

Programación de Sistemas y Concurrency

Tema 6: Comunicación y Sincronización en Memoria Compartida

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

Índice

- Regiones críticas
- Mecanismos de sincronización en Java
- Métodos sincronizados y Monitores
- Interrupciones
- Referencias

Regiones críticas condicionales

- Críticas a los semáforos

- Son primitivas de muy bajo nivel, eficientes desde el punto de vista de la implementación, pero no adecuadas cuando hay que resolver problemas de sincronización complejos (lectores/escritores).
- A veces, los semáforos están sobrecargados (se utilizan para más de una tarea: exclusión mutua y sincronización). Esto hace que sea difícil entender la interacción entre los procesos que los utilizan, lo que dificulta la modificación del código.
- Errores pequeños pueden tener consecuencias impredecibles:
 - Olvidar una operación **acquire** en un preprotocolo puede violar la exclusión mutua;
 - Olvidar una operación **release** en un postprotocolo puede producir bloqueos.

Regiones críticas

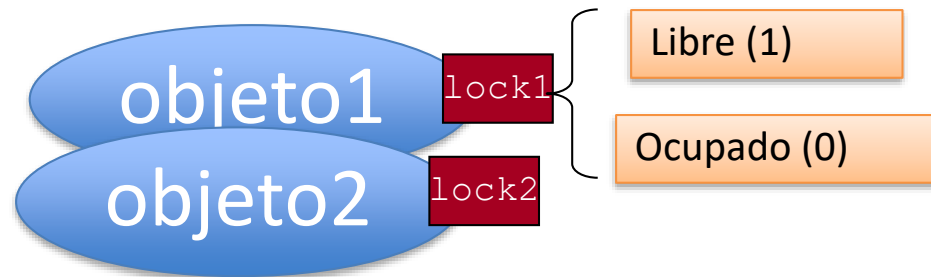
- Una *sección crítica* es una secuencia de instrucciones en un proceso (que típicamente acceden a un recurso compartido) que *debería* ejecutarse en *exclusión mutua* con otras secciones críticas de otros procesos.
 - El programador debe asegurarlo utilizando mutex.
- Una *región crítica* es una secuencia de instrucciones en un proceso, que accede a un recurso compartido, y que, por definición *se ejecuta* en *exclusión mutua*.
 - El lenguaje lo asegura porque pone automáticamente (de forma transparente al programador) los mutex necesarios para asegurar la exclusión mutua.

Regiones críticas

- Para definir una región crítica necesitamos utilizar un constructor del lenguaje especial que indique qué secuencia de instrucciones corresponden a una región crítica.

```
region (Object v) {  
    S;  
}
```

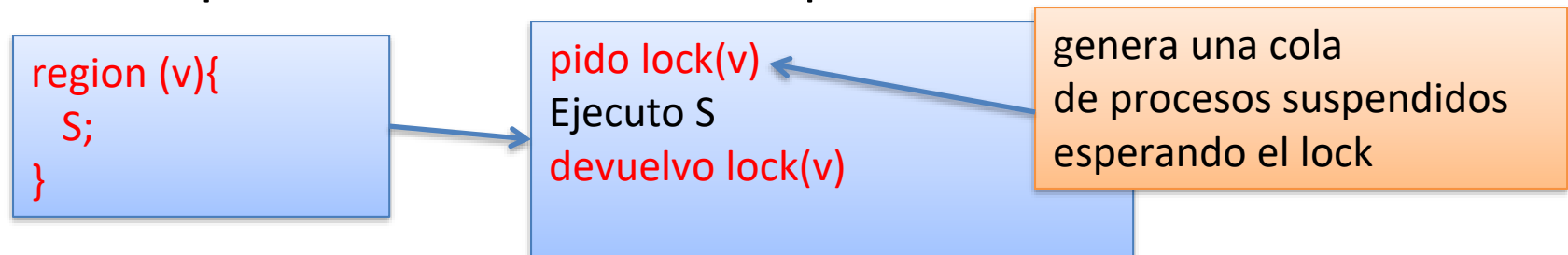
- Java no tiene regiones críticas, sin embargo, cada objeto Java tiene asociado un **lock** (cerrojo), al que no se puede acceder directamente.



Regiones críticas

```
region (Object v) {  
    S;  
}
```

- Cualquier proceso que quiera ejecutar **S** debe conseguir primero el *lock* del recurso **v**.
- Si no puede porque otro proceso tiene el *lock*, se bloquea en una **cola** asociada al recurso.
- Cuando el proceso que tiene el *lock* termina su acceso al recurso, libera el *lock* permitiendo entrar al primer proceso de los que están en la cola de espera.



Regiones críticas

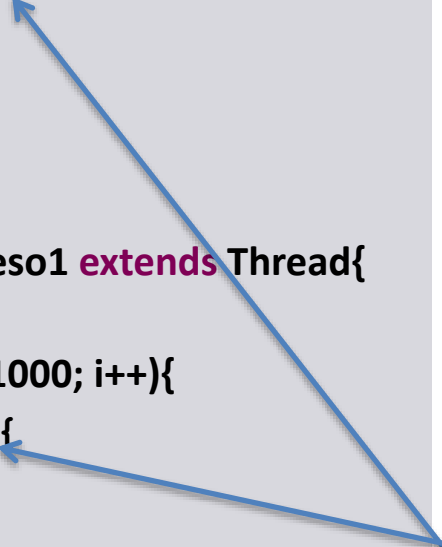
- Sólo las regiones críticas etiquetadas con el mismo recurso se ejecutan en exclusión mutua, por ejemplo,

<pre>region (v1) { S1; }</pre>	<pre>region (v2) { S2; }</pre>
--	--

pueden solaparse en el tiempo.

Regiones críticas: el problema de los jardines

```
public class ExclusionMutua {  
    private static int cont = 0;  
    public static class Proceso1 extends Thread{  
        public void run(){  
            for (int i = 0; i<1000; i++){  
                region (cont){  
                    cont++;  
                }  
            }  
        }  
    }  
    public static class Proceso2 extends Thread{  
        public void run(){  
            for (int i = 0; i<1000; i++){  
                region (cont){  
                    cont++;  
                }  
            }  
        }  
    }  
}
```



```
public static void main(String[] args){  
    Proceso1 p1 = new Proceso1();  
    Proceso2 p2 = new Proceso2();  
    p1.start();  
    p2.start();  
    try{  
        p1.join();  
        p2.join();  
    }catch (InterruptedException ie){}  
    System.out.println(cont);  
}
```

Como sabemos que los incrementos se hacen en exclusión mutua, estamos seguros de que al final de la ejecución $cont = 2000$

Regiones críticas condicionales

- Además de la exclusión mutua, una primitiva de sincronización debe ser capaz de resolver también **condiciones de sincronización** como las del problema del productor/consumidor.
- Para ello, **extendemos las regiones críticas con guardas** que deben evaluarse a true antes de que un proceso pueda hacerse con el lock del recurso compartido.

```
region (v) when (B) {  
    S;  
}
```

Regiones críticas condicionales

```
region (v) when (B) {  
    S;  
}
```

- Un proceso que quiere ejecutar una RCC debe
 - Conseguir el lock del recurso compartido. Si no está disponible, el proceso se bloquea en la cola de entrada asociada al recurso.
 - Cuando el proceso tiene el lock, evalúa la expresión booleana B.
 - Si es cierta, ejecuta S
 - Si es falsa, libera el lock y se bloquea en la cola de entrada asociada a v
- Un proceso sale de una RCC debe liberar el lock del recurso

Regiones críticas condicionales

```
public class ProdConsSem {  
    private static final int N = 10;  
    private static int[] buffer = new int[N];  
    private static int elem = 0;  
  
    public static class Productor extends Thread{  
        private int i = 0;  
        Random r = new Random();  
        public void run(){  
            while (true){  
                produce(aux);  
                region (buffer) when (elem<N){  
                    buffer[i] = aux;  
                    i = (i + 1)% N;  
                    elem++;  
                }  
            }  
        }  
    }  
    ....  
}
```

```
public static class Consumidor extends Thread{  
    private int i = 0;  
    public void run(){  
        while (true){  
            region (buffer) when (elem>0){  
                int aux = buffer[i];  
                i = (i + 1)% N;  
                elem--;  
            }  
            consume(aux);  
        }  
    }  
}
```

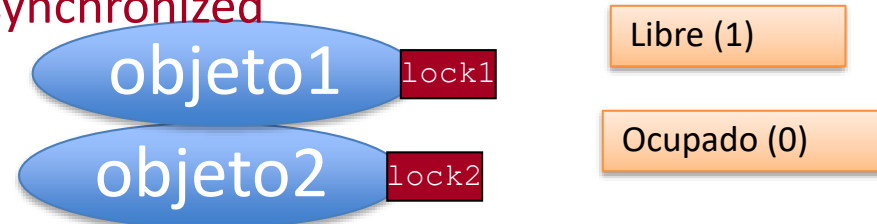
Regiones críticas condicionales

- Críticas a las RCC

- El procedimiento de bloqueo de los procesos no es muy eficiente. Cuando un proceso evalúa a false la expresión booleana de una RCC se vuelve a bloquear en la cola de entrada. Cuando lo desbloquean, no hay seguridad de que la expresión booleana sea ahora cierta. Este comportamiento es similar al de una espera activa.
 - Una solución es que el proceso que sale de la RCC evalúe las expresiones booleanas de los procesos que esperan, y despierte al primero que pueda continuar. El problema de esta solución es que, muchas veces, en las expresiones booleanas aparecen variables locales, a las que no tienen acceso el resto de los procesos.
- El código de acceso al recurso protegido está distribuido por el código de los procesos. Desde el punto de vista de la seguridad, esto no es adecuado, ya que cualquiera puede modificar el procedimiento de acceso a un recurso, violando alguna propiedad básica.

Mecanismos de sincronización en Java

- Cada objeto Java tiene asociado un **lock** (cerrojo) al que no se puede acceder directamente, pero que afecta a
 - los métodos declarados como **synchronized**
 - los bloques sincronizados



- Antes de ejecutar un método declarado como **synchronized** hay que competir para conseguir el lock del objeto al que pertenece.
- Por lo tanto, los métodos sincronizados aseguran el **acceso en exclusión mutua** a los datos del objeto, si sólo puede accederse a estos datos a través de métodos sincronizados
- Los **métodos no sincronizados** no necesitan adquirir el lock del objeto y, por lo tanto, pueden ser ejecutados en cualquier momento.

Mecanismos de sincronización en Java

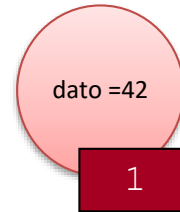
- Los *métodos sincronizados* de java se sincronizan con cualquier *otro método sincronizado del mismo objeto*.
- De esta forma, un objeto puede contener recursos cuyo acceso requiera un control (los métodos de acceso deben declararse como sincronizados), y otras entidades neutras, a las que no hace falta sincronizar. Estas últimas pueden ser ejecutadas en cualquier momento.
- Desde el punto de vista de la organización del código esta solución es más razonable, pues el recurso compartido se introduce en un objeto, y son los métodos de este objeto los que permiten su manipulación.

Mecanismos de sincronización en Java

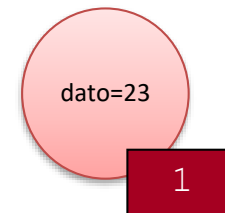
```
class VarCompEntera {  
  
    private int dato;  
    public VarCompEntera (int valorInicial) {  
        dato = valorInicial;  
    }  
    public synchronized int read() {  
        return dato;  
    }  
    public synchronized void write(int nuevoValor) {  
        dato = nuevoValor;  
    }  
    public synchronized void inc (int increm) {  
        dato = dato + increm;  
    }  
}  
VarCompEntera miDato1 = new VarCompEntera(42);  
VarCompEntera miDato2 = new VarCompEntera(23);
```

Los accesos a cada dato de cada instancia de la clase VarCompEntera se realizan en exclusión mutua

miDato1



miDato2



Cuando se crea un objeto de una clase, el lock correspondiente está inicialmente libre (1)

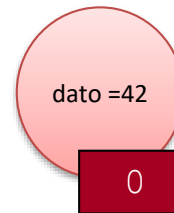
Nota: el constructor del objeto no puede sincronizarse

Mecanismos de sincronización en Java

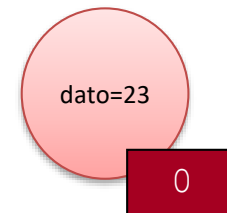
```
class VarCompEntera {  
    ....  
    public synchronized void write(int nuevoValor) {  
        dato = nuevoValor;  
    }  
    public synchronized void inc (int increm) {  
        dato = dato + increm;  
    }  
    ....  
}
```

```
VarCompEntera miDato1 = new VarCompEntera(42);  
VarCompEntera miDato2 = new VarCompEntera(23);
```

miDato1



miDato2



hebra1

```
public void run(){  
    ...  
    miDato1.write(5);  
}
```

hebra2

```
public void run(){  
    ...  
    miDato1.inc(9);  
}
```

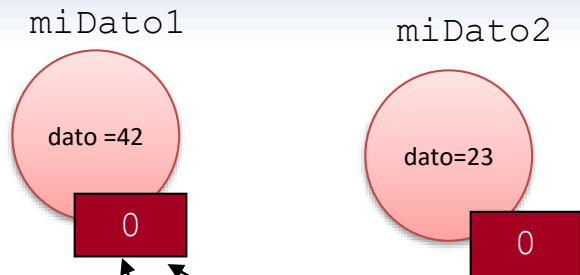
```
public synchronized void write(int nuevoValor){  
    cogeLock();  
    dato = nuevoValor;  
    devuelveLock()  
}
```

cogeLock se realiza de forma **atómica**, por lo que solo una de las hebras puede ejecutar el código inmediatamente, la otra debe esperar

Mecanismos de sincronización en Java

Tenemos dos posibilidades...

```
class VarCompEntera {  
    ....  
    public synchronized void write(int nuevoValor) {  
        dato = nuevoValor;  
    }  
    public synchronized void inc (int increm) {  
        dato = dato + increm;  
    }  
    ....  
}  
  
VarCompEntera miDato1 = new VarCompEntera(42);  
VarCompEntera miDato2 = new VarCompEntera(23);
```



```
hebra1  
public void run(){  
    ...  
    miDato1.write(5);  
}
```

```
hebra2  
public void run(){  
    ...  
    miDato1.inc(9);  
}
```

```
public synchronized void write(int nuevoValor){  
    cogeLock();  
    dato = nuevoValor;  
    devuelveLock()  
}
```

hebra1: cogeLock (0)
hebra2: cogeLock (0) espera
hebra1: dato = 5
hebra1: devuelveLock (0)
hebra2: dato = dato + 9
hebra2: devuelveLock (1)

hebra2: cogeLock (0)
hebra1: cogeLock (0) espera
hebra2: dato = dato + 9 (dato = 51)
hebra2: devuelveLock (0)
hebra1: dato = 5
hebra1: devuelveLock (1)

Bloques sincronizados

- Un **secuencia de instrucciones (bloque)** dentro de cualquier método puede también ejecutarse en exclusión mutua si lo etiquetamos con la palabra **synchronized** parametrizado con respecto al **objeto** cuyo lock necesitamos para ejecutar el bloque de forma segura

```
public synchronized void write(int nuevoValor) {  
    dato = nuevoValor;  
}
```

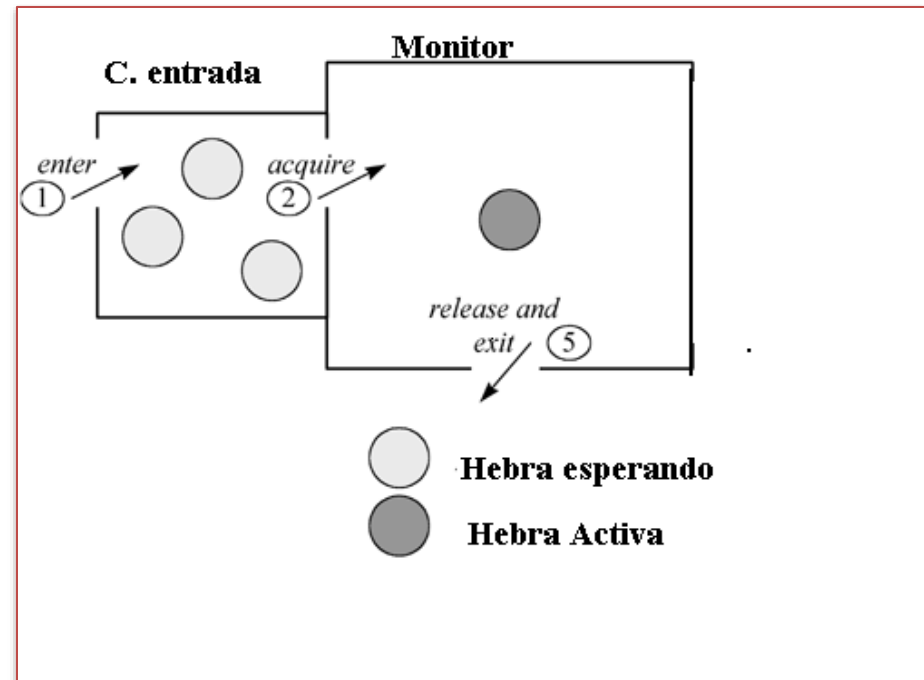
es equivalente a

```
public void write(int nuevoValor) {  
    synchronized (this){  
        dato = nuevoValor;  
    }  
}
```

this es una referencia
al nodo actual

Mecanismos de sincronizacion en java

- Podemos suponer que cada objeto tiene una estructura de entrada donde esperan las hebras que esperan adquirir el lock del objeto
- En cada momento, ejecutando un método sincronizado del objeto puede haber a lo sumo una hebra
- En la estructura de entrada puede haber múltiples hebras
- Java no define ninguna estructura concreta para almacenar a los procesos que esperan. Podría ser una fifo, una cola de prioridades, etc.



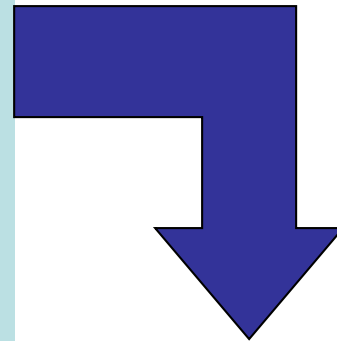
Métodos sincronizados y Monitores

- Los **métodos sincronizados** son una versión simplificada (en las primeras versiones de java) de la estructura de **monitor** definida por **Hoare** y **Hansen** a mediados de los 70.
- Un **monitor** es una estructura proporcionada por un lenguaje de programación concurrente que **permite encapsular todos aspectos de sincronización asociados a un recurso**, habitualmente también anidado en el monitor. Por definición, todos los procedimientos, métodos o funciones exportados (públicos) por el monitor son ejecutados siempre en **exclusión mutua**. Además proporciona variables de tipo **condición** para modelar fácilmente las **condiciones de sincronización**.
- Por esto, siempre que sea posible es mejor implementar los mecanismos de sincronización de cada objeto **dentro** de la clase que lo define, a través de métodos o bloques sincronizados.
- Si no lo hacemos, será imposible conocer los mecanismos de sincronización asociados a un objeto mirando sólo la clase que lo define, ya que otros objetos podrían tener un bloque sincronizado parametrizado por el objeto.
- Si utilizamos los bloques sincronizados con cuidado, podemos diseñar condiciones de sincronización más complejas de forma sencilla.

Bloques sincronizados: ejemplo

- Considera una clase que representa unas coordenadas bidimensionales que pueden ser compartidas por dos o más hebras.

```
public class Coodenadas {  
    private int x, y;  
  
    public Coordenadas(int initX, int initY) {  
        x = initX;  
        y = initY;  
    }  
    public synchronized void write(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
    ...  
}
```



La actualización de las coordenadas es fácil, basta con asignarles los nuevos valores en un método sincronizado

Bloques sincronizados: ejemplo

- Pero, ¿cómo leemos el valor de las coordenadas?
- Las funciones en java sólo pueden devolver un valor, y el paso de parámetros es por valor
- No es posible tener un método único que devuelva simultáneamente **x** e **y**.
- Además, si usamos dos funciones sincronizadas, **readX** y **readY**, es posible que una llamada a **write** interfiera entre dos llamadas a estas funciones, y por lo tanto el valor devuelto será inconsistente.

```
public class Coodenadas {  
    private int x, y;  
  
    public Coodenadas(int initX, int initY) {  
        x = initX;  
        y = initY;  
    }  
  
    public synchronized void write(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
    ...  
}
```

Bloques sincronizados: ejemplo

- PRIMERA SOLUCIÓN: Definir un método que devuelva un nuevo objeto **Coordenadas** con los valores **x** e **y** idénticos al objeto original (una copia del objeto actual)
- Aunque la **coordenada devuelta es consistente**, estos valores podrían ser cambiados inmediatamente después de ejecutar el método (y la coordenada nueva no reflejaría el cambio pues es una copia)
- Una vez que se ha utilizado la coordenada, puede desecharse, y ser recogida por el recolector de basura
- Desde el punto de vista de la eficiencia, podríamos decir que no es recomendable crear un objeto para tirarlo a la basura inmediatamente después.

```
public class Coodenadas {  
    private int x, y;  
    .....  
    public synchronized Coordenadas read() {  
        return new Coordenadas(x, y);  
    }  
    public int readX() { return x; }  
    public int readY() { return y; }  
}
```

Bloques sincronizados: ejemplo

- Segunda solución:
- Podemos suponer que la hebra cliente utilizará bloques sincronizados para conseguir la atomicidad

```
public class Coordenadas {  
    ...  
    public synchronized void write(int newX, int newY) {  
        x = newX; y = newY;  
    }  
  
    public int readX() { return x; } // no synchronized  
    public int readY() { return y; } // no synchronized  
}  
  
Coordenadas point1 = new Coordenadas(0,0);  
  
synchronized(point1) {  
    Coordenadas point2 = new Coordenadas(  
        point1.readX(), point1.readY());  
}
```


Datos estáticos sincronizados

- Las variables estáticas son compartidas por todos los objetos creados de una clase dada.
- En Java, las clases son también objetos, y como tales tienen un lock de sincronización asociado.
- Podemos acceder a este lock definiendo como sincronizados a los métodos estáticos de la clase, o pasando como parámetro a un bloque sincronizado el objeto clase.
- Este lock de una clase no puede obtenerse cuando se realizan sincronizaciones de objetos de la clase.

```
class VarCompEstatica {  
    private static int varComp;  
    ...  
  
    public int read() {  
        synchronized(VarCompEstatica.class) {  
            return varComp;  
        };  
    }  
  
    public synchronized static void write(int l) {  
        varComp = l;  
    }  
}
```

El problema de los jardines

```
public class JardinesSinc {  
    public class Contador{  
        private int cont = 0;  
        public synchronized void inc(int c){  
            cont+=c;  
        }  
        public int cont(){  
            return cont;  
        }  
    }  
    public static class Puerta extends Thread{  
        private Contador contador;  
        public Puerta(Contador contador){  
            this.contador = contador;  
        }  
        public void run(){  
            for (int i=0; i<100; i++)  
                contador.inc(1);  
        }  
    }  
    .....  
}
```

```
public static void main(String[] args){  
    Contador cont = new Contador();  
    Puerta[] p = new Puerta[25];  
    for (int i = 0; i<25; i++)  
        p[i] = new Puerta(cont);  
    for (int i = 0; i<25; i++)  
        p[i].start();  
    try{  
        for (int i = 0; i<25; i++)  
            p[i].join();  
    }catch (InterruptedException ie){}  
    System.out.println(cont.cont());  
}
```

todos los accesos a contador.cont se realizan en exclusión mutua

Condiciones de sincronización

- Para definir **condiciones de sincronización** utilizamos los métodos **wait** y **notify** de la clase **Object**

```
public class Object {  
    ...  
    public final void notify();  
    public final void notifyAll();  
    public final void wait() throws InterruptedException;  
    public final void wait(long millis) throws InterruptedException;  
    public final void wait(long millis, int nanos)  
        throws InterruptedException;  
    ...  
}
```

- Estos métodos deberían utilizarse sólo desde métodos en los que se mantiene el lock del objeto.
Si se utilizan sin tener el lock se lanza la excepción no comprobada **IllegalMonitorStateException**

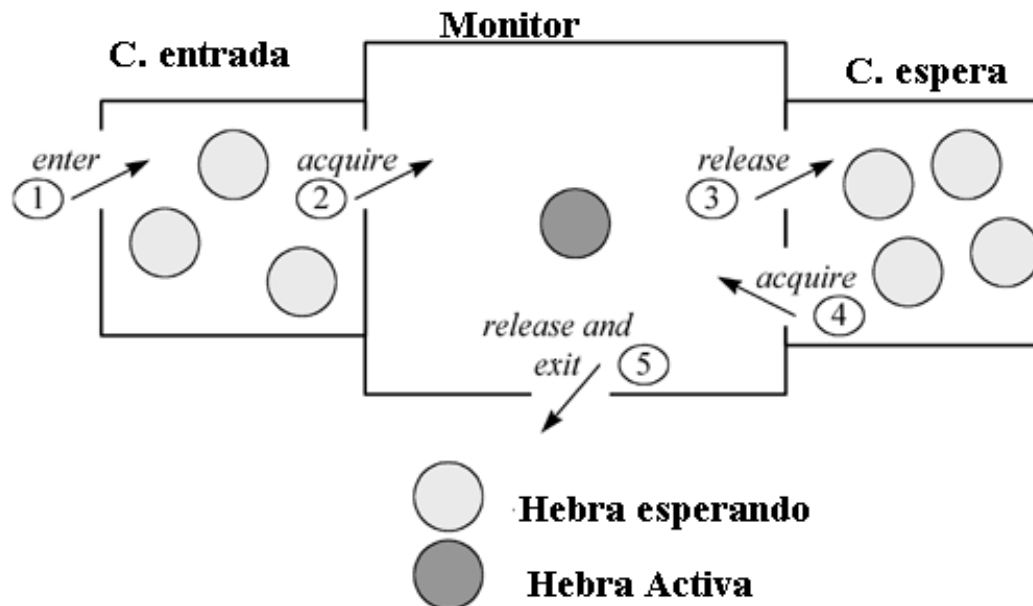
Condiciones de sincronización: wait

- El método **wait()** **suspende siempre** a la hebra que lo ejecuta.
- Se utiliza cuando se está ejecutando un método sincronizado y, por alguna razón, **el estado del objeto compartido no es el adecuado** para continuar la ejecución. En ese caso, **la hebra se suspende** a la espera de que cambie el estado del objeto y pueda continuar su ejecución.
- Cuando una hebra ejecuta **wait()**:
 - Libera el lock del objeto sincronizado
 - Se bloquea en el *conjunto de espera*

Condiciones de sincronización: wait

Asociados a un objeto compartido puede haber

- varias hebras esperando en el conjunto de entrada
- una hebra (a lo sumo) ejecutando un método sincronizado
- varias hebras esperando en el conjunto de espera



Condiciones de sincronización: notify

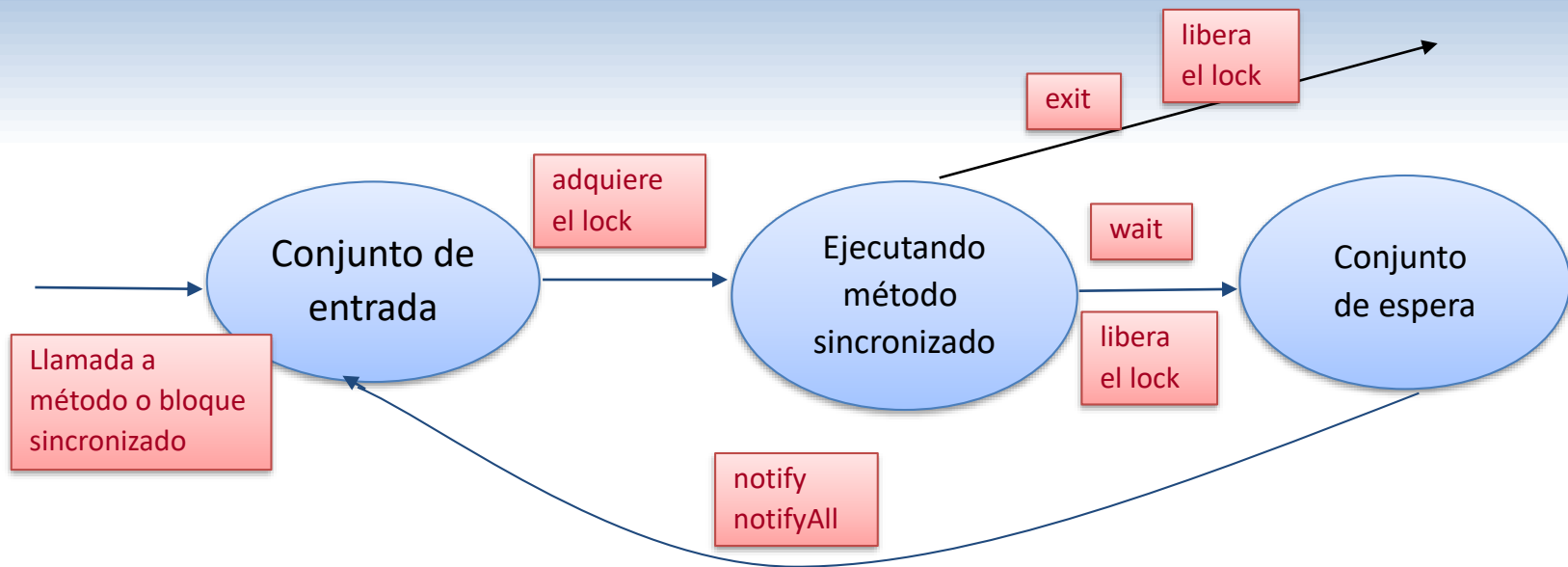
- El método **notify** despierta a una (cualquiera) de las hebras que esperan.
- Para despertar a todas las hebras utilizamos **notifyAll**
- Si no hay ninguna hebra esperando estos métodos no tienen efecto
- Una hebra que espera se despierta si es interrumpida por otra hebra. En ese caso se lanza la excepción **InterruptedException**

Condiciones de sincronización: notify

- La hebra que ejecuta **notify** no libera el **lock** del objeto compartido, por lo que la hebra despertada debe volver a conseguirlo para continuar su ejecución en el método sincronizado.
- Esta disciplina se denomina **notify-and-continue**
- Bajo esta disciplina, desde que el objeto es despertado hasta que vuelve de nuevo al monitor ha podido haber muchas otras hebras dentro del objeto compartido que han podido modificar su estado
- Por lo tanto, el objeto despertado antes de continuar su ejecución en el método/bloque sincronizado debe volver a comprobar si la condición por la que se suspendió sigue siendo cierta.

```
while (condicion)  
    wait();
```

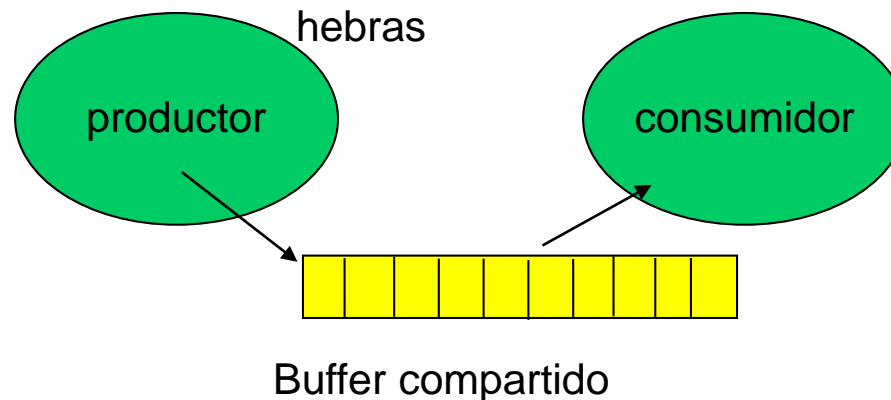
Disciplina notify-and-continue



- Cuando una hebra llama a un método o bloque sincronizado de un objeto espera en el conjunto de entrada del objeto hasta obtener el lock
- Cuando tiene el lock, ejecuta en exclusión mutua el método o bloque sincronizado
- Durante la ejecución del método sincronizado la hebra puede
 - Ejecutar **wait**, en ese caso libera el lock del objeto y se va al conjunto de espera
 - Ejecutar **notify/notifyAll**, en ese caso continúa en el monitor pero una/todas las hebras del conjunto de espera son transferidas al conjunto de entrada

Productor consumidor con Buffer Acotado

- Un proceso **productor** produce de forma ininterrumpida datos que deben ser consumidos por otro proceso **consumidor**
- Los procesos utilizan un buffer intermedio donde se almacenan temporalmente los datos producidos, que todavía no se han consumido
- El sistema debe satisfacer las siguientes propiedades
 - El buffer se utiliza en exclusión mutua
 - El productor no puede escribir sobre el buffer, si está lleno
 - El consumidor no puede extraer un dato, si el buffer está vacío



Productor consumidor con Buffer Acotado

```
public class Buffer <T>{  
  
    private T[] b;  
    private int i=0,j=0;  
    private int numElementos = 0;  
    public Buffer(int tam){  
        b = (T[]) new Object[tam];  
    }  
    public synchronized T get()  
        throws InterrutpedException{  
        while (numElementos == 0)  
            wait();  
        int aux = i;  
        i = (i + 1) % b.length;  
        numElementos--;  
        notifyAll();  
        return b[aux];  
    }  
}
```

CS2



Launch

```
public synchronized void put(T nDato)  
    throws InterruptedException{  
  
    while (numElementos == b.length)  
        wait();  
  
    b[j]=nDato;  
    j = (j + 1) % b.length;  
    numElementos++;  
    notifyAll();  
  
    }  
  
    }
```

CS1

Condiciones de sincronización:

CS1: el productor no puede escribir si el buffer está lleno

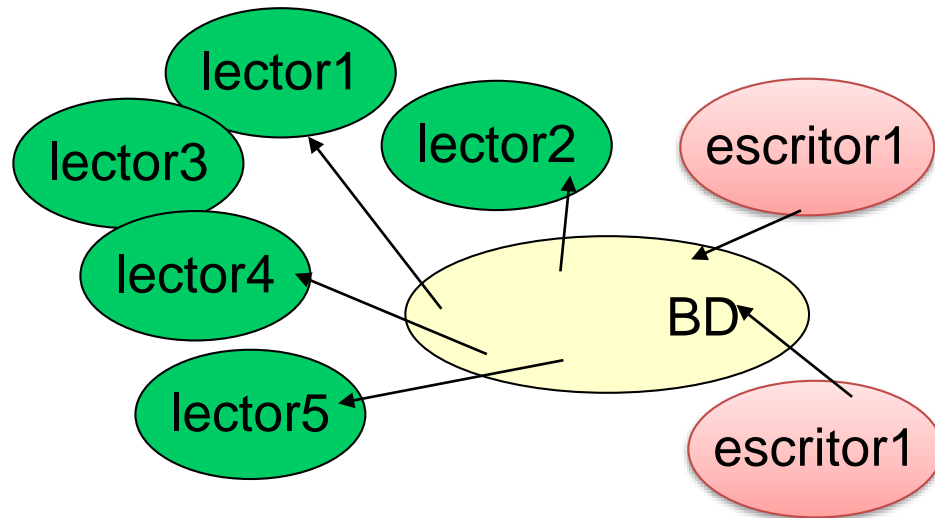
CS2: el consumidor no puede leer si el buffer está vacío

El problema de tener sólo un conjunto de espera

- Cuando se despierta una hebra, no puede suponer que la condición por la que se suspendió se satisface, ya que todas las hebras se despiertan con notify/notifyAll con independencia de las condiciones por las que esperan.
- Para algunos algoritmos esto no es un problema ya que las condiciones no pueden ser ciertas simultáneamente.
- Por ejemplo, en el problema de productor consumidor, si una hebra está esperando una condición, la otra hebra no puede estar suspendida (el buffer no puede estar lleno y vacío a la vez)
- Pero para otros escenarios, la situación puede ser diferente

El problema de los lectores/escritores

- Dos tipos de hebras (tipos lector y escritor) comparten un recurso (típicamente una Base de Datos)
- El sistema debe satisfacer las siguientes condiciones de sincronización
 - Varios lectores pueden acceder a la BD simultáneamente
 - Cada escritor debe acceder a la BD en exclusión mutua con cualquier otra hebra lector o escritor



Clases Lector y Escritor

```
public class Lector extends Thread{
    private int mild; // id de la hebra
    private ControlBD c;
    // objeto que controla el acceso a la BD
    private static Random r = new Random();
    // para dormir un tiempo aleatorio
    public Lector(int id,ControlBD c){
        mild = id;
        this.c = c;
    }

    public void run(){

        for (int i = 0; i<10;i++){
            try{
                c.openL(mild);
                // Lector mild en BD
                Thread.sleep(r.nextInt(500));
                c.CloseL(mild);
            } catch (InterruptedException ie){};
        }
    }
}
```

```
public class Escritor extends Thread{
    private int mild;
    private ControlBD c;
    private static Random r = new Random();
    public Escritor(int id,ControlBD c){
        mild = id;
        this.c = c;
    }

    public void run(){

        for (int i = 0; i<10;i++){
            try{
                c.openE(mild);
                // Escritor mild en BD
                Thread.sleep(r.nextInt(500));
                c.CloseE(mild);
            } catch (InterruptedException ie){};
        }
    }
}
```

openL, closeL, openE, closeE implementan las condiciones de sincronización

Condiciones de Sincronización: Lectores Escritores

```
public class ControlBD {  
    private int nLectores=0;  
    private boolean escribiendo=false;  
  
    public synchronized void openL(int id)  
        throws InterruptedException{  
        while (escribiendo)  
            wait();  
        nLectores++;  
    }  
    public synchronized void openE(int id)  
        throws InterruptedException{  
        while (escribiendo || nLectores > 0)  
            wait();  
        escribiendo = true;  
    }  
    ....  
}
```

```
....  
public synchronized void CloseL(int id) {  
    nLectores--;  
    if (nLectores == 0) notify();  
}  
  
public synchronized void CloseE(int id) {  
    escribiendo = false;  
    notifyAll();  
}  
}
```

- Si hay un escritor en la BD, los lectores deben esperar
- Si hay un escritor en la BD o hay lectores, los escritores deben esperar
- Observa que esta clase sólo modela el acceso a la BD, pero no la incluye ¿por qué?
- Cuando sale un escritor de la BD despierta todos las hebras que esperan.
 - 1) si un lector coge el lock del controlBD, ningún escritor puede entrar, pero sí el resto de lectores
 - 2) si un escritor coge el lock le cierra el paso al resto de lectores y escritores
- Si hay un lector en la BD, no hay ningún lector en el conjunto de espera
- Cuando sale el último lector despierta a una hebra únicamente (si hay alguna suspendida debe ser un escritor)

Condiciones de Sincronización: Lectores Escritores

- Problema de la solución
 - El escritor que sale de la BD debe hacer **notifyAll** porque si despierta a un lector deben pasar a la BD todos los lectores que estén esperando en el conjunto de espera del objeto controlBD
 - Si notifyAll() despierta primero a un escritor, el resto de hebras lectores han sido despertadas, pero posiblemente tendrán que volver a suspenderse. **Esta solución es ineficiente porque implica la reactivación innecesaria de muchas hebras.**
 - El código anterior es injusto para las hebras escritores. Si los lectores entran en la BD de forma ininterrumpida y continua, los escritores no pueden acceder a ella (se quedarían suspendidos de forma indefinida).



Condiciones de Sincronización: Lectores Escritores (versión justa)

```
....  
public synchronized void CloseL(int id) {  
    nLectores--;  
    if (nLectores == 0) notifyAll();  
}  
  
public synchronized void CloseE(int id) {  
    nEscritores--;  
    escribiendo = false;  
    notifyAll();  
}  
}
```

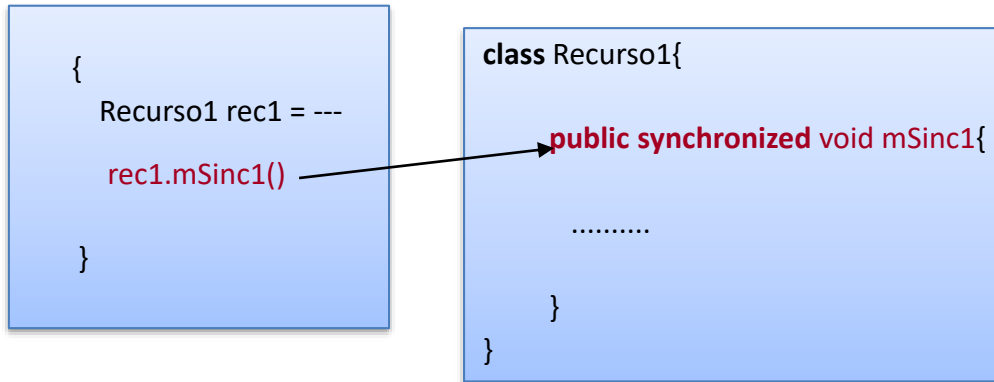
```
public class ControlBD {  
    private int nLectores=0;  
    private int nEscritores = 0;  
    private boolean escribiendo=false;  
  
    public synchronized void openL(int id)  
        throws InterruptedException{  
        while (escribiendo || nEscritores > 0)  
            wait();  
        nLectores++;  
    }  
    public synchronized void openE(int id)  
        throws InterruptedException{  
        nEscritores++;  
        while (escribiendo || nLectores > 0)  
            wait();  
        escribiendo = true;  
    }  
}
```

- La variable nEscritores cuenta el número de escritores que quieren actualizar la BD
- Si hay algún escritor esperando, aunque haya lectores en la BD, el siguiente lector espera
- Cuando un escritor sale de la BD, decrementa la variable nEscritores
- Esta solución es justa para los escritores, pero aún tiene el problema de que todas las hebras que esperan lo hacen en el mismo sitio, independientemente de la condición por la que esperan
- La suspensión y despertado de las hebras sigue siendo ineficiente.



Llamadas anidadas a métodos sincronizados

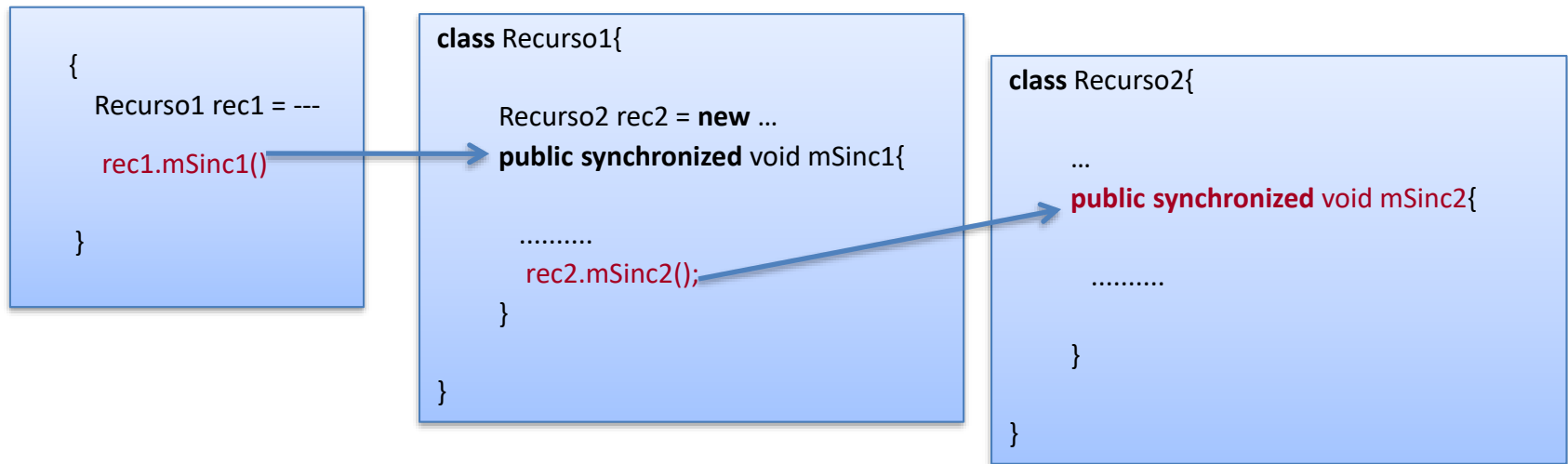
Objeto obj



- Para que el método **mSinc1** pueda ser ejecutado, **obj** debe hacerse primero con el lock del objeto **rec1**

Llamadas anidadas a métodos sincronizados

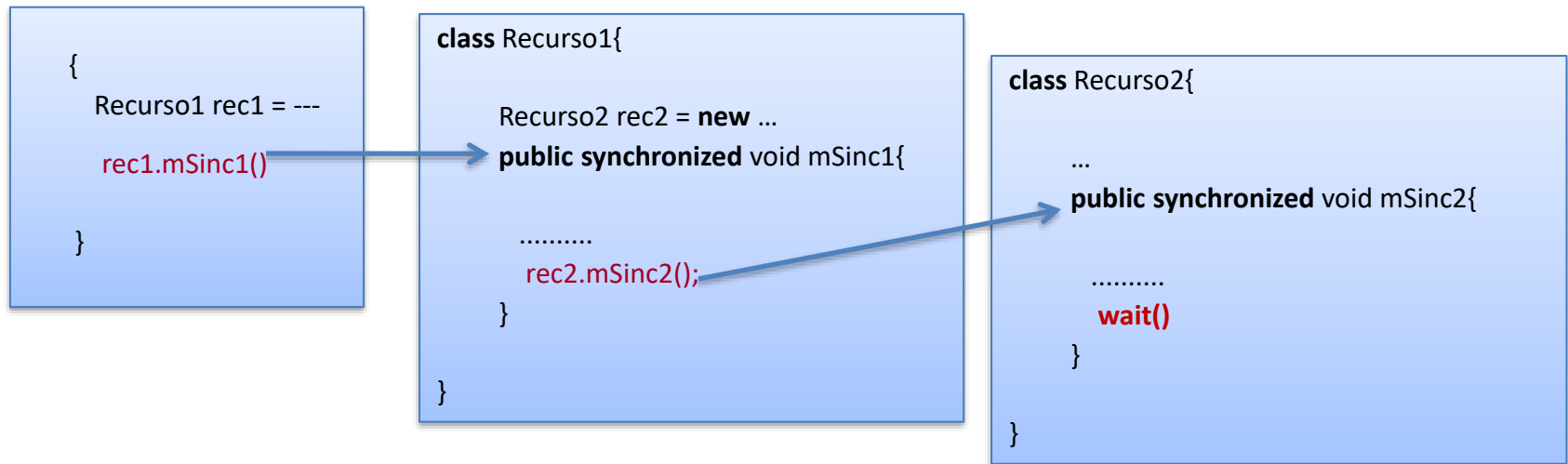
Objeto obj



- Para que el método **mSinc1** pueda ser ejecutado, **obj** debe hacerse primero con el lock del objeto **rec1**
- Para que el método **mSinc2** pueda ser ejecutado, **obj** debe hacerse también con el lock del objeto **rec2**. En este momento, **obj** tiene bloqueados a los dos recursos **rec1** y **rec2**.

Llamadas anidadas a métodos sincronizados

Objeto obj



- Para que el método **mSinc1** pueda ser ejecutado, **obj** debe hacerse primero con el lock del objeto **rec1**
- Para que el método **mSinc2** pueda ser ejecutado, **obj** debe hacerse también con el lock del objeto **rec2**. En este momento, **obj** tiene bloqueados a los dos recursos **rec1** y **rec2**.
- Si **mSinc2** llama a **wait()**, se libera el lock de **rec2**, pero **obj** sigue manteniendo el lock de **rec1**. Esta situación hay que tratarla con mucho cuidado ya que puede producir bloqueos.

Variables condición: primer intento

- La siguiente clase Condition intenta modelar una condición de sincronización por la que pueden esperar las hebras.
- Como cada objeto tiene su propio lock, varias variables de tipo Condition nos permiten agrupar a los objetos en distintos conjuntos de espera, según la condición por la que esperan.
- Por ejemplo, okLeer y okEscribir podrían servirnos para bloquear a las hebras Lector y Escritor, respectivamente.
- La ventaja es que cuando despertamos a una hebra, sabemos de qué tipo es.

```
public class Condition {  
    public synchronized void cwait()  
        throws InterruptedException{  
        wait();  
    }  
    public synchronized void cnotify(){  
        notify();  
    }  
    public synchronized void cnotifyAll(){  
        notifyAll();  
    }  
}
```

```
Condition okLeer = new Condition();  
Condition okEscribir = new Condition();
```

Variables condición: primer intento

```
public class ControlBDCondicion {
    private int nLectores=0;
    private int nEscritores = 0;
    private boolean escribiendo=false;
    private Condicion okLeer = new Condicion();
    private Condicion okEscribir = new Condicion();

    public synchronized void openL(int id)
        throws InterruptedException{
        while (escribiendo || nEscritores > 0){
            okLeer.cwait();
        }
        nLectores++;
    }
    public synchronized void openE(int id)
        throws InterruptedException{
        nEscritores++;
        while (escribiendo || nLectores > 0){
            okEscribir.cwait();
        }
        escribiendo = true;
    }
    ....
}
```

```
....
public synchronized void CloseL(int id) {
    nLectores--;
    if (nLectores == 0) okEscribir.cnotify();
}

public synchronized void CloseE(int id) {
    nEscritores--;
    escribiendo = false;
    if (nEscritores > 0) okEscribir.cnotify();
    else okLeer.notifyAll();
}
```

Sin embargo esta solución no es válida ya que puede bloquear al sistema, debido a que hay llamadas anidadas a métodos sincronizados

Variables condición: primer intento

```
public class ControlBDCondicion {
    private int nLectores=0;
    private int nEscritores = 0;
    private boolean escribiendo=false;
    private Condicion okLeer = new Condicion();
    private Condicion okEscribir = new Condicion();
    public synchronized void openL(int id)
        throws InterruptedException{
        while (escribiendo || nEscritores > 0)
            okLeer.cwait();
        nLectores++;
    }
    public synchronized void openE(int id)
        throws InterruptedException{
        nEscritores++;
        while (escribiendo || nLectores > 0)
            okEscribir.cwait();
        escribiendo = true;
    }
    public synchronized void CloseL(int id) {
        nLectores--;
        if (nLectores == 0) okEscribir.cnotify();
    }
    public synchronized void CloseE(int id) {
        nEscritores--;
        escribiendo = false;
        if (nEscritores > 0) okEscribir.cnotify();
        else okLeer.notifyAll();
    }
}
```

Acción	nLec	nEsc	Lock Control	lock okEscribir
Inicialmente	0	0	libre	libre
L1: c.openL()	1	0	libre	Libre
L2: c.openL()	2	0	libre	Libre
E1: c.openE()	2	1	E1	E1 espera notify
L1: c.closeL()	2	1	E1 L1 espera	E1 espera notify
L2: c.closeL()	2	1	E1 L1 espera L2 espera	E1 espera notify

```
public class Condicion {
    public synchronized void cwait()
        throws InterruptedException{
        wait();
    }
    public synchronized void cnotify(){
        notify();
    }
    public synchronized void cnotifyAll(){
        notifyAll();
    }
}
```

Los tres procesos
están bloqueados

Detección de bloqueo en el depurador de Eclipse

Debug - prEjemplos/src/capitulo6/ControlBDCondicion.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Variables Breakpoints

UsaBD [Java Application]
capitulo6.UsaBD at localhost: 1457
Daemon System Thread [Attach Listener] (Running)
Daemon System Thread [Signal Dispatcher] (Running)
Daemon System Thread [Finalizer] (Running)
Daemon System Thread [Reference Handler] (Running)

Thread [Thread-0] (Suspended)
waiting for: ControlBDCondition (id=24)
owned by: Thread [Thread-2] (Suspended)
waiting for: Condition (id=33)
ControlBDCondition.CloseL(int) line: 33
Lector.run() line: 22

Thread [Thread-2] (Suspended)
owns: ControlBDCondition (id=24)
waited by: Thread [Thread-1] (Suspended)
waited by: Thread [Thread-0] (Suspended)
waiting for: Condition (id=33)
Object.wait(long) line: not available [native method]
Condition(Object).wait() line: 485
Condition.cwait() line: 5
ControlBDCondition.openE(int) line: 24
Escritor.run() line: 18

Thread [Thread-1] (Suspended)
waiting for: ControlBDCondition (id=24)
owned by: Thread [Thread-2] (Suspended)
waiting for: Condition (id=33)
ControlBDCondition.openL(int) line: 11
Lector.run() line: 18

Thread [DestroyJavaVM] (Running)

C:\Archivos de programa\Java\jdk1.6.0\bin\javaw.exe (12/11/2008 15:36:53)

View Management...

Java

✓ Show Monitors
✓ Show System Threads
Show Qualified Names
Show Thread Groups

Para ver los locks de los monitores hay que activar Show Monitors en el menú View del depurador

Las hebras 0 y 1 son lectores y esperan el lock del objeto ControlBDCondition asignado actualmente a la hebra 2, que espera en el objeto condición (id=33)

Las hebras 2 es un escritor, posee el lock del objeto ControlBDCondition, que esperan las hebras 0 y 1 y actualmente espera el objeto condición (id=33)

Launch

Condiciones de Sincronización: Java 1.5

- Java 1.5 incluye nuevas funcionalidades para la programación concurrente
- Se distribuyen en tres paquetes:
 - `java.util.concurrent` - proporciona varias clases que soportan paradigmas típicos de programación concurrente como son buffers acotados, conjuntos y mapas, pools de hebras, etc.
 - `java.util.concurrent.atomic` - proporciona acceso seguro (sin necesidad de locks explícitos) a tipos de datos simples como atomic integers, atomic booleans etc.
 - `java.util.concurrent.locks` - proporciona varios tipos de locks que mejoran el mecanismo básico de Java, como locks de lectura/escritura y variables condición.

Locks

```
package java.util.concurrent.locks;  
public interface Lock {  
    public void lock();  
        // Espera hasta se obtiene el lock.  
    public Condition newCondition();  
        // Crea una nueva variable condición  
        // asociada al lock.  
    public void unlock();  
        // devuelve el control del lock  
    ...  
}
```

```
package java.util.concurrent.locks;  
public interface Condition {  
    public void await()  
        throws InterruptedException;  
        // De forma atómica libera el lock asociado  
        // a la condición y la hebra actual bloquea  
    public void signal();  
        // Despierta a una hebra de las que esperan  
        // en la condición.  
    public void signalAll();  
        // Despierta a todas las hebras que esperan  
        // en la condición  
    ...  
}
```

Java.util.concurrent.locks proporciona dos interfaces.

- una para crear nuevos locks
- otra para crear condiciones asociadas a los locks

Locks

```
public void synchronized incrementa (){  
    c++;  
}
```

Pido el lock

devuelvo el lock

```
public void incrementa(){  
    synchronize(this){  
        c++;  
    }  
}
```

Pido el lock

devuelvo el lock

Lock l = ...

```
public void incrementa(){  
    l.lock();  
    try{  
        c++;  
    } finally{  
        l.unlock();  
    }  
}
```

Pido el lock
explícitamente

Devuelvo el lock
explícitamente

La cláusula **try/finally** es necesaria para devolver el lock pase lo que pase en el cuerpo del try

- Los métodos/bloques **synchronized** permiten el acceso exclusivo al **lock implícito** asociado a un objeto, típicamente un recurso compartido por varias hebras
- Esta técnica es **transparente y elegante** y debería usarse siempre que se pueda. Sin embargo, presenta problemas como los mostrados en el ejemplo de los Lectores/Escritores
- Los métodos y bloques sincronizados **cogen y sueltan el lock del objeto considerado en puntos muy concretos** del código que el programador no puede cambiar. Además cuando hay llamadas anidadas a métodos/bloques sincronizados, los **locks de los objetos adquiridos se devuelven en orden inverso a como se cogieron**.
- Sin embargo, puede haber casos en que este mecanismo no sea el más conveniente.
- Los objetos Lock resuelven este problema, liberalizan el uso y posición de los locks y unlocks dentro del código, pero con la responsabilidad por parte del programador de asegurarse de que cuando se ha adquirido un lock debe liberarse, en algún momento.

Locks

- El paquete `java.util.concurrent.lock` proporciona varias implementación de la interfaz `Lock` con distintas características.

```
package java.util.concurrent.locks;
public class ReentrantLock implements Lock {
    public ReentrantLock();
    public ReentrantLock(boolean fair);

    ...
    public void lock();
    public Condition newCondition();
    public void unlock();
}
```

- `ReentrantLock` tiene un constructor que permite indicar que se quiere que el lock se comporte de forma justa.
- En este caso, la justicia significa que si hay varias hebras esperando en `lock`, el método `unlock` despierta a la hebra que lleva más tiempo esperando.
- Este tipo de locks evita que se produzca el error conocido como `starvation` o posposición indefinida que ocurre cuando una hebra espera indefinidamente a que la despierten para continuar su ejecución.
- Además la clase `ReentrantLock` resuelve satisfactoriamente las llamadas a `locks anidadas` que ocurren cuando un objeto llama de forma recursiva a locks (o métodos sincronizados que ya posee).

Locks: El problema de los Lectores Escritores

```
import java.util.concurrent.locks.*;
public class ControlBDLocks {
    private int nLectores=0;
    private int nEscritores = 0;
    private boolean escribiendo=false;
    private Lock l = new ReentrantLock(true);

    private Condition okLeer = l.newCondition();
    private Condition okEscribir = l.newCondition()

    public void openL(int id)
        throws InterruptedException{
        l.lock();
        try{
            while (escribiendo || nEscritores > 0){
                okLeer.await();
            }
            nLectores++;
        } finally {
            l.unlock();
        }
    }
    public void CloseL(int id) {
        l.lock();
        try{
            nLectores--;
            if (nLectores == 0) okEscribir.signal();
        } finally {
            l.unlock();
        }
    }
}
```

```
public void openE(int id)
    throws InterruptedException{
    l.lock()
    try{
        nEscritores++;
        while (escribiendo || nLectores > 0){
            okEscribir.await();
        }
        escribiendo = true;
    } finally {
        l.unlock();
    }
}

public void CloseE(int id) {
    l.lock();
    try{
        nEscritores--;
        escribiendo = false;
        if (nEscritores > 0) okEscribir.signal();
        else okLeer.signalAll();
    } finally{
        l.unlock();
    }
}
}
```



ReadWriteLock

- Interfaz ReadWriteLock y clase ReentrantReadWriteLock

```
package java.util.concurrent.locks;  
public interface ReadWriteLock {  
    Lock readLock();  
    Lock writeLock();  
}
```

```
package java.util.concurrent.locks;  
public class ReentrantReadWriteLock {  
    public ReentrantReadWriteLock();  
    public ReentrantReadWriteLock(boolean fair);  
    public Lock readLock();  
    public Lock writeLock();  
}
```

```
import java.util.concurrent.locks.*;  
public class ControlBDRWLock implements Control {  
    private ReadWriteLock l = new ReentrantReadWriteLock(true);  
    private Lock lr = l.readLock();  
    private Lock lw = l.writeLock();  
  
    public void openL(int id) throws InterruptedException {  
        lr.lock();  
    }  
    public void openE(int id) throws InterruptedException {  
        lw.lock();  
    }  
    public void CloseL(int id) {  
        lr.unlock();  
    }  
    public void CloseE(int id) {  
        lw.unlock();  
    }  
}
```

Java define también una interfaz ReadWriteLock que permite definir locks asociados a estructuras de datos con el protocolo de lectores/escritores descrito



Locks reentrantes

- Como los locks son reentrantes (reentrant), si una hebra pide un lock que ya tiene asignado ella misma, la petición tiene éxito inmediatamente.
- Cuando una hebra pide un lock que ya está asignado a otra hebra suspende, pero..
- El carácter reentrante de los lock se implementa asociando un contador a cada lock, así como una hebra propietaria.
 - Cuando el contador es cero, el lock está disponible
 - Cuando una hebra adquiere un lock que antes estaba disponible, JVM registra la hebra como propietaria del lock, e inicializa el contador a 1.
 - Si la hebra adquiere el lock de nuevo, el contador se incrementa
 - Si la hebra sale de un método sincronizado, el contador se decrementa en 1.
 - Cuando el contador se queda a 0, el lock vuelve estar disponible.
- Este comportamiento lo tienen también los locks de la clase `java.util.concurrent.locks.ReentrantLock`


Métodos/Objetos sincronizados frente a Lock

- Métodos/objetos sincronizados frente a Lock
 - Usar implementaciones de la interfaz Lock implica tener que pedir y liberar el lock explícitamente, lo que puede llevar a errores si no tenemos cuidado:
 - Olvidar pedir un lock puede llevar a violar la exclusión mutua de algún objeto
 - Olvidar devolver un lock puede llevar a bloqueos
 - En general, si en una clase no hace falta diferenciar entre varios tipos de condiciones de sincronización es mejor utilizar métodos u objetos sincronizados frente a Lock.
 - Otra característica de los métodos/objetos sincronizados es que los locks se devuelven en orden inverso a cómo fueron adquiridos, que es lo que normalmente nos interesa.
 - Sin embargo, si hay varias condiciones de sincronización o si queremos devolver los locks en otro orden tenemos que utilizar alguna implementación de la interfaz Lock.

Implementación de semáforos binarios con monitores: dos soluciones

```
public static class Semaforo{
    private int valor;

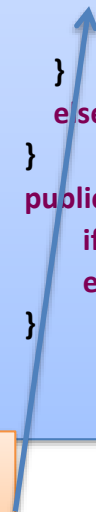
    public Semaforo(int v){
        valor = v;
    }
    public synchronized void acquire()
        throws InterruptedException{
        while (valor==0) wait();
        valor=0;
    }
    public synchronized void release(){
        valor=1;
        notify();
    }
}
```



En la primera solución, la hebra despertada puede tener que volver a bloquearse si se le ha colado otra

```
public static class Semaforo{
    private int valor;
    private int enEspera = 0;

    public Semaforo(int v){
        valor = v;
    }
    public synchronized void acquire()
        throws InterruptedException{
        if (valor==0) {
            enEspera++;
            wait();
            enEspera--;
        }
        else valor=0;
    }
    public synchronized void release() {
        if (enEspera>0) notify();
        else valor = 1;
    }
}
```



En esta segunda solución, la hebra despertada es la primera que completa el método acquire() porque es la única que sabe que el semáforo no es nulo

Semáforos vs Condiciones

- Supongamos que el proceso P1 no puede continuar su ejecución hasta que P2 haya ejecutado cierto código

```
Semaphore s = new Semaphore(0); {s = 0}
```

Acción	P1	P2	S
Inicialmente			0
P1:s.acquire()	Suspendido en s		0
P2:s.release()		Despierta a P1	0

Acción	P1	P2	S
Inicialmente			0
P2:s.release()			1
P1:s.acquire()			0

El orden de ejecución no es relevante porque el semáforo registra que se ha realizado una operación release.

Semáforos vs Condiciones

- La misma condición de sincronización con condiciones

```
Lock l = new ReentrantLock(true);  
Condition c = l.newCondition();
```

Acción	P1	P2	c
Inicialmente			
P1:c.await()	Suspendido en c		P1
P2:c.signal()			

Acción	P1	P2	c
Inicialmente			
P2: c.signal()			
P1:c.await()	Suspendido en c		P1

Cuando se usan condiciones, el orden de ejecución **sí** es importante. En el segundo caso, P1 queda suspendido, mientras que en el primer caso no.

Semáforos vs Condiciones

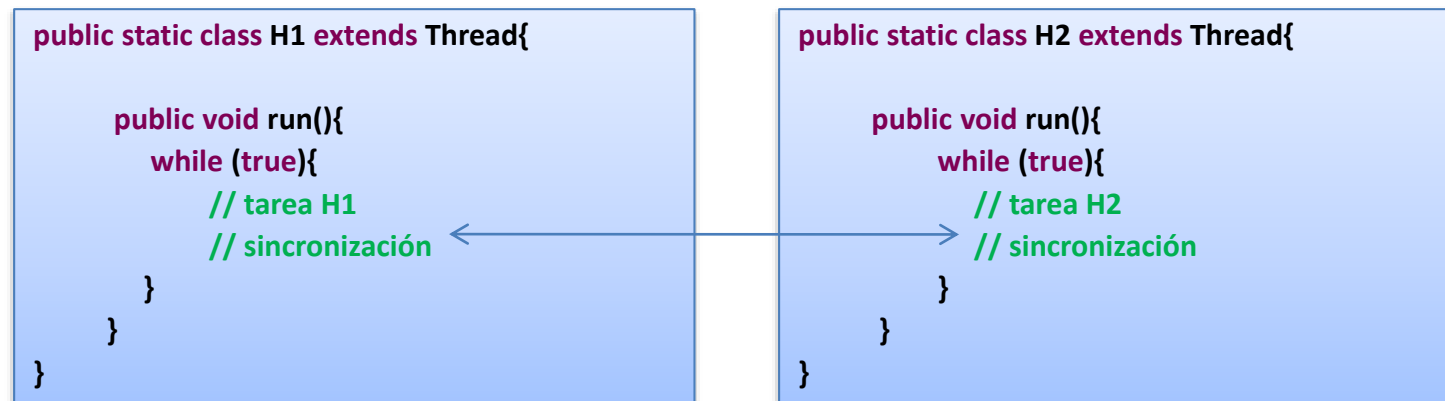
- Equivalencia

```
boolean ejecP2 = false;
```

Acción	Semáforos	Monitores
Suspensión de P1	s.acquire()	if (!ejecP2) c.await();
Reactivación de P1	s.release()	ejecP2 = true; c.signal()
Reactivación de P1	s.release()	ejecP2 = true; c.signal()
Suspensión de P1	s.acquire()	if (!ejecP2) c.await();

Implementación de una cita con condiciones

- Supongamos que tenemos dos hebras H1 y H2 que realizan infinitas veces dos tareas respectivas de forma asíncrona. Queremos que sus ejecuciones se sincronicen (se citen) en un punto determinado de cada uno de los códigos.




Implementación de una cita con condiciones

- La cita se convierte en dos condiciones de sincronización:
 - C1: H1 no puede continuar hasta que H2 no haya terminado la tarea 2
 - C2: H2 no puede continuar hasta que H1 no haya terminado la tarea 1

```
public static class H1 extends Thread{
    private Random r = new Random();
    private Sincro sinc = new Sincro();
    public H1(Sincro sinc){
        this.sinc = sinc;
    }
    public void run(){
        while (true){
            try {
                Thread.sleep(r.nextInt(500));
                System.out.println("fin H1");
                sinc.llegaH1();
                sinc.esperaH2();
            } catch (InterruptedException e) {}
        }
    }
}
```

```
public static class H2 extends Thread{
    private Random r = new Random();
    private Sincro sinc = new Sincro();
    public H2(Sincro sinc){
        this.sinc = sinc;
    }
    public void run(){
        while (true){
            try {
                Thread.sleep(r.nextInt(500));
                System.out.println("fin H2");
                sinc.llegaH2();
                sinc.esperaH1();
            } catch (InterruptedException e) {}
        }
    }
}
```



Implementación de una cita con condiciones

C1: Si H1 se adelanta ha de esperar a que H2 llegue a su punto de cita.

C2: Si H2 se adelanta ha de esperar a que H1 llegue a su punto de cita.

```
public static class Sincro{
```

```
    private boolean fin1 = false, fin2 = false;
```

```
    public synchronized void llegaH1(){
```

```
        fin1 = true;
```

```
        notify();
```

```
    }
```

```
    public synchronized void llegaH2(){
```

```
        fin2 = true;
```

```
        notify();
```

```
    }
```

```
    ...
```

```
}
```

```
...
```

```
public synchronized void esperaH1()
```

```
    throws InterruptedException{
```

```
    if (!fin1) wait();
```

```
    fin1 = false;
```

```
}
```

```
public synchronized void esperaH2()
```

```
    throws InterruptedException{
```

```
    if (!fin2) wait();
```

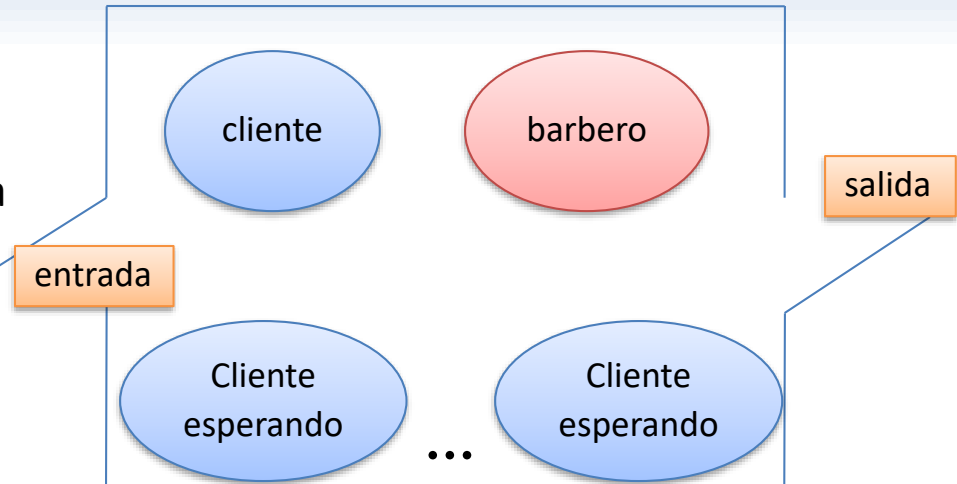
```
    fin2 = false;
```

```
}
```

```
...
```

El problema del barbero dormilón

- Supongamos que una ciudad tiene una barbería con dos puertas, una silla para el barbero, y unas sillas para los clientes que esperan. Los clientes entran por una puerta y salen por la otra.
- El barbero se pasa la vida pelando clientes, pero cuando no hay nadie en la barbería se echa un sueño en la silla del barbero.
- Cuando llega un cliente, si el barbero está durmiendo, lo despierta, se sienta en la silla del barbero y espera mientras que el barbero lo pela.
- Si llega un cliente y el barbero está ocupado, el cliente se duerme en una de las otras sillas.



- Cuando el barbero termina de pelar a un cliente, le abre la puerta de salida, que es cerrada por el cliente que sale.
- Si hay más clientes esperando el barbero despierta a uno de ellos, sino se va a dormir.

El problema del barbero dormilón

- Ejecución del sistema:
- *barbero* disponible → nuevo *cliente* se sienta en la silla → *barbero* pela cliente → *barbero* abre la puerta de salida → *cliente* se va cierra la puerta → *barbero* disponible
- Por lo tanto tenemos cuatro condiciones de sincronización:

El problema del barbero dormilón

- Ejecución del sistema:
- *barbero disponible* → *nuevo cliente se sienta en la silla* → *barbero pela cliente* → *barbero abre la puerta de salida* → *cliente se va cierra la puerta* → *barbero disponible*
- Por lo tanto tenemos cuatro condiciones de sincronización:
 - C1: Un nuevo cliente no se sienta en la silla del barbero hasta que el barbero no está disponible (boolean blibre; condition cblibre)

El problema del barbero dormilón

- Ejecución del sistema:
- *barbero* disponible → **nuevo *cliente* se sienta en la silla** → ***barbero* pela *cliente*** → *barbero* abre la puerta de salida → *cliente* se va cierra la puerta → *barbero* disponible
- Por lo tanto tenemos cuatro condiciones de sincronización:
 - C1: Un nuevo cliente no se sienta en la silla hasta que el barbero no esté disponible (boolean blibre; condition cblibre)
 - **C2: El barbero no pela al cliente hasta que no se ha sentado en su silla (boolean socupada; condition csocupada)**

El problema del barbero dormilón

- Ejecución del sistema:
- *barbero* disponible → nuevo *cliente* se sienta en la silla → *barbero* pela cliente → *barbero abre la puerta de salida* → *cliente se va cierra la puerta* → *barbero* disponible
- Por lo tanto tenemos cuatro condiciones de sincronización:
 - C1: Un nuevo cliente no se sienta en la silla hasta que el barbero no esté disponible (boolean blibre; condition cblibre)
 - C2: El barbero no pela al cliente hasta que no se ha sentado en su silla (boolean socupada; condition csocupada)
 - C3: El cliente no se va hasta que el barbero no le ha abierto la puerta (boolean pabierta; condition cpabierta)

El problema del barbero dormilón

- Ejecución del sistema:
- *barbero* disponible → nuevo *cliente* se sienta en la silla → *barbero* pela cliente → *barbero* abre la puerta de salida → *cliente se va cierra la puerta* → *barbero disponible*
- Por lo tanto tenemos cuatro condiciones de sincronización:
 - C1: Un nuevo cliente no se sienta en la silla hasta que el barbero no esté disponible (boolean blibre; condition cblibre)
 - C2: El barbero no pela al cliente hasta que no se ha sentado en su silla (var socupada:boolean; csocupada:condition)
 - C3: El cliente no se va hasta que el barbero no le ha abierto la puerta (boolean socupada; condition csocupada)
 - C4: El barbero no está disponible hasta que el cliente ha cerrado la puerta (condition cciclo)

El problema del barbero dormilón

```
public static class Barbero extends Thread{
    private Random r = new Random();
    private Barberia barb;
    public Barbero(Barberia barb){
        this.barb = barb;
    }
    public void run(){
        while (true){
            try {
                barb.siguiente();
                Thread.sleep(r.nextInt(500));
                //barbero pelando
                barb.finPelar();
            } catch (InterruptedException e) {}
        }
    }
}
```

```
public static class Cliente extends Thread{
    private Random r = new Random();
    private Barberia barb;
    private int id;
    public Cliente(Barberia barb,int id){
        this.barb = barb;
        this.id = id;
    }
    public void run(){
        try {
            barb.cortar(id);
        } catch (InterruptedException e) {}
    }
}
```

El problema del barbero dormilón

```
public static class Barberia{
    private Lock l = new ReentrantLock(true);
    private boolean bLibre = false, sOcupada = false,
        pAbierta = false;
    private Condition cbLibre = l.newCondition(),
        csOcupada = l.newCondition(),
        cpAbierta = l.newCondition(),
        cCiclo = l.newCondition();
    public void cortar(int id) throws InterruptedException{
        try{
            l.lock();
            while (!bLibre)
                cbLibre.await();
            bLibre = false;

            sOcupada = true;
            csOcupada.signal();

            while (!pAbierta)
                cpAbierta.await();
            pAbierta = false;
            cCiclo.signal();
        } finally {
            l.unlock();
        }
    } ....
}
```

C1

C2

C3

C4

```
public void siguiente() throws InterruptedException{
    try{
        l.lock();
        bLibre = true;
        cbLibre.signal();

        while (!sOcupada)
            csOcupada.await();
        sOcupada = false;
    } finally {
        l.unlock();
    }
}

public void finPelar() throws InterruptedException{
    try{
        l.lock();
        pAbierta = true;
        cpAbierta.signal();

        while (pAbierta)
            cCiclo.await();
    } finally {
        l.unlock();
    }
}
```

C1

C2

C3

C4

Interrupciones

```
public class Thread extends Object implements Runnable {  
    ...  
    public void interrupt();  
        // Envía una interrupción a la hebra receptora  
        // La hebra receptora tiene un flag booleano de interrupción que  
        // se pone a true  
  
    public boolean isInterrupted();  
        // Devuelve true si la hebra receptora ha sido interrumpida  
        // El flag de interrupción de la hebra no se modifica  
  
    public static boolean interrupted();  
        // Devuelve true si la hebra actual (currentThread),  
        // la que se está ejecutando, ha sido interrumpida  
        // y cambia el flag de interrupción de la hebra a no interrumpida.  
    ...  
}
```

Interrupciones

- Cada hebra tiene asociada internamente una variable booleana, denominada flag de interrupción, que puede utilizarse para parar su ejecución, y que se modifica cuando se ejecutan los métodos `interrupt()`, `isInterrupted()` y `interrupted()`

Cuando una hebra interrumpe a otra hebra:

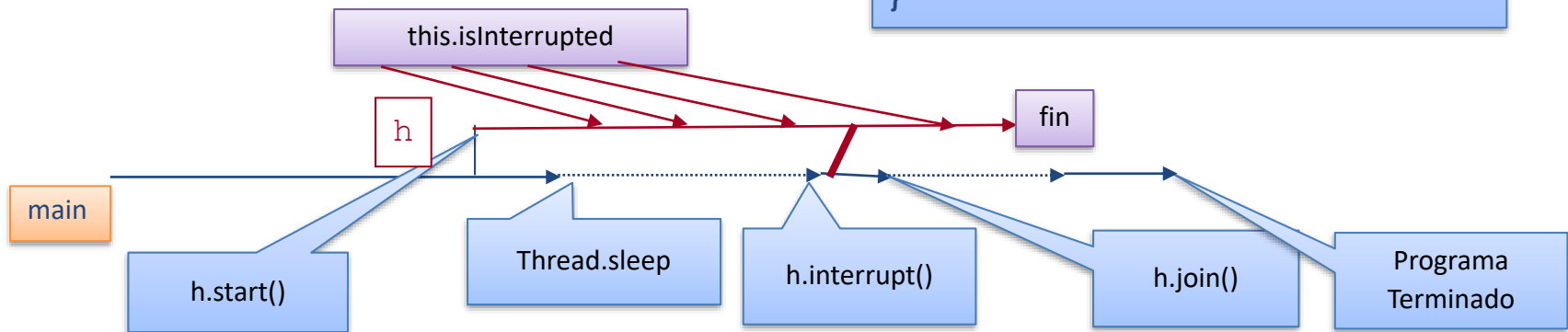
- Si la hebra interrumpida está bloqueada (en `wait`, `await`, `sleep`, `join`,...) pasa al estado “ejecutable” y se lanza la excepción comprobada `InterruptedException`
- Si la hebra está en ejecución en ese momento,
 - El flag de interrupción se pone a `true` para indicar que se ha recibido una interrupción, pero **la hebra sigue ejecutándose como si no hubiera sido interrumpida.**
 - Si a continuación la hebra intenta bloquearse (en `wait`, `await`, `sleep`, `join`,...) se hace “ejecutable” inmediatamente, se lanza la excepción comprobada `InterruptedException` y **además el flag de interrupción se resetea poniéndose de nuevo a false.**
- Por esto, si la hebra quiere terminar su ejecución cuando es interrumpida,
 - debe utilizar la excepción `InterruptedException`, para detectar la interrupción, y
 - debe chequear periódicamente su flag de interrupción a través de los métodos `isInterrupted` o `interrupted`

Interrupciones: Ejemplo

- La clase Hebra permite crear hebras muy simples, sólo escriben una secuencia infinita de números en la pantalla. Esta tarea no terminaría a menos que la hebra sea interrumpida.
- La clase Principal crea un objeto de la clase hebra, y lo pone en ejecución. Luego espera durante unos segundos, y la interrumpe. A continuación, espera a que la hebra termine para imprimir el mensaje "Programa terminado".

```
public class Hebra extends Thread{  
    private int i = 0;  
    public void run(){  
        while (!isInterrupted()){  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

```
public class Principal {  
    public static void main(String[] args) {  
        Hebra h = new Hebra();  
        h.start();  
        try{  
            Thread.sleep(1000);  
            h.interrupt();  
            h.join();  
        } catch (InterruptedException ie){}  
        System.out.println("Programa terminado");  
    }  
}
```



Interrupciones: El problema del productor consumidor

```
import java.util.concurrent.*;
import java.util.*;
public class Productor extends Thread{

    private BlockingQueue<Integer> buffer;

    private Random r = new Random();

    public Productor (BlockingQueue<Integer> buffer){
        this.buffer = buffer;
    }

    public void run(){
        boolean fin = false;
        int cont = 0;
        while (!isInterrupted() && !fin){
            try{
                Thread.sleep(50);
                buffer.put(r.nextInt(25));
                cont++;
            } catch (InterruptedException ie){fin = true;}
        }
        System.out.println("He generado "+cont+" elementos");
    }
}
```

- Utilizamos la interfaz **BlockingQueue** del paquete **java.util.concurrent**
- Esta interfaz implementa un Buffer Acotado (como el visto en clase).
- Las operaciones para insertar y extraer elementos del buffer se denominan **put** y **take**.
- Estas dos operaciones conllevan la suspensión de la hebra que las ejecuta, si el buffer no se encuentra en el estado adecuado. Lanza la excepción **InterruptedException**, si la hebra es interrumpida, mientras que está bloqueada en **put** o **take**.
- El buffer se pasa como parámetro a los constructores de las clases Productor y Consumidor

Interrupciones. El problema del productor consumidor

```
import java.util.concurrent.*;
import java.util.*;
public class Productor extends Thread{

    private BlockingQueue<Integer> buffer;

    private Random r = new Random();

    public Productor (BlockingQueue<Integer> buffer){
        this.buffer = buffer;
    }

    public void run(){
        boolean fin = false;
        int cont = 0;
        while (!isInterrupted() && !fin){
            try{
                Thread.sleep(50);
                buffer.put(r.nextInt(25));
                cont++;
            } catch (InterruptedException ie){fin = true;}
        }
        System.out.println("He generado "+cont+" elementos");
    }
}
```

- El productor produce e inserta en el buffer números aleatorios hasta que alguien (otra hebra) interrumpe su ejecución.
- Si la hebra es interrumpida y no está bloqueada en el método **put**, el **flag de interrupción** permanece a **true**, y por lo tanto el método **isInterrupted** devuelve **true**, y la hebra termina su ejecución.
- Por el contrario, si la hebra está bloqueada en el método **put** (o va a ejecutarlo a continuación) se lanza la excepción **InterruptedException**, y se cambia el **flag de interrupción** de la hebra a **false**. Por lo tanto, la llamada al método **isInterrupted** devuelve **false**, y es necesario usar la variable booleana **fin** para terminar la iteración de la hebra.

Interrupciones. El problema del productor consumidor

```
public class Consumidor extends Thread{

    private BlockingQueue<Integer> buffer;

    public Consumidor (BlockingQueue<Integer> buffer){
        this.buffer = buffer;
    }

    public void run(){

        boolean fin = false;
        int cont = 0;
        while (!isInterrupted() && (!fin) || buffer.size() > 0){
            try{
                Thread.sleep(50);
                System.out.println(isInterrupted());
                int i = buffer.take();
                cont++;
                System.out.println(i);
            } catch (InterruptedException ie){fin = true;}
        }
        System.out.println("He consumido "+cont+" elementos.");
    }
}
```

- La terminación de la hebra consumidora es un poco más complicada.
- Debe acabar, cuando la interrumpen, y además no haya datos en el buffer sin consumir.

Interrupciones. El problema del productor consumidor

```
public static void main(String[] args){
    BlockingQueue<Integer> buffer = new ArrayBlockingQueue<Integer>(5);

    Productor p = new Productor(buffer);
    Consumidor c = new Consumidor(buffer);
    p.start();
    c.start();
    try{
        Thread.sleep(1000);
        p.interrupt();
        p.join();
        c.interrupt();
        c.join();
    }catch (InterruptedException ie){}
    System.out.println("Programa terminado");
}
```

- Este es un ejemplo de uso e inicialización del sistema.
- Utilizamos la implementación **ArrayBlockingQueue** para crear el buffer. Le damos como parámetro el tamaño que queremos que tenga el buffer. Hay otros parámetros, que pueden verse en la documentación.
- El programa principal, crea las hebras, y las interrumpe por orden: primero al productor y luego al consumidor.

Referencias

- Concurrent Programming
Alan Burns, Geoff Davies, Ed. Addison Wesley
- Concurrent and Real Time Programming in Java
Andy Wellings, Ed. Willey
- Foundations of Multithreaded, Parallel and
Distributed Programming
Andrews, G. R., Addison-Wesley (2000)