# Universidad Nacional de San Antonio Abad del Cusco
## Departamento Académico de Informática
## Programación Concurrente y Distribuida
## Práctica N° 4

# SEMÁFOROS
# Especificacion JSR 166 Concurrency Utilities.

Ing. Iván Medrano Valencia

## 1. OBJETIVO.

Conocer y utilizar las utilidades de concurrencia que proporciona el Java a través de la Especificación JSR 166.

## 2. BASE TEORICA COMPLEMENTARIA

Java es un lenguaje de programación multiproceso que facilita la programación con hilos, proporcionando compatibilidad integrada para los subprocesos. Las primitivas integradas sin embargo para la sincronización de bloques, Object.wait() y Object.notify() son insuficientes para muchas tareas de programación. Esto lleva a los desarrolladores a implementar sus propias librerías de sincronización de alto nivel, pero dada la dificultad de problemas de concurrencia, estas implementaciones pueden no ser correctas, eficientes y de alta calidad.

El Java 2 Platform, Standard Edition versión 5.0 (J2SE 5.0), que es también conocido como Tigre, ha proporcionado un nuevo camino para el subprocesamiento múltiple en el lenguaje de programación Java. Los originales mecanismos de coordinación de subprocesos con wait() y notify() son ahora mejorados con nuevos y sofisticados mecanismos para trabajar con subprocesos. Los nuevos mecanismos son parte del paquete java.util.concurrent, que tiene como objetivo ofrecer un conjunto estándar de utilidades de concurrencia que facilitará la tarea de desarrollar servidores y aplicaciones multiproceso. Además, dichas normas mejorarán la calidad de dichas aplicaciones. El paquete ha sido definido mediante el proceso de comunidad Java JSR 166: Utilidades de concurrencia.

Al escribir aplicaciones multiproceso, las situaciones que pueden crear dificultades están relacionadas con sincronización de datos. Estos producen errores que dificultan el diseño, y son difíciles de detectar. La sincronización integrada (de métodos y bloques) están muy bien para muchas aplicaciones basadas en el bloqueo, pero tienen sus limitaciones, tales como:

- De ningún modo se debe intentar nuevamente adquirir un bloqueo que ya se ha realizado, o se debe renunciar después de esperar un período de tiempo especificado o cancelar un bloqueo después de una interrupción.
- No hay control de acceso para la sincronización. Cualquier método puede realizar `synchronized(obj)` para cualquier objeto accesible. La sincronización se realiza en bloques, lo que limita el uso estricto del bloqueo estructurado y de métodos. En otras palabras, no puede adquirir un bloqueo en un método y lanzarlo en otro.

Tales problemas pueden superarse mediante clases de utilidad para control de bloqueo, como la exclusión mutua mediante semáforos.

```
public class Semaphore {
   public void acquire() throws InterruptedException { }
   public void release() { }
   public boolean attempt(long msec) throws
InterruptedException { }
}
```

Que pueden ser utilizados como:

```
....
Semaphore mutex;
.....
try {
  mutex.acquire();
  try {
    // do something
  } finally {
    mutex.release();
  }
} catch(InterruptedException ie) {
  // ...
}
```

## 3. EJERCICIO PRÁCTICO Nº 1

En una fábrica de bicicletas hay 3 operarios y 1 montador. El primer operario (OP1) construye ruedas, el segundo (OP2) construye marcos de bicicleta y el tercero (OP3) construye manillares. El montador se encarga de coger 2 ruedas, un marco y un manillar y monta la bicicleta. La actuación concurrente de estas tres personas está sujeta a las siguientes restricciones:

- El montador no podrá coger ningún material, si dicho material no ha sido fabricado de antemano por el operario correspondiente.
- El operario que fabrica marcos no tiene espacio más que para almacenar 4 de estos objetos; deberá esperar si en cualquier momento ha fabricado 4 marcos, sin que ninguno haya sido tomado por el montador.

- El número máximo de objetos que pueden almacenar los operarios que fabrican ruedas y manillares es de 10.

```
package appbicicletas;
import java.util.concurrent.Semaphore;
public class COperario1  extends Thread
{
  Semaphore hayEspRuedas;
  Semaphore hayRueda;

 public COperario1(Semaphore p_hayEspRuedas, Semaphore p_hayRueda)
  {
     hayEspRuedas = p_hayEspRuedas;
     hayRueda = p_hayRueda;
  }

  public void run ()
  {
    while (true)
    {
     try
     {
       hayEspRuedas.acquire();
       //--Hacer Rueda
       sleep(100);
       System.out.println("Se fabricó una rueda");
       hayRueda.release();
     }
     catch (InterruptedException e)
     {

     }
    }
  }
 }
```

```
package appbicicletas;
import java.util.concurrent.Semaphore;
public class COperario2 extends Thread
{ //--fabrica marcos para bicicletas
    Semaphore hayEspMar;
    Semaphore hayMarco;

    public COperario2 (Semaphore p_hayEspMar, Semaphore p_hayMarco)
    {
      hayEspMar = p_hayEspMar;
      hayMarco = p_hayMarco;
    }

   public void run()
   {
    while (true)
    {
     try
      {
        hayEspMar.acquire();
```

```
        //---fabricar marcos
         sleep(500);
         System.out.println("Se fabricó un marco...");
        hayMarco.release();
       }
      catch (InterruptedException e)
      {
      }
     }
    }
}
```

```
package appbicicletas;
import java.util.concurrent.Semaphore;
public class COperario3 extends Thread
{ //--fabrica marcos para bicicletas
    Semaphore hayEspMani;
    Semaphore hayManillar;

    public COperario3 (Semaphore p_hayEspMani, Semaphore p_hayManillar)
    {
      hayEspMani = p_hayEspMani;
      hayManillar = p_hayManillar;
    }

   public void run()
   {
    while (true)
    {
     try
      {
        hayEspMani.acquire();
        //---fabricar manillar
         sleep(400);
         System.out.println("Se fabricó un manillar");
        hayManillar.release();
      }
     catch (InterruptedException e)
     {
     }
    }
   }
}
```

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

public class CMontador extends Thread
{ //--ensambla bicicletas utilizando ruedas, marcos y manillasres
    Semaphore hayEspMar;
    Semaphore hayMarco;
    Semaphore hayEspRuedas;
    Semaphore hayRueda;
    Semaphore hayEspMani;
    Semaphore hayManillar;
```

```
  public CMontador(Semaphore p_hayEspRuedas, Semaphore p_hayRueda,
                   Semaphore p_hayEspMar,    Semaphore p_hayMarco,
                   Semaphore p_hayEspMani,   Semaphore p_hayManillar)
    {
      hayEspRuedas = p_hayEspRuedas;
      hayRueda  = p_hayRueda;
      hayEspMar = p_hayEspMar;
      hayMarco = p_hayMarco;
      hayEspMani = p_hayEspMani;
      hayManillar = p_hayManillar;
    }

   public void run()
   {
    while (true)
    {
     try
      {
        hayRueda.acquire();
        //--coger rueda
        System.out.println("Se tomó una rueda");
        sleep(50);
        hayEspRuedas.release();  //--libera un espacio para una rueda

        hayRueda.acquire();
        //--coger rueda
        System.out.println("Se tomó una rueda");
        sleep(50);
        hayEspRuedas.release();  //--libera un espacio para otra rueda

        hayMarco.acquire();
        //--coge un marco
        sleep(100);
        System.out.println("Se tomó un marco");
        hayEspMar.release(); //--libera un espacio para un marco

        hayManillar.acquire();
        //--coge un manillar
        System.out.println("Se tomó un manillar");
        sleep(60);
        hayEspMani.release(); //--libera un espacio para un manillar
        //--MONTAR LA BICICLETA
        sleep(200);

        System.out.println("SE ARMO UNA BICICLETA...");

      }
     catch (InterruptedException e)
     {
     }
    }
   }
}
```

```
package appbicicletas;
import java.util.concurrent.Semaphore;
public class Main {
 public static void main(String[] args)
 {
   Semaphore hayEspMar = new Semaphore(4);
   Semaphore hayMarco  = new Semaphore(0);
   Semaphore hayEspRuedas = new Semaphore(10);
   Semaphore hayRueda     = new Semaphore(0);
   Semaphore hayEspMani   = new Semaphore(10);
   Semaphore hayManillar  = new Semaphore(0);

  COperario1 op1 = new COperario1(hayEspRuedas, hayRueda);
  COperario2 op2 = new COperario2(hayEspMar, hayMarco);
  COperario3 op3 = new COperario3(hayEspMani, hayManillar);
  CMontador  mon = new CMontador(hayEspRuedas, hayRueda,
                                 hayEspMar, hayMarco,
                                 hayEspMani, hayManillar);

   //--iniciar los hilos
    op1.start();
    op2.start();
    op3.start();
    mon.start();
  }
}
```

## 4. EJERCICIO PRÁCTICO Nº 2

En el patio de una casa se encuentran cuatro niños que continuamente efectúan las siguientes tareas. Inicialmente se encuentran jugando. Cuando un niño tiene hambre recurre a un plato y toma una porción de queso del mismo. Acto seguido duerme durante cierto tiempo y a continuación vuelve a sus juegos. Este ciclo se repite indefinidamente.

Si cuando un niño intenta comer, encuentra el plato vacío, avisa a la madre de este suceso, esperando entonces a que la madre ponga comida en el plato. La madre, entre otras cosas, se encarga de realizar las siguientes dos tareas:

- Despertar a los niños que, después de comer, han dormido cierto tiempo (los niños son muy dormilones y no se despertarían de otra forma).
- Reponer el plato, cuando esté vacío, con cuatro porciones de queso, si es que algún niño solicita comida.

Implemente un programa utilizando semáforos en el que existan cuatro procesos "niño" y un proceso "madre", que actúen siguiendo el comportamiento anteriormente descrito. La solución propuesta debe estar libre de interbloqueo e inanición.

```
package ninosmadre;
public class Contador {
int numero;

public Contador(int numero)
```

```
    {
      this.numero=numero;
    }

    public void setNumero(int numero)
    {
       this.numero = numero;
    }

    public int getNumero()
    {
           return numero;
    }

 }



 package ninosmadre;
import java.util.concurrent.Semaphore;
public class HiloNinos extends Thread
{
    Semaphore mutex,esperar,dormir;
    int num;
    Contador nroRaciones;
   public HiloNinos(Semaphore pmutex, Contador raciones,
                    Semaphore pesperar, Semaphore pdormir)
    {
      mutex=pmutex;
      nroRaciones=raciones;
      esperar=pesperar;
      dormir=pdormir;
    }

    public void run()
    {
      while (true)
      {
          try
          {
            //--niños jugando
            Thread.sleep(100);
            //--niño tiene hambre
            mutex.acquire();
            while (nroRaciones.getNumero() <=0) //--no hay raciones
             {
                mutex.release();
                System.out.println("NIÑO ESPERANDO POR RACIONES");
                esperar.acquire();
                System.out.println("NIÑO DESPIERTA LUEGO DE ESPERAR");
                mutex.acquire();
             }

            //--si hay raciones
            System.out.println("niño comió una ración");
            num= nroRaciones.getNumero();
            num--;
```

```
            nroRaciones.setNumero(num);
            System.out.println("Nro Raciones: "+nroRaciones.getNumero());

            mutex.release();
            //--niño se va a dormir
            System.out.println("NIÑO SE VA A DORMIR");
            dormir.acquire();
          }

          catch (InterruptedException e)
          {

          }
      }

   }
}
```

```
package ninosmadre;
import java.util.concurrent.Semaphore;
public class HiloMadre extends Thread
{
  Semaphore mutex,espera,dormir;
  Contador nroRaciones;
  public HiloMadre(Semaphore pmutex, Contador raciones,
                   Semaphore pespera, Semaphore pdormir)
  {
    mutex=pmutex;
    nroRaciones=raciones;
    espera = pespera;
    dormir= pdormir;
   }

  public void run()
  {
    while (true)
    {
       System.out.println("MADRE VERIFICANDO SI HAY RACIONES");
       try
       {
         Thread.sleep(150);
         mutex.acquire();
          while (nroRaciones.getNumero()<=0)
          {
            System.out.println("REPONIENDO RACIONES");
            nroRaciones.setNumero(4);
            espera.release();
          }
         mutex.release();

         //--depierta si hay niños durmiendo

         dormir.release();
        }
      catch (InterruptedException e)
      {
```

```
      }

    }

  }
}
```

```
package ninosmadre;
import java.util.concurrent.Semaphore;
public class Main {

    public static void main(String[] args) {

     HiloNinos n1, n2,n3, n4;
     HiloMadre m1;

     Semaphore mutex,esperar,dormir;
     mutex = new Semaphore(1);
     esperar = new Semaphore(0);
     dormir = new Semaphore(0);
     Contador nroRaciones;
     nroRaciones= new Contador(4);
     n1 = new HiloNinos(mutex,nroRaciones,esperar,dormir);
     n2 = new HiloNinos(mutex,nroRaciones,esperar,dormir);
     n3 = new HiloNinos(mutex,nroRaciones,esperar,dormir);
     n4 = new HiloNinos(mutex,nroRaciones,esperar,dormir);
     m1 = new HiloMadre(mutex,nroRaciones,esperar,dormir);

     //--EJECUTAR 4 NIÑOS
     n1.start();
     n2.start();
     n3.start();
     n4.start();

       //--EJECUTAR LA MADRE
     m1.start();

   }
}
```

## 5. BIBLIOGRAFÍA


- Doug Lea "Programación Concurrente en Java" 2da. Edición. Addison Wesley 2001.

- http://java.sun.com/developer/technicalArticles/J2SE/concurrency/

- http://today.java.net/pub/a/today/2004/03/01/jsr166.html

- http://www.jcp.org/en/jsr/detail?id=166