code URJC

Programación Concurrente

Tema 5

Programación Concurrente en Java

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

code URJC

Programación Concurrente

Tema 5.5

Sincronización de Hilos

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

Sincronización de hilos

- **Semáforos**
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - Monitores
  - Sincronización de berrara (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Semáforos

- Los semáforos en Java se representan como objetos de la clase **java.util.concurrent.Semaphore**
- Se comportan igual que los semáforos diseñados por Dijkstra en el 68
  - Internamente tienen un contador de permisos (*permits*)
  - El método **adquire()** se bloquea hasta que adquiere un permiso
  - El método **release()** incrementa el número de permisos del semáforo

http://download.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

Semáforos

- **Justicia**
  - En el constructor se puede indicar la justicia (*fairness*) del semáforo
  - Si es justo (**fair = true**) entonces los hilos se desbloquean en orden de llegada (**cola FIFO**)
  - Si no es justo (**fair = false**) entonces los hilos se desbloquean de forma **aleatoria**
  - Por defecto los semáforos en Java no son justos porque es más eficiente y en los usos normales no plantea problemas de **inanición (*starvation*)**

Semáforos

- Versiones del método **adquire()**
  - **adquire():** Eleva la excepción **InterruptedException**
  - **acquireUninterruptibly():** El hilo no se desbloquea en caso de que se interrumpa al hilo. No se lanza la InterruptexException.
  - **tryAcquire()**
    - Si hay permisos devuelve **true** y decrementa los permisos del semáforo
    - Si no hay permisos, devuelve **false** inmediatamente
  - **tryAcquire(long timeout, TimeUnit unit):** Versión de tryAcquire() con tiempo de bloqueo máximo antes de devolver **false**

Semáforos

- **Operaciones en bloque**
  - Permiten adquirir o liberar varios permisos en una única llamada
  - relese(int permits)
  - acquire(int permits)
  - tryAcquire(int permits)
  - tryAcquire(int permits, long timeout, TimeUnit unit)

Semáforos

- **Métodos avanzados**
  - Gestión de permisos
    - **int drainPermits():** si hay permisos, ejecuta de forma atómica un acquire() por cada permiso. Si no, no hace nada
  - Gestión de hilos bloqueados
    - **getQueueLength():** Devuelve el número de hilos bloquedos
    - **hasQueuedThreads():** Indica si hay hilos bloqueados

Ejercicio 1

- Implementa el problema de Productores Consumidores con buffer usando únicamente la API de Java, sin usar **SimpleConcurrent**

Sincronización de hilos

- Semáforos
- **Exclusión mutua**
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - Monitores
  - Sincronización de berrara (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Exclusión Mutua

- La exclusión mutua es un tipo de sincronización que **limita** el número de hilos que pueden utilizar un recurso de forma **simultánea**
- Normalmente **sólo un hilo** puede usar un recurso. En ese caso la exclusión mutua permite crear **sentencias atómicas de grano grueso**
- Otras veces la exclusión mutua permite otros esquemas como en el problema de los **lectores/escritores**

Exclusión Mutua

- Se puede implementar "manualmente" con **semáforos**
- En Java existen mecanismos **específicos** para implementar exclusión mutua:
  - **Métodos y bloques sincronizados** *(synchronized)*
    - El **lenguaje** tiene construcciones **sintácticas** para representar exclusión mutua
  - **Cerrojos** (*Locks*)
    - La **API** dispone de diferentes **clases específicas** para implementar una exclusión mutua **avanzada**

Sincronización de hilos

- Semáforos
- Exclusión mutua
  - **Exclusión mutua con *synchronized***
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - Monitores
  - Sincronización de berrara (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Exclusión mutua con *synchronized*

code URJC

- La **exclusión mutua** es un tipo de sincronización tan **básica** en la programación concurrente que los lenguajes de programación ofrecen una **sintaxis especial** para conseguirlo
- En muchos lenguajes se denomina **cerrojo** (***lock***) al objeto que controla la exclusión mutua. En otros se llama **mutex**.
- El hilo que **posee el cerrojo** es el que está en la zona de exclusión mutua

Exclusión mutua con *synchronized*

code URJC

- Los bloques sincronizados (***synchronized***) son bloques de código que ponen bajo **exclusión mutua las sentencias** que contienen
- Para permitir **varias** exclusiones mutuas en un mismo programa, en los bloques sincronizados se especifica un **objeto** (de cualquier clase) que actuará como **cerrojo**

Exclusión mutua con *synchronized*

```
public class IncSynchronized {

   private double x = 0;
   private Object xLock = new Object();

   public void inc() {
      for (int i = 0; i < 10000000; i++) {
         synchronized (xLock) {
            x = x + 1;
         }
      }
   }

   public static void main(String[] args) throws InterruptedException {

      //Crear 3 hilos que ejecutan inc() y esperar a que acaben
      ...

      System.out.println("x:" + x);
   }
}
```

El cerrojo puede se un objeto de cualquier clase

En el bloque sincronizado se indica el cerrojo

Exclusión mutua con *synchronized*

```
public class IncDecSynchronized {

   private double x = 0;
   private Object xLock = new Object();

   public void inc() {
      for (int i = 0; i < 10000000; i++) {
         synchronized (xLock) {
            x = x + 1;
         }
      }
   }

   public void dec() {
      for (int i = 0; i < 10000000; i++) {
         synchronized (xLock) {
            x = x - 1;
         }
      }
   }
   ...
}
```

El mismo cerrojo se puede usar en varios bloques sincronizados que identifican secciones críticas diferentes de la misma exclusión mutua

Exclusión mutua con *synchronized*

```
public class IncSynchronized2 {

   private double x = 0;
   private Object xLock = new Object();

   private double y = 0;
   private Object yLock = new Object();

   public void incX() {
      for (int i = 0; i < 10000000; i++) {
         synchronized (xLock) {
            x = x + 1;
         }
      }
   }

   public void incY() {
      for (int i = 0; i < 10000000; i++) {
         synchronized (yLock) {
            y = y + 1;
         }
      }
   }

   ...
}
```

Se pueden tener tantos cerrojos como se quieran

Cada cerrojo se usa en una exclusión mutua diferente

Exclusión mutua con *synchronized*

code URJC

- Una forma de implementar una **clase *thread-safe*** consiste en poner bajo **exclusión mutua** todas las sentencias de un **método**
- En ese caso existe una **sintaxis más compacta**
- Se pone la palabra reservada ***synchronized*** al método y el **objeto que recibe el mensaje** actúa como **cerrojo**
- En los métodos **estáticos** la propia **clase** actúa como cerrojo

Exclusión mutua con *synchronized*

```
class Counter {
   private int x = 0;
   public synchronized void inc(){
      x = x + 1;
   }
   public synchronized int getValue(){
      return x;
   }
}

public class IncSynchronizedMethod {

   private Counter c = new Counter();

   public void inc() {
      for (int i = 0; i < 10000000; i++) {
         c.inc();
      }
   }
   ...
}
```

El objeto que recibe el mensaje actúa como cerrojo para los métodos sincronizados

La llamada al método no cambia

Exclusión mutua con *synchronized*

- Las sentencias y métodos **sincronizados** son **reentrantes**
- Si un hilo que ha **adquirido el cerrojo** en un método **sincronizado** llama a otro método **sincronizado** no se queda bloqueado porque ya tiene el cerrojo
- Esto permite la **reutilización** de métodos sincronizados
- Los **semáforos** no se comportan de esta forma, si un hilo ejecuta dos veces **acquire()** sobre un semáforo con 1 permiso, la segunda vez quedará bloqueado

Sincronización de hilos

- Semáforos
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - **Exclusión mutua con cerrojos (*Locks*)**
- Sincronización condicional
  - Monitores
  - Sincronización de berrara (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Exclusión mutua con cerrojos (*Locks*)

- Los **bloques sincronizados** son muy sencillos de usar pero tienen **limitaciones:**
  - La exclusión mutua no se puede **adquirir en un método y liberar en otro**
  - No se puede especificar un **tiempo máximo de espera** para adquirir el cerrojo
  - No se puede crear una **exclusión mutua extendida** como los lectores escritores de forma sencilla

Exclusión mutua con cerrojos (*Locks*)

- Los objetos **Lock** implementan la misma funcionalidad que los bloques sincronizados pero son más versátiles
- El paquete que contiene las clases de locks es **java.util.concurrent.locks**
- **Lock** es una interfaz y la clase por defecto que la implementa es **ReentrantLock**

Exclusión mutua con cerrojos (*Locks*)

```
public class IncLock {

   private double x = 0;
   private Lock xLock = new ReentrantLock();

   public void inc() {
      for (int i = 0; i < 10000000; i++) {
         xLock.lock();
         x = x + 1;
         xLock.unlock();
      }
   }

   //Crear 3 hilos que ejecutan inc() y esperar a que acaben

}
```

Creamos el cerrojo como un objeto de la clase ReentrantLock

Se adquiere y libera el cerrojo con métodos de la clase Lock

Exclusión mutua con cerrojos (*Locks*)

- En general es buena idea meter la **sección bajo exclusión mutua** en un bloque **try/finally** para liberar el cerrojo tanto si se produce una excepción como si no

```
lock.lock(); // Espera para adquirir el cerrojo
try {
    // Sección bajo exclusión mutua
} finally {
   lock.unlock()
}
```

Con los bloques y métodos sincronizados el cerrojo se libera automáticamente cuando se produce una excepción.

Exclusión mutua con cerrojos (*Locks*)

- Otros métodos de **Lock**
  - **lockInterruptibly():** Intenta adquirir el cerrojo pero eleva una InterruptedException si se interrumpe el hilo que está a la espera
  - **tryLock():** Adquiere el cerrojo si está disponible y devuelve true. Si no está disponible, devuelve false
  - **tryLock(long time, TimeUnit unit):** Espera para adquirir el cerrojo el tiempo indicado, si lo consigue devuelve true. Si no, devuelve false. Este método eleva InterruptedException si es interrumpido el hilo

Exclusión mutua con cerrojos (*Locks*)

- Métodos específicos de **ReentrantLock**
  - ReentrantLock(boolean fair)
  - getQueueLength()
  - isHeldByCurrentThread()
  - isLocked()
  - isFair()

Ejercicio 2

- Implementa el problema del museo con regalo del Tema2 usando únicamente la API de Java, sin usar **SimpleConcurrent**

Exclusión mutua con cerrojos (*Locks*)

- Con los cerrojos (*Locks*) también se pueden implementar otras formas de **exclusión mutua generalizada**
- La clase **ReentrantReadWriteLock** (que implementa el interfaz **ReadWriteLock)** permite la implementar la exclusión mutua extendida de **lectores y escritores**
- Los lectores usan el ***lock* de lectura** y los escritores el ***lock* de escritura**

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html

Exclusión mutua con cerrojos (*Locks*)


```
ReadWriteLock rw = new ReentrantReadWriteLock();
```

```
rw.readLock().lock();
try {
    // Lectura
} finally {
   rw.readLock().unlock();
}
```

Proceso lector

```
rw.writeLock().lock();
try {
    // Escritura
} finally {
   rw.writeLock().unlock();
}
```

Proceso escritor

Ejercicio 3

- Implementa el problema de los lectores y escritores con **ReadWriteLock** sin usar **SimpleConcurrent**

Sincronización de hilos

- Semáforos
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- **Sincronización condicional**
  - Monitores
  - Sincronización de barrera (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Sincronización condicional

- La **Sincronización Condicional** se produce cuando un proceso debe esperar a que se cumpla una cierta condición para proseguir su ejecución
- Esta condición sólo puede ser activada por otro proceso

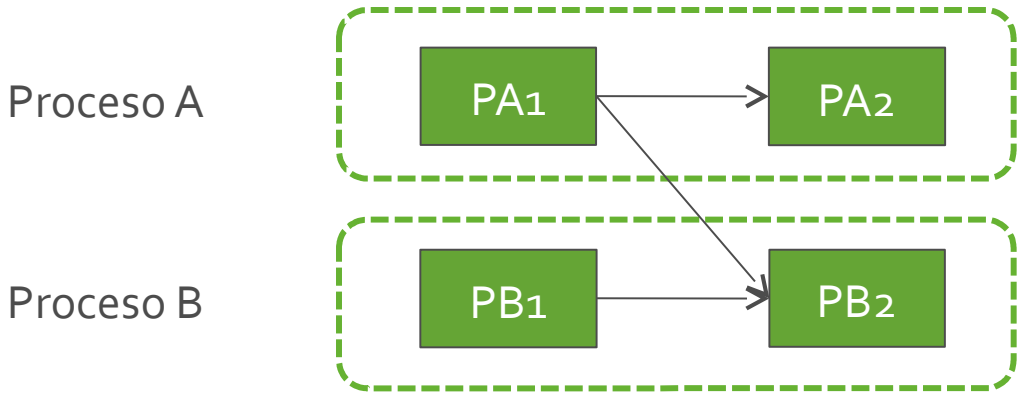


Diagrama de Precedencia

Sincronización condicional

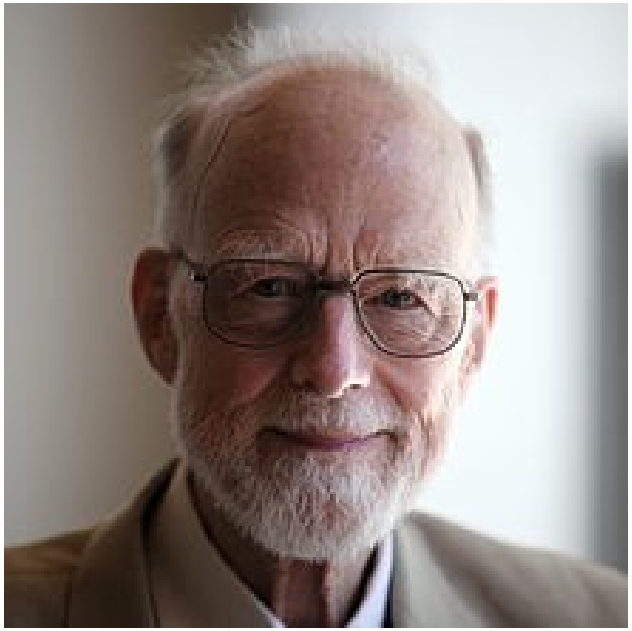
- En Java se puede implementar la sincronización condicional de múltiples formas
  - Usando **semáforos**
  - Usando **monitores** (con *synchronize* o *locks*)
  - Usando **clases** que implementan tipos específicos de sincronización
    - **CyclicBarrier**
    - **Exchanger**
    - **CountDownLatch**

Sincronización de hilos

- Semáforos
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - **Monitores**
  - Sincronización de barrera (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Monitores

- **Hoare** introdujo en 1974 una primitiva de Sincronización de hilos llamada **Monitor**
- El monitor se diseñó originalmente como un **tipo abstracto de datos** (**TAD**) ya que no se había desarrollado todavía la orientación a objetos



Tony Hoare
(1934-)

Inventó en método de ordenación *quicksort*

Monitores

- Un monitor implementado como un **TAD** tiene las siguientes características:
  - Proporciona **exclusión mutua** de forma automática en los **procedimientos** del TAD
  - Proporciona **sincronización condicional** en unos atributos del TAD llamados **variables de condición** (***condition***)
  - Las **variables de condición** se comportan como un semáforo inicializado a 0, permitiendo que un proceso quede **bloqueado** y otro proceso lo **desbloquee**

Monitores

- En **Java** lo **natural** hubiera sido implementar los monitores como **clases especiales** con todos sus **métodos** bajo **exclusión mutua** y unos atributos especiales para las condiciones (*conditions*)
- No obstante, para ofrecer **más versatilidad**, en Java los monitores no están asociados a las clases
- En Java los monitores se implementan en las zonas de **exclusión mutua** (bloques **sincronizados** o **cerrojos**)

Monitores

- Un monitor es esencialmente una forma de **bloquear a un hilo dentro de la exclusión mutua**
- Para poder **bloquearse** dentro de la exclusión mutua, el hilo usa un objeto especial llamado **condición** (*condition*)
- Para evitar que otros hilos queden **esperando indefinidamente** fuera de la exclusión mutua, cuando un hilo queda bloqueado en la *condition*, automáticamente **libera la exclusión mutua** (sin que nosotros tengamos que hacer nada)

Monitores

- Existen dos formas de implementar monitores en Java
  - **Bloques sincronizados (*synchronized*):** Las condiciones son métodos especiales del cerrojo
  - **Cerrojos (*locks*):** Las condiciones son objetos independientes.

Monitores con *synchronized*

- Los métodos para la gestión de las **variables de condición** están en la clase **Object** (padre de todas las clases en Java) :
  - **wait():** Un hilo que ejecuta este método quedará bloqueado. Este método eleva **InterruptedException**. Existen versiones de wait con tiempo de espera.
  - **notify():** Si un hilo ejecuta este método, desbloqueará a **uno de los hilos** que invocó el método wait() en ese objeto.
  - **notifyAll():** Si un hilo ejecuta este método, desbloqueará a **todos los hilos** que invocaron el método wait() en ese objeto

Primero intento incorrecto de Sincronización de barrera con Monitores

```
public class SincBarreraBien1 {

  static final int NPROCESOS = 3;
  static int nProcesos;
  static Semaphore sb;
  static Semaphore emNProcesos;

  public static void proceso() {
    print("A");
    emNProcesos.acquire();
    nProcesos++;
    if (nProcesos < NPROCESOS) {
      emNProcesos.release();
      sb.acquire();
    } else {
      sb.release(NPROCESOS-1);
      emNProcesos.release();
    }
    print("B");
  }

  public static void main(String[] args){
    nProcesos = 0;
    sb = new SimpleSemaphore(0);
    emNProcesos = new SimpleSemaphore(1);
    createThreads(NPROCESOS, "proceso");
    startThreadsAndWait();
  }
}
```

```
public class SincBarreraSynchronizedMal {

  static final int NPROCESOS = 3;
  private int nProcesos;
  private Object procesosLock;

  public void proceso() {
    System.out.println("A");

    synchronized (procesosLock) {
      nProcesos++;
      if (nProcesos < NPROCESOS) {
        try {
          procesosLock.wait();
        }catch(InterruptedException e){}
      } else {
        procesosLock.notifyAll();
      }
    }
    System.out.println("B");
  }

  public void exec(){
    nProcesos = 0;
    procesosLock = new Object();
    for (int i = 0; i < NPROCESOS; i++) {
      new Thread(()->proceso()).start();
    }
  }
}
```

1er intento incorrecto de

Sincronización de Barrera con Monitores

```
public static void proceso(){

  print("A");

  emNProcesos.acquire();
  nProcesos++;
  if(nProcesos < NPROCESOS){

    emNProcesos.release();
    sb.acquire();

  } else {
    sb.release(NPROCESOS-1);
    emNProcesos.release();
  }

  print("B");
}
```

```
public void proceso() {

  System.out.println("A");

  synchronized(procesosLock) {
    nProcesos++;
    if(nProcesos < NPROCESOS) {
      try {
        procesosLock.wait();
      }catch(InterExc e){}

    } else {
      procesosLock.notifyAll();
    }
  }

  System.out.println("B");
}
```

Monitores con *synchronized*

- Un hilo sólo puede ejecutar **wait()** si está en la zona de **exclusión mutua** (no desde fuera)
- Un **hilo desbloqueado** que esperaba en un **wait()**
  - Queda a la **espera** de entrar de nuevo en la exclusión mutua
  - Cuando lo **consigue**, continúa la ejecución en la **sentencia siguiente** al wait()
  - No tiene ninguna prioridad sobre los hilos que esperaban **fuera** de la **exclusión mutua,** se seleccionan de forma aleatoria

Monitores con *synchronized*

- **Activaciones inesperadas** o Despertares espúreos (*spurious wakeup)*
  - Un **hilo bloqueado** en un wait() se puede **desbloquear** incluso cuando ningún otro hilo ha ejecutado un notify(), notifyAll() o se haya interrumpido el hilo
  - Esto se debe a que las herramientas de sincronización de la **JVM** se implementan usando **rutinas del sistema operativo** (por ejemplo, las bibliotecas de hilos **POSIX**)
  - En estas herramientas se permiten las denominadas "**activaciones inesperadas**", aunque son raras, se producen ocasionalmente

http://en.wikipedia.org/wiki/Spurious_wakeup

Monitores con *synchronized*

- Debido a que se pueden producir **activaciones inesperadas**, cuando un hilo se desbloquea tiene que verificar si realmente debería haberse desbloqueado
- Para ello, las llamadas a **wait()** siempre deben realizarse dentro de un **bucle while** en el que el hilo se bloquea de nuevo si no debería estar desbloqueado

Monitores con *synchronized*

- Implementación correcta de sincronización de barrera (no cíclica) con monitores

```
public void proceso() {

  System.out.println("A");

  synchronized (procesosLock) {
    nProcesos++;
    if(nProcesos < NPROC) {
      try {
        while(nProcesos < NPROC) {
          procesosLock.wait();
        }
      } catch (IntException e) {}
    } else {
      procesosLock.notifyAll();
    }
  }

  System.out.println("B");
}
```

Se verifica que realmente se cumple la condición de desbloqueo

NOTA: Esta implementación no se puede usar para una barrera cíclica

Monitores con *synchronized*

- Se **desaconseja*** el uso de variables de condición en los monitores (wait, notify y notifyAll) debido a las **activaciones inesperadas** y a que puede ser **complejo** implementar operaciones básicas de sincronización
- Para realizar Sincronización de hilos, siempre que sea posible es **mejor** utilizar las herramientas de alto nivel como ***CyclicBarrier***, ***CountDownLatch*** y ***Exchanger***

* Item 69 en Joshua Bloch's "Effective Java Programming Language Guide, second edition", (Addison-Wesley,2008)

Monitores con cerrojos (*locks*)

- En la exclusión mutua obtenida con **cerrojos** (***locks***) también se pueden usar **condiciones**
- Con ***synchronized***, sólo se puede tener **una única variable** en la que bloquear y desbloquear hilos (la variable representada por el cerrojo)
- Con cerrojos, se pueden tener varias variables **variables condicionales independientes** para bloquear y desbloquear hilos

Monitores con cerrojos (*locks*)

- Cada variable de condición se crea invocando el método **newCondition()** en la clase **Lock**
- Devuelve un objeto que implementa en interfaz **Condition** con los métodos
  - **await():** Equivalente a wait. Tiene versiones con tiempo de espera y sin interrupciones
  - **signal():** Equivalente a notify()
  - **signalAll():** Equivalente a notifyAll()

Monitores con cerrojos (*locks*)

- En el método **await()** también pueden producirse **activaciones inesperadas**, así que también es necesario invocar el método dentro de un **bucle**
- En la medida de lo posible, siempre es **recomendable** utilizar herramientas de sincronización de más **alto nivel**

Monitores con cerrojos

- Implementación correcta de sincronización de barrera (no cíclica) con cerrojos

Se verifica que realmente se cumple la condición de desbloqueo

```
Lock lock = new ReentrantLock();
Condition sb = lock.newCondition();

public void proceso() {
  System.out.println("A");
   try {
     lock.lock();
     nProcesos++;
     if (nProcesos < NPROC) {
       try {
         while (nProcesos < NPROC) {
           sb.await();
         }
       } catch (IntExc e) {}
     } else {
       sb.signalAll();
     }
   } finally {
     lock.unlock();
   }
  System.out.println("B");
}
```

Ejercicio 4

- Implementar la comunicación con buffer del tema 2 con productores y consumidores usando ***Locks*** y ***Conditions***

Ejercicio 5

- Implementar el ejercicio de lectores escritores con ***Locks*** y ***Conditions***

Ejercicio 6

- Se dispone de un programa, **DataAccess**, que tiene un único hilo de ejecución (es *single threaded*) y accede a dos fuentes de datos
- Para mejorar el rendimiento del programa en sistemas con varios núcleos se ha decidido **paralelizarle**
- El problema es que las fuentes de datos no permiten acceso concurrente (**no son *thread-safe***), sólo pueden usarse por un único hilo cada vez

Ejercicio 6

code URJC

- En el programa DataAccess, a veces se necesita acceder a **DataSource1** y otras veces a **DataSource2**
- No obstante, en algunos casos, se puede utilizar indistintamente **cualquiera de los *data sources***
- Es decir, si un hilo quiere acceder a un *data source*, pero **no le importa** cual de ellos es, si alguno de los *data sources* están siendo **usado por otro hilo**, entonces podrá usar **el que esté libre**

Ejercicio 6

code URJC

- Se pide implementar una clase **DataSourceManager** que controle el acceso a los *data sources*
- Cuando un hilo quiera usar un data source concreto, invocará el método

  void accessDataSource(int dataSource)

- Cuando un hilo quiera usar cualquier *data source* invocará el método que devuelve el id del *data source* libre

  int accessAnyDataSource()

- Cuando cualquier hilo termine de usar el data source, deberá invocar el método

  void freeDataSource(int dataSource)

Ejercicio 6

- Estructura de la clase

```
public class DataSourceManager {

   //Atributos necesarios

   public DataSourceManager(){
      //Inicializaciones necesarias
   }

   public void accessDataSource(int dataSource){
      //Este método será invocado cuando se quiera
      //usar un DataSource concreto.
      //Se bloqueará hasta que ese dataSource esté
      //libre.
   }

   public int accessAnyDataSource(){
      //Este método será invocado cuando se quiera
      //usar cualquier DataSource.
      //Se bloqueará hasta que cualquiera de los
      //DataSource esté libre.
   }

   public void freeDataSource(int dataSource)
      //Este método se utilizará para liberar el
      //DataSource para que otros hilos lo puedan
      //usar.
   }
}
```

Ejercicio 6

code URJC

- Uso de la clase **DataSourceManager**
  - Creación

```
DataSourceManager dsManager = new DataSourceManager();
```

  - Proceso que sólo puede usar el **data source 1**

```
dsManager.accessDataSource(1);
//Uso el data source 1
dsManager.freeDataSource(1);
```

  - Proceso que puede usar **cualquier data source**

```
int ds = dsManager.accessAnyDataSource();
//Uso el data source ds
dsManager.freeDataSource(ds);
```

Ejercicio 6

- Los métodos de la clase **DataSourceManager** tiene que ser ***thread-safe*** porque sus métodos serán invocados por **diferentes hilos de ejecución**
- La **asignación** de los ***data source*** a los hilos debe realizarse en **orden de llegada (FIFO)**
- Cuando un ***data source*** queda **libre**, primero se intentará asignar a un proceso que sólo pueda acceder a ese ***data source***. Si no hay ningún proceso de ese tipo esperando, se asignará a un proceso que pueda utilizar cualquier ***data source***. En otro caso, el ***data source*** quedará **libre**

Sincronización de hilos

- Semáforos
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - Monitores
  - **Sincronización de barrera (*CyclicBarrier*)**
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Sincronización de barrera (*CyclicBarrier*)

- La clase **CyclicBarrier** implementa un sincronización de barrera cíclica
- Al construir un objeto se indica el **número de hilos** que tienen que llegar a la barrera antes de poder continuar
- Opcionalmente se puede pasar un **código** (en forma de **Runnable***)* que se ejecutará cuando todos los procesos han llegado a la barrera y antes de que reanuden la ejecución

Sincronización de barrera (*CyclicBarrier*)

- **CyclicBarrier(int parties)**: Número de hilos de la barrera
- **CyclicBarrier(int parties, Runnable r)**: Número de hilos de la barrera y código ejecutado al sincronizar los procesos
- **await()**: bloquea el hilo hasta que lleguen los demás hilos
- **getNumberWaiting()**: Número de bloqueados

Ejercicio 7

code URJC

- Implementar el ejercicio de descarga de ficheros usando **CyclicBarrier** y las demás herramientas de sincronización que sean necesarias
- No se permite el uso de SimpleConcurrent

Sincronización de hilos

- Semáforos
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - Monitores
  - Sincronización de barrera (*CyclicBarrier*)
  - **Intercambiador (*Exchanger*)**
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- Conclusiones

Intercambiador (*Exchanger*)

code URJC

- Un **intercambiador** es un objeto de la clase **Exchanger** que permite que dos hilos intercambien objetos entre sí
- Ambos hilos invocan el método

```
public V exchange(V x) throws InterruptedException
```

- El **primer** hilo que ejecuta **exchange(…)** queda **bloqueado** hasta que el otro hilo ejecuta también ese método
- Cuando el segundo hilo ejecuta **exchange(…)** ambos hilos **intercambian** los valores pasados como parámetro y continúan su ejecución

Intercambiador (*Exchanger*)

- Aunque el intercambio es bidireccional, esta clase permite implementar un esquema de **productor/consumidor** sin buffer si uno de los dos hilos pasa **null**
- El **productor** queda bloqueado hasta que el consumidor obtiene el valor
- El **consumidor** queda bloqueado hasta que el productor ha producido el valor

Intercambiador (*Exchanger*)

- Como en la mayoría de los métodos bloqueantes de Java, los métodos **exchange(…)**:
  - Elevan la InterruptedException
  - Existen versiones en las que se puede especificar un tiempo máximo de espera

Intercambiador (*Exchanger*)

```
public class ProdConsInf {
  private Exchanger<Double> exchanger = new Exchanger<Double>();

  public void productor() throws InterruptedException {
    double productoL = 0;
    for(int i=0; i<10; i++) {
      productoL++;
      Thread.sleep(1000);
      exchanger.exchange(productoL);
    }
  }

  public void consumidor() throws InterruptedException {
    for(int i=0; i<10; i++) {
      double producto = exchanger.exchange(null);
      System.out.println("Producto: " + producto);
      Thread.sleep(1000);
    }
  }

  //Se crean inician dos hilos que llaman
  //a productor() y consumidor()
}
```

Se ignora el intercambio en un sentido

Ejercicio 8

- Implementar el ejercicio de cliente servidor con proxy del Tema 2 usando objetos **Exchanger**

Sincronización de hilos

- Semáforos
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - Monitores
  - Sincronización de barrera (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - **Cerrojo de cuenta atrás (*CountDownLatch*)**
- Conclusiones

Cerrojo de cuenta atrás (*CountDownLatch*)

- Un **CountDownLatch** permite implementar una **cuenta atrás** (como las salidas de las carreras o de los cohetes espaciales)
- Los hilos que invoquen **await()** quedarán bloqueados y podrán continuar cuando la cuenta llegue a cero (como en las carreras)
- La salida de la carrera (el desbloqueo de los hilos) se produce cuando se invoca **countDown()** tantas veces como se haya especificado en el constructor del objeto (p.e: 3, 2, 1… Go!)

Cerrojo de cuenta atrás (*CountDownLatch*)

```
public class CountDownLatchDemo {

  private CountDownLatch latch = new CountDownLatch(4);

  public void corredor() throws InterruptedException {
    System.out.println("Preparado");
    latch.await();
    System.out.println("Corriendo");
  }

  public void juez() throws InterruptedException {
    for (int i=3; i >= 0; i--) {
      System.out.println(i);
      latch.countDown();
      Thread.sleep(500);
    }
  }

  //Crear e iniciar 3 hilos que ejecutan corredor()
  //Crear e iniciar 1 hilo que ejecuta juez()
}
```

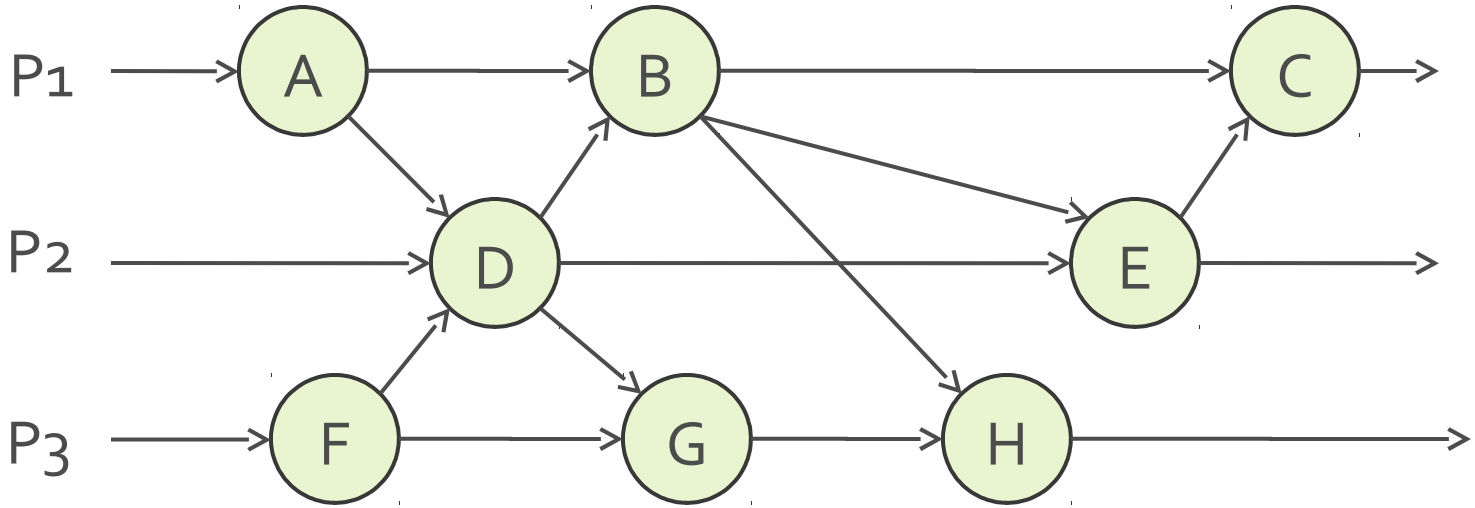
Se crea el objeto indicando el número de countDown necesarios para desbloquear a los await()

Cerrojo de cuenta atrás (*CountDownLatch*)

- Un objeto de la clase **CountDownLatch** sólo se puede utilizar para un única cuenta atrás
- La cuenta atrás **no se puede reiniciar**
- Se puede usar con los siguientes **esquemas**:
  - Varios hilos esperan y un hilo desbloquea a todos los demás (carrera)
  - Un hilo espera y varios hilos invocan **countDown()**. Cuando todos los hilos han dado la señal, el hilo que espera se desbloquea

Ejercicio 9

- Implementar el siguiente diagrama de precedencia con objetos **CountDownLatch**



Sincronización de hilos

- Semáforos
- Exclusión mutua
  - Exclusión mutua con *synchronized*
  - Exclusión mutua con cerrojos (*Locks*)
- Sincronización condicional
  - Monitores
  - Sincronización de barrera (*CyclicBarrier*)
  - Intercambiador (*Exchanger*)
  - Cerrojo de cuenta atrás (*CountDownLatch*)
- **Conclusiones**

Conclusiones

- Hemos visto que en **Java** podemos utilizar la **herramienta** de más **bajo nivel** de sincronización de hilos: El **semáforo**
- Pero también hemos visto herramientas **especializadas** para implementar los dos tipos **básicos** de **sincronización**
  - Exclusión Mutua
  - Sincronización condicional

Conclusiones

- **Exclusión mutua**
  - Soporte en el código fuente (*synchronized*)
    - Más compacta y fácil de usar
  - Implementada con clases de la librería (*Locks*)
    - Lock, ReadWriteLock
    - Más potente y versátil pero más difícil de usar (try/finally)

Conclusiones

- **Sincronización condicional**
  - Monitores
    - Bloqueo dentro de la exclusión mutua que libera automáticamente dicha exclusión mutua
    - Los despertares inesperados y la espera en bucle
  - Herramientas específicas
    - Sincronización de barrera (*CyclicBarrier*)
    - Intercambiador (*Exchanger*)
    - Cerrojo de cuenta atrás (*CountDownLatch*)

Conclusiones

- **Recomendaciones**
  - Siempre que sea posible, hay que usar las herramientas de más **alto nivel** para la **sincronización de hilos**
  - Hacen el código **más legible** y son más fáciles de entender, implementar y depurar
  - Si se usan **monitores**, pensar siempre en los **despertares inesperados**