



UD5. Otras herramientas de sincronización



Concurrencia y Sistemas Distribuidos



Objetivos de la Unidad Didáctica

- ▶ Conocer otras herramientas de sincronización
 - ▶ Semáforos
 - ▶ Barreras
- ▶ Utilizar la biblioteca *java.util.concurrent*, que ofrece herramientas de sincronización de alto nivel
 - ▶ Locks
 - ▶ Variables condición
 - ▶ Colecciones concurrentes
 - ▶ Variables atómicas
 - ▶ Sincronización: Semáforos y Barreras
 - ▶ Entorno para la gestión de hilos (Executors)



Bibliografía

- ▶ Java concurrency: *High-level concurrency objects* → *Lock Objects, Executors, Concurrent Collections, Atomic Variables*
<http://docs.oracle.com/javase/tutorial/essential/concurrency/>
 - ▶ También disponible en Poliformat asignatura: documento “Java Concurrency”
- ▶ Capítulo 5 libro base asignatura (“Concurrencia y Sistemas Distribuidos”).
- ▶ Vídeos unidad 5



Contenido

- ▶ Inconvenientes de las primitivas básicas de Java
- ▶ Otras herramientas de sincronización (y qué nos ofrece *java.util.concurrent*)
 - ▶ Locks y variables condición
 - ▶ Colecciones concurrentes
 - ▶ Variables atómicas
 - ▶ Semáforos y barreras
 - ▶ Entorno para la gestión de hilos (Executor)



Inconvenientes de las primitivas básicas de Java

- ▶ **Actividad:** Enumere los distintos inconvenientes que ofrecen las primitivas básicas de Java (i.e. monitores) para la sincronización de tareas.
 - ▶ a) Limitaciones de las primitivas básicas de Java relacionadas con la exclusión mutua
 - ▶ b) Limitaciones de las primitivas básicas de Java relacionadas con la sincronización condicional



Inconvenientes de las primitivas básicas de Java

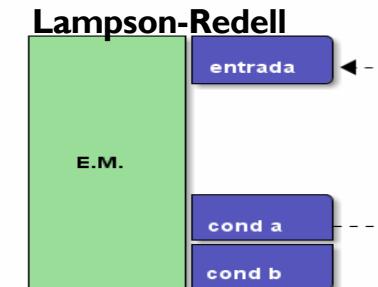
- ▶ a) Limitaciones primitivas básicas de Java respecto a la **exclusión mutua**:
 - I. No se puede establecer un plazo máximo de espera a la hora de “solicitar” la entrada al monitor
 - ▶ Si el monitor está ocupado, el hilo se queda esperando y no se le puede interrumpir voluntariamente.
 2. No se puede preguntar por el estado del monitor antes de solicitar el acceso al mismo.
 - ▶ En ocasiones interesa consultar si el lock (o monitor) está libre u ocupado sin necesidad de bloquearse (Ejemplo: usando *tryLock*)
 - Especialmente útil para evitar posibles interbloqueos
 - Con métodos *synchronized* esta consulta no es posible.



Inconvenientes de las primitivas básicas de Java

3. Las herramientas que garantizan exclusión mutua están orientadas a bloques
 - ▶ Es decir, están orientadas a métodos completos o a secciones de códigos más pequeñas
 - ▶ No podemos adquirir un *lock* en un método y liberarlo dentro de otro método
4. No podemos extender su semántica.
 - ▶ Ejemplo: en problema lectores-escritores, no podemos utilizar estas construcciones para definir:
 - exclusión mutua entre hilos escritores o entre escritores y lectores, pero no entre múltiples lectores.

- ▶ b) Limitaciones relacionadas con la **sincronización condicional**:
 - I. Solo podrá existir una única condición en cada monitor.
 - ▶ Todos los hilos que se suspenden en un monitor van a parar a una misma cola, con independencia del motivo (condición) por el que se suspenden.
 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.





Biblioteca `java.util.concurrent`

- ▶ J2SE 5.0 introduce el paquete **java.util.concurrent**
 - ▶ Ofrece construcciones de alto nivel
 - ▶ Garantizan:
 - ▶ + Productividad
 - Facilita desarrollo/mantenimiento de aplicaciones concurrentes de calidad
 - ▶ + Prestaciones
 - Más eficiente y escalable que las implementaciones típicas
 - ▶ + Fiabilidad
 - Comprobaciones extensivas contra interbloqueos, condiciones de carrera, inanición



Herramientas de java.util.concurrent

- ▶ La biblioteca incluye varias herramientas útiles:

- ▶ Locks
- ▶ Variables condición
- ▶ Colecciones concurrentes
- ▶ Variables atómicas
- ▶ Sincronización: Semáforos y Barreras
- ▶ Entorno para la gestión de hilos (Executor)



- ▶ **java.util.concurrent.locks** proporciona diferentes clases e interfaces para la gestión y desarrollo de múltiples tipos de *locks*.
- ▶ Características:
 - ▶ Permite especificar si se requiere una gestión equitativa (**fair**) de la cola de espera mantenida por el lock
 - ▶ Se facilitan distintos tipos de locks, con semántica diferente.
 - ▶ Ejemplo: locks orientados a exclusión mutua, locks que resuelven el problema de lectores-escritores.
 - ▶ Ofrece un método **tryLock()** que no suspende al invocador si el lock ya ha sido cerrado por otro hilo.
 - ▶ Se rompe así la condición de retención y espera (vista en la Unidad 4 – interbloqueos).



Locks: ReentrantLocks

- ▶ Ejemplo de clase lock ofrecida: **ReentrantLock**
 - ▶ Implementa un lock reentrante:
 - ▶ Dentro de la sección de código protegida por el lock se podrá volver a utilizar ese mismo lock sin que haya problemas de bloqueo.
 - ▶ Resuelve las limitaciones de la sentencia 'synchronized'
 - ▶ Permite especificar un **plazo máximo de espera** para obtener el lock → *tryLock()* con *timeout*.
 - ▶ Definir distintas variables condición → método *newCondition()*
 - ▶ Cerrar y abrir los locks en diferentes métodos de la aplicación → *lock()*, *unlock()*
 - No restringe el uso de los objetos lock a la construcción de monitores.
 - ▶ Soporte para interrupciones sobre hilos que esperan adquirir un lock
 - Se pueden interrumpir las esperas usando *Thread.interrupt()*



Locks: ReentrantLocks

- ▶ Inconvenientes:
 - ▶ El lock no se abre por defecto al terminar un bloque de código. El programador debe abrirlo explícitamente.
 - ▶ Asegurarse de **abrir el lock** correspondiente al liberar un recurso que se use en exclusión mutua.
 - ▶ Controlar las excepciones que se puedan generar dentro de una sección crítica protegida por un **ReentrantLock**
 - Asegurarse de **abrir el lock** en el código de la excepción.
 - El código asociado a la excepción debe asegurar que se ejecute el método *unlock()* sobre dicho *ReentrantLock*.



Locks: ReentrantLocks

- ▶ Recomendación: seguir esta estructura de los protocolos de entrada y salida

```
Lock x = new ReentrantLock();
x.lock();
try {
```

Protocolo de entrada

.. //Sección crítica (donde se actualiza el estado del objeto)

```
} finally {
    x.unlock();
}
```

Protocolo de salida



Variables condición

- ▶ Recordemos que la clase **Object** incluye métodos especiales para sincronización entre hilos: *wait*, *notify*, *notifyAll*.
 - ▶ Es fácil usarlos de forma incorrecta
 - ▶ Existe interacción entre notificación y locking
 - ▶ Para esperar o notificar hay que cerrar previamente el “lock” asociado al objeto
- ▶ Con *java.util.concurrent* tenemos construcciones de mayor nivel, y por tanto se reduce la necesidad de utilizar *wait/notify/notifyAll*.



Variables condición

- ▶ La interfaz **Condition** permite declarar cualquier número de **variables condición** en un lock
 - ▶ En muchos casos utilizar múltiples variables condición permite un código más legible y eficiente
 - Ej. en productor/consumidor
- ▶ El objeto lock actúa como una factoría para variables condición asociadas al mismo
 - ▶ Sólo podemos definir variables **condición** dentro de un lock



Variables condición

- ▶ Método **newCondition()** de la clase **ReentrantLock**:
 - ▶ permite generar todas las colas de espera (condiciones) que resulten necesarias
 - ▶ devuelve un objeto que implementa la interfaz **Condition**
- ▶ Métodos de la interfaz **Condition**:
 - ▶ **await()**: suspende a un hilo en la condición
 - ▶ **signal()**: notifica a **uno** de los hilos que estuviera esperándolo
 - ▶ **signalAll()**: notifica a **todos** los hilos suspendidos en la condición
- ▶ En los monitores básicos, los métodos similares son: **wait()**, **notify()**, **notifyAll()**



Ejemplo.- Lock y condition

```
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();  
    } finally {lock.unlock();}  
    return x;  
}  
  
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();}  
}
```

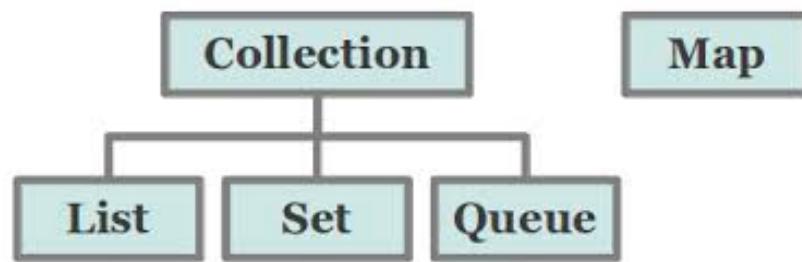


Herramientas de java.util.concurrent

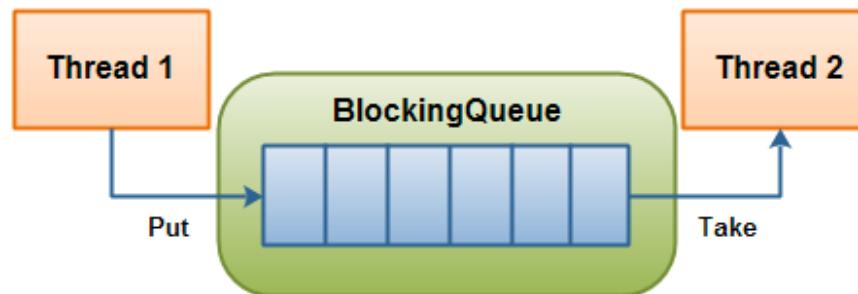
- ▶ La biblioteca incluye varias herramientas útiles:

- ▶ Locks
- ▶ Variables condición
- ▶ Colecciones concurrentes
- ▶ Variables atómicas
- ▶ Sincronización: Semáforos y Barreras
- ▶ Entorno para la gestión de hilos (Executor)

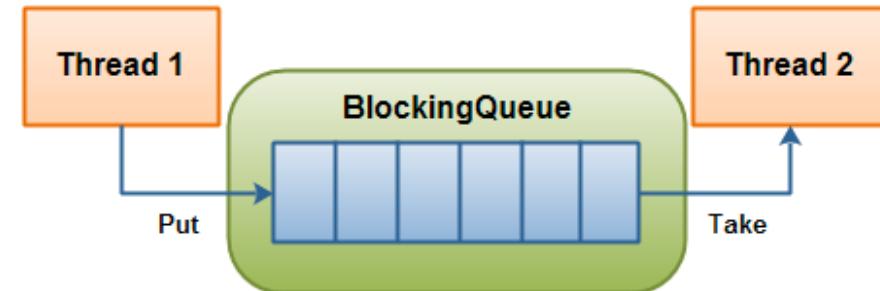
- ▶ Muchas aplicaciones requieren acceso a colecciones de objetos
 - ▶ *Interfaz List* (*lista dinámica*) → *ArrayList*, *LinkedList*, *Vector*...
 - ▶ *Interfaz Map* (*vector asociativo*) → *EnumMap*, *HashMap*, *TreeMap*
 - ▶ *Interfaz Queue* (*política FIFO*)
- ▶ No son **thread-safe** (*no se pueden compartir de forma segura entre varios hilos*)



- ▶ *Java.util.concurrent* incluye versiones **Thread-safe**:
 - ▶ Clases **ConcurrentHashMap**, **ConcurrentSkipListMap**, implantando la interfaz **Map**
 - ▶ **CopyOnWriteArrayList**, para implantar la interfaz **List**
 - ▶ Interfaz **BlockingQueue**, extiende a **Queue**.



- ▶ *Java.util.concurrent* incluye versiones **Thread-safe**:
 - ▶ Implementaciones concurrentes, eficientes de *Queue*:
 - *ArrayBlockingQueue*
 - *ConcurrentLinkedQueue*
 - *LinkedBlockingQueue*
 - *PriorityBlockingQueue*
 - *DelayQueue*
 - *SynchronousQueue*





Colecciones concurrentes.- Ejemplo: BlockingQueue

► Métodos de la interfaz **BlockingQueue**:

	Genera Excepción	Devuelve Valor (true/false)	Bloques	TimeOuts
Insertar	<code>add(o)</code>	<code>offer(o)</code>	<code>put(o)</code>	<code>offer(o, timeout, timeunit)</code>
Extraer	<code>remove(o)</code>	<code>poll(o)</code>	<code>take(o)</code>	<code>poll(timeout, timeunit)</code>
Consultar	<code>element(o)</code>	<code>peek(o)</code>		

► Otros métodos:

- **remainingCapacity():**# elementos que se pueden insertar en la cola.
- **contains():** indica si la cola contiene un objeto determinado.
- **drainTo():** traslada los elementos de la cola a la colección indicada.



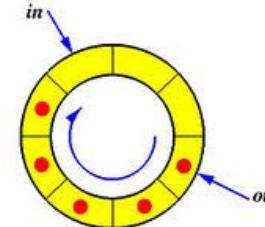
Colecciones concurrentes.- Ejemplo: BlockingQueue

► Métodos de la interfaz **BlockingQueue**:

- ▶ Para insertar elementos en la cola:
 - ▶ **add()**: si no hay espacio, se genera una excepción.
 - ▶ **offer()**: si no hay espacio, devuelve false.
 - ▶ **put()**: si no hay espacio, el hilo se queda esperando.
- ▶ Para extraer elementos de la cola:
 - ▶ **take()**: recupera y elimina el primer elemento de la cola. Espera si fuera necesario hasta que haya algún elemento que extraer.
 - ▶ **poll()**: igual que *take*, pero si no hay elementos en cola, se espera como máximo el intervalo especificado.
 - ▶ **remove()**: elimina de la cola la instancia del objeto suministrada.
 - ▶ **peek()**: retorna el primer elemento de la cola, sin extraerlo de ella.

- ▶ Ejemplo: **Problema del productor/consumidor**

- ▶ El buffer es una **BlockingQueue**:



BlockingQueue queue;

- ▶ El hilo que quiere extraer un ítem espera si la cola está vacía
consume(queue.take())
 - ▶ El hilo que quiere insertar un ítem espera si no hay espacio
queue.put(produce())

Sincronización condicional controlada en los métodos proporcionados por BlockingQueue:

- **take()**
- **put()**



Colecciones concurrentes.- Ejemplo: BlockingQueue

```
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) { ... }  
}
```

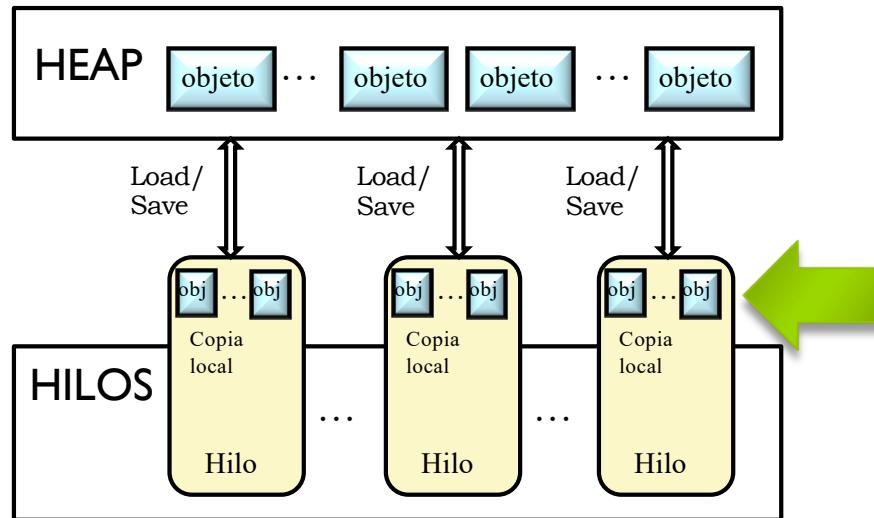
```
class Setup {  
    void main() {  
        BlockingQueue q = new  
            someQueueImpl();  
        Producer p = new Producer(q);  
        Consumer c1 = new Consumer(q);  
        Consumer c2 = new Consumer(q);  
        new Thread(p).start();  
        new Thread(c1).start();  
        new Thread(c2).start();  
    }  
}
```



Herramientas de java.util.concurrent

- ▶ La biblioteca incluye varias herramientas útiles:
 - ▶ Locks
 - ▶ Variables condición
 - ▶ Colecciones concurrentes
 - ▶ Variables atómicas
 - ▶ Sincronización: Semáforos y Barreras
 - ▶ Entorno para la gestión de hilos (Executor)

► Revisión del modelo de memoria en Java:



- Instrucciones no atómicas: `int i=0; i++;`
 - ▶ Cargar valor del entero desde heap (copia local=0);
 - ▶ Incrementar valor en 1 (copia local=1);
 - ▶ Cargar valor al heap (i=1);

**Sección
crítica**

requiere
synchronized

o protegerla con
ReentrantLock



Variables atómicas (`java.util.concurrent.atomic`)

- ▶ Define clases que soportan el acceso concurrente seguro a variables simples
 - ▶ Se aprovechan las instrucciones atómicas de bajo nivel de los procesadores
 - ▶ Garantizan: si un hilo lee un determinado valor de una variable, ningún otro hilo podrá leer más tarde un valor más antiguo
- ▶ Tipos primitivos (`AtomicBoolean`, `AtomicInteger`, `AtomicLong`)
- ▶ Referencias (`AtomicReference`)
- ▶ Permite tratar atómicamente distintos operadores
 - ▶ La operación `++` sobre un entero no es atómica, pero `AtomicInteger` posee una operación de incremento atómica
 - ▶ También operaciones como `getAndSet`, `compareAndSet`, etc.



Variables atómicas (*java.util.concurrent.atomic*)

- ▶ Útil para:
 - ▶ implementar algoritmos concurrentes de forma eficiente
 - ▶ implementación de contadores
 - ▶ generadores de secuencias de números
- ▶ La implementación es muy eficiente
 - ▶ Más de lo que podemos obtener utilizando *synchronized*



Variables Atómicas.- Ejemplo: **AtomicLong**

- ▶ **AtomicLong**: Permite actualizar de forma atómica un valor long
- ▶ Métodos que ofrece:
 - ▶ *addAndGet()* → atómicamente añade el valor dado al actual
 - ▶ *decrementAndGet()*, *getAndDecrement()* → atómicamente decrementa en 1
 - ▶ *incrementAndGet()*, *getAndIncrement()* → atómicamente incrementa en 1
 - ▶ *getAndSet()*
 - ▶ *compareAndSet()*
 - ▶ *toString()*
 - ▶ ...



Variables Atómicas.- Ejemplo

► Con 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();  
    }  
}
```

► Con AtomicLong

- Métodos *addAndGet*, *compareAndSet*, *decrementAndGet*, *incrementAndGet*, *getAndDecrement*, *getAndIncrement*, *getAndSet*, *toString*, *get*, ...

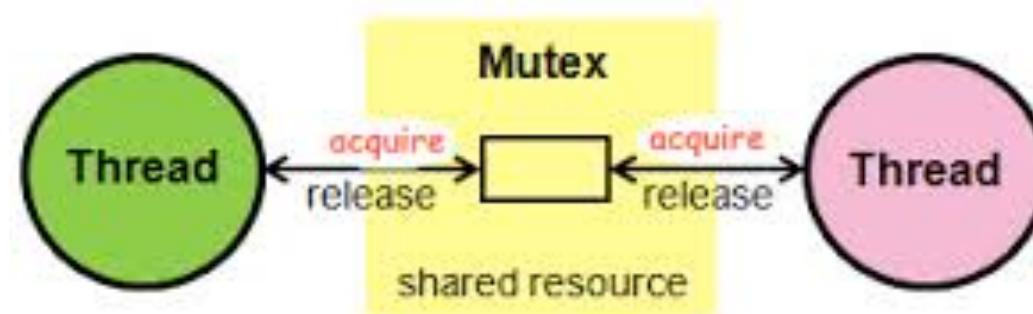
```
public class EjCounter extends Thread {  
    AtomicLong counter;  
    public EjCounter(AtomicLong c) {counter=c;}  
    public void run() {  
        System.out.println("counter value: "+  
            (counter.getAndIncrement()));  
    }  
    public static void main(String[] args) {  
     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



Herramientas de java.util.concurrent

- ▶ La biblioteca incluye varias herramientas útiles:
 - ▶ Locks
 - ▶ Variables condición
 - ▶ Colecciones concurrentes
 - ▶ Variables atómicas
 - ▶ Sincronización: Semáforos y Barreras
 - ▶ Entorno para la gestión de hilos (Executor)

- ▶ La clase **Semaphore** permite la sincronización entre hilos
 - ▶ Se usan los métodos **acquire()** y **release()**
 - ▶ **acquire()**: espera hasta disponer de un permiso, y entonces lo consume
 - ▶ **release()**: añade un permiso, y posiblemente libera a un hilo esperando tras **acquire()**





Semáforos



- ▶ Se puede emplear para proteger secciones críticas (ej: mutexes) o para sincronizar hilos.
- ▶ Lleva asociado un **contador** que se inicializa en la creación.
- ▶ Si se inicializan a uno → garantiza exclusión mutua

Semaphore sem = new Semaphore(1, true);

sem.acquire();

Sección Crítica



sem.release();



Semáforos



- ▶ Se puede emplear para proteger secciones críticas (ej: mutexes) o para sincronizar hilos.
- ▶ Lleva asociado un **contador** que se inicializa en la creación.
- ▶ Si se inicializan a **valor positivo >1** → limita el grado de concurrencia

Semaphore sem = new Semaphore(4, true);

sem.acquire();

Sección Limitada

sem.release();

Sólo 4 hilos
a la vez



- ▶ Se puede emplear para proteger secciones críticas (ej: mutexes) o para sincronizar hilos.
- ▶ Lleva asociado un **contador** que se inicializa en la creación.
- ▶ Si se inicializan a cero → garantiza cierto orden de ejecución entre dos o más hilos

Semaphore sem = new Semaphore(0, true);

H1

```
sem.acquire();
```

S1

H2

S2

```
sem.release();
```

Garantizar
que “S2 antes
que S1”



Sincronización.- Clase Semaphore

► Ejemplo 1: proteger secciones críticas

```
import java.util.concurrent.Semaphore;  
  
class Process2 extends Thread {  
    private int id;  
    private Semaphore sem;  
  
    public Process2(int i, Semaphore s) {  
        id = i; sem = s;}  
  
    private void busy() {  
        try {  
            sleep(new  
java.util.Random().nextInt(500));  
        } catch (InterruptedException e) {}}  
    private void msg(String s) {  
        System.out.println("Thread " + id + s);}  
  
    private void noncritical() {  
        msg(" is NON critical");  
        busy();}
```

```
private void critical() {  
    msg(" entering CS");  
    busy();  
    msg(" leaving CS");}  
  
public void run() {  
    for (int i = 0; i < 2; ++i) {  
        noncritical();  
        try {sem.acquire();}  
        catch (InterruptedException e) {}  
        critical();  
        sem.release();  
    }  
}  
public static void main(String[] args) {  
    Semaphore sem = new Semaphore(1, true);  
    for (int i = 0; i < 4; i++)  
        new Process2(i, sem).start();  
} }
```

Fairness: hilos seleccionados en orden (cola FIFO)



Sincronización.- Clase Semaphore

► Ejemplo 1: proteger secciones críticas

```
import java.util.concurrent.*;  
class Process2 extends Thread {  
    private int id;  
    private Semaphore sem;  
  
    public Process2(int i, Semaphore s) {  
        id = i; sem = s;  
    }  
  
    private void busy() {  
        try {  
            sleep(new java.util.Random().nextInt(500));  
        } catch (InterruptedException e) {}  
    }  
  
    private void msg(String s) {  
        System.out.println("Thread " + id + " " + s);  
    }  
  
    private void noncritical() {  
        msg(" is NON critical");  
        busy();  
    }  
  
    private void critical() {  
        msg(" entering CS");  
        busy();  
        msg(" leaving CS");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Semaphore sem = new Semaphore(1, true);  
        for (int i = 0; i < 4; i++)  
            new Process2(i, sem).start();  
    }  
}
```

```
C:\CSD>java Process2  
Thread 0 is NON critical  
Thread 3 is NON critical  
Thread 1 is NON critical  
Thread 2 is NON critical  
Thread 2 entering CS  
Thread 2 leaving CS  
Thread 2 is NON critical  
Thread 1 entering CS  
Thread 1 leaving CS  
Thread 1 is NON critical  
Thread 3 entering CS  
Thread 3 leaving CS  
Thread 3 is NON critical  
Thread 0 entering CS  
Thread 0 leaving CS  
Thread 0 is NON critical  
Thread 2 entering CS  
Thread 2 leaving CS  
Thread 3 entering CS  
Thread 3 leaving CS  
Thread 0 entering CS  
Thread 0 leaving CS  
Thread 1 entering CS  
Thread 1 leaving CS
```

```
    private void critical() {  
        msg(" entering CS");  
        busy();  
        msg(" leaving CS");  
    }  
  
    private void run() {  
        for (int i = 0; i < 2; ++i) {  
            noncritical();  
            try {sem.acquire();}  
            catch (InterruptedException e) {}  
            critical();  
            sem.release();  
        }  
    }  
}
```



Sincronización.- Clase *Semaphore*

► Ejemplo 2: sincronización de tareas

```
import java.util.concurrent.Semaphore;
class ProdCons extends Thread {
    static final int N=6; // buffer size
    static int head=0, tail=0, elems=0;
    static int[] data= new int[N];
    static Semaphore item= new Semaphore(0,true);
    static Semaphore slot= new Semaphore(N,true);
    static Semaphore mutex= new
Semaphore(1,true);

    public static void main(String[] args) {
        new Thread(new Runnable() { // producer
            public void run() {
                for (int i=0; i<10; i++) {
                    busy();
                    put(i);
                }
            }
        }).start();
        new Thread(new Runnable() { // consumer
            public void run() {
                for (int i=0; i<10; i++) {
                    busy();
                    System.out.println(get());
                }
            }
        }).start();
    }
}
```

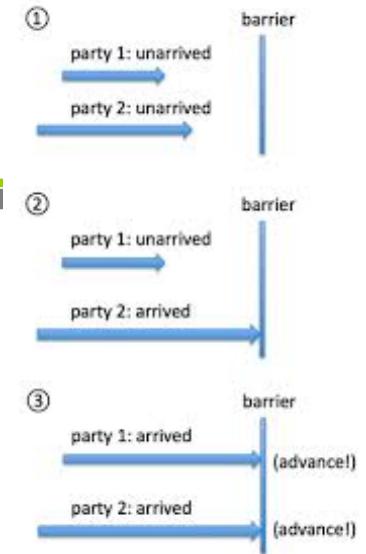
```
private static void busy() {
    try {sleep(new java.util.Random().nextInt(500));}
    catch (InterruptedException e) {}
}

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

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

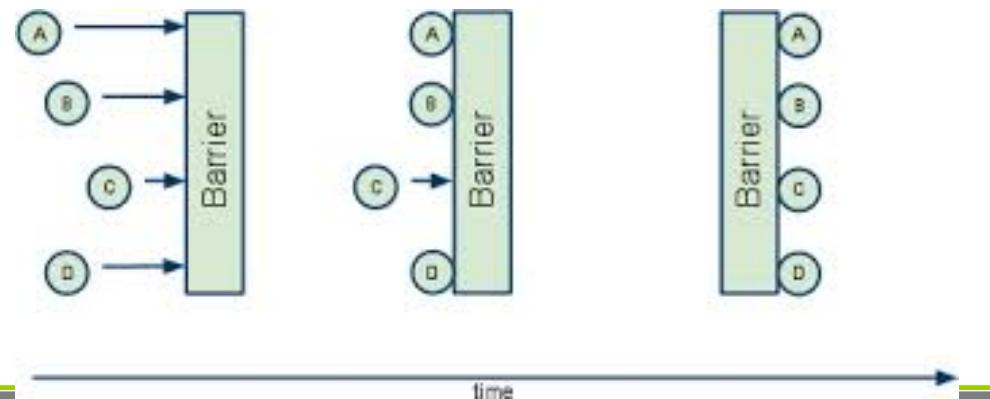
- ▶ Permiten sincronizar a múltiples hilos de ejecución
- ▶ En *java.util.concurrent* existen dos tipos diferentes de barreras:
 - ▶ CyclicBarrier
 - ▶ CountdownLatch





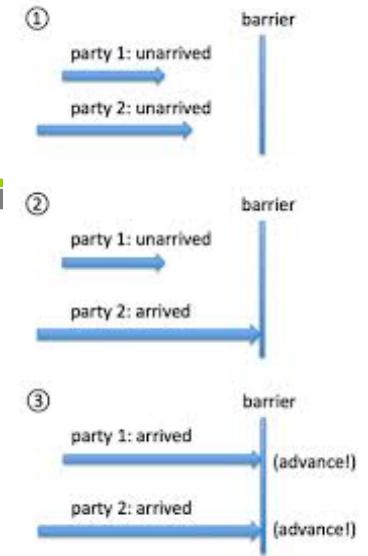
▶ CyclicBarrier:

- ▶ Permite que varios hilos se esperen mutuamente en un punto
- ▶ Se abre cuando determinado número de hilos llega a la barrera
 - ▶ Método **await()** de la barrera: suspender al hilo
 - ▶ Cuando último hilo invoca **await()**, la barrera se abre → los hilos se reactivan





Barreras: CyclicBarrier



► **CyclicBarrier:**

- ▶ Es reutilizable (por eso se dice que es cíclica)
- ▶ Al abrirse y pasar a los hilos, se restauran las condiciones iniciales
- ▶ Ideal para aplicaciones iterativas
 - ▶ Ej.- en cada iteración un hilo trabajador (Worker) procesa una columna de la matriz, y un coordinador (Solver) mezcla los resultados



Barreras: CyclicBarrier → Ejemplo

```
class Solver {  
    final int N;  
    final float[][] data;  
    final CyclicBarrier barrier;  
  
    class Worker implements Runnable {  
        int myRow;  
        Worker(int row) { myRow = row; }  
        public void run() {  
            while (!done()) {  
                processRow(myRow);  
                try { barrier.await();  
                } catch (InterruptedException e)  
                    {return;}  
                catch (BrokenBarrierException e)  
                    {return;}  
            } } } }
```

```
public Solver(float[][] matrix) {  
    data = matrix;  
    N = matrix.length;  
    barrier = new CyclicBarrier(N,  
        new Runnable() {  
            public void run() {  
                mergeRows(...);  
            }  
        });  
  
    for (int i = 0; i < N; ++i)  
        new Thread(new Worker(i)).start();  
    waitUntilDone();  
}
```



Barreras: CyclicBarrier → Ejemplo

```
class Solver {  
    final int N;  
    final float[][] data;  
    final  
    class Worker implements Runnable {  
        int myRow;  
        public void run() {  
            while (...) {  
                processRow(myRow);  
                try { barrier.await(); }  
                catch (InterruptedException e) {return;}  
                catch (BrokenBarrierException e) {return;}  
            } } } }
```

Método principal:
clases para que esperen en
Worker barrera

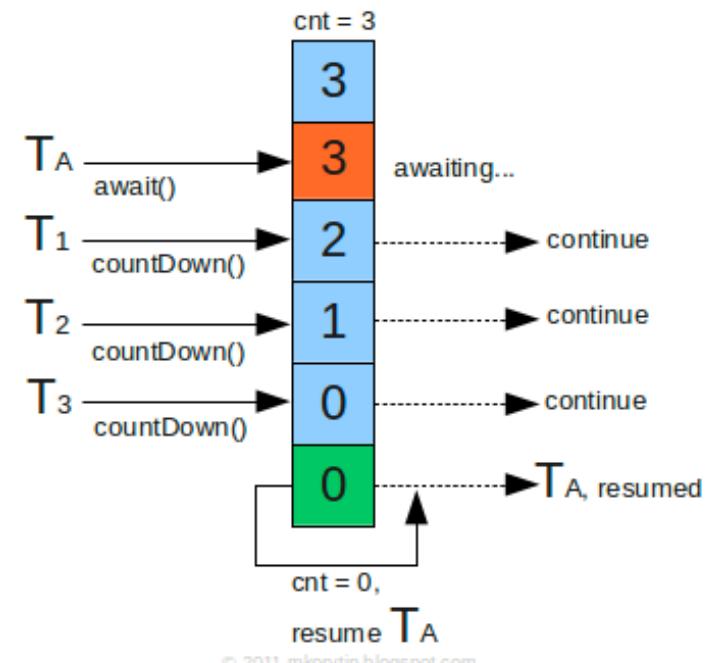
Número de hilos
que esperarán

```
public void solve() {  
    data = ...;  
    N = matrix.length;  
    barrier = new CyclicBarrier(N,  
        new Runnable() {  
            public void run() {  
                mergeRows(...);  
            }  
        });  
    for (int i = 0; i < N; ++i)  
        workers[i].start();  
}
```

Opcional: Tarea a realizar
cuando la barrera se abre
(antes de reactivar los hilos)

▶ CountDownLatch:

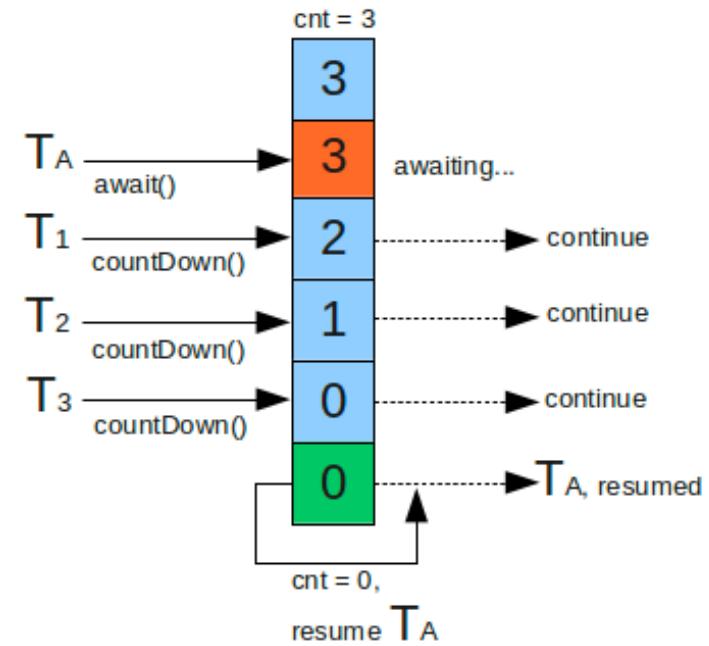
- ▶ Permite suspender a un grupo de hilos, a la espera de que suceda algún evento generado por un hilo ajeno al grupo
- ▶ Se mantiene un contador de eventos, inicializado en el constructor



© 2011 mkorytin.blogspot.com

▶ CountDownLatch:

- ▶ **Métodos que ofrece:**
 - ▶ **await():** los hilos se bloquean en la barrera mientras esté cerrada
 - ▶ **countDown():** decrementa en 1 el valor del contador (si superior a cero).
 - Si contador pasa a 0 →
 - la barrera se abre,
 - liberando todos los hilos
- ▶ **Barrera de un solo uso:**
 - ▶ Una vez a cero, permanece así.
 - ▶ Si se desea reutilización, usar *CyclicBarrier*.





Barreras: CountDownLatch → Ejemplo

```
class Driver { // ...
    void main() throws InterruptedException {
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new CountDownLatch(N);
        for (int i = 0; i < N; ++i)
            new Thread(new Worker(startSignal, doneSignal)).start();
        doSomethingElse();
        startSignal.countDown();
        doSomethingElse();
        doneSignal.await();
    }
}
```

startSignal: para que todos los Worker comiencen a la vez

¿Por qué tienen estos valores?

```
class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch start, CountDownLatch done) {
        startSignal = start;
        doneSignal = done;
    }
    public void run() {
        try { startSignal.await(); ←
              doWork();
              doneSignal.countDown();
            } catch (InterruptedException ex) {return;}
        }
    void doWork() { ... }
}
```



Barreras: CountDownLatch → Ejemplo

```
class Driver { // ...
void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);
    for (int i = 0; i < N; ++i)
        new Thread(new Worker(startSignal, doneSignal)).start();
    doSomethingElse();
    startSignal.countDown();
    doSomethingElse();
    doneSignal.await();
}
```

doneSignal:: donde
Driver espera a que todos
los Worker finalicen su
trabajo.

```
class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch start, CountDownLatch done) {
        startSignal = start;
        doneSignal = done;
    }
    public void run() {
        try { startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch (InterruptedException ex) {return;}
    }
    void doWork() { ... }
}
```

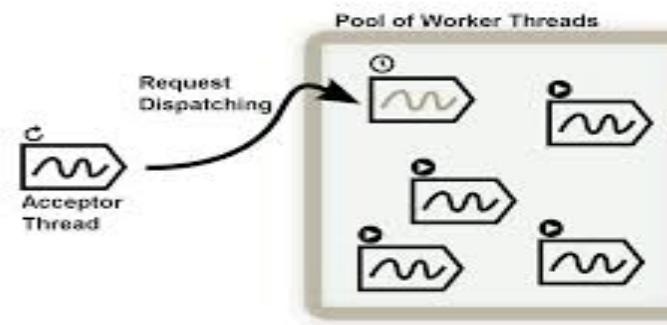


Herramientas de java.util.concurrent

- ▶ La biblioteca incluye varias herramientas útiles:
 - ▶ Locks
 - ▶ Variables condición
 - ▶ Colecciones concurrentes
 - ▶ Variables atómicas
 - ▶ Sincronización: Semáforos y Barreras
 - ▶ Entorno para la gestión de hilos (Executor)

Entorno para la ejecución de hilos

- ▶ La **creación de hilos** es una operación costosa, que requiere muchos recursos y puede resultar lenta.
- ▶ **Interfaz Executor**: ofrece un entorno para la creación y gestión de hilos en Java
 - ▶ Permite su invocación, planificación, ejecución y control de políticas de ejecución.



- ▶ Ventajas:
 - ▶ Previene el consumo desmedido de recursos
 - ▶ Se dispone de una biblioteca ya implementada (**Executor**) para crear tareas de una forma muy flexible
- ▶ Este tipo de entorno está ampliamente utilizado



Entorno para la ejecución de hilos

- ▶ La interfaz **Executor** ofrece la subinterfaz **ExecutorService**, que permite crear diferentes tipos de **Executors** :
 - ▶ Un hilo único (*SingleThreadExecutor*): utiliza un solo hilo de trabajo que opera en una cola sin límites.
 - ▶ Se garantiza que las tareas se ejecutan de forma secuencial, y que no habrá más de una tarea activa en un momento dado.

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

Crea un solo hilo en el que las tareas quedan en cola y se ejecutan de forma secuencial

- ▶ Un *thread-pool* (ej.- para servidor)
 - ▶ Permite mantener un conjunto de hilos ya creados, reciclandolos para que ejecuten nuevas tareas.

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

Crea un pool con 10 hilos



Entorno para la ejecución de hilos

- ▶ Ejemplo sencillo de uso de un **ExecutorService**:

```
ExecutorService executorService =  
    Executors.newFixedThreadPool(10);  
  
executorService.execute(new Runnable() {  
    public void run() {  
        System.out.println("Asynchronous task");  
    }  
});  
  
executorService.shutdown();
```



Entorno para la ejecución de hilos

- ▶ Existen diferentes formas para delegar tareas para la ejecución de un ExecutorService:
 - ▶ execute(Runnable), submit(Runnable), submit(Callable), invokeAny(...), invokeAll(...)
- ▶ Nos centraremos aquí en el uso de Runnable

```
Runnable command = new Runnable() {  
    public void run() {  
        doLongWork();  
    }  
};
```

```
ExecutorService executor = Executors.newSingleThreadExecutor();  
executor.execute(command);
```



Entorno para la ejecución de hilos

```
package executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {

    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(10);
        executor.execute(() -> doLongWork("hi 1"));
        executor.execute(() -> doLongWork("hi 2"));
        executor.execute(() -> doLongWork("hi 3"));
    }

    private static void doLongWork(String hola) {
        System.out.println("Running " + hola);
        try {
            Thread.sleep(1000l);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Nota: se ha utilizado notación Lambda (no entraremos en el detalle)



Entorno para la ejecución de hilos

- ▶ Por defecto, **ExecutorService** se queda “escuchando” nuevas tareas
 - ▶ Queda activo para recibir nuevas tareas y asignarlas a su pool
 - ▶ Aunque acabe el hilo main, **ExecutorService** continúa activo
- ▶ Existen dos formas para detener el **ExecutorService**:
 - ▶ **shutdown()** → permite detener los hilos del Executor.
 - ▶ No se detiene inmediatamente, pero ya no aceptará nuevas tareas
 - ▶ Cuando todos los hilos hayan acabado con sus tareas actuales, entonces se detendrá
 - ▶ **shutdownNow()** → trata de detener todas las tareas en ejecución
 - ▶ Las tareas presentadas pero no procesadas las ignora
 - ▶ Intenta parar las tareas en ejecución (aunque algunas podrían ejecutarse hasta su finalización)
 - ▶ Devuelve la lista de tareas que no finalizaron



Entorno para la ejecución de hilos

- ▶ Importante: aunque el hilo principal (main) de la aplicación termine, el ExecutorService continuará ejecutándose (y, por tanto, la aplicación), mientras esté activo.
- ▶ Podemos utilizar `awaitTermination()` para esperar a que ExecutorService se detenga.
- ▶ Ejemplo:

```
executorService.shutdown();
executorService.awaitTermination();
```



Resultados de aprendizaje de la Unidad Didáctica

- ▶ Al finalizar esta unidad, el alumno deberá ser capaz de:
 - ▶ Identificar los inconvenientes de las primitivas básicas de Java.
 - ▶ Describir las herramientas del package `java.util.concurrent`, que facilitan el desarrollo de aplicaciones concurrentes:
 - ▶ Ilustrar la utilización de los *locks* y las *condiciones*. Compararlos con los monitores.
 - ▶ Interpretar uso de colecciones concurrentes *thread-safe* (e.g. `BlockingQueue`)
 - ▶ Describir el funcionamiento de las clases atómicas. Comparar `AtomicInteger` con los monitores.
 - ▶ Ilustrar el uso de semáforos (*Semaphore*) para sincronización.
 - ▶ Ilustrar el funcionamiento de las barreras, distinguiendo entre `CyclicBarrier` y `CountDownLatch`.
 - ▶ Conocer el entorno de gestión de hilos `Executor`