PROGRAMACION CONCURRENTE

Sincronización basada en memoria compartida:

Sincronización y Semáforos

2020

Algunos términos clave relacionados con concurrencia

Atomic operation

- Las operaciones atómicas son aquellas operaciones que SIEMPRE se ejecutan hasta terminar todas. O todos ellos se ejecutan juntos, o ninguno de ellos se ejecuta. Si una operación es atómica, entonces no puede estar parcialmente completa, estará completa o no comenzará en absoluto, pero no estará incompleta.

Critical section

- Los recursos compartidos pueden conducir a un comportamiento inesperado o erróneo, por lo que las partes del programa donde se accede al recurso compartido deben protegerse de manera que se evite el acceso concurrente. Esta sección protegida es la **sección crítica** o **región crítica.** No puede ser ejecutado por más de un proceso/hilo a la vez.

Mutual exclusion

- Es una propiedad para el control de concurrencia, que se instituye con el propósito de prevenir las condiciones de carrera . Es el requisito de que un hilo de ejecución nunca entre en su sección crítica al mismo tiempo que otro hilo la este ejecutando.

Race condition

- Cuando el resultado de la ejecución de varios hilos concurrentes puede ser correcto o incorrecto dependiendo de cómo se entrelacen los *hilos*. Básicamente el problema aparece cuando varios *hilos* hacen ciclos de modificación de una variable compartida (lectura + modificación + escritura)

Algunos términos clave relacionados con concurrencia

Deadlock

- Un punto muerto es un estado en el que cada miembro de un grupo de procesos/hilos está esperando que otro miembro, incluido él mismo, tome medidas, como enviar un mensaje o, más comúnmente, liberar un bloqueo .
  - El punto muerto es un problema común en los sistemas de multiprocesamiento, la computación paralela y los sistemas distribuidos, donde se utilizan bloqueos de software y hardware para arbitrar recursos compartidos e implementar la sincronización de procesos .

Livelock

- Se producen cuando dos o más procesos repiten continuamente la misma interacción en respuesta a cambios en los otros procesos sin realizar ningún trabajo útil. Estos procesos no están en estado de espera y se ejecutan simultáneamente. Esto es diferente de un punto muerto porque en un punto muerto todos los procesos están en estado de espera.

Starvation

- Es un problema que se encuentra en computación concurrente donde un proceso se le niega perpetuamente recursos para procesar su trabajo.
  - La inanición puede ser causada por errores en un algoritmo de programación o exclusión mutua , pero también puede ser causada por fugas de recursos, y puede ser causada intencionalmente a través de un ataque de denegación de servicio

Procesos concurrentes y memoria compartid

- Si los diferentes hilos de un programa concurrente tienen acceso a variables globales o secciones de memoria comunes, la transferencia de datos a través de ella es una vía habitual de comunicación y sincronización entre ellos.

Ejemplo

```
1  package default_package;
2
3  public class Main {
4
5      /**
6       * @param args
7       */
8      public static void main(String[] args) {
9          System.out.println("Hello World!");
10
11         Variable variable = new Variable();
12
13         Process1 p1 = new Process1(variable);
14         Process2 p2 = new Process2(variable);
15
16         Thread t1 = new Thread(p1);
17         Thread t2 = new Thread(p2);
18
19         t1.start();
20         t2.start();
21
22         System.out.println("Bye World!");
23     }
24
25
26 }
```

Ejemplo

```java
package default_package;

public class Process1 implements Runnable {

    private Variable v;

    public Process1(Variable variable) {
        this.v = variable;
    }

    public void run() {
        while (true) {
            v.y1 = v.y2 + 1;

            while (!(v.y2 == 0 || v.y1 < v.y2)) {}

            v.inCritical ++;
            v.inCritical --;
            v.y1 = 0;

            System.out.print("Prcess 1 - inCritical    ");
            System.out.println(v.inCritical);

        }
    }
}
```

```java
package default_package;

public class Process2 implements Runnable {

    private Variable v;
    private String s;

    public Process2(Variable v) {
        this.v = v;
    }


    public void run() {
        while (true) {
            v.y2 = v.y1 + 1;

            while (!(v.y1 == 0 || v.y2 < v.y1)) {}

            v.inCritical ++;
            v.inCritical --;
            v.y2 = 0;
            System.out.print("Prcess 2 - inCritical    ");
            System.out.println(v.inCritical);
        }
    }

}
```

```java
package default_package;

public class Variable {

    int y1 = 0;
    int y2 = 0;
    int inCritical = 0;
}
```

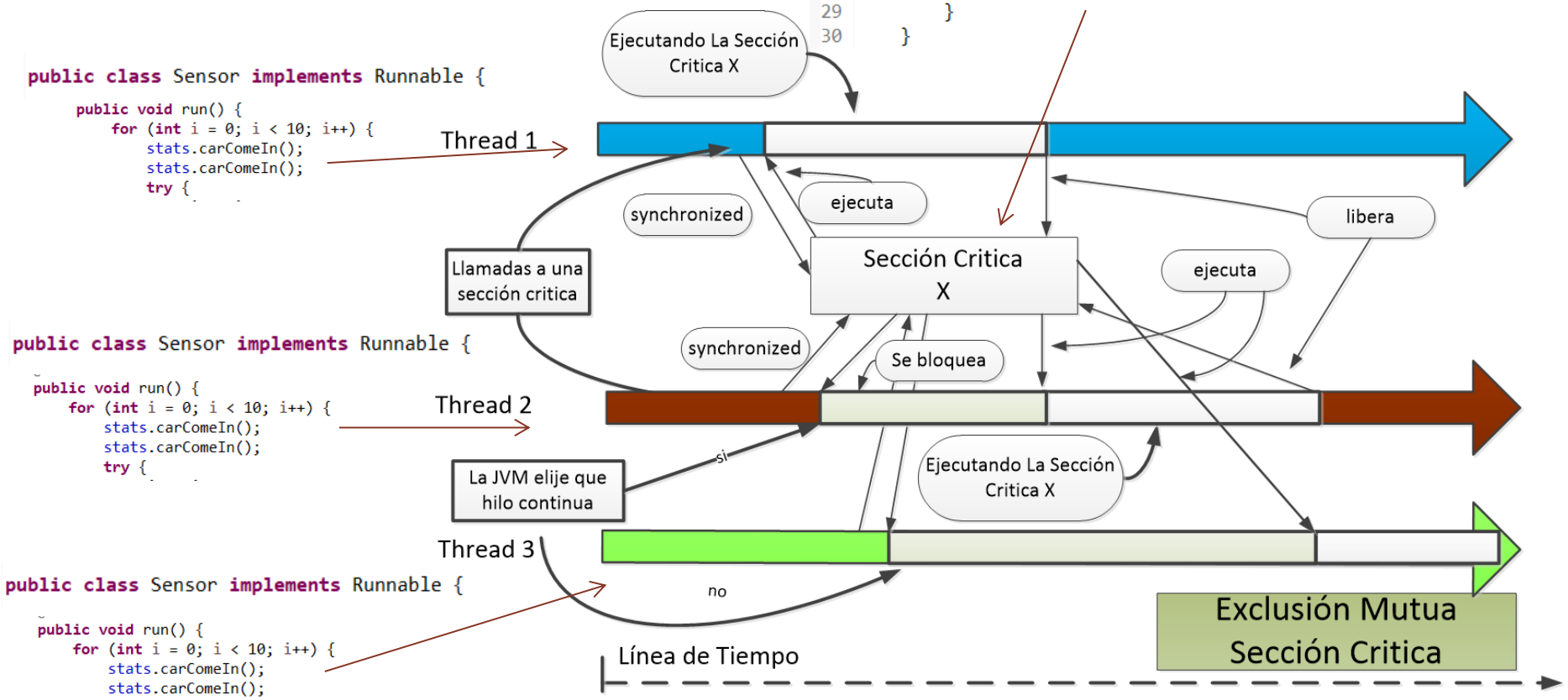
Procesos concurrentes y memoria compartid

- Las primitivas para programación concurrente basada en memoria compartida resuelven los problemas de:
  - sincronización entre procesos
  - exclusión mutua.
- **Secciones críticas**: Son mecanismos de nivel medio de abstracción orientados a su implementación en el contexto de un lenguaje y que permiten la ejecución de un bloque de sentencias de forma segura.
- **Semáforos:** Los semáforos tienen una variable que no es negativa y se comparte entre hilos, se utiliza para resolver el problema de la sección crítica y para lograr la sincronización del proceso en entornos de multiprocesamiento. Java proporciona la clase Semaphore en el paquete java.util.concurrent que implementa este mecanismo.
- **Monitores:** Son módulos de alto nivel de abstracción orientados a la gestión de recursos que van a ser usados concurrentemente. (otra clase)

Recursos de Java para sincronizar threads

- Todos los objetos tienen un bloqueo asociado, lock o cerrojo, que puede ser adquirido y liberado mediante el uso de métodos y sentencias **synchronized**.
- La sincronización fuerza a que la ejecución de los dos hilos sea mutuamente exclusiva en el tiempo.
- **Mecanismos de bloqueo**:
  - **Métodos synchronized** (exclusión mutua).
  - **Bloques synchronized** (regiones críticas).

```
3  public class ParkingStats {
   private final Object controlCars, controlMotorcycles;
26     public void carComeIn() {
27         synchronized (controlCars) {
28             numberCars++;
29         }
30     }
```



J. F. González, *Java 9 Concurrency Cookbook*: Packt Publishing Ltd, 2017. Capitulo 2 ejemplo 2

Bloques synchronized

- Es el mecanismo para implementar **regiones críticas** en Java las.
  - **Métodos synchronized** (exclusión mutua) o un
  - **Bloques synchronized** (regiones críticas).

- Synchronized puede ser con el uso del lock de un **objeto** (this o explicito otro objeto.
  - En ese caso solo se ejecuta si se obtiene el lock asociado al objeto

```
synchronized (this) {
    totalAmmount=cash;
    cash=0;
}
```

```
26    public void carComeIn() {
27        synchronized (controlCars) {
28            numberCars++;
29        }
30    }
```

- Se utiliza en un entorno concurrente y cuando el **objeto** es explicito, este es definido con el fin de usar su lock y preferentemente es **private**.
  - La sincronización fuerza a que la ejecución de los dos hilos sea mutuamente exclusiva en el tiempo.

Ejemplo de utilización de bloque synchronized

- Hacemos que totalAmmount y cash sean atómicas

```
16
17      public void close() {
18          System.out.printf("Closing accounting");
19          long totalAmmount;
20          synchronized (this) {
21              totalAmmount=cash;
22              cash=0;
23          }
24          System.out.printf("The total ammount is : %d",totalAmmount);
25      }
```

- Hacemos el método recoger sean atómico

```
 8      public synchronized char recoger(){
 9          while(!disponible){
10              try{
11                  wait();
12              }catch(InterruptedException ex){}
13          }
14          disponible=false;
15          notify();
16          return contenido;
17      }
```

Bloques synchronized

Métodos synchronized

Lock asociado a los objetos Java.

- Cada objeto derivado de la clase object (esto es, prácticamente todos) tienen asociado un elemento de sincronización o lock intrínseco, que afecta a la ejecución de los métodos definidos como synchronized en el objeto:
  - Cuando un objeto ejecuta un **método synchronized**, toma el lock, y cuando termina de ejecutarlo lo libera.
  - Cuando un **thread tiene tomado el lock**, ningún otro thread puede ejecutar ningún otro método synchronized por el mismo objeto.
  - El thread que **ejecuta un método synchronized** de un objeto cuyo lock se encuentra tomado, se suspende hasta que el objeto es liberado y se le concede el acceso.
- **Cada clase Java derivada de Object**
  - tiene también un mecanismo lock asociado a ella (que es independiente del asociado a los objetos de esa clase) y que afecta a los procedimientos estáticos declarados synchronized.

Métodos synchronized

- Los **métodos** de una clase Java se pueden declarar como **synchronized**. Esto significa que **se garantiza que se ejecutará con régimen de exclusión mutua** respecto de otro método del mismo objeto que también sea synchronized.

```
public static MyStaticCounter{

  private static int count = 0;

  public static synchronized void add(int value){
      count += value;
  }


}
```

- Si el método **synchronized es estático** (static), el lock al que hace referencia es **de clase y no de objeto**, por lo que se hace en exclusión mútua con cualquier otro método estático synchronized de la misma clase.

Consideraciones sobre métodos synchronized.

- El lock es tomado por el thread, por lo que mientras un thread tiene tomado el lock de un objeto puede acceder a otro método synchronized del mismo objeto.
- El lock es por cada instancia del objeto.
- Los métodos de clase (static) también pueden ser synchronized.
  - Por cada clase hay un lock y es relativo a todos los métodos synchronized de la clase.
  - Este lock no afecta a los accesos a los métodos synchronized de los objetos que son instancia de la clase.
- Cuando una clase se extiende y un método se sobreescribe, este se puede definir como synchronized o nó, con independencia de cómo era y como sigue siendo el método de la clase madre.

Recursos de Java para sincronizar threads

- Todos los objetos tienen un bloqueo asociado
  - los hilos pueden comunicarse entre sí a través de los métodos definidos en la clase **Object** y solo se pueden llamar desde un contexto **sincronizado**
  - Están definidos e implementados en la clase **Object :**
    - public final void **wait()** throws InterruptedException
      - Le dice al hilo de llamada que abandone el bloqueo y se vaya a dormir hasta que algún otro hilo ingrese al mismo monitor y llame notify().
      - libera el bloqueo antes de esperar y vuelve a adquirir el bloqueo antes de regresar de dormir.
      - está estrechamente integrado con el bloqueo de sincronización, utilizando como una función que no está disponible directamente desde el mecanismo de sincronización.
    - public final void **notify()**
      - Despierta un solo hilo que invocó wait()el mismo objeto. Cabe señalar que las llamadas en notify()realidad no ceden el bloqueo de un recurso. Le dice a un hilo en espera que ese hilo puede despertarse. Sin embargo, el bloqueo no se abandona hasta que el bloqueo sincronizado del notificador se ha completado.
    - public final void **notifyAll()**
      - Despierta todos los hilos que llamaron wait()al mismo objeto.
  - Estos métodos son básico y engloban los conceptos mas importantes

Java: Métodos de Object para sincronización

- Todos son métodos de la clase object. Solo se pueden invocar por el thread propietario del lock (p.e. dentro de métodos synchronized).
- En caso contrario lanzan la excepción IllegalMonitorStateException

public final void wait() throws InteruptedException
- Espera indefinida hasta que reciba una notificación.

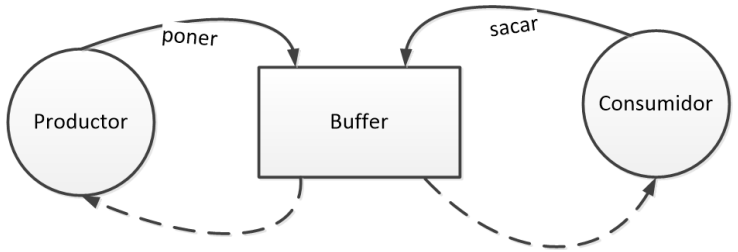
public final void wait(long timeout) throws InteruptedException
- El thread que ejecuta el método se suspende hasta que, o bien recibe una notificación, o bien,transcurre el timeout establecido en el argumento.
- wait(0) representa una espera indefinida hasta que llegue la notificación.

public final wait(long timeout, int nanos)
- Wait en el que el tiempo de timeout es 1000000*timeout+ nanos nanosegundos

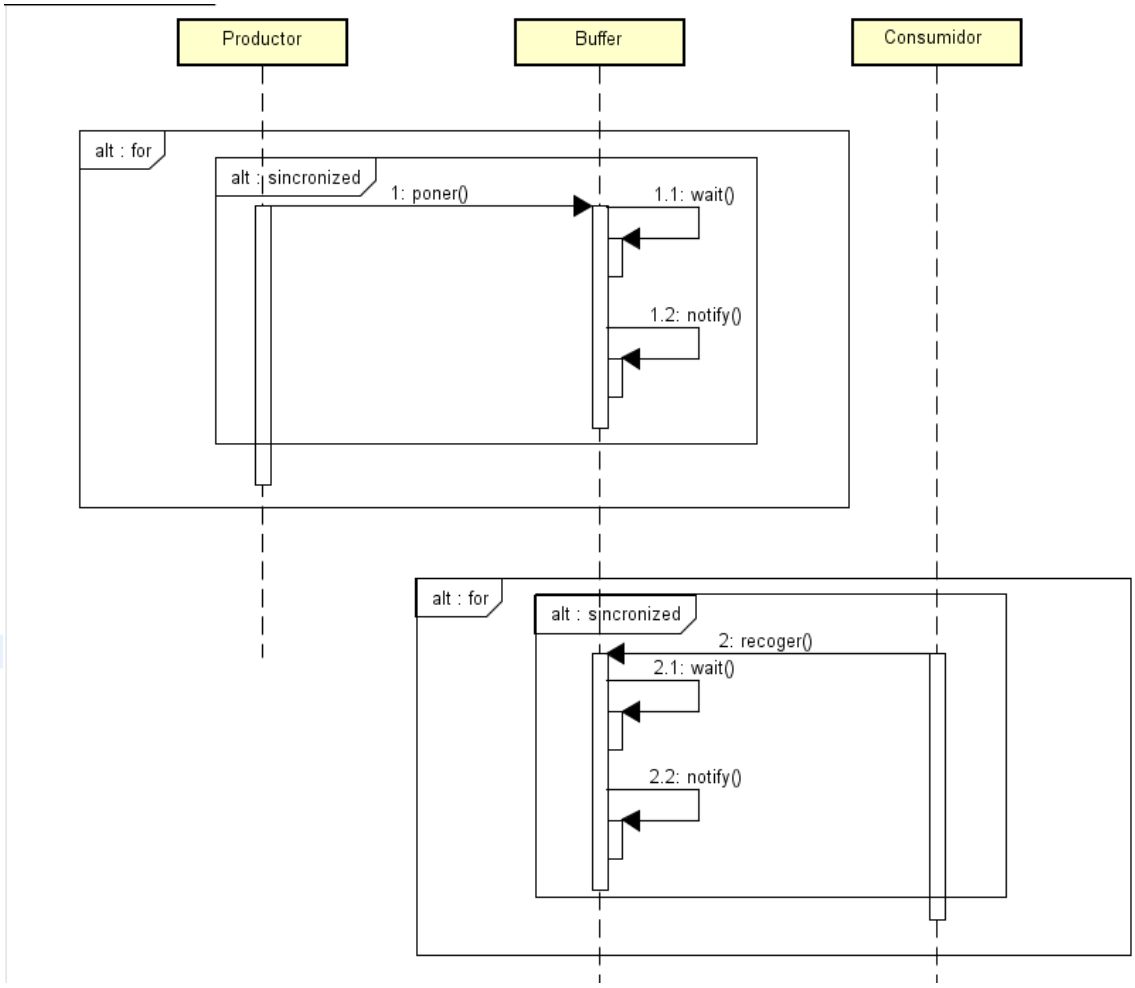
public final void notify()
- Notifica al objeto un cambio de estado, esta notificación es transferida a solo uno de los threads que esperan (han ejecutado un wait) sobre el objeto. No se puede especificar a cual de los objetos que esperan en el objeto será despertado.

public final void notifyAll()
- Notifica a todos los threads que esperan (han ejecutado un wait) sobre el objeto.

Productor consumidor




```java
package productorConsumidor;

public class Buffer {
    private char contenido;
    private boolean disponible=false;
    public Buffer() {
    }
    public synchronized char recoger(){
        while(!disponible){
            try{
                wait();
            }catch(InterruptedException ex){}
        }
        disponible=false;
        notify();
        return contenido;
    }
    public synchronized void poner(char valor){
        while(disponible){
            try{
                wait();
            }catch(InterruptedException ex){}
        }
        contenido=valor;
        disponible=true;
        notify();
    }

}
```



Productor consumidor

```java
package productorConsumidor;

public class ThreadApp2 {

      public static void main(String[] args) {
        Buffer b=new Buffer();
        Productor p=new Productor(b);
        Consumidor c=new Consumidor(b);
        p.start();
        c.start();

        try  {
    //espera la pulsación de una tecla y luego RETORNO
            System.in.read();
        }catch (Exception e) {  }
      }
     }
```

```java
package productorConsumidor;

public class Buffer {
    private char contenido;
    private boolean disponible=false;
    public Buffer() {
    }
    public synchronized char recoger(){
        while(!disponible){
            try{
                wait();
            }catch(InterruptedException ex){}
        }
        disponible=false;
        notify();
        return contenido;
    }
    public synchronized void poner(char valor){
        while(disponible){
            try{
                wait();
            }catch(InterruptedException ex){}
        }
        contenido=valor;
        disponible=true;
        notify();
    }

}
```

```java
package productorConsumidor;

public class Productor extends Thread {
      private Buffer buffer;
      private final String letras="abcdefghijklmnopqrstuvxyz";
      public Productor(Buffer buffer) {
        this.buffer=buffer;
      }
        public void run() {
            for(int i=0; i<10; i++){
                char c=letras.charAt((int)(Math.random()*letras.length()));
                buffer.poner(c);
                System.out.println(i+" Productor: " +c);
                try {
                    sleep(400);
                } catch (InterruptedException e) { }
            }
        }
    }
```

```java
package productorConsumidor;

public class Consumidor extends Thread {
      private Buffer buffer;
      public Consumidor(Buffer buffer) {
        this.buffer=buffer;
      }
      public void run(){
        char valor;
        for(int i=0; i<10; i++){
            valor=buffer.recoger();
            System.out.println(i+ " Consumidor: "+valor);
            try{
                sleep(100);
            }catch (InterruptedException e) { }
        }
      }
    }
```

Definición de semáforo

- **Un semáforo es** un mecanismo de mas alto nivel que synchronized
- **Un semáforo es** un tipo de objeto, y tiene un contador que protege el acceso a uno o más recursos compartidos
- **Un semáforo puede** controla el acceso a uno o mas recursos que no necesariamente están en el mismo bloque.
- **Un semáforo como** cualquier tipo de datos, queda definido por:
  - Conjunto de valores que se le pueden asignar.
  - Conjunto de operaciones que se le pueden aplicar.
- **Un semáforo tiene** *asociada una lista/cola de procesos*, en la que se incluyen los procesos suspendidos a la espera de su cambio de estado.
- **Los semáforos son** herramientas básicas de la programación concurrente y son proporcionados por la mayoría de los lenguajes de programación
- Siempre asegúrese de seleccionar el mecanismo apropiado de acuerdo con las características de aplicación que este diseñando.

- El concepto de un semáforo fue introducido por Edsger Dijkstra en 1965

Valores de un semáforo.

- En función del rango de valores que puede tomar, los semáforos se clasifican en: „
  - Semáforos **binarios**: Pueden tomar solo los valores 0 y 1.

    var mutex: BinSemaphore; „
  - Semáforos **general**: Puede tomar cualquier valor Natural (entero no negativo, 0,1,2,…).

    var escribiendo: Semaphore;

    Sus metodos: acquire()
    release()

- Valores de un semáforo:
  - **semáforo=0 ⇒ lock cerrado**
  - **semáforo > 0 ⇒ lock abierto**

- Es posible demostrar que un semáforo General se puede implementar utilizando semáforos Binarios.
- Los sistemas suelen ofrecer como componente primitivo semáforos generales, y su uso, lo convierte de hecho en semáforo binario.

Semaphore

- Para declarar un semaphore

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

Semaphore semaphore = new Semaphore(1);
```

- Cuando un hilo quiere acceder a uno de los recursos compartidos, primero debe adquirir el semáforo.

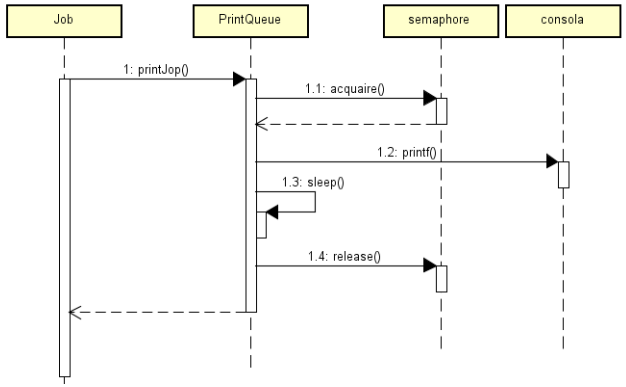
```
semaphore.acquire();
```

- Cuando el hilo ha terminado de usar el recurso compartido, debe liberar el semáforo con lo que libera el recurso.

```
semaphore.release();
```

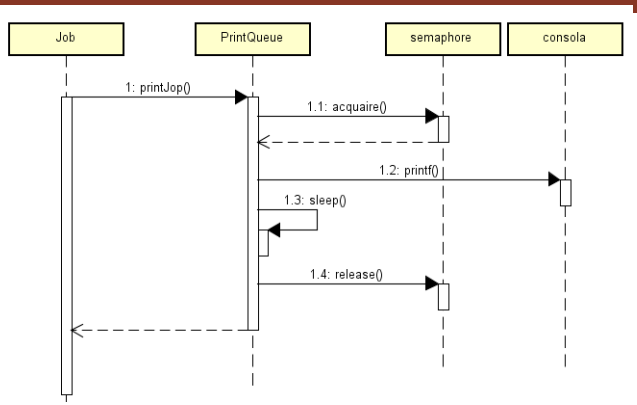
  - Si el contador interno del semáforo es mayor que 0, el semáforo disminuye el contador y permite el acceso al recurso compartido.
    - Un contador mayor que 0 implica que hay recursos que pueden ser utilizados, por lo que el hilo puede acceder y utilizar uno de ellos.
  - Si el contador es 0, el semáforo pone el hilo a dormir hasta que el contador sea mayor que 0.
    - Un valor de 0 en el contador significa que todos los recursos compartidos están siendo utilizados, así que el hilo que quiera usar uno de ellos debe esperar hasta que uno esté libre.

Ejemplo con semaphore



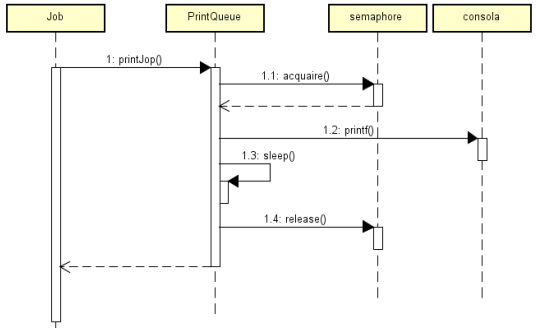

```java
public class Main {

    /**
     * Main method of the class. Run ten jobs in parallel that
     * send documents to the print queue at the same time.
     */
    public static void main (String args[]){

        // Creates the print queue
        PrintQueue printQueue=new PrintQueue();

        // Creates ten Threads
        Thread thread[]=new Thread[10];
        for (int i=0; i<10; i++){
            thread[i]=new Thread(new Job(printQueue),"Thread "+i);
        }

        // Starts the Threads
        for (int i=0; i<10; i++){
            thread[i].start();
        }
    }

}
```

Ejemplo con semaphore



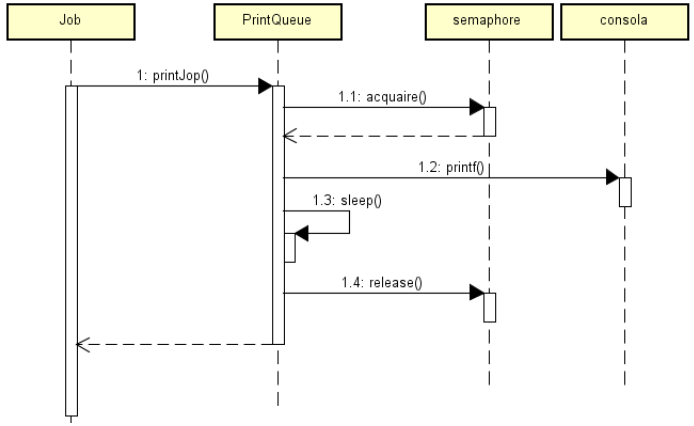

```
4   * This class simulates a job that send a document to print.
5   *
6   */
7  public class Job implements Runnable {
8
9      /**
10      * Queue to print the documents
11      */
12     private PrintQueue printQueue;
13
14     /**
15      * Constructor of the class. Initializes the queue
16      * @param printQueue
17      */
18     public Job(PrintQueue printQueue){
19         this.printQueue=printQueue;
20     }
21
22     /**
23      * Core method of the Job. Sends the document to the print queue and waits
24      *  for its finalization
25      */
26     @Override
27     public void run() {
28         System.out.printf("%s: Going to print a job\n",Thread.currentThread().getName());
29         printQueue.printJob(new Object());
30         System.out.printf("%s: The document has been printed\n",Thread.currentThread().getName());
31     }
32 }
33
```

Ejemplo con semaphore




```
1  package com.packtpub.java7.concurrency.chapter3.recipe1.task;
2
3  import java.util.concurrent.Semaphore;
4  import java.util.concurrent.TimeUnit;
5
6  /**
7   * This class implements the PrintQueue using a Semaphore to control the
8   * access to it.
9   *
10  */
11 public class PrintQueue {
12
13     /**
14      * Semaphore to control the access to the queue
15      */
16     private final Semaphore semaphore;
17
18     /**
19      * Constructor of the class. Initializes the semaphore
20      */
21     public PrintQueue(){
22         semaphore=new Semaphore(1);
23     }
24
25     /**
26      * Method that simulates printing a document
27      * @param document Document to print
28      */
```

Ejemplo con semaphore



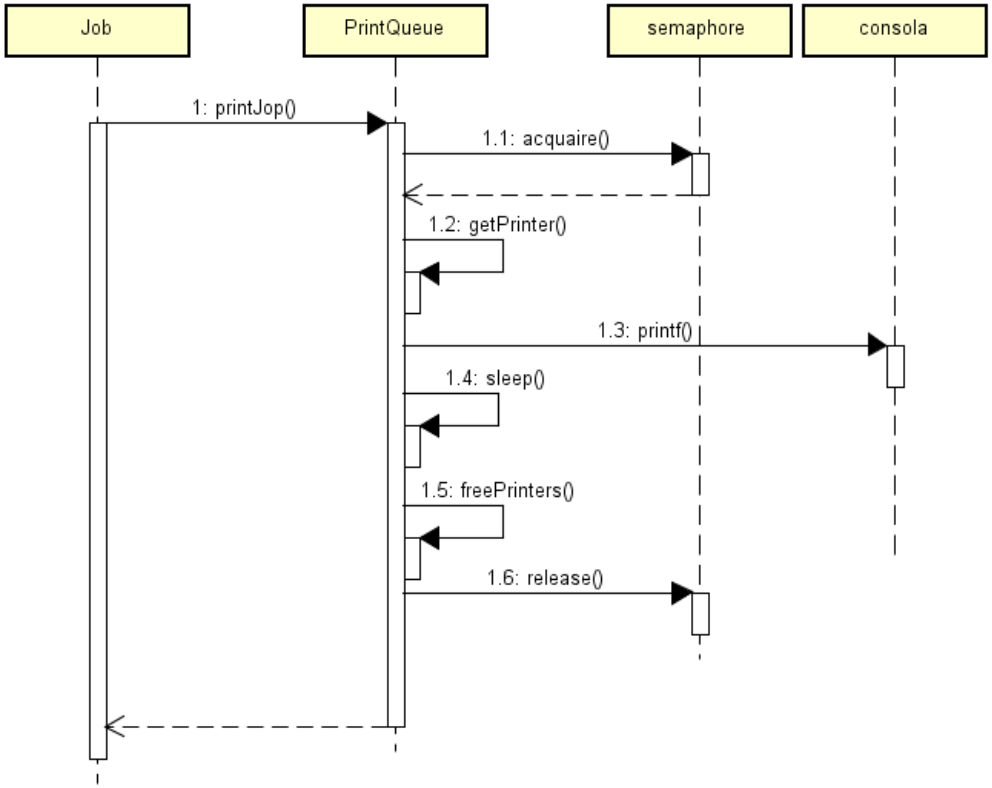

```
29    public void printJob (Object document){
30        try {
31            // Get the access to the semaphore. If other job is printing, this
32            // thread sleep until get the access to the semaphore
33            semaphore.acquire();
34
35            Long duration=(long)(Math.random()*3);
36            System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",Thread.currentThread().getName(),duration);
37            Thread.sleep(duration);
38            TimeUnit.SECONDS.sleep(duration);
39        } catch (InterruptedException e) {
40            e.printStackTrace();
41        } finally {
42            // Free the semaphore. If there are other threads waiting for this semaphore,
43            // the JVM selects one of this threads and give it the access.
44            semaphore.release();
45        }
46    }
47
48 }
```

Control del acceso simultáneo a múltiples copias de un recurso

- Los semáforos pueden ser usados cuando necesita proteger varias copias de un recurso, o cuando tiene una sección crítica que puede ser ejecutado por más de un hilo al mismo tiempo.
- Para el caso anterior, donde como recurso se tiene solo una impresora, ahora disponemos de tres impresoras. El tipo de recurso es el mismo pero y lo que hacemos con este es lo mismo pero cada impresora es diferente por lo que las tres pueden estar trabajando en paralelo
- Lo que se debe hacer es administrar los recursos y cuidar la concurrencia de los trabajos.
- Lock interface
  - **ReentrantLock**, es una clase:, sus metodos: lock(), unlock(), isLocked(), tryLock(), isHeldByCurrentThread(), lockInterruptibly, isLocked
    - es un bloqueo de exclusión mutua con el mismo comportamiento básico synchronized, solo un hilo puede mantener el bloqueo en un momento dado
    - Es importante envolver su código en un try/finallybloque para garantizar el desbloqueo en caso de excepciones.
  - ReadWriteLock es una interface
    - especifica otro tipo de bloqueo que mantiene un par de bloqueos para acceso de lectura y escritura.
    - La idea detrás de los bloqueos de lectura y escritura es que, por lo general, es seguro leer variables mutables simultáneamente, siempre y cuando nadie escriba en esta variable.
    - Por lo tanto, el bloqueo de lectura puede ser mantenido simultáneamente por múltiples hilos, siempre y cuando ningún hilo mantenga el bloqueo de escritura.
    - Esto puede mejorar el rendimiento
      - lock.writeLock().lock(); lock.writeLock().unlock(); lock.readLock().lock(); lock.readLock().unlock();

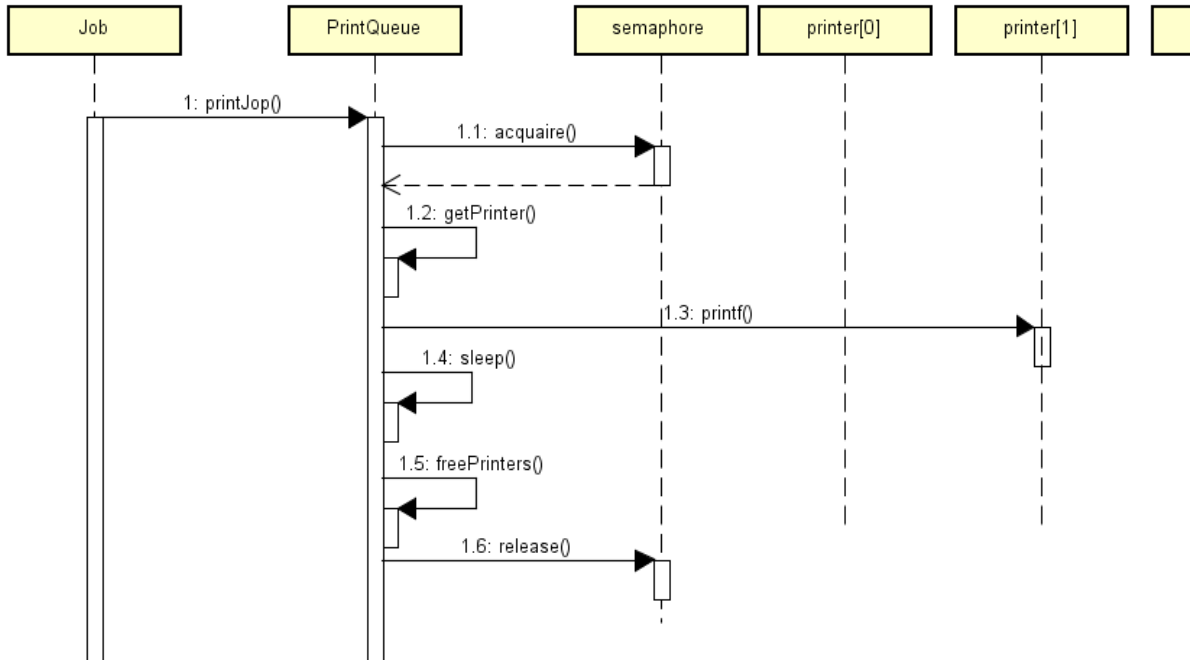
Control del acceso simultáneo a múltiples copias de un recurso

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

/**
 * This class implements a PrintQueue that have access to three printers.
 *
 * We use a Semaphore to control the access to one of the printers. When
 * a job wants to print, if there is one or more printers free, it has access
 * to one of the free printers. If not, it sleeps until one of the printers
 * is free.
 *
 */
public class PrintQueue {

    /**
     * Semaphore to control the access to the printers
     */
    private Semaphore semaphore;

    /**
     * Array to control what printer is free
     */
    private boolean freePrinters[];

    /**
     * Lock to control the access to the freePrinters array
     */
    private Lock lockPrinters;

    /**
     * Constructor of the class. It initializes the three object
     */
    public PrintQueue(){
        semaphore=new Semaphore(3);
        freePrinters=new boolean[3];
        for (int i=0; i<3; i++){
            freePrinters[i]=true;
        }
        lockPrinters=new ReentrantLock();
    }
```



Control del acceso simultáneo a múltiples copias de un recurso

```
17 public class PrintQueue {

46     public void printJob (Object document){
47         try {
48             // Get access to the semaphore. If there is one or more printers free,
49             // it will get the access to one of the printers
50             semaphore.acquire();
51
52             // Get the number of the free printer
53             int assignedPrinter=getPrinter();
54
55             Long duration=(long)(Math.random()*10);
56             System.out.printf("%s: PrintQueue: Printing a Job in Printer %d during %d seconds\n",Thread.currentThread().getNa
57             TimeUnit.SECONDS.sleep(duration);
58
59             // Free the printer
60             freePrinters[assignedPrinter]=true;
61         } catch (InterruptedException e) {
62             e.printStackTrace();
63         } finally {
64             // Free the semaphore
65             semaphore.release();
66         }
67     }
68
69     private int getPrinter() {
70         int ret=-1;
71
72         try {
73             // Get the access to the array
74             lockPrinters.lock();
75             // Look for the first free printer
76             for (int i=0; i<freePrinters.length; i++) {
77                 if (freePrinters[i]){
78                     ret=i;
79                     freePrinters[i]=false;
80                     break;
81                 }
82             }
83         } catch (Exception e) {
84             e.printStackTrace();
85         } finally {
87             lockPrinters.unlock();
88         }
89         return ret;
90     }
91
```



Esperando múltiples eventos simultáneos

- clase CountDownLatch
  - uno o más hilos esperan hasta que se realizan un conjunto de operaciones
  - Para que un hilo espere la ejecución de estas operaciones, ejecuta el método await().
  - El hilo duerme hasta que las operaciones se completen
  - Cuando se termina una de las operaciones el hilo que la termino ejecuta countDown() para disminuir el contador interno de la clase CountDownLatch
  - Cuando el contador llega a 0, la clase despierta todos los hilos que estaban durmiendo en el método await()
  - Esta incluida en la API de concurrencia de Java
- Como ejemplo implementaremos un sistema de videoconferencia, este esperará la llegada de todos los participantes antes de comenzar.

Esperando múltiples eventos simultáneos

```
10 public class Main {
11
13      * Main method of the example
16     public static void main(String[] args) {
17
18         // Creates a VideoConference with 10 participants.
19         Videoconference conference=new Videoconference(10);
20         // Creates a thread to run the VideoConference and start it.
21         Thread threadConference=new Thread(conference);
22         threadConference.start();
23
24         // Creates ten participants, a thread for each one and starts them
25         for (int i=0; i<10; i++){
26             Participant p=new Participant(conference, "Participant "+i);
27             Thread t=new Thread(p);
28             t.start();
29         }
30
31     }
32
33 }
34
```

Esperando múltiples eventos simultáneos

```
6  * This class implements a participant in the VideoConference
9  public class Participant implements Runnable {
10
11     /**
12      * VideoConference in which this participant will take part off
13      */
14     private Videoconference conference;
15
16     /**
17      * Name of the participant. For log purposes only
18      */
19     private String name;
20
21     /**
22      * Constructor of the class. Initialize its attributes
23      * @param conference VideoConference in which is going to take part off
24      * @param name Name of the participant
25      */
26     public Participant(Videoconference conference, String name) {
27         this.conference=conference;
28         this.name=name;
29     }
30
31     /**
32      * Core method of the participant. Waits a random time and joins the VideoConference
33      */
34     @Override
35     public void run() {
36         Long duration=(long)(Math.random()*10);
37         try {
38             TimeUnit.SECONDS.sleep(duration);
39         } catch (InterruptedException e) {
40             e.printStackTrace();
41         }
42         conference.arrive(name);
43
44     }
45 }
```

Esperando múltiples eventos simultáneos

```
3  import java.util.concurrent.CountDownLatch;
4
6  * This class implements the controller of the Videoconference..
12 public class Videoconference implements Runnable{
13
15     * This class uses a CountDownLatch to control the arrivel of all..
18     private final CountDownLatch controller;
19
21     * Constructor of the class. Initializes the CountDownLatch..
24     public Videoconference(int number) {
25         controller=new CountDownLatch(number);
26     }
27
29     * This method is called by every participant when he incorporates to the VideoConference..
32     public void arrive(String name){
33         System.out.printf("%s has arrived.\n",name);
34         // This method uses the countDown method to decrement the internal counter of the
35         // CountDownLatch
36         controller.countDown();
37         System.out.printf("VideoConference: Waiting for %d participants.\n",controller.getCount());
38     }
39
41     * This is the main method of the Controller of the VideoConference. It waits for all..
44     @Override
45     public void run() {
46         System.out.printf("VideoConference: Initialization: %d participants.\n",controller.getCount());
47         try {
48             // Wait for all the participants
49             controller.await();
50             // Starts the conference
51             System.out.printf("VideoConference: All the participants have come\n");
52             System.out.printf("VideoConference: Let's start...\n");
53         } catch (InterruptedException e) {
54             e.printStackTrace();
55         }
56     }
57
58
59 }
```

Sincronizar tareas en un punto común

- class CyclicBarrier
  - Se utiliza para sincronización de dos o más hilos en un punto determinado, similar CountDownLatch pero mas poderosa
  - se inicia con el número de hilos que se sincronizarán en un punto determinado.
  - Cuando uno de esos hilos llega a la punto, llama al método await() para esperar a los otros hilos.
    - Cuando el hilo llama a ese método, la clase de Barrera Cíclica bloquea el hilo.
  - Cuando el último hilo llama al método await() de la clase CyclicBarrier, despierta todos los hilos que estaban esperando y continúa con su trabajo.
  - Una ventaja de esta clase es que puede pasar una Objeto ejecutable como parámetro de inicialización, y la clase CyclicBarrier ejecuta este objeto como un hilo cuando todos los hilos han llegado al punto común.
- Ejemplo
  - Se busca un número en una matriz de los números aleatorios.
  - La matriz se dividirá en subconjuntos (utilizando la técnica de dividir y conquistar), así que cada hilo buscará el número en un subconjunto.
  - Una vez que todos los hilos hayan terminado su trabajo, una tarea final unificará los resultados de los mismos.

Sincronizar tareas en un punto común

la longitud de cada fila y el número que vamos a buscar como parámetros

```
3  import java.util.Random;
4
6  * This class generates a random matrix of integer numbers between 1 and 10
9  public class MatrixMock {
11     * Bi-dimensional array with the random numbers
13     private int data[][];
14
16     * Constructor of the class. Generates the bi-dimensional array of numbers.
23     public MatrixMock(int size, int length, int number){
24
25         int counter=0;
26         data=new int[size][length];
27         Random random=new Random();
28         for (int i=0; i<size; i++) {
29             for (int j=0; j<length; j++){
30                 data[i][j]=random.nextInt(10);
31                 if (data[i][j]==number){
32                     counter++;
33                 }
34             }
35         }
36         System.out.printf("Mock: There are %d ocurrences of number in generated data.\n",counter,number);
37     }
38
40     * This methods returns a row of the bi-dimensional array
44     public int[] getRow(int row){
45         if ((row>=0)&&(row<data.length)){
46             return data[row];
47         }
48         return null;
49     }
50
51 }
```

matriz aleatoria
de números
entre el 1 y el 10

El parámetro es el número de una fila en la matriz y devuelve la fila si existe, y null si no existe

Sincronizar tareas en un punto común

Esta clase almacenará, en una matriz, el número de ocurrencias del número buscado en cada fila de la matriz.

```
 4⊕ * This class is used to store the number of occurrences of the number
 8  public class Results {
 9
11⊕     * Array to store the number of occurrences of the number in each row of the array
13      private int data[];
14
16⊕     * Constructor of the class. Initializes its attributes
19⊖     public Results(int size){
20          data=new int[size];
21      }
22
24⊕     * Sets the value of one position in the array of results
28⊖     public void  setData(int position, int value){
29          data[position]=value;
30      }
31
33⊕     * Returns the array of results
36⊖     public int[] getData(){
37          return data;
38      }
39  }
40
```

Sincronizar tareas en un punto común

método que buscará el número. Utiliza una variable interna llamada contador que almacenará el número de ocurrencias del número en cada fila y lo almacene en la posición correspondiente del objeto Resultados..

await() del objeto CyclicBarrier y añade el código necesario para procesar las excepciones InterruptedException y BrokenBarrierException que este método puede lanzar

```
10  * Class that search for a number in a set of rows of the bi-dimensional array
13  public class Searcher implements Runnable {
14      /**
15       * First row where look for, Last row where look for, Number to look for
16       */
17      private int firstRow, lastRow, number;
18      /**
19       * Bi-dimensional array with the numbers
20       */
21      private MatrixMock mock;
22
24       * Array to store the results
26      private Results results;
27      /**
28       * CyclicBarrier to control the execution
29       */
30      private final CyclicBarrier barrier;
31
33       * Constructor of the class. Initializes its attributes
41      public Searcher(int firstRow, int lastRow, MatrixMock mock, Results results, int number, CyclicBarrier barrier){
42          this.firstRow=firstRow; this.lastRow=lastRow; this.mock=mock;
43          this.results=results; this.number=number; this.barrier=barrier;
44      }
45
47       * Main method of the searcher. Look for the number in a subset of rows. For each row, saves the
50      @Override
51      public void run() {
52          int counter;
53          System.out.printf("%s: Processing lines from %d to %d.\n",Thread.currentThread().getName(),firstRow,lastRow);
54          for (int i=firstRow; i<lastRow; i++){
55              int row[]=mock.getRow(i);
56              counter=0;
57              for (int j=0; j<row.length; j++){
58                  if (row[j]==number){
59                      counter++;
60                  }
61              }
62              results.setData(i, counter);
63          }
64          System.out.printf("%s: Lines processed.\n",Thread.currentThread().getName());
65          try {
66              barrier.await();
67          } catch (InterruptedException e) {
68              e.printStackTrace();
69          } catch (BrokenBarrierException e) {
70              e.printStackTrace();
71          }
72      }
73
74  }
```

Sincronizar tareas en un punto común

calculará el número total de ocurrencias del número en el conjunto de resultados

Obtener el número de ocurrencias del número en cada fila usando el método getData() del objeto de resultados. Luego, procesa todos los elementos de la matriz y añade su valor a la variable finalResultado.

```
7  * Group the results of each Searcher. Sum the values stored in the Results object
11 public class Grouper implements Runnable {
12
13     /**
14      * Results object with the occurrences of the number in each row
15      */
16     private Results results;
17
18     /**
19      * Constructor of the class. Initializes its attributes
20      * @param results Results object with the ocurrences of the number in each row
21      */
22     public Grouper(Results results){
23         this.results=results;
24     }
25
26     /**
27      * Main method of the Grouper. Sum the values stored in the Results object
28      */
29     @Override
30     public void run() {
31         int finalResult=0;
32         System.out.printf("Grouper: Processing results...\n");
33         int data[]=results.getData();
34         for (int number:data){
35             finalResult+=number;
36         }
37         System.out.printf("Grouper: Total result: %d.\n",finalResult);
38     }
39
40 }
```

Sincronizar tareas en un punto común

```
20    public static void main(String[] args) {
21
22        /*
23         * Initializes the bi-dimensional array of data
24         *      10000 rows
25         *      1000 numbers in each row
26         *      Looking for number 5
27         */
28        final int ROWS=10000;
29        final int NUMBERS=1000;
30        final int SEARCH=5;
31        final int PARTICIPANTS=5;
32        final int LINES_PARTICIPANT=2000;
33        MatrixMock mock=new MatrixMock(ROWS, NUMBERS,SEARCH);
34
35        // Initializes the object for the results
36        Results results=new Results(ROWS);
```

objeto de Barrera Cíclica llamado barrera. Este objeto esperará cinco hilos. Cuando este hilo termine, ejecutará el objeto Grouper creado previamente.

Crear cinco objetos del Buscador, cinco hilos para ejecutarlos, e iniciar los cinco hilos

```
38        // Creates an Grouper object
39        Grouper grouper=new Grouper(results);
40
41        // Creates the CyclicBarrier object. It has 5 participants and, when
42        // they finish, the CyclicBarrier will execute the grouper object
43        CyclicBarrier barrier=new CyclicBarrier(PARTICIPANTS,grouper);
44
45        // Creates, initializes and starts 5 Searcher objects
46        Searcher searchers[]=new Searcher[PARTICIPANTS];
47        for (int i=0; i<PARTICIPANTS; i++){
48            searchers[i]=new Searcher(i*LINES_PARTICIPANT, (i*LINES_PARTICIPANT)+LINES_PARTICIPANT, mock, results, 5,barrier);
49            Thread thread=new Thread(searchers[i]);
50            thread.start();
51        }
52        System.out.printf("Main: The main thread has finished.\n");
53    }
54
55 }
```

Sincronización

- Java 7 Concurrency Cookbook Basic Thread, Javier Fernández González, 2012
- Java 9 Concurrency Cook, González, Javier Fernández, 2017

- Mirar este documento
  - https://www.securecoding.cert.org/confluence/display/java/LCK01-J.+Do+not+synchronize+on+objects+that+may+be+reused