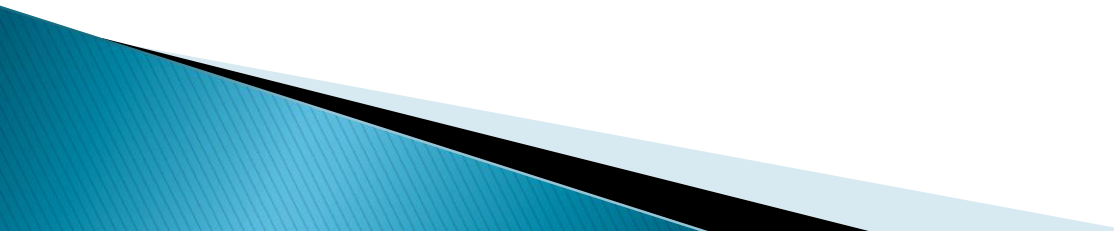


# CAPÍTULO 2: PROGRAMACIÓN DE HILOS

- ▶ Programación de Servicios y Procesos

# ÍNDICE

- ▶ Concepto Hilo
  - ▶ Multitarea
  - ▶ Recursos compartidos por Hilos
  - ▶ Estados de un Hilo
  - ▶ Gestión de Hilos
  - ▶ Planificación de Hilos
  - ▶ Sincronización de Hilos
  - ▶ Mecanismos de sincronización
- 

# Hilo

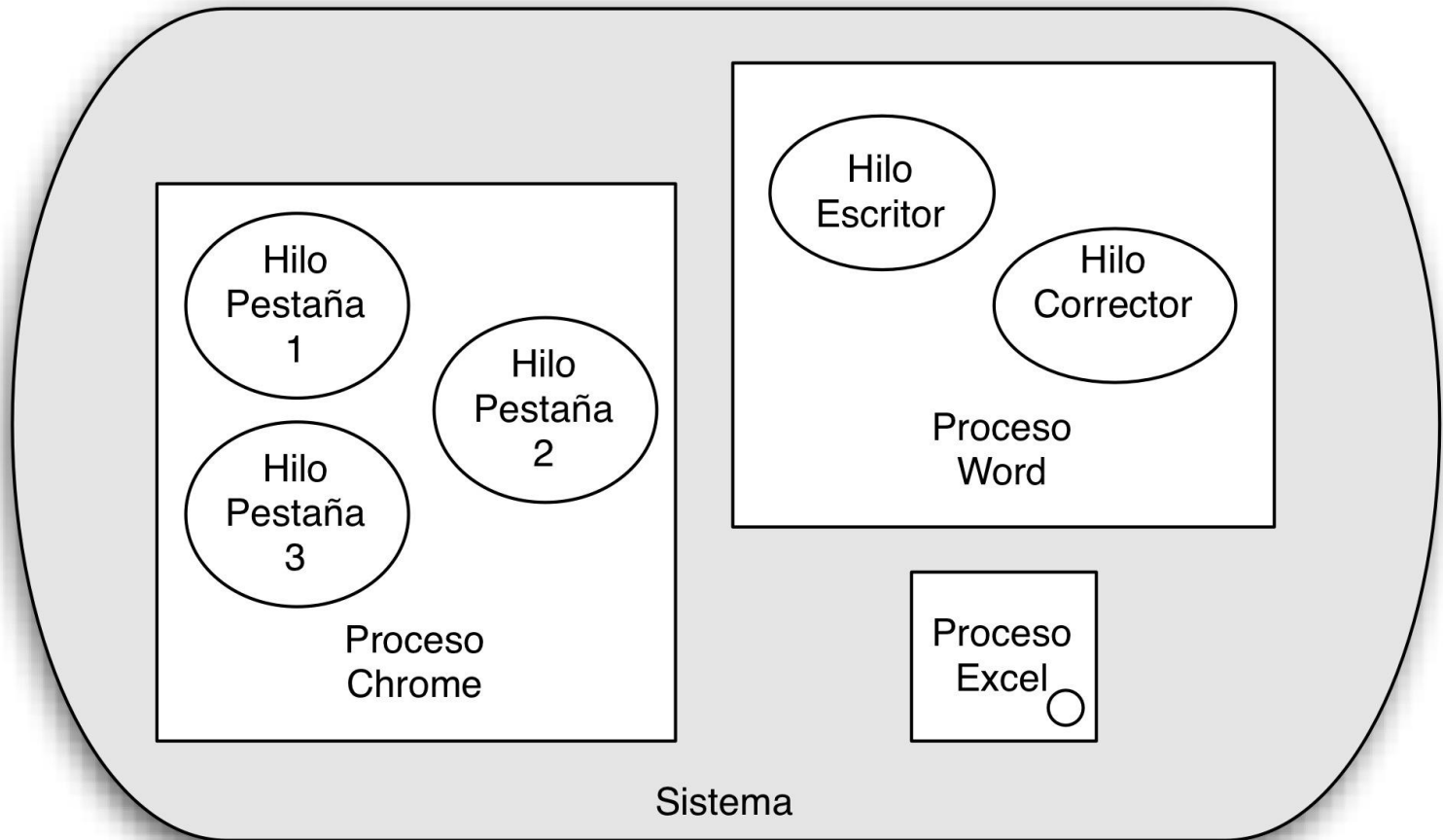
## ▶ Hilo (thread):

- Unidad básica de utilización de la CPU, y más concretamente de un *core* del procesador.
- Secuencia de código que está en ejecución, pero dentro del contexto de un proceso.

## ▶ Diferencia con procesos:

- Los hilos se ejecutan dentro del contexto de un proceso. Dependen de un proceso para ejecutarse.
- Los procesos son independientes y tienen espacios de memoria diferentes.
- Dentro de un mismo proceso pueden coexistir varios hilos ejecutándose que compartirán la memoria de dicho proceso.

# Hilo

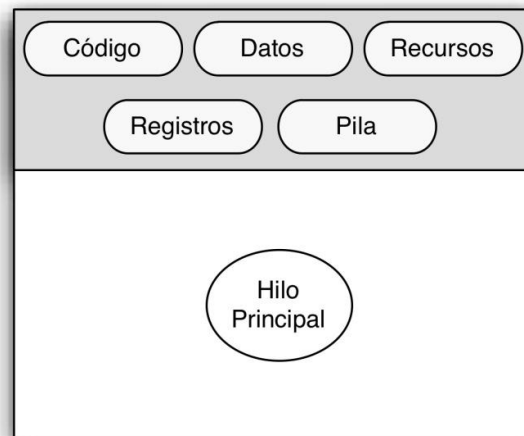


# Multitarea

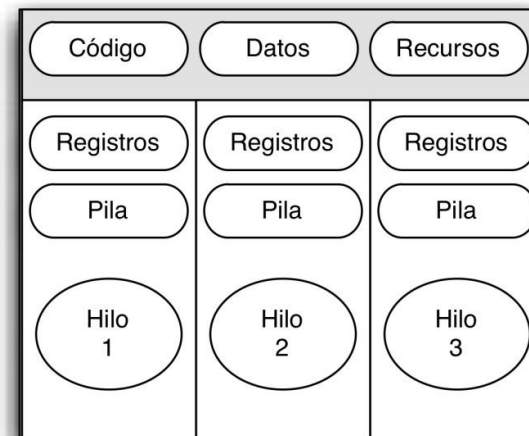
- ▶ Multitarea: ejecución simultánea de varios hilos:
  - Capacidad de respuesta. Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas (hilo) que esté realizando el programa sea muy larga.
  - Compartición de recursos. Por defecto, los *threads* comparten la memoria y todos los recursos del proceso al que pertenecen.
  - La creación de nuevos hilos no supone ninguna reserva adicional de memoria por parte del sistema operativo.
  - Paralelismo real. La utilización de *threads* permite aprovechar la existencia de más de un núcleo en el sistema en arquitecturas *multicore*.

# Recursos compartidos por Hilos

- ▶ Los procesos mantienen su propio espacio de direcciones y recursos de ejecución mientras que los hilos dependen del proceso.
  - Comparten con otros hilos la sección de código, datos y otros recursos.
  - Cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.



Proceso con un único thread



Proceso con varios threads

# Estados de un hilo

- ▶ Los hilos pueden cambiar de estado a lo largo de su ejecución.
- ▶ Se definen los siguientes estados:
  - Nuevo: el hilo está preparado para su ejecución pero todavía no se ha realizado la llamada correspondiente en la ejecución del código del programa.
  - Listo: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
  - *Runnable*: el hilo está preparado para ejecutarse y puede estar ejecutándose.
  - Bloqueado: el hilo está bloqueado por diversos motivos esperando que el suceso suceda para volver al estado *Runnable*.
  - Terminado: el hilo ha finalizado su ejecución.

# Gestión de Hilos

## ► Operaciones básicas:

- Creación y arranque de hilos. Cualquier programa a ejecutarse es un proceso que tiene un hilo de ejecución principal. Este hilo puede a su vez crear nuevos hilos que ejecutarán código diferente o tareas, es decir el camino de ejecución no tiene por qué ser el mismo.
- Espera de hilos. Como varios hilos comparten el mismo procesador, si alguno no tiene trabajo que hacer, es bueno suspender su ejecución para que haya un mayor tiempo de procesador disponible.



# Creación y Arranque de hilos

- ▶ Creación de hilos en Java:
  - Implementando la interfaz *Runnable*. La interfaz *Runnable* proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz. La interfaz *Runnable* debería ser utilizada si la clase solamente va a utilizar la funcionalidad *run* de los hilos.
  - Extendiendo de la clase *Thread* mediante la creación de una subclase. La clase *Thread* es responsable de producir hilos funcionales para otras clases e implementa la interfaz *Runnable*
- ▶ El método *run()* implementa la operación *create* conteniendo el código a ejecutar por el hilo. Dicho método contendrá el hilo de ejecución.
- ▶ La clase *Thread* define también el método *start()* para implementar la operación *create*. Este método es que comienza la ejecución del hilo de la clase correspondiente.

# Creación y Arranque de hilos

- ▶ Implementando la interfaz *Runnable*:

```
class HelloThread implements Runnable {  
  
    Thread t;  
    HelloThread () {  
        t = new Thread(this, "Nuevo Thread");  
        System.out.println("Creado hilo: " + t);  
        t.start(); // Arranca el nuevo hilo de ejecución. Ejecuta run  
    }  
  
    public void run() {  
        System.out.println("Hola desde el hilo creado!");  
        System.out.println("Hilo finalizando.");  
    }  
}  
  
class RunThread {  
    public static void main(String args[]) {  
        new HelloThread(); // Crea un nuevo hilo de ejecución  
        System.out.println("Hola desde el hilo principal!");  
        System.out.println("Proceso acabando.");  
    }  
}
```

# Creación y Arranque de hilos

- ▶ Extendiendo la clase *Thread*

```
class HelloThread extends Thread {
```

```
    public void run() {  
        System.out.println("Hola desde el hilo creado!");  
    }
```

```
}
```

```
public class RunThread {
```

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

```
        new HelloThread().start();// Crea y arranca un nuevo hilo de ejecución  
        System.out.println("Hola desde el hilo principal!");  
        System.out.println("Proceso acabando.");  
    }
```

```
}
```



# Creación y Arranque de hilos

- ▶ De las dos alternativas, ¿cuál utilizar? Depende de la necesidad:
  - La utilización de la interfaz *Runnable* es más general, ya que el objeto puede ser una subclase de una clase distinta de *Thread*
  - La utilización de la interfaz *Runnable* no tiene ninguna otra funcionalidad además de *run()* que la incluida por el programador.
  - La extensión de la clase *Thread* es más fácil de utilizar, ya que está definida una serie de métodos útiles para la administración de hilos,
  - La extensión de la clase *Thread* está limitada porque las clases creadas como hilos deben ser descendientes únicamente de dicha clase.

# Espera de hilos

- ▶ Operaciones:
  - Join: La ejecución del hilo puede suspenderse esperando hasta que el hilo correspondiente por el que espera finalice su ejecución.
  - Sleep: duerme un hilo por un período especificado
- ▶ Ambas operaciones de espera pueden ser interrumpidas, si otro hilo interrumpe al hilo actual mientras está suspendido por dichas llamadas.
  - Una interrupción es una indicación a un hilo que debería dejar de hacer lo que esté haciendo para hacer otra cosa.
  - Un hilo envía una interrupción mediante la invocación del método *interrupt()* en el objeto del hilo que se quiere interrumpir.
- ▶ *isAlive()*: comprueba si el hilo no ha finalizado su ejecución antes de trabajar con él.

# Espera de hilos

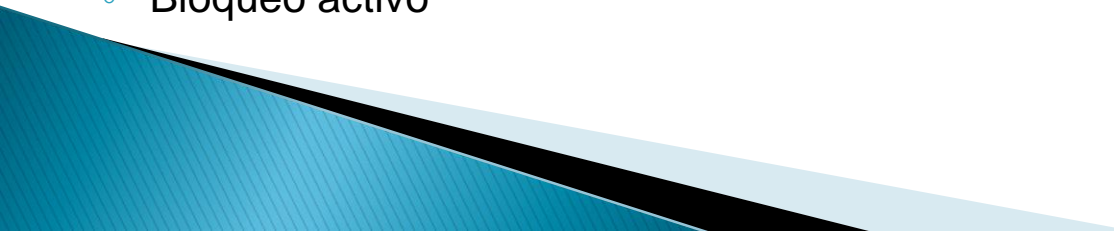
## ► Gestión de interrupciones

```
public void run() {  
    for (int i = 0; i < NDatos; i++) {  
        try {  
            System.out.println("Esperando a recibir dato!");  
            Thread.sleep(500);  
        } catch (InterruptedException e) {  
            System.out.println("Hilo interrumpido.");  
            return;  
        }  
        // Gestiona dato i  
    }  
    System.out.println("Hilo finalizando correctamente.");  
}
```

# Planificación de Hilos

- ▶ Cuando se trabaja con varios hilos, a veces es necesario pensar en la planificación de *threads*, para asegurarse de que cada hilo tiene una oportunidad justa de ejecutarse.
- ▶ El planificador del sistema operativo determina qué proceso es el que se ejecuta en un determinado momento en el procesador (uno y solamente uno).
  - Dentro de ese proceso, el hilo que se ejecutará estará en función del número de núcleos disponibles y del algoritmo de planificación que se esté utilizando.
  - Java, por defecto, utiliza un planificador apropiativo cuando un hilo que se está ejecutando pasa al estado *Runnable*. Si los hilos tienen la misma prioridad, será el planificador el que asigne a uno u otro el núcleo correspondiente para su ejecución utilizando tiempo compartido.

# Sincronización de Hilos

- ▶ Los *threads* se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria.
    - Los *threads* pertenecen al mismo proceso, y pueden acceder a toda la memoria asignada a dicho proceso utilizando las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente.
  - ▶ Cuando varios hilos manipulan a la vez objetos compartidos, pueden ocurrir diferentes problemas:
    - Condición de carrera
    - Inconsistencia de memoria
    - Inanición
    - Interbloqueo
    - Bloqueo activo
- 



# Problemas de Sincronización

## ▶ CONDICIONES DE CARRERA

- Si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.

## ▶ Ej: sumar o restar 1 en ensamblador

```
registroX = cuenta  
registroX = registroX (operación: suma o resta) 1  
cuenta = registroX
```

## ▶ Supongamos *cuenta* vale 10, y dos hilos *sumador* y *restador* ejecutándose a la vez sobre *cuenta*. Puede suceder

T0: <i>sumador</i>	registro1 = cuenta {registro1 = 10}
T1: <i>sumador</i>	registro1 = registro1 + 1 {registro1 = 11}
T2: <i>restador</i>	registro2 = cuenta {registro2 = 10}
T3: <i>restador</i>	registro2 = registro2 - 1 {registro2 = 9}
T4: <i>sumador</i>	cuenta = registro 1 {cuenta = 11}
T5: <i>restador</i>	cuenta = registro2 {cuenta = 9}

## ▶ El resultado final *cuenta* = 9 es incorrecto. Condición de carrera

# Problemas de Sincronización

## ▶ INCONSISTENCIA DE MEMORIA

Se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato.

## ▶ INANICIÓN

- Cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto siempre toman el control antes que él por diferentes motivos.

## ▶ INTERBLOQUEO

- Se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado

## ▶ BLOQUE ACTIVO

- Es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro.

# Sincronización de hilos

- ▶ Las condiciones de carrera y las inconsistencias de memoria se producen porque se ejecutan varios hilos concurrentemente pudiendo ser ordenados de forma diferente a la esperada.
  - La solución pasa por provocar que cuando los hilos accedan a datos compartidos, los accesos se produzcan de forma ordenada o **síncrona**.
  - Cuando estén ejecutando código que no afecte a datos compartidos, podrán ejecutarse libremente en paralelo, proceso también denominado ejecución **asíncrona**.

# Sincronización de hilos

## ► CONDICIONES DE BERNSTEIN

Las condiciones de Bernstein describen que dos segmentos de código  $i$  y  $j$  son independientes y pueden ejecutarse en paralelo de forma asíncrona en diferentes hilos sin problemas, si cumplen una serie de condiciones:

### 1. Dependencia de flujo.

Todas las variables de entrada del segmento  $j$  tienen que ser diferentes de las variables de salida del segmento  $i$ . Si no fuera así, el segmento  $j$  dependería de la ejecución de  $i$ .

### 2. Antidependencia.

Todas las variables de entrada del segmento  $i$  tienen que ser diferentes de las variables de salida del segmento  $j$ . Es el caso contrario a la primera condición, ya que si no fuera así, el segmento  $i$  tendría una dependencia del otro segmento

### 3. Dependencia de salida.

todas las variables de salida del segmento  $i$  tienen que ser diferentes de las variables de salida del segmento  $j$ . En caso contrario, si dos segmentos de código escriben en el mismo lugar, el resultado será dependiente del último segmento que ejecutó.

## ► OPERACIÓN ATÓMICA

Es una operación que sucede completa sin interrupciones, por lo que ningún otro hilo puede leer o modificar datos relacionados mientras se esté realizando la operación. En sistemas multiprocesador, con múltiples hilos de ejecución, asegurar atomicidad es muy complicado. Una forma de asegurar atomicidad es declarando las variables como *volatile*.

# Sincronización de hilos

Los desarrolladores que buscan aprovechar los beneficios de la programación paralela para acelerar sus aplicaciones se enfrentan a una serie de desafíos a la hora de compartir datos entre unidades de ejecución — *conocidos como hilos* — , uno de ellos es el de garantizar la consistencia de los datos que comparten. Para esto, Java provee dos mecanismos básicos para asegurar la consistencia de datos cuando se comparte datos con múltiples hilos: los métodos sincronizados con *synchronized* y las variables compartidas con *volatile*.

Mirad esta página y revisad el código de ejemplo. Si tenéis tiempo, lo probáis

<https://medium.com/@pablocastelnovo/variables-volátiles-en-java-f5ae078bf8b9>



# Sincronización de hilos

## ▶ SECCIÓN CRÍTICA

Se denomina **sección crítica** a una región de código en la cual se accede de forma ordenada a variables y recursos compartidos, de forma que se puede diferenciar de aquellas zonas de código que se pueden ejecutar de forma asíncrona. Este concepto se puede aplicar tanto a hilos como a procesos concurrentes, la única condición es que compartan datos o recursos.

```
do {  
    Sección de entrada;  
    SECCIÓN CRÍTICA  
    Sección de salida;  
  
    SECCIÓN RESTANTE  
} while (TRUE);
```

- Cuando un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su correspondiente sección crítica, ordenando de esta forma la ejecución concurrente

# Sincronización de hilos

- ▶ El problema de la sección crítica consiste en diseñar un protocolo que permita a los procesos cooperar. Cualquier solución al problema de la sección crítica debe cumplir:
  1. Exclusión mutua: si un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su sección crítica.
  2. Progreso: si ningún proceso está ejecutando su sección crítica y hay varios procesos que quieren entrar en su sección crítica, solo aquellos procesos que están esperando para entrar pueden participar en la decisión de quién entra definitivamente.
  3. Espera limitada: debe existir un número limitado de veces que se permite a otros procesos entrar en su sección crítica después de que otro proceso haya solicitado entrar en la suya y antes de que se le conceda.
- ▶ Para la implementación de la sección crítica se necesita un mecanismo de sincronización tanto que actúe tanto antes de entrar en la sección crítica como después de salir de ejecutarla.

# Mecanismos de sincronización

## ▶ SEMAFOROS

- Un semáforo se representa como:
  - Variable entera donde su valor representa el número de instancias libres o disponibles en el recurso compartido.
  - Cola donde se almacenan los procesos o hilos bloqueados esperando para usar el recurso.

- Se accede mediante dos operaciones atómicas

```
wait(Semaphore S) {  
    S.valor--;  
    if (S.valor < 0) {  
        Añadir el proceso o hilo a la lista S.col  
        Bloquear la tarea  
    }  
}
```

```
signal(Semaphore S) {  
    S.valor++;  
    if(S.valor <= 0) {  
        Sacar una tarea P de la lista S.col  
        Despertar a P  
    }  
}
```



# Mecanismos de sincronización

## ► MUTEX

- Semáforo binario, también denominado *mutex* (*MUTual EXclusion*, “exclusión mutua” en español) que registra si un único recurso está disponible o no. Un *mutex* solo puede tomar los valores 0 y 1.
- Con un semáforo binario se puede resolver el problema de la sección crítica:

```
//Inicializado a 1 y compartido por varios procesos  
Semáforo S;
```

```
wait(S);  
    SECCIÓN CRÍTICA  
signal(S);
```

# Mecanismos de sincronización

## ▶ MONITORES

- Conjunto de métodos atómicos que proporcionan de forma sencilla exclusión mutua a un recurso. Los métodos indicados permiten que cuando un hilo ejecute uno de los mismos, solamente ese hilo pueda estar ejecutando un método del monitor.
- Mientras que un monitor no puede ser utilizado incorrectamente, los semáforos dependen del programador ya que debe proporcionar la correcta secuencia de operaciones para no bloquear el sistema.
- Para utilizar un monitor en Java se utiliza la palabra clave *synchronized* sobre una región de código para indicar que se debe ejecutar como si de una sección crítica se tratase. Existen dos formas de utilizarlo:
  - Métodos sincronizados.
  - Sentencias sincronizadas.

# Mecanismos de sincronización

## ▶ MÉTODOS SINCRONIZADOS

- Mecanismo para construir una sección crítica de forma sencilla. La ejecución por parte de un hilo de un método sincronizado de un objeto en Java imposibilita que se ejecute a la vez otro método sincronizado del mismo objeto por parte de otro hilo.
- Para crear un método sincronizado, solo es necesario añadir la palabra clave *synchronized* en la declaración del método, sabiendo que los constructores ya son síncronos por defecto.

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

- Cuando un hilo invoca un método sincronizado, adquiere automáticamente el monitor que el sistema crea específicamente para todo el objeto que contiene ese método.

# Mecanismos de sincronización

## ▶ SENTENCIAS SINCRONIZADAS

- Permite una sincronización de grano fino al sincronizar únicamente una región específica de código.
- Esta funcionalidad permite especificar el objeto que proporciona el monitor en vez de ser el objeto por defecto que se está ejecutando como ocurre en métodos sincronizados.

```
public void increment() {  
    synchronized(mutex1) {  
        GlobalVar.c1++;  
    }  
}
```

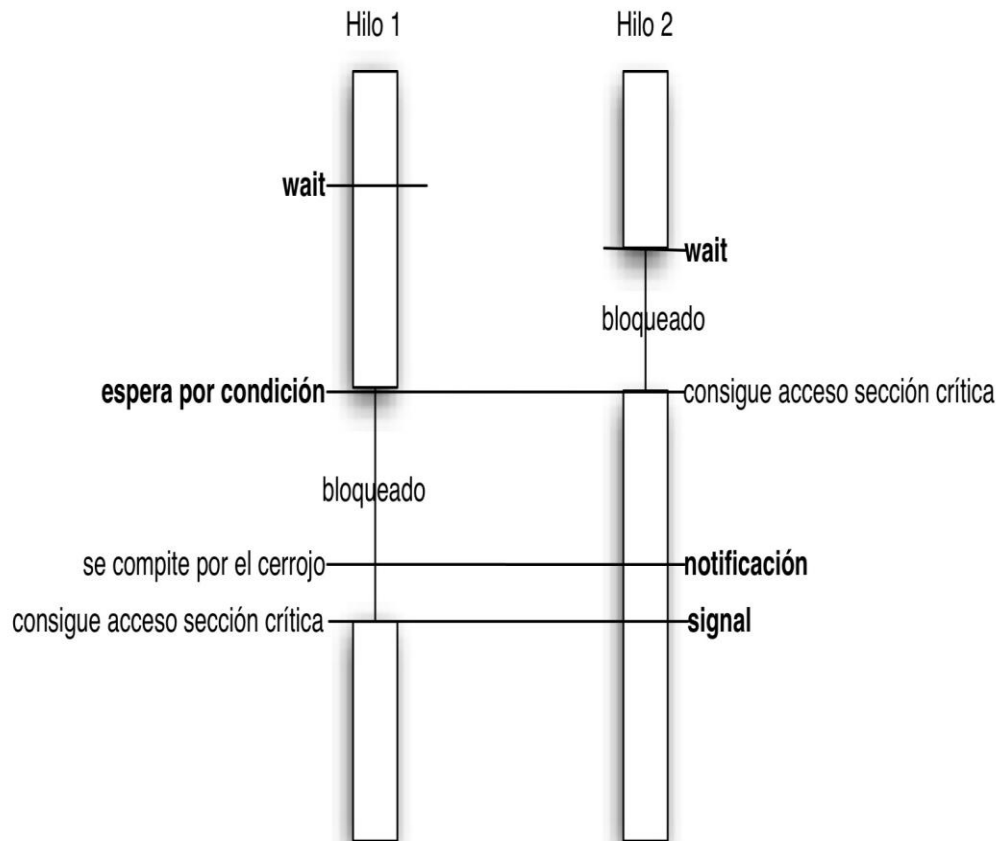
- **Sincronización reentrante:** permitir que un hilo pueda adquirir un monitor que ya tiene. El hilo está ejecutando código sincronizado, que, directa o indirectamente, invoca un método que también contiene código sincronizado, y ambos conjuntos de código utilizan el mismo monitor.

# Mecanismos de sincronización

## ► CONDICIONES

- A veces el hilo que se encuentra dentro de una sección crítica no puede continuar, al no cumplirse una **condición**. Sin embargo, esta condición solo puede ser cambiada por otro hilo desde dentro de su correspondiente sección crítica.
  - Es necesario que el hilo que no puede continuar libere temporalmente el cerrojo que protege la sección crítica mientras espera a que alguien le avise de la ocurrencia de dicha condición.
- Para implementar condiciones se utiliza:
  - *wait*: libera automáticamente el cerrojo sobre la sección crítica que se está ejecutando.
  - *notify*: implementa *wait*. Avisa de la ocurrencia de la condición por la que otro hilo espera.

# Mecanismos de sincronización



```
synchronized public void
comprobacion_ejecucion()
{
    // Seccion critica
    while (condicion) //no pueda continuar
    {
        wait();
    }
    // Seccion critica
}
```

```
synchronized public void aviso_condicion()
{
    // Seccion critica
    if (condicion se cumple)
        notify();
    // Seccion critica
}
```

# Programación de aplicaciones multihilo

- ▶ La **programación multihilo** permite la ejecución de varios hilos al mismo tiempo, tantos hilos como núcleos tenga el procesador, para realizar una tarea común.
- ▶ A la hora de realizar un programa multihilo cooperativo, se deben seguir las fases:
  - Descomposición funcional. Es necesario identificar previamente las diferentes tareas que debe realizar la aplicación y las relaciones existentes entre ellas.
  - Partición. La comunicación entre hilos se realiza principalmente a través de memoria. Los tiempos de acceso son muy rápidos por lo que no supone una pérdida de tiempo significativa. Ojo: Problemas de sincronización.
  - Implementación. Se utiliza la clase *Thread* o la interfaz *Runnable* como punto de partida. Utilizar mecanismos de sincronización.