Trabajaremos con el paquete java.util.concurrent.Semaphore;

Algunos conecptos basicos sobre Semaforos:

Un semáforo es una variable especial protegida 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). Fueron inventados por Edsger Dijkstra .

Un semáforo dispone de un contador de tipo Integer y de un array de procesos bloqueados, si el contador del semáforo esta a 0 , los procesos que hagan acquire() en el , se dormiran en el array de procesos bloqueados, esperando a que otro proceso haga un release().

Cuando instanciamos un objeto de la clase Semaphore, podemos inicializar su contador, asi si lo hacemos a 2, los dos primeros procesos que hagan acquire() sobre el no se dormiran pero excrementaran el contador, el tercero consecutivamente que realice un acquire() sobre el contador, quedara dormido en el array de procesos bloqueados.

Un libro completo sobre semáforos lo podéis encontrar aquí: The Little Book of Semaphores

Un ejemplo de sincronización, es el mostrado en la siguiente figura:

Observamos como los Procesos P1y P3 precede al Proceso 2 y el 4 , es decir; los Procesos P2 y P4 no pueden arrancar hasta que P1 y P3 hayan terminado su seccion critica.

El código fuente sin semáforos de esta aplicación seria; 4 clases que lanzaremos como procesos, ejecutaran un sleep() simulando un procesamiento y luego imprimirá el número de proceso que es, identificando así la salida.

1. Creación de los procesos y la clase lanzadora:

public class Proceso1 extends Thread {

public void run() {

try {

sleep((int) Math.round(500 \* Math.random() - 0.5));

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("P1");

}

}

public class Proceso2 extends Thread {

public void run() {

try {

sleep((int) Math.round(500 \* Math.random() - 0.5));

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("P2");

}

}

public class Proceso3 extends Thread {

public void run() {

try {

sleep((int) Math.round(500 \* Math.random() - 0.5));

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("P3");

}

}

public class Proceso4 extends Thread {

public void run() {

try {

sleep((int) Math.round(500 \* Math.random() - 0.5));

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("P4");

}

}

public class go {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

//Throw concurrente proccess

(new Thread(new Proceso1())).start();

(new Thread(new Proceso2())).start();

(new Thread(new Proceso3())).start();

(new Thread(new Proceso4())).start();

}

}

Al ejecutar la salida obtendremos distintos resultados, porque cada vez, cada proceso tardara mas o menos en ejecutar.

Posibles salidas: {P1,P2,P3,P4},{P2,P4,P1,P3},{P4,P3,P1,P2},..... cualquier combinación es posible.

2. Sincronización de los procesos

/\*\*

\* @(#)Proceso1.java

\* http://mmengineer.blogspot.com/2007/10/java-semaphores-productor-consumidor.html

\* adaptado por Alvaro Pino N.

\* @author

\* @version 1.00 2015/11/23

\*/

import java.util.concurrent.Semaphore;

public class Proceso1 extends Thread{

protected Semaphore oFinP1;

public Proceso1(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);

}

}

/\*\*

\* @(#)Proceso2.java

\* http://mmengineer.blogspot.com/2007/10/java-semaphores-productor-consumidor.html

\* adaptado por Alvaro Pino N.

\*

\* @author

\* @version 1.00 2015/11/23

\*/

import java.util.concurrent.Semaphore;

public class Proceso2 extends Thread{

protected Semaphore oFinP1;

protected Semaphore oFinP3;

public Proceso2(Semaphore oFinP1,Semaphore oFinP3 ) {

this.oFinP3 = oFinP3;

this.oFinP1 = oFinP1;

}

public void run()

{

try{

this.oFinP1.acquire();

this.oFinP3.acquire();

}catch (InterruptedException e)

{ e.printStackTrace(); }

try{

sleep((int) Math.round( 500\* Math.random() - 0.5 ));

}catch (InterruptedException e)

{ e.printStackTrace(); }

System.out.println("P2");

}

}

/\*\*

\* @(#)Proceso3.java

\* http://mmengineer.blogspot.com/2007/10/java-semaphores-productor-consumidor.html

\* adaptado por Alvaro Pino N.

\*

\* @author

\* @version 1.00 2015/11/23

\*/

import java.util.concurrent.Semaphore;

public class Proceso3 extends Thread{

protected Semaphore oFinP3;

public Proceso3(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);

}

}

/\*\*

\* @(#)Proceso4.java

\* http://mmengineer.blogspot.com/2007/10/java-semaphores-productor-consumidor.html

\* adaptado por Alvaro Pino N.

\*

\* @author

\* @version 1.00 2015/11/23

\*/

import java.util.concurrent.Semaphore;

public class Proceso4 extends Thread{

protected Semaphore oFinP1;

protected Semaphore oFinP3;

public Proceso4(Semaphore oFinP1,Semaphore oFinP3 ) {

this.oFinP3 = oFinP3;

this.oFinP1 = oFinP1;

}

public void run()

{

try{

this.oFinP1.acquire();

this.oFinP3.acquire();

}catch (InterruptedException e)

{ e.printStackTrace(); }

try{

sleep((int) Math.round( 500\* Math.random() - 0.5 ));

}catch (InterruptedException e)

{ e.printStackTrace(); }

System.out.println("P4");

}

}

/\*\*

\* @(#)goTest.java

\* http://mmengineer.blogspot.com/2007/10/java-semaphores-productor-consumidor.html

\* adaptado por Alvaro Pino N.

\* @author

\* @version 1.00 2015/11/23

\*/

import java.util.concurrent.Semaphore;

public class goTest {

protected static Semaphore oFinP1,oFinP3;

/\*\*

\* Creates a new instance of <code>goTest</code>.

\*/

public goTest() {

}

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

oFinP1 = new Semaphore(0,true);

oFinP3 = new Semaphore(0,true);

( new Thread ( new Proceso1(oFinP1))).start();

( new Thread ( new Proceso2(oFinP1,oFinP3))).start();

( new Thread ( new Proceso3(oFinP3))).start();

( new Thread ( new Proceso4(oFinP1,oFinP3))).start();

}

}

Ahora las posibles salidas son : {P3,P1,P2,P4},{P1,P3,P4,P2},... como podemos observar se cumple perfectamente la sincronización requerida.