esta clase debe incluir los métodos públicos sincronizados **`p()`** y **`v();`** en estos métodos tome como referencia las diapositivas usadas en clase por el profesor. Para lograr la sincronización utilice wait()/notify()

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

```
private int contador ;
private LinkedList <Object> colaEspera ;
```

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.

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:

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`**.

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

5. Clase **`MutexThread`**. Método constructor.

6. 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). Escriba un mensaje al principio y al final de cada método y asegúrese que no se mezclan los mensajes. Pruébelo sin sincronización para validar el resultado.

Parte 3: Manejo de Semáforos - Usos

El propósito de esta sección es entender la forma de utilizar semáforos para control de precedencia. Aunque no lo revisaremos en este ejemplo, tenga en cuenta que los semáforos también se pueden usar en señalización para encuentros (Rendez-vous) y señalización para control de barrera.

Para esta sección, utilice la siguiente implementación de semáforo:

```
public class Semaforo {
	private int contador;
	private LinkedList<Object> colaEspera;

	public Semaforo(int contador) {
		this.contador = contador;
		this.colaEspera = new LinkedList<Object>();
	}

	// Solicitar turno
	public void p() {
		while (contador <= 0) {
		try {
			Object o = new Object();
			colaEspera.add(o);
			synchronized(o) {
				System.out.println("hi"+colaEspera.size());
				o.wait();
			}
			} catch (InterruptedException e) {
			}
		}
		contador--;
	}

	// Liberar turno
	public void v() {
		//Vuelve a haber recurso y habia alguien en espera
		if (contador >= 0) {
			Object o = colaEspera.poll();
			synchronized (o) {
				contador++;
				o.notifyAll();
			}
		}
	}
}
```

Para asegurar que cada thread tenga una demora de ejecución, puede usar el siguiente método.

```
public static void esperaAleatoria(int tiempoMaximo, String mensaje) {
	long espera = (long) (Math.random() * tiempoMaximo);
	System.out.println("Esperando " + espera + " milisegundos " + mensaje);
	try {
		Thread.sleep(espera);
	} catch (InterruptedException e) {
		e.printStackTrace();
	}
}
```

Parte 3.1: Señalamiento para control de precedencia

El objetivo de esta parte es entender cómo funcionan los semáforos para controlar el orden de ejecución de tareas.

Suponga que se tienen dos threads:
(1) un thread **`A`** que ejecuta un método **`a()`**, y (2) un thread **`B`** que ejecuta un método **`b()`**.

Para desarrollar:

Vamos a desarrollar dos clases: (1) La clase **`Signaling`** y (2) la clase **`SignalingThread`**.

La clase **`SignalingThread`** implementa los threads. La clase **`Signaling`** 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.

Clase **`SignalingThread`**.

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

2. Clase **`SignalingThread`**. Método constructor.

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

Clase **`Signaling`**.

4. Clase **`Signaling`**. Variable compartida con su valor inicial.

5. Clase **`Signaling`**. Método **`a()`**.

6. Clase **`Signaling`**. Método **`b()`**.

7. Clase **`Signaling`**. Método **`main()`**.

Se quiere que el método **`a()`** se ejecute antes que el método **`b()`**. Para esto, escriba los dos métodos no sincronizados **`a()`** y **`b()`** que realicen alguna acción sobre una variable compartida, por ejemplo, que el método **`a()`** sume 100 y el método **`b()`** sume 10.

Inicialmente, ejecute el programa varias veces, sin usar sincronización, y observe los resultados.

A continuación, sincronice **a()** y **b()** usando semáforos para que **b()** no se pueda ejecutar a menos que **a()** ya se haya ejecutado. Note que, para poder hacer el control de precedencia, el semáforo se inicializará en 0.