

Programación Concurrente en Java

PROGRAMACIÓN CONCURRENTE – TEMA 4



Universidad
Rey Juan Carlos

Sincronización de Procesos

PROGRAMACIÓN CONCURRENTE EN JAVA – TEMA 4.2



Universidad
Rey Juan Carlos

Sincronización de Procesos

PROGRAMACIÓN CONCURRENTE EN JAVA

- Introducción a la PC en Java
- **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

SINCRONIZACIÓN DE PROCESOS

- 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/escriptores**

Exclusión Mutua

SINCRONIZACIÓN DE PROCESOS

- Se puede implementar con semáforos
- En Java existen mecanismos específicos para implementar exclusión mutua:
 - *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**

Exclusión mutua con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- La **exclusión mutua** es un tipo de sincronización tan **básico** 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.
- El hilo que **posee el cerrojo** es el que está en la zona de exclusión mutua (ejecuta las sentencias de la sección crítica)

Exclusión mutua con *synchronized*

SINCRONIZACIÓN DE PROCESOS

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

Exclusión mutua con *synchronized*

SINCRONIZACIÓN DE PROCESOS

```
public class IncSynchronized {
```

```
    private static double x = 0;
```

```
    private static Object xLock = new Object();
```

El cerrojo puede ser un objeto de cualquier clase

```
    public static void inc() {
```

```
        for (int i = 0; i < 10000000; i++) {
```

```
            synchronized (xLock) {
```

```
                x = x + 1;
```

```
            }
```

```
        }
```

```
    }
```

En la sentencia sincronizada se indica el cerrojo

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

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

```
        ...
```

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

```
    }
```

```
}
```


Exclusión mutua con *synchronized*

SINCRONIZACIÓN DE PROCESOS

```
public class IncDecSynchronized {  
  
    private static double x = 0;  
    private static Object xLock = new Object();  
  
    public static void inc() {  
        for (int i = 0; i < 10000000; i++) {  
            synchronized (xLock) {  
                x = x + 1;  
            }  
        }  
    }  
  
    public static void dec() {  
        for (int i = 0; i < 10000000; i++) {  
            synchronized (xLock) {  
                x = x - 1;  
            }  
        }  
    }  
    ...  
}
```

El mismo cerrojo se puede usar en varias sentencias sincronizadas que identifican secciones críticas diferentes de la misma exclusión mutua

Exclusión mutua con *synchronized*

SINCRONIZACIÓN DE PROCESOS

```
public class IncSynchronized2 {  
  
    private static double x = 0;  
    private static Object xLock = new Object();  
  
    private static double y = 0;  
    private static Object yLock = new Object();  
  
    public static void incX() {  
        for (int i = 0; i < 10000000; i++) {  
            synchronized (xLock) {  
                x = x + 1;  
            }  
        }  
    }  
  
    public static 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*

SINCRONIZACIÓN DE PROCESOS

- Es muy habitual que un objeto tenga 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** actúa como **cerrojo**
- En los métodos estáticos, el objeto que representa la **clase** actúa como cerrojo

Exclusión mutua con *synchronized*

SINCRONIZACIÓN DE PROCESOS

```
class Counter {  
    private int x = 0;  
    public synchronized void inc() {  
        x = x + 1;  
    }  
    public int getValue() {  
        return x;  
    }  
}  
  
public class IncSynchronizedMethod {  
  
    private static Counter c = new Counter();  
  
    public static void inc() {  
        for (int i = 0; i < 10000000; i++) {  
            c.inc();  
        }  
    }  
    ...  
}
```

El propio objeto actúa
como cerrojo para los
métodos sincronizados

La llamada al
método no cambia

Exclusión mutua con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- Las sentencias y métodos **sincronizados** son **reentrantes**
- Si un hilo que ha **adquirido el cerrojo** en un método **sincronizado**, llama a otro método **sincronizado** vuelve a coger el cerrojo, no se queda bloqueado
- 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 Procesos

PROGRAMACIÓN CONCURRENTE EN JAVA

- Introducción a la PC en Java
- 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)

SINCRONIZACIÓN DE PROCESOS

- Las sentencias sincronizadas son muy sencillas 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 en el caso de los lectores escritores

Exclusión mutua con cerrojos (*Locks*)

SINCRONIZACIÓN DE PROCESOS

- Los objetos **Lock** implementan la misma funcionalidad que las sentencias sincronizadas pero con más posibilidades
- El paquete de los 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)

SINCRONIZACIÓN DE PROCESOS

```
public class IncLock {  
  
    private static double x = 0;  
    private static Lock xLock = new ReentrantLock();  
  
    public static void inc() {  
        for (int i = 0; i < 10000000; i++) {  
            xLock.lock();  
            x = x + 1;  
            xLock.unlock();  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException{  
        //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)

SINCRONIZACIÓN DE PROCESOS

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

```
xLock.lock(); // Espera para adquirir el cerrojo
try {
    // Sección crítica
} finally {
    lock.unlock()
}
```

Exclusión mutua con cerrojos (Locks)

SINCRONIZACIÓN DE PROCESOS

- 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*)

SINCRONIZACIÓN DE PROCESOS

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

Exclusión mutua con cerrojos (*Locks*)

SINCRONIZACIÓN DE PROCESOS

- La interfaz **ReadWriteLock** permite la implementación de la exclusión mutua extendida de lectores y escritores
- Permite acceder al *lock* de lectura y al *lock* de escritura por separado
- Su implementación por defecto es **ReentrantReadWriteLock**

Sincronización de Procesos

PROGRAMACIÓN CONCURRENT EN JAVA

- Introducción a la PC en Java
- 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

Sincronización condicional

SINCRONIZACIÓN DE PROCESOS

- 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

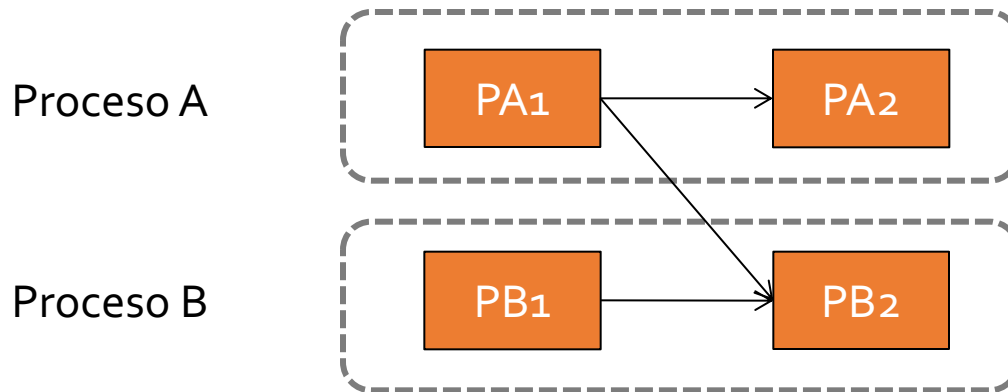


Diagrama de Precedencia

Sincronización de Procesos

PROGRAMACIÓN CONCURRENT EN JAVA

- Introducción a la PC en Java
- 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

Sincronización condicional

SINCRONIZACIÓN DE PROCESOS

- 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**

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



C. A. R. Hoare
(1934-)

Inventó en método de
ordenación *quicksort*

- 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 cero, permitiendo que un proceso quede bloqueado y otro proceso lo desbloquee

- En **Java** lo **natural** hubiera sido implementar los monitores como **clases** con todos sus **métodos** bajo **exclusión mutua** y definir una clase para las condiciones
- No obstante, para ofrecer **más versatilidad**, en Java los monitores son independientes de las clases y funcionan con la exclusión mutua definida por
 - Métodos o sentencias sincronizadas (*synchronized*)
 - Cerrojos (*locks*)

Monitores con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- La **variables de condición** diseñadas por Hoare tienen esa característica, cuando un hilo se bloquea en ellas, se **libera la exclusión mutua** de forma automática
- En **Java** las variables de condición se implementan con **métodos** en el objeto que actúa como **cerrojo** (*lock*) en el bloque sincronizado

Monitores con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- Métodos de la clase Object que implementan las variables de condición de los monitores:
 - **wait()**: Un hilo que ejecute 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()
 - **notifyAll()**: Si un hilo ejecuta este método, desbloqueará a **todos los hilos** que invocaron el método wait()

Estructura sintáctica de un Monitor en JAVA

SINCRONIZACIÓN DE PROCESOS

```
public class Monitor {
    // Recursos compartido
    private ...;
    private ...;

    // Cosntructor por defecto
    public Monitor(...) {
        ...
    }
    // Métodos exportados en exclusión mútua
    public synchronized void metodo1() throws InterruptedException {
        while (!condicion) {
            ...
            this.wait();
            ...
        }
        ...
        this.notifyAll();
        ...
    }
}
// Métodos privados o que no deben estar en excusión mútla

    public synchronized String toString() {
        ...
    }
}
```

Monitores con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- Al existir sólo una variable de condición (`this`), es recomendable utilizar a `notifyAll()`, para que todos los hilos comprueben la condición que los bloqueó
- No es posible identificar un hilo en particular. Por lo tanto
 - Es aconsejable bloquear a los hilos con una condición de guarda (todos los hilos despertados y se bloquearan de nuevo salvo el que cumpla la condición

```
while (condicion) {  
    try {  
        this.wait();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```


Monitores con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- Implementar un programa concurrente que gestione la venta de entradas online
 - Un **monitor** simple que contenga una única lista (**como recurso compartido**) de solicitudes
 - Tres métodos en exclusión mutua
 - Comprar: añadir una persona a la lista de solicitudes
 - Vender: asignarle la entrada y sacarle de la lista de solicitudes
 - Mostrar el estado del recurso y el dinero ganado
- Implementar dos clases que contengan un **hilo** que, usando el **monitor**, compre y venda respectivamente, actualizando el recurso compartido
- Implementar un **programa principal** que simule 1 grupo de gente comprando y 4 proceso que los atiendan.

Monitores con *synchronized*

SINCRONIZACIÓN DE PROCESOS

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

Monitores con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- Debido a que se pueden producir **activaciones inesperadas**, tenemos que implementar un mecanismo de protección en los `wait()`
- Las llamadas a **`wait()`** siempre deben realizarse dentro de un **bucle `while`** de forma que el hilo se bloquee hasta que **no se cumpla** la condición de **desbloqueo**

Monitores con *synchronized*

Se verifica que realmente se cumple la condición de desbloqueo

```
public class SincBarreraSynchronizedBien {  
  
    static final int NPROCESOS = 3;  
    static volatile int nProcesos;  
    static Object procesosLock;  
  
    public static void proceso() {  
        System.out.println("A");  
  
        synchronized (procesosLock) {  
            nProcesos++;  
            if (nProcesos < NPROCESOS) {  
                try {  
                    while (nProcesos < NPROCESOS) {  
                        procesosLock.wait();  
                    }  
                } catch (InterruptedException e) {}  
            } else {  
                procesosLock.notifyAll();  
            }  
        }  
        System.out.println("B");  
    }  
  
    public static void main(String[] args) {  
        //Crear NPROCESOS que ejecutan proceso()  
    }  
}
```

Monitores con *synchronized*

SINCRONIZACIÓN DE PROCESOS

- 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 procesos, 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*)

SINCRONIZACIÓN DE PROCESOS

- En la exclusión mutua obtenida con **cerrojos** (***locks***) también se puede implementar sincronización condicional
- Se pueden crear **tantas variables condicionales** como sea necesario
- Con ***synchronized***, sólo se puede tener **una única variable** en la que bloquear y desbloquear hilos (la variable representada por el cerrojo)

Monitores con cerrojos (*locks*)

SINCRONIZACIÓN DE PROCESOS

- 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*)

SINCRONIZACIÓN DE PROCESOS

- 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 (*locks*)

SINCRONIZACIÓN DE PROCESOS

- Implementar la clase Servidor (monitor) para controlar el acceso a un servidor heterogéneo
 - Atiende peticiones web
 - Realiza operaciones de mantenimiento
- Utilizar un monitor con 1 Lock, 2 condiciones y 2 variables enteras que controlan en número de operaciones de mantenimiento y de peticiones web
- Métodos del monitor. Empezar y terminar tanto peticiones web como operaciones de mantenimiento
- Se producen bloqueos si:
 - Hay peticiones web, no se puede hacer mantenimiento
 - Si hay operaciones de mantenimiento, las peticiones web se bloquean

Sincronización de Procesos

PROGRAMACIÓN CONCURRENTE EN JAVA

- Introducción a la PC en Java
- Exclusión mutua
 - Exclusión mutua con *synchronized*
 - Exclusión mutua con cerrojos (*Locks*)
- **Sincronización condicional**
 - Monitores
 - **Intercambiador (Exchanger)**
 - Cerrojo de cuenta atrás (*CountDownLatch*)
 - Sincronización de berrara (*CyclicBarrier*)
- Conclusiones

Intercambiador (*Exchanger*)

SINCRONIZACIÓN DE PROCESOS

- 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*)

SINCRONIZACIÓN DE PROCESOS

- Aunque el intercambio es bidireccional, esta clase permite implementar un esquema de **productor/consumidor** sin buffer (usando sólo un sentido de la comunicación)
- 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*)

SINCRONIZACIÓN DE PROCESOS

- 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)

SINCRONIZACIÓN DE PROCESOS

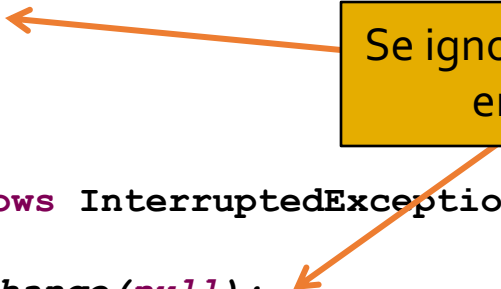
```
public class ProdConsInf {
    static Exchanger<Double> exchanger = new Exchanger<Double>();

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

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

    public static void main(String[] args) {
        //Se crean e inician dos hilos que llaman
        //a productor() y consumidor()
    }
}
```

Se ignora el intercambio
en un sentido



Sincronización de Procesos

PROGRAMACIÓN CONCURRENTE EN JAVA

- Introducción a la PC en Java
- Exclusión mutua
 - Exclusión mutua con *synchronized*
 - Exclusión mutua con cerrojos (*Locks*)
- **Sincronización condicional**
 - Monitores
 - Intercambiador (Exchanger)
 - **Cerrojo de cuenta atrás (CountDownLatch)**
 - Sincronización de berrara (CyclicBarrier)
- Conclusiones

Cerrojo de cuenta atrás (*CountDownLatch*)

SINCRONIZACIÓN DE PROCESOS

- Un objeto de la clase **CountDownLatch** permite implementar una **cuenta atrás** (como las salidas de las carreras)
- Uno o más hilos invocan **await()** y eso le/s bloquea a la espera de que le/s den la salida
- 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

Cerrojo de cuenta atrás (*CountDownLatch*)

SINCRONIZACIÓN DE PROCESOS

```
public class CountDownLatchDemo {  
  
    static CountDownLatch latch = new CountDownLatch(4);  
  
    public static void runner() throws InterruptedException {  
        System.out.println("Ready");  
        latch.await();  
        System.out.println("Running");  
    }  
  
    public static void judge() throws InterruptedException {  
        for (int i=3; i >= 0; i--) {  
            System.out.println(i);  
            latch.countDown();  
            Thread.sleep(500);  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        //Create and start 3 threads executing runner()  
        //Create and start 1 thread executing judge()  
    }  
}
```

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

Cerrojo de cuenta atrás (*CountDownLatch*)

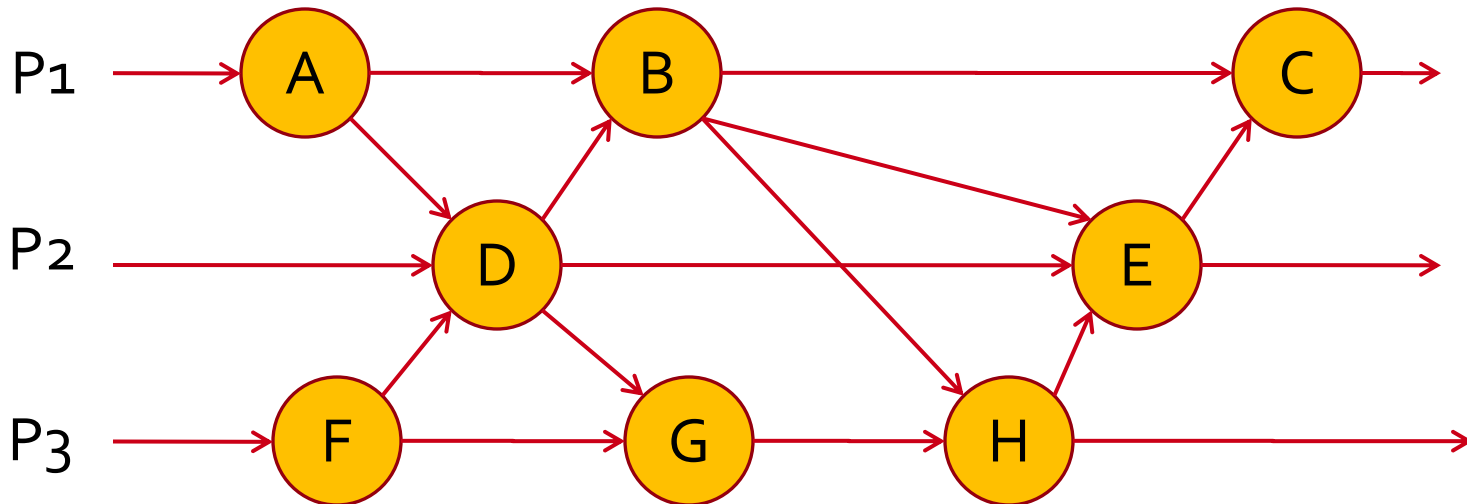
SINCRONIZACIÓN DE PROCESOS

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

Ejercicio 4.7

SINCRONIZACIÓN DE PROCESOS

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



Sincronización de Procesos

PROGRAMACIÓN CONCURRENT EN JAVA

- Introducción a la PC en Java
- Exclusión mutua
 - Exclusión mutua con *synchronized*
 - Exclusión mutua con cerrojos (*Locks*)
- **Sincronización condicional**
 - Monitores
 - Intercambiador (*Exchanger*)
 - Cerrojo de cuenta atrás (*CountDownLatch*)
 - **Sincronización de berrara (*CyclicBarrier*)**
- Conclusiones

- En Java es siempre recomendable utilizar las herramientas de sincronización de procesos de alto nivel presentes en la librería
- La clase ***CyclicBarrier*** implementa una sincronización de barrera cíclica
- Al construir un objeto se indica el número de hilos de la barrera y un ***Runnable*** que se ejecutará cuando todos los procesos han llegado a 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

- Implementar un programa concurrente donde NPROC hilos escriban una A y se queden bloqueados. El último hilo escribirá (además de la A) un *.
- Utilizar CyclicBarrier para determinar la sincronización de barrera.
- El programa debe ejecutarse de forma indefinida

Sincronización de Procesos

PROGRAMACIÓN CONCURRENTE EN JAVA

- Introducción a la PC en Java
- 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**

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

- **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*)

• **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*)

• **Recomendaciones**

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