Actividades UD 5. Otras herramientas de sincronización. *Java.util.concurrent*

Concurrencia y Sistemas Distribuidos

Actividad 1

- Enumere los distintos inconvenientes que ofrecen las primitivas básicas de Java (i.e. monitores) para la sincronización de tareas.
- Limitaciones relacionadas con la exclusión mutua
  1. No se puede establecer un plazo máximo de espera a la hora de "solicitar" la entrada al monitor
  2. No se puede preguntar por el estado del monitor antes de solicitar el acceso al mismo.
  3. Las herramientas que garantizan exclusión mutua están orientadas a bloques
  4. No podemos extender su semántica.

     P. ejemplo: proporcionar con un synchronized exclusión mutua entre hilos escritores o entre escritores y lectores, pero no entre múltiples lectores.
- Limitaciones relacionadas con la sincronización condicional
  1. Solo podrá existir una única condición en cada monitor.
  2. Se utiliza la variante de Lampson y Redell.
     - El programador está obligado a utilizar una estructura del tipo:

       while (expresión lógica) wait();

       para consultar el estado del monitor y suspenderse.

Actividad 2

- Indique brevemente qué realiza cada uno de los siguientes métodos de la clase **ReentrantLock**:

ReentrantLock(boolean fair) Crea una instancia de un loock reentrante con la gestión de la cola indicada.	¿Para qué sirve el parámetro "fair"? Permite especificar si se requiere una gestión equitativa (fair) de la cola de espera mantenida por el lock. Cuando es true, se favorece al hilo que ha esperado más tiempo en la cola.

tryLock()
Adquiere el lock solo si está libre, retornando true. Si no está libre retorna false.
	¿Podemos usar este método para romper la condición de "retención y espera"? ¿Cómo? Sí.
	`Lock l=new ReentrantLock();`
	`if (l.trylock()) {`
	`  try { …. Usar recurso ….`
	`} finally { l.unlock()};`
	`} else  … no lo puedo usar hago otra cosa … ;`

tryLock(long timeout, TimeUnit unit) Adquiere el lock si está libre en el tiempo indicado, retornando true. Si no, si pasa la cantidad de tiempo indicada antes de que pueda adquirir el lock retorna false.	¿Para qué sirve el timeout? Permite especificar un plazo máximo de espera para obtener el lock
newCondition() Crea una instancia de una variable Condition asociada a un lock	¿Cuántas veces lo podríamos utilizar dentro de un lock? Todas las que queramos

Actividad 3

- Para cada una de las siguientes afirmaciones, indique si se corresponden con características de los Locks de la biblioteca java.util.concurrent, de los monitores básicos de Java, o bien de ambos.

	Reentrant Lock	Monitores básicos
Se ofrecen distintos tipos, con semántica diferente (orientados a exclusión mutua, orientados a resolver el problema de lectores-escritores).	X	
Dispone de un método que no suspende al invocador si el "lock" ya ha sido cerrado por otro hilo.	X	
Ofrece un método que sí suspende al invocador, si el "lock" ya ha sido cerrado por otro hilo.	X	X
Se puede establecer un plazo máximo de espera al solicitar la entrada al monitor.	X	
Se puede preguntar por el estado del monitor antes de solicitar acceso al mismo.	X	
Se puede utilizar más de una variable condición asociada al monitor.	X	
Se puede cerrar el monitor utilizando un método de la una clase A, y abrirlo en un método de la clase B.	X	

Actividad 3

- Para cada una de las siguientes afirmaciones, indique si se corresponden con características de los Locks de la biblioteca java.util.concurrent, de los monitores básicos de Java, o bien de ambos.

	ReentrantLock	Monitores básicos de Java
Se puede cerrar el monitor utilizando un método de la una clase A, y abrirlo con otro método de la clase A.	X	
Todos los métodos de la clase monitor deben llevar la etiqueta synchronized.		X
Se pueden interrumpir las esperas de los hilos que desean adquirir el lock.	X	
El programador no debe preocuparse del cierre y apertura de los locks. La gestión es implícita.		X
Cuando se produce una excepción, se debe controlar que, en el código asociado a la excepción se abra el lock.	X	
Todos los hilos que se suspenden en alguna condición van a parar a una única cola (la misma para todos).		X

Actividad 4

- Dado el siguiente ejemplo de utilización de las clases Locks y Conditions:

```
class BufferOk implements Buffer {
   private int elems, cabeza, cola, N;
   private int[] datos;
   Condition noLleno, noVacio;
   ReentrantLock lock;
   public BufferOk(int N) {
      datos= new int[N];
      this.N=N;
      cabeza = cola = elems = 0;
      lock= new ReentrantLock();
      noLleno=lock.newCondition();
      noVacio=lock.newCondition();
   }
```

a) Explique para qué sirve la clase BufferOK. ¿Quién podrá hacer uso de dicha clase?.

BufferOk implementa un buffer circular de capacidad N, proporcionando los métodos para insertar (put) y extraer (get) elementos enteros. Lo utilizarán tanto hilos productores como consumidores.

```
public int get() {
   int x;
   try {
      lock.lock();
      while (elems==0) {
         System.out.println("consumidor esperando ..");
         try {noVacio.await();}
          catch(InterruptedException e) {}
      }
      x=datos[cabeza];  cabeza= (cabeza+1)%N;
      elems--;
      noLleno.signal();
     return x;
   } finally {lock.unlock();}
}
public void put(int x) {
    try{
      lock.lock();
      while (elems==N) {
         System.out.println("productor esperando ..");
         try {noLleno.await();}
          catch(InterruptedException e) {}
      }
      datos[cola]=x; cola= (cola+1)%N;  elems++;
      noVacio.signal();
   } finally {lock.unlock();}
}
```

Actividad 4

- Dado el siguiente ejemplo de utilización de las clases Locks y Conditions:

```
class BufferOk implements Buffer {
   private int elems, cabeza, cola, N;
   private int[] datos;
   Condition noLleno, noVacio;
   ReentrantLock lock;
   public BufferOk(int N) {
      datos= new int[N];
      this.N=N;
      cabeza = cola = elems = 0;
      lock= new ReentrantLock();
      noLleno=lock.newCondition();
      noVacio=lock.newCondition();
   }
```

b) Explique para qué sirven las condiciones noLLeno, noVacio.

En **noLleno esperarán** los hilos **productores** cuando quieren poner un elemento y el buffer está lleno. Cuando un hilo consumidor extraiga un elemento despertará a un hilo productor que esté esperando en noLLeno.

```
public int get() {
   int x;
   try {
      lock.lock();
      while (elems==0) {
         System.out.println("consumidor esperando ..");
         try {noVacio.await();}
          catch(InterruptedException e) {}
      }
      x=datos[cabeza];  cabeza= (cabeza+1)%N;
      elems--;
      noLleno.signal();
     return x;
   } finally {lock.unlock();}
}
public void put(int x) {
    try{
      lock.lock();
      while (elems==N) {
         System.out.println("productor esperando ..");
         try {noLleno.await();}
          catch(InterruptedException e) {}
      }
      datos[cola]=x; cola= (cola+1)%N;  elems++;
      noVacio.signal();
   } finally {lock.unlock();}
}
```

Actividad 4

- Dado el siguiente ejemplo de utilización de las clases Locks y Conditions:

```
class BufferOk implements Buffer {
   private int elems, cabeza, cola, N;
   private int[] datos;
   Condition noLleno, noVacio;
   ReentrantLock lock;
   public BufferOk(int N) {
      datos= new int[N];
      this.N=N;
      cabeza = cola = elems = 0;
      lock= new ReentrantLock();
      noLleno=lock.newCondition();
      noVacio=lock.newCondition();
   }
```

b) Explique para qué sirven las condiciones noLLeno, noVacio.

En **noVacío esperarán** los hilos **consumidores** cuando quieren extraer un elemento y el buffer está vacío. Cuando un hilo productor extraiga un elemento, despertará a un hilo consumidor que esté esperando en noVacio.

```
public int get() {
   int x;
   try {
      lock.lock();
      while (elems==0) {
         System.out.println("consumidor esperando ..");
         try {noVacio.await();}
          catch(InterruptedException e) {}
      }
      x=datos[cabeza];  cabeza= (cabeza+1)%N;
      elems--;
      noLleno.signal();
     return x;
   } finally {lock.unlock();}
}
public void put(int x) {
    try{
      lock.lock();
      while (elems==N) {
         System.out.println("productor esperando ..");
         try {noLleno.await();}
          catch(InterruptedException e) {}
      }
      datos[cola]=x; cola= (cola+1)%N;  elems++;
      noVacio.signal();
   } finally {lock.unlock();}
}
```

Actividad 4

- Dado el siguiente ejemplo de utilización de las clases Locks y Conditions:

```
class BufferOk implements Buffer {
  private int elems, cabeza, cola, N;
  private int[] datos;
  Condition noLleno, noVacio;
  ReentrantLock lock;
  public BufferOk(int N) {
      datos= new int[N];
      this.N=N;
      cabeza = cola = elems = 0;
      lock= new ReentrantLock();
      noLleno=lock.newCondition();
      noVacio=lock.newCondition();
  }
```

```
public int get() {
   int x;
   try {
      lock.lock();
      while (elems==0) {
         System.out.println("consumidor esperando ..");
         try {noVacio.await();}
          catch(InterruptedException e) {}
      }
      x=datos[cabeza];  cabeza= (cabeza+1)%N;
      elems--;
      noLleno.signal();
     return x;
   } finally {lock.unlock();}
}
public void put(int x) {
    try{
      lock.lock();
      while (elems==N) {
         System.out.println("productor esperando ..");
         try {noLleno.await();}
          catch(InterruptedException e) {}
      }
      datos[cola]=x; cola= (cola+1)%N;  elems++;
      noVacio.signal();
   } finally {lock.unlock();}
}
```

c) ¿Podrían producirse condiciones de carrera? ¿Por qué?.

No, puesto que el acceso a los atributos compartidos sólo se puede realizar invocando a los métodos put y get, y éstos están protegidos por el ReentrantLock **lock.** Las respectivas **secciones críticas se ejecutan en exclusión mútua.**

Actividad 5

- Complete el código del monitor Crosswalk con las instrucciones necesarias que faltan (atendiendo a los comentarios)

```
public class Crosswalk {
    private int c, c_waiting, p, p_waiting;
    private Condition OKcars, OKpedestrians;
    private ReentrantLock lock;
    public Crosswalk() {
      c = c_waiting = p = p_waiting = 0;
      lock=new ReentrantLock();
      OKcars=lock.newCondition();
      OKpedestrians=lock.newCondition();
    }
    public void enterC() {
                                    //cerrar lock

       c_waiting++;
       while (p >0)
                                  //esperar a que los coches puedan pasar

       c_waiting--;
       c++;
                                  //notificar que los coches podrían pasar
                                  //abrir lock
  }
```

Actividad 5

- Complete el código del monitor Crosswalk con las instrucciones necesarias que faltan (atendiendo a los comentarios)

```
public class Crosswalk {
    private int c, c_waiting, p, p_waiting;
    private Condition OKcars, OKpedestrians;
    private ReentrantLock lock;
    public Crosswalk() {
      c = c_waiting = p = p_waiting = 0;
      lock=new ReentrantLock();
      OKcars=lock.newCondition();
      OKpedestrians=lock.newCondition();
    }
    public void enterC() {
    try {
       lock.lock();                  //cerrar lock
       c_waiting++;
       while (p >0)
       try { Okcars.await()          //esperar a que los coches puedan pasar
       } catch (InterruptedException e) {}
       c_waiting--;
       c++;
       Okcars.signal();               //notificar que los coches podrían pasar
    } finally {lock.unlock()}        //abrir lock
   }
```

Actividad 6

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { queue.put(produce()); }
    } catch (InterruptedException ex) {...}
  }
  Object produce() { ... }
}
class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { consume(queue.take()); }
    } catch (InterruptedException ex) {...}
  }
  void consume(Object x) { ... }
}
class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

- a) ¿Qué problema se pretende resolver en este ejemplo? ¿Cuántos hilos hay? ¿Qué representan? ¿Qué información/recurso comparten esos hilos?

  - El problema del productor/consumidor con buffer acotado
  - Hay 3 hilos, aparte de main.
  - 1 Hilo productor (p) y 2 hilos Consumidores (c1 y c2)
  - Comparten la BlockingQueue q

Actividad 6

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { queue.put(produce()); }
    } catch (InterruptedException ex) {...}
  }
  Object produce() { ... }
}
class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { consume(queue.take()); }
    } catch (InterruptedException ex) {...}
  }
  void consume(Object x) { ... }
}
class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

- b) Si un hilo quiere extraer un ítem de una cola que está vacía, ¿tendrá que esperar? ¿Dónde se controla esto?

- Sí que tendrá que esperar.
- Se controla en el método **take**

Actividad 6

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { queue.put(produce()); }
    } catch (InterruptedException ex) {...}
  }
  Object produce() { ... }
}
class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { consume(queue.take()); }
    } catch (InterruptedException ex) {...}
  }
  void consume(Object x) { ... }
}
class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

- c) Si un hilo quiere insertar un ítem en una cola que está llena, ¿tendrá que esperar? ¿Dónde se controla esto?

  - Sí que tendrá que esperar.
  - Se controla en el método **put**

Actividad 6

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { queue.put(produce()); }
    } catch (InterruptedException ex) {...}
  }
  Object produce() { ... }
}
class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while(true) { consume(queue.take()); }
    } catch (InterruptedException ex) {...}
  }
  void consume(Object x) { ... }
}
class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

- d) ¿Podrían producirse condiciones de carrera? ¿Por qué?
  - No se pueden producir condiciones de carrera.
  - BlockingQueue lo garantiza: proporciona acceso en exclusión mutua a la cola y sincronización condicional con los métodos put y take.
- ¿Podemos decir que la cola BlockingQueue es Thread-Safe? ¿Por qué?
  - Sí. Puede utilizarse por varios hilos concurrentemente sin que se produzcan condiciones de carrera

Actividad 7

- a) Analice las dos opciones. ¿Qué es lo que hacen? ¿Cuál será el valor de la variable *counter* en ambos casos? ¿Se pueden producir condiciones de carrera en algún caso?

OPCIÓN A: Usando nuestra propia clase

```
class ID {
   private static long nextID = 0;
   public static synchronized long getNext() {
        return nextID++;
   }
}
public class EjCounter extends Thread{
   ID counter;
   public EjCounter(ID c) {counter=c;}
   public void run() {
     System.out.println("counter value: "+
       (counter.getNext()));
   }
   public static void main(String[] args) {
     ID counter= new ID();
     new EjCounter(counter).start();
     new EjCounter(counter).start();
     new EjCounter(counter).start();
   }
}
```

OPCIÓN B: Usando variables atómicas

```
public class EjCounter extends Thread{
   AtomicLong counter;
   public EjCounter(AtomicLong c) {counter=c;}
   public void run() {
     System.out.println("counter value: "+
       (counter.getAndIncrement()));
   }
   public static void main(String[] args) {
     AtomicLong counter=
          new AtomicLong(0);
     new EjCounter(counter).start();
     new EjCounter(counter).start();
     new EjCounter(counter).start();
   }
}
```

- Se lanzan **3 hilos** EjCounter, que **comparten el mismo *counter***. En la opción A, *counter* es de la **clase ID**, mientras que en la opción B es un **AtomicLong**.
- Los hilos incrementan en uno el valor de *counter,* cuyo **valor final será 3**.
- **No se pueden producir condiciones de carrera**. En la opción A, el código de getNext está etiquetado como synchronized. En la opción B, se hace uso de la función getAndIncrement, que es atómica.

Actividad 7

- b) ¿Para qué cree que sirven los siguientes métodos de la clase AtomicLong? Indique cuál sería su instrucción equivalente, haciendo uso de la clase ID definida con la opción A. Para ello, añada nuevos métodos que permitan implementar esa misma funcionalidad.

Método	*Funcionamiento*	*Método equivalente en la clase ID*
counter.addAndGet(5);	Añade de forma atómica el valor indicado (5 en este caso) y nos devuelve el resultado	`public static synchronized long Add(long value) { return (nextID+value); }`
counter.getAndDecrement();	Decrementa en uno el valor actual. Nos devuelve el valor anterior.	`public static synchronized long Decrement() { return (nextID--); }`
counter.incrementAndGet();	Incrementa en uno el valor actual. Nos devuelve el resultado.	`public static synchronized long IncrementAndGet() { return (++nextID); }`

Actividad 8

- A continuación se muestra un código Java que trata de resolver el problema "Productor-Consumidor con buffer acotado":

```
class Buffer {
   private int head, tail, elems,size;
   private int[] data;
   private Semaphore item;
   private Semaphore slot;
   private Semaphore mutex;

   public Buffer(int s) {
      head=tail=elems=0;  size=s;
      data=new int[size];
      item=new Semaphore(0,true);
      slot=new Semaphore(size,true);
   mutex=new Semaphore(1,true);
   }
```

```
public int get() {
    try {item.acquire();} catch (InterruptedException e) {}
    try {mutex.acquire();} catch (InterruptedException e) {}
    int x=data[head]; head= (head+1)%size;  elems--;
    mutex.release();
    slot.release();
    return x;
 }
public void put(int x) {
    try {slot.acquire();} catch (InterruptedException e) {}
    try {mutex.acquire();} catch (InterruptedException e) {}
    data[tail]=x; tail= (tail+1)%size;  elems++;
    mutex.release();
    item.release();
  }
}
```

Actividad 8

- Dadas las siguientes afirmaciones, indique si son VERDADERAS o FALSAS

El código es incorrecto, pues cada objeto Buffer debería tener un *ReentrantLock* como atributo interno para poder generar semáforos dentro de él.	F
El semáforo slot proporciona sincronización condicional, suspendiendo al hilo productor cuando no haya huecos libres en el buffer.	V
El semáforo item proporciona sincronización condicional, suspendiendo al hilo consumidor cuando no haya elementos en el buffer.	V
El código es incorrecto, pues los métodos put() y get() deberían estar calificados como "synchronized" para poder utilizar los semáforos.	F

Actividad 9

- Complete la siguiente tabla de comparación entre estas dos clases

	CyclicBarrier	*CountdownLatch*
¿Lleva un contador?	Sí	Sí
¿Permite inicializar el contador en la creación?	Sí CyclicBarrier barrier = new CyclicBarrier(2);	Sí CountDownLatch latch = new CountDownLatch(3);
¿Se puede incrementar el contador? ¿Cómo?	NO	NO
¿Se puede decrementar el contador? ¿Cómo?	Se decrementa automáticamente con `barrier.await();`	Sí. Con llamadas a countDown() `latch.countDown();`
Método que se emplea para que el hilo se quede esperando	`barrier.await();`	`latch.await();`

¿Permite ejecutar alguna acción cuando el último hilo llega al punto de sincronización?	SÍ
	barrier = new CyclicBarrier(N,
	new Runnable() {
	public void run() {
	mergeRows(...);
	}
		NO

¿Se puede reutilizar?	Sí. Cuando se abre y reactivan todos los hilos, automáticamente se vuelve a cerrar	NO. Cuando el contador llega a 0, la barrera se abre y todos los que lleguen después la encuentran abierta

Actividad 10

- A. La clase buffer ha de ser *thread-safe*. Utilice las clases *ReentrantLock* y *Condition*.

```
public class Buffer {
  private int store = 0;
  private boolean full = false;




public int get()  {



    int value = store;
    store = 0;
    full = false;

    return value;

  }
```

```
public void put(int value)   {



    full = true;
    store = value;


 }
}
```

Actividad 10

- A. La clase buffer ha de ser *thread-safe*. Utilice las clases *ReentrantLock* y *Condition*.

```
public class Buffer {
  private int store = 0;
  private boolean full = false;
 Lock l=new ReentrantLock();
 Condition noLleno=l.newCondition();
 Condition noVacio=l.newCondition();

public int get()  {
 try {
   l.lock();
   while (!full) try {noVacio.await()} catch …
    int value = store;
    store = 0;
    full = false;
    noLleno.signal();
    return value;
  } finally { l.unlock();}
 }
```

```
public void put(int value)  {
 try {
    l.lock();
    while (full) try {noLleno.await()} catch …
     full = true;
     store = value;
     noVacio.signal();
  } finally { l.unlock();}
 }
}
```

Actividad 10

- El hilo principal debe escribir su último mensaje una vez que los otros hilos hayan acabado. Utilice la clase: *Semaphore*

```
import java.util.concurrent.Semaphore;
public class Main {
    public static void main(String[] args)  {
        Buffer c = new Buffer();
        Semaphore s = new Semaphore(0);
        Consumer c1 = new Consumer(c, 1, s);
        Producer p1 = new Producer(c, 2, s);
        c1.start();
        p1.start();
        try {
            s.acquire(); s.acquire();
         } catch (InterruptedException e) {};
        System.out.println("Producer and " +
          "Consumer have terminated.");
    }
}
```

```
import java.util.concurrent.Semaphore;
public class Consumer extends Thread {
    private Buffer b; Semaphore sem;
    private int number;
    public Consumer(Buffer ca, int id, Semaphore s)  {
        b = ca;
        number = id; sem=s;
    }
    public void run()  {
        int value = 0;
        for (int i = 1; i < 101; i++)
        {
          value = b.get();
          System.out.println("Consumer #" + number +
          " gets: " + value);
        }
        sem.release();
    }
}
```

Actividad 10

- El hilo principal debe escribir su último mensaje una vez que los otros hilos hayan acabado. Utilice la clase: *CyclicBarrier*

```
import java.util.concurrent.*;
public class Main {
    public static void main(String[] args)  {
       Buffer c = new Buffer();
       CyclicBarrier cb = new CyclicBarrier (3);
       Consumer c1 = new Consumer(c, 1, cb);
       Producer p1 = new Producer(c, 2, cb);
       c1.start();
       p1.start();
       try {
          cb.await();
        } catch (InterruptedException |
                  BrokenBarrierException e) {};
       System.out.println("Producer and " +
         "Consumer have terminated.");
    }
}
```

```
import java.util.concurrent.*;
public class Consumer extends Thread {
    private Buffer b; CyclicBarrier c;
    private int number;
    public Consumer(Buffer ca, int id, CyclicBarrier cb)  {
       b = ca;
       number = id; c=cb;
    }
    public void run()  {
       int value = 0;
       for (int i = 1; i < 101; i++)
       {
         value = b.get();
         System.out.println("Consumer #" + number +
         " gets: " + value);
       }
     try {c.await();} catch (InterruptedException |
                              BrokenBarrierException e) {};
    }
}
```

Actividad 10

- El hilo principal debe escribir su último mensaje una vez que los otros hilos hayan acabado. Utilice la clase: *CountDownLatch*

```
import java.util.concurrent.*;
public class Main {
    public static void main(String[] args)  {
        Buffer c = new Buffer();
        CountDownLatch cdl=new CountDownLatch(2);
        Consumer c1 = new Consumer(c, 1, cdl);
        Producer p1 = new Producer(c, 2, cdl);
        c1.start();
        p1.start();
        try {
           cdl.await();
        } catch (InterruptedException |e) {};
        System.out.println("Producer and " +
          "Consumer have terminated.");
    }
}
```

```
import java.util.concurrent.*;
public class Consumer extends Thread {
    private Buffer b; CountDownLatch c;
    private int number;
 public Consumer(Buffer ca, int id, CountDownLatch cdl)
   {
        b = ca;
        number = id; c=cdl;
    }
    public void run()  {
        int value = 0;
        for (int i = 1; i < 101; i++)
        {
          value = b.get();
          System.out.println("Consumer #" + number +
          " gets: " + value);
        }
        c.countDown();
    }
}
```

Actividad 12

- Dadas las siguientes afirmaciones, indique si son VERDADERAS o FALSAS

El objeto CountDownLatch es una barrera que una vez abierta ya no puede ser utilizada de nuevo. Justificación: No es reutilizable, porque una vez abierta no vuelve al estado inicial, sino que permanece abierta para siempre (ni la operación await ni countDown tendrían ningún efecto)	V
Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo CountDownLatch inicializado a 5, sabiendo que los hilos A ejecutan c.await() y los hilos B ejecutan c.countDown(), todos los hilos A quedarán suspendidos. Justificación: Como el valor inicial es 5, y sólo se ejecutarían 3 operaciones countDown (por parte de cada uno de los 3 hilos de tipo B), el contador no llega a 0 y la barrera no se abre: los 5 hilos tipo A que habían ejecutado await quedan suspendidos.	V

Actividad 12

- Dadas las siguientes afirmaciones, indique si son VERDADERAS o FALSAS

Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo CyclicBarrier inicializado a 4, sabiendo que los hilos A ejecutan c.await() y los hilos B también ejecutan c.await(), algún hilo de la clase A podrá quedarse suspendido indefinidamente. Justificación: Dado que todos los hilos ejecutan await, no distinguimos entre hilos A y B. Los 3 primeros esperan, y en cuanto llega el cuarto la barrera se abre y pasan los 4: en ese momento se restauran las condiciones iniciales, y con los 4 restantes pasa exactamente lo mismo.	F
Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo Semaphore inicializado a 3, sabiendo que los hilos A ejecutan c.acquire() y los hilos B ejecutan c.release(), todos los hilos B quedarán suspendidos, pues un hilo no puede realizar el método "release()" sobre un semáforo si previamente no ha adquirido un permiso de dicho semáforo. Justificación: La operación release no provoca la suspensión del hilo, y tampoco existe ninguna limitación en cuanto a la posibilidad de invocar release	F

Actividad 12

- Dadas las siguientes afirmaciones, indique si son VERDADERAS o FALSAS

Un objeto "c" de la clase CountDownLatch se puede utilizar para que M hilos de clase B esperen a que otro hilo A les avise. Para ello, inicializamos "c" a 1, los hilos B usarán c.await() y el hilo A llamará a c.countDown() una sola vez.	V
Un objeto "c" de la clase CountDownLatch se puede utilizar para garantizar exclusión mutua, inicializándolo a 1 en su constructor, y protegiendo la sección crítica entre un c.await() a su entrada y un c.countDown() a su salida. Justificación: Con esta implementación, todos los hilos que lleguen a la sección crítica se quedarán bloqueados, pues el código para abrir la barrera (el c.countDown) está justo a la salida de la sección crítica, de modo que ningún hilo podrá invocarla.	F
Para que un hilo A espere hasta que otros N hilos de una misma clase (H1..Hn) hayan ejecutado una sentencia B dentro de su código se puede utilizar un Semaphore S inicializado a 0; A invoca S.release(), mientras que H1..Hn invocan S.acquire() tras la sentencia B. Justificación: Con esta implementación, A no espera a nadie, y n-1 hilos H quedan bloqueados al realizar S.acquire().	F

ReentrantLock & Condition & Semahore

```
ReentrantLock rl=new ReentrantLock(true);
  try {
       rl.lock();
              -- S.C.
  } finally {
       rl.unlock();
  }
```

```
Condition c;
c=rl.newCondition();
```

- try { c.**await**();} catch(…) {…};
- c.**signal**();
- c.**signalAll**();

```
Semaphore s=new Semaphore(cont, fair);
```

- try {s.**acquire**();} catch(…) {…};
- s.**release**()

Decrementa el contador y si contador <0 => suspende al hilo que lo invoca

Incrementa el contador y si hay alguien bloqueado lo activa

CyclicBarrier & CountDownLatch

```
CyclicBarrier b=new CyclicBarrier (N, Runnable() );
```

- try {b.**await**();} catch(…) {…};

Suspende hasta que se hagan N operaciones await, => se abre la barrera y automáticamente se vuelve a cerrar

Código que se realizará cuando todos los hilos hayan llegado al punto de sincronización y antes de reactivarlos

Valor positivo

```
CountDownLatch cdl=new CountDownLatch(cont);
```

- try { cdl.**await**();} catch(…) {…};
- cdl.**countDown**();

Suspende si contador > 0

Decrementa el contador. Cuando llega a cero reactiva a todos los hilos suspendidos