Benemérita Universidad Autónoma del Estado de Puebla

Facultad de Cs. De la Computación

Programación Concurrente y Paralela

Práctica de Laboratorio No. 4

Profr:

María del Carmen Cerón Garnica

Alumno:

Roberto Alejandro Bravo Arredondo

Matricula:

200824268

20 de septiembre de 2011

Introducción

En esta práctica vamos a hacer uso de los Semáforos para realizar la exclusión mutua en la región critica, para esto hay que dar una breve introducción de lo que es un semáforo;

Un semáforo es una variable especial (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutarán varios procesos concurrentemente).

Java desde la versión 1.5 contiene la clase **java.util.concurrent.Semaphore** para el uso de Semáforos, o podemos crear nuestra propia clase Semáforos con los métodos WAIT() y SIGNAL() los cuales bloquean y liberan a un proceso respectivamente…

Objetivo

Hacer uso y entender el comportamiento de los Semáforos como herramienta para la programación concurrente de hilos en Java, también en esta práctica debemos notar la diferencia entre dos programas que realizan lo mismo, pero uno usando métodos de sincronización con primitivas de Java y el otro con semáforos.

Código

1. Explica la diferencia entre los dos códigos de semáforos

R= La diferencia es que un código utiliza las primitivas de Java para lograr la sincronización de los hilos, para ser exactos, utiliza el método sleep(), y el otro código hace uso de la clase **java.util.concurrent.Semaphore** para implementar los Semáforos de Java y con éstos logras la exclusión mutua de los hilos.

2. Explica los métodos de Java utilizados en el Código de Con semáforos.

R= Cada semáforo tiene estos 2 métodos: **acquire ()** bloquea si es necesario un hilo hasta que un permiso esté disponible. Y **release ()** añade un permiso, el cual puede desbloquear un acquire ().

3. ¿Por qué se utilizan variables protegidas y estáticas?

R= Porque las variables protegidas, que son los Semáforos, se usan así para que otras subclases de otros paquetes no puedan acceder, modificar su valor y causar un comportamiento inesperado, las estáticas se usan porque son variables “de clase” y así se podrán usar en el Main de Programa.

4. Especifica que recurso se comparte entre los semáforos

R= Se comparte el valor del semáforo, si el contador de éste es 0 puede acceder a la región critica, de lo contrario tendrá que esperar un release().

5. ¿Qué se utilizó para la sincronización y cómo se implementó?

R= Para la sincronización con primitivas de Java se utilizó solo el método sleep(), sin ninguna condición, solo el hilo dormía un tiempo aleatorio, luego imprimía el texto asignado y finalmente terminaba, esto es lo mismo para los 4 hilos que se crean.

6. Se presenta la exclusión mutua en este ejercicio, ¿explica de qué manera?

R= Si se presenta, ya que los hilos no pueden acceder al mismo recurso compartido al mismo tiempo, para esto se utilizó la sincronización por primitivas de Java y los Semáforos para realizar la exclusión mutua y que uno, y solo un hilo accediera a la región critica por vez.

Sin Semáforos:

```
public class p1 extends Thread
{
	public void run()
	{
		try
		{
			sleep((int) Math.round(500 * Math.random() - 0.5));
		}
		catch (InterruptedException e)
		{
			e.printStackTrace();
		}

		System.out.println("P1");
	}
}
public class p2 extends Thread
{
	public void run()
	{
		try
		{
			sleep((int) Math.round(500 * Math.random() - 0.5));
		}
		catch (InterruptedException e)
		{
			e.printStackTrace();
		}

		System.out.println("P2");
	}
}
public class p3 extends Thread
{
	public void run()
	{
```

```
                try
                {
                        sleep((int) Math.round(500 * Math.random() - 0.5));
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }

                System.out.println("P3");
        }
}
public class p4 extends Thread
{
        public void run()
        {
                try
                {
                        sleep((int) Math.round(500 * Math.random() - 0.5));
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }

                System.out.println("P4");
        }
}
public class SinSemaforos
{
        public static void main(String[] args)
        {
                (new Thread(new p1())).start();
                (new Thread(new p2())).start();
                (new Thread(new p3())).start();
                (new Thread(new p4())).start();
        }
}
```

Con Semáforos:

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

public class p1 extends Thread
{
        protected Semaphore oFinP1;
```

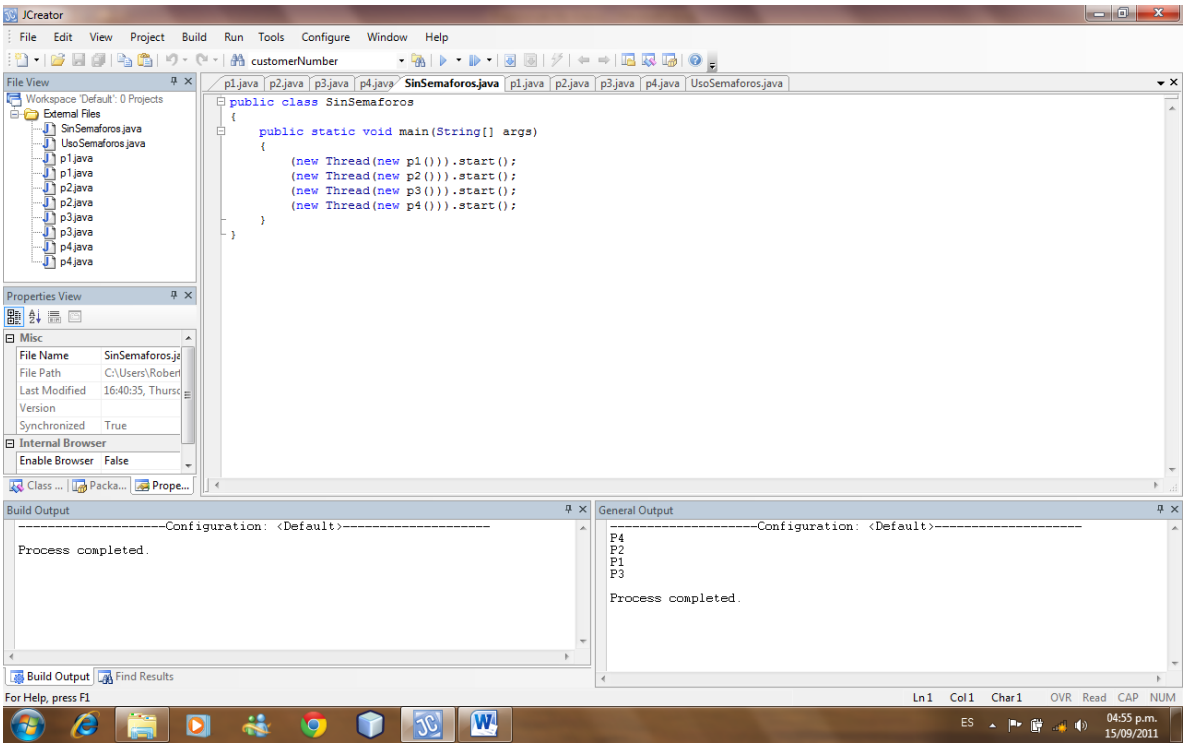
```
	public p1(Semaphore oFinP1)
	{
		this.oFinP1 = oFinP1;
	}

	public void run()
	{
		try
		{
			sleep((int) Math.round(500 * Math.random() - 0.5));
		}
		catch (InterruptedException e)
		{
			e.printStackTrace();
		}

		System.out.println("P1");
		this.oFinP1.release(2);
	}
}
import java.util.concurrent.Semaphore;

public class p2 extends Thread
{
	protected Semaphore oFinP1;
	protected Semaphore oFinP3;

	public p2(Semaphore oFinP1,Semaphore oFinP3)
	{
		this.oFinP3 = oFinP3;
		this.oFinP1 = oFinP1;
	}

	public void run()
	{
		try
		{
			this.oFinP1.acquire();
			this.oFinP3.acquire();
		}
		catch(Exception e)
		{
			e.printStackTrace();
		}

		try
```

```
                {
                        sleep((int) Math.round(500 * Math.random() - 0.5));
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }

                System.out.println("P2");
        }
}
import java.util.concurrent.Semaphore;

public class p3 extends Thread
{
        protected Semaphore oFinP3;

        public p3(Semaphore oFinP3)
        {
                this.oFinP3 = oFinP3;
        }

        public void run()
        {
                try
                {
                        sleep((int) Math.round(500 * Math.random() - 0.5));
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }

                System.out.println("P3");
                this.oFinP3.release(2);
        }
}
import java.util.concurrent.Semaphore;

public class p4 extends Thread
{
        protected Semaphore oFinP1;
        protected Semaphore oFinP3;

        public p4(Semaphore oFinP1,Semaphore oFinP3)
        {
                this.oFinP3 = oFinP3;
```

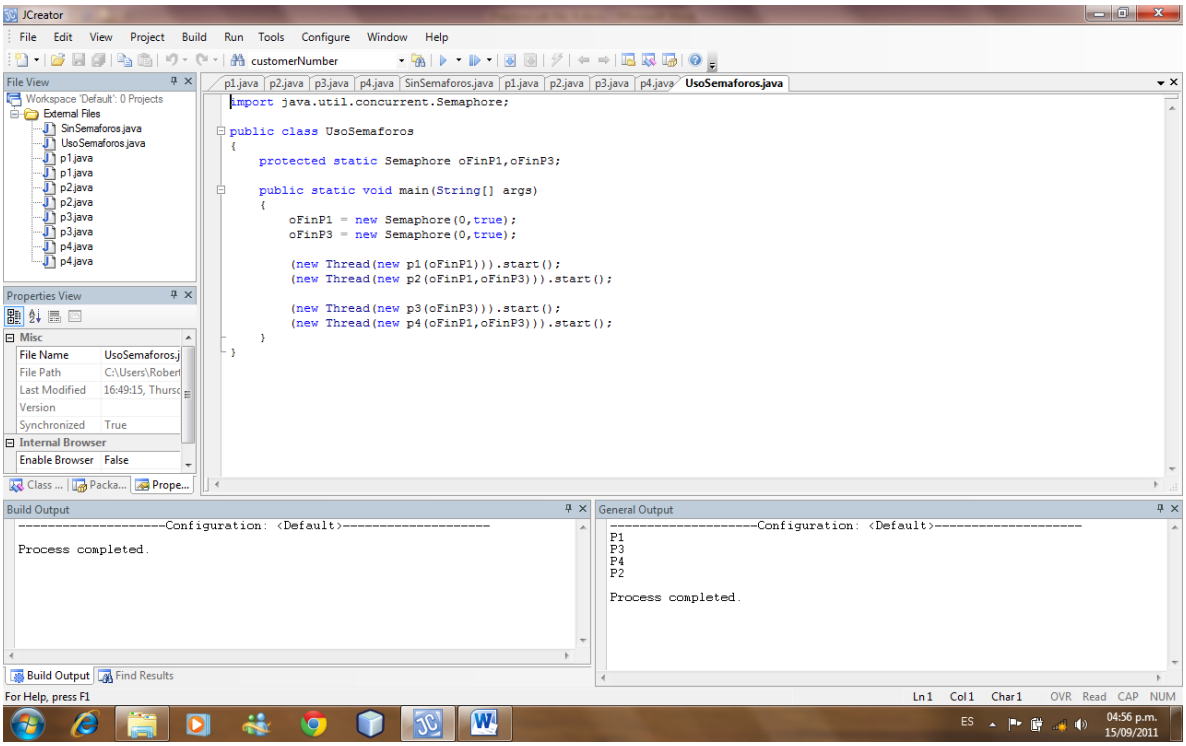
```
		this.oFinP1 = oFinP1;
	}

	public void run()
	{
		try
		{
			this.oFinP1.acquire();
			this.oFinP3.acquire();
		}
		catch(Exception e)
		{
			e.printStackTrace();
		}

		try
		{
			sleep((int) Math.round(500 * Math.random() - 0.5));
		}
		catch (InterruptedException e)
		{
			e.printStackTrace();
		}

		System.out.println("P4");
	}
}
import java.util.concurrent.Semaphore;

public class UsoSemaforos
{
	protected static Semaphore oFinP1,oFinP3;

	public static void main(String[] args)
	{
		oFinP1 = new Semaphore(0,true);
		oFinP3 = new Semaphore(0,true);

		(new Thread(new p1(oFinP1))).start();
		(new Thread(new p2(oFinP1,oFinP3))).start();

		(new Thread(new p3(oFinP3))).start();
		(new Thread(new p4(oFinP1,oFinP3))).start();
	}
}
```

Capturas de Pantalla

Sin Semáforos:



Con Semáforos:



Conclusión

Esta práctica principalmente me ayudo a entender los métodos de los Semáforos implementados en la clase **java.util.concurrent.Semaphore**, ya que los nombres son distintos a el tipo de Semáforos que vimos en clase, aunque en esencia realizan lo mismo, el acquire() es similar al wait() y el release() es muy parecido al signal().

También se puede notar la diferencia en programar utilizando Semáforos a comparación de hacerlo con primitivas de Java.