

Programación de hilos

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender los conceptos básicos del funcionamiento de los sistemas en lo relativo a la ejecución de diferentes hilos.
- ✓ Comprender el concepto de paralelismo y cómo el sistema puede proporcionar multitarea al usuario.
- ✓ Saber utilizar los mecanismos de sincronización de hilos para construir aplicaciones paralelas.
- ✓ Familiarizarse con la programación de hilos entendiendo sus principios y formas de aplicación.

2.1

CONCEPTOS BÁSICOS

De igual manera que en el caso del capítulo anterior, antes de profundizar en la programación de hilos, es necesario tener claros algunos de los conceptos claves.

- **Hilo:** los hilos, o *threads*, son la unidad básica de utilización de la CPU, y más concretamente de un *core* del procesador. Así un *thread* se puede definir como la secuencia de código que está en ejecución, pero dentro del contexto de un proceso.
- **Hilo vs. Proceso:** hasta ahora hemos visto que el sistema operativo gestiona procesos, asignándoles la memoria y recursos que necesiten para su ejecución. En este sentido, el sistema operativo planifica únicamente procesos.

Los hilos se ejecutan dentro del contexto de un proceso, por lo que dependen de un proceso para ejecutarse. Mientras que 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. Esto sirve para que el mismo programa en ejecución (proceso) pueda realizar diferentes tareas (hilos) al mismo tiempo. Un proceso siempre tendrá, por lo menos, un hilo de ejecución que es el encargado de la ejecución del proceso.



EJEMPLO 2.1

Gracias al uso de hilos podemos tener diferentes pestañas abiertas en el navegador Google Chrome, cada una cargando a la vez una página web diferente, o Microsoft Word puede tener un hilo comprobando automáticamente la gramática, a la vez que se está escribiendo un documento.

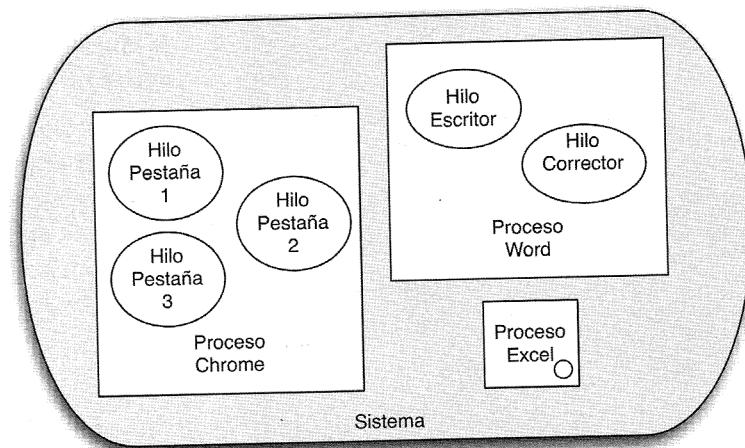


Figura 2.1. Relación entre hilos y procesos

Frente a la multiprogramación de procesos, la multitarea presenta bastantes ventajas:

- ✓ Capacidad de respuesta. Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas que esté realizando el programa sea muy larga. Este modelo se utiliza mayoritariamente en la programación de un servicio en servidores. Un hilo se encarga de recibir todas las peticiones del usuario y por cada petición se lanza un nuevo hilo para responderla. De esta forma se están tratando varias peticiones al mismo tiempo.
- ✓ Compartición de recursos. Por defecto, los *threads* comparten la memoria y todos los recursos del proceso al que pertenecen. No necesitan ningún medio adicional para comunicarse información entre ellos ya que todos pueden ver la información que hay en la memoria del proceso. Sin embargo, debido a que todos los hilos pueden acceder y modificar los datos al mismo tiempo, se necesitan medios de sincronización adicionales para evitar problemas en el acceso.
- ✓ Como los hilos utilizan la misma memoria del proceso del cual dependen, la creación de nuevos hilos no supone ninguna reserva adicional de memoria por parte del sistema operativo. Es más barato en términos de uso de memoria y otros recursos crear nuevos *threads* que crear nuevos procesos.
- ✓ Paralelismo real. La utilización de *threads* permite aprovechar la existencia de más de un núcleo en el sistema en arquitecturas *multicore*. Sabemos que el sistema operativo planifica procesos de forma concurrente, intercambiando los procesos uno por otro mediante un cambio de contexto. En este sentido, solamente puede haber un proceso en ejecución en un momento dado independientemente del número de núcleos del procesador ya que solo existe una memoria (cuando se desea acceder a un dato se realiza mediante la traducción de las referencias a direcciones de memoria que se encuentran en el código del proceso a las direcciones efectivas en memoria principal). Es decir, la gestión de procesos no puede aprovecharse de la existencia de varios núcleos. Sin embargo, varios hilos pueden ejecutarse en el contexto de un mismo proceso. Cuando ese proceso está en ejecución, los hilos programados del mismo pueden utilizar todos los núcleos del procesador de forma paralela, permitiendo que se ejecuten varias instrucciones (una por cada *thread* y núcleo) a la vez.



¿SABÍAS QUE...?

La tendencia del mercado es añadir cada vez un número mayor de núcleos en el sistema en vez de aumentar las prestaciones de los mismos. La idea es que la suma de las potencias de los núcleos sea mayor que la prestación ofrecida por un único procesador aunque los núcleos que se utilicen sean menos potentes de forma individual. La razón es que los procesadores menos potentes gastan menos energía por lo que se reduce el consumo. Así, se ha pasado de procesadores Intel Pentium 4 Prescott 570J a 3,8 GHz (3,8 millones de operaciones por segundo) en 2005 a Intel i7 Ivy Bridge de 2-3,8 GHz (8 núcleos de 2 hasta 3,8 millones de operaciones con *hyperthreading*, 2 *threads* en ejecución por cada núcleo) en 2012.

2.2

RECURSOS COMPARTIDOS POR HILOS

Un hilo es muy similar a un proceso pero con la diferencia de que un hilo siempre se ejecuta dentro del contexto de un proceso. Los procesos mantienen su propio espacio de direcciones y recursos de ejecución mientras que los hilos dependen del proceso, por lo que comparten con otros hilos la sección de código, datos y otros recursos. Únicamente cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.

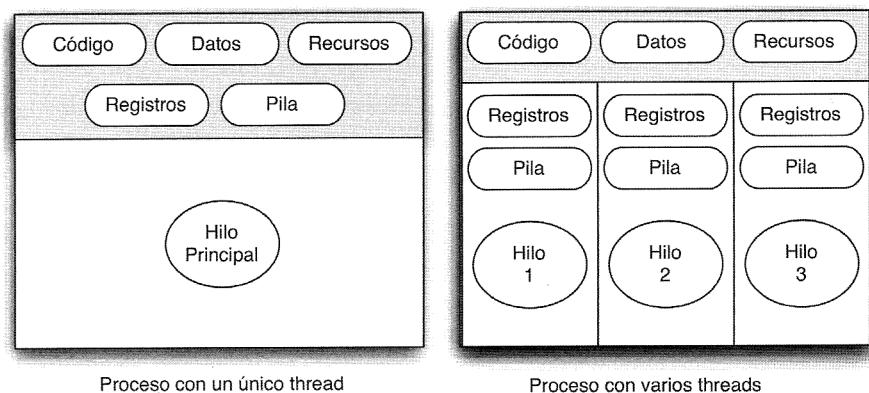


Figura 2.2. Recursos compartidos por hilos

2.3

ESTADOS DE UN HILO

Al igual que los procesos, los hilos pueden cambiar de estado a lo largo de su ejecución. Su comportamiento dependerá del estado en el que se encuentren:

- **Nuevo:** el hilo está preparado para su ejecución pero todavía no se ha realizado la llamada correspondiente en la ejecución del código del programa. Los hilos se inicializan en la creación del proceso correspondiente, ya que forman parte de su espacio de memoria pero no empiezan a ejecutar hasta que el proceso lo indica.
- **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. El planificador del sistema operativo es el encargado de elegir cuándo el proceso pasa a ejecutarse.
- **Pudiendo ejecutar (Runnable):** el hilo está preparado para ejecutarse y puede estar ejecutándose. A todos los efectos sería como si el *thread* estuviera en ejecución, pero debido al número limitado de núcleos no se puede saber si se está ejecutando o esperando debido a que no hay hardware suficiente para hacerlo. En general, por simplicidad se puede considerar que todos los *threads* del proceso se ejecutan al mismo tiempo (en paralelo) sabiendo que los hilos deben compartir los recursos del sistema.

- **Bloqueado:** el hilo está bloqueado por diversos motivos (esperando por una operación de E/S, sincronización con otros hilos, dormido, suspendido) esperando que el suceso suceda para volver al estado *Runnable*. Cuando ocurre el evento que lo desbloquea, el hilo pasaría directamente a ejecutarse.
- **Terminado:** el hilo ha finalizado su ejecución. Sin embargo, frente a los procesos que liberan los recursos que mantenían cuando finalizan, el hilo no libera ningún recurso ya que pertenecen al proceso y no a él mismo. Para terminar un hilo, él mismo puede indicarlo o puede ser el propio proceso el que lo finalice mediante la llamada correspondiente.

ACTIVIDADES 2.1

- Enumera en una lista las principales diferencias entre hilo y proceso.

2.4 GESTIÓN DE HILOS



OPERACIONES BÁSICAS

2.4.1.1 Creación y arranque de hilos (operación create)

Los *threads* comparten tanto el espacio de memoria del proceso como los recursos asociados (entorno de ejecución), siendo su creación más eficiente que la creación de procesos. Estos últimos tienen que crear estructuras especiales en el sistema además de hacer la reserva de memoria y recursos correspondiente. 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.



¿SABÍAS QUE?

La GPU (*Graphics Processing Unit*, en inglés, por similitud con el nombre de CPU, *Central Processing Unit*) es el coprocesador dedicado al procesamiento gráfico que se encuentra en las tarjetas gráficas de última generación, para aligerar la carga de trabajo de la CPU al ejecutar videojuegos, animaciones 3D, videos de alta definición, etc. Frente a la CPU, que puede contar con unos pocos núcleos, las GPU presentan múltiples núcleos (del orden de más de 100), lo que permite ejecutar un elevado número de hilos en paralelo. Sin embargo, las operaciones que pueden realizar estos hilos están limitadas y en general deben realizar todos la misma tarea con distintos datos. Si el algoritmo paralelo se adapta a estas condiciones, su implementación en GPU mediante los lenguajes de programación CUDA (*Compute Unified Device Architecture*, en inglés) de NVidia o OpenCL (*Open Computing Language*, en inglés) puede obtener una mejora muy significativa frente a su implementación en CPU mediante los lenguajes de programación tradicionales.

A la hora de crear los nuevos hilos de ejecución dentro de un proceso, existen dos formas de hacerlo en Java: implementando la interfaz *Runnable*, o extendiendo de la clase *Thread* mediante la creación de una subclase.

La interfaz *Runnable* proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz. Java proporciona soporte para hilos a través esta interfaz. La interfaz de Java necesaria para cualquier clase que implemente hilos es *Runnable*.

Las clases que implementan la interfaz *Runnable* proporcionan una forma de realizar la operación *create*, encargada de crear nuevos hilos. La operación *create* inicia un *thread* de la clase correspondiente, pasándolo del estado “nuevo” a “pudiendo ejecutar”.

En la utilización de la interfaz *Runnable*, el método *run()* implementa la operación *create* contenido el código a ejecutar por el hilo. Dicho método contendrá el hilo de ejecución, es decir viene a ser como el método *main()* en el hilo. Esto implica que el hilo finaliza su ejecución cuando finaliza el método *run()*. A menudo se denomina a este método “el cuerpo del hilo”.

Además de la interfaz *Runnable*, otra clase principal para el uso de hilos es la clase *Thread*. La clase *Thread* es responsable de producir hilos funcionales para otras clases e implementa la interfaz *Runnable*. La interfaz *Runnable* debería ser utilizada si la clase solamente va a utilizar la funcionalidad *run* de los hilos. En otro caso se debería derivar de la clase *Thread* modificando los métodos que se consideren. Eso sí, hay que tener en cuenta que Java no soporta herencia múltiple de forma directa, es decir, no se puede derivar una clase de varias clases padre. Para poder añadir la funcionalidad de hilo a una clase que deriva de otra clase, siendo esta distinta de *Thread*, se debe utilizar la interfaz *Runnable*.

Para añadir la funcionalidad de hilo a una clase mediante la clase *Thread* simplemente se deriva de dicha clase ignorando el método *run()* (proveniente de la interfaz *Runnable*). La clase *Thread* define también un método para implementar la operación *create* para comenzar la ejecución del hilo. Este método es *start()*, que comienza la ejecución del hilo de la clase correspondiente. Al ejecutar *start()*, la JVM llama al método *run()* del hilo que contiene el código de la tarea.

Para crear un hilo utilizando la interfaz *Runnable* se debe crear una nueva clase que implemente la interfaz, teniendo que implementar únicamente el método *run()* con la tarea a realizar. Además se debe crear una instancia de la clase *Thread* dentro de la nueva clase, la cual representará el hilo a ejecutar. Como dicho hilo pertenece a la clase *Thread* se debe utilizar *start()* para ponerlo en ejecución o arrancarlo.

```
public class NuevoThread implements Runnable {  
  
    Thread hilo;  
    public void run() {  
        // Código a ejecutar por el hilo  
    }  
}
```

El siguiente ejemplo muestra el proceso:

EJEMPLO 2.2

Creación de un hilo 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.");
    }
}
```

El otro mecanismo de creación de hilos, como ya hemos dicho, consistiría en la creación previa de una subclase de la clase *Thread*, la cual podríamos instanciar después. Es necesario sobrecargar el método *run()* con la implementación de lo que se desea que el hilo ejecute. Como hemos visto, nunca se ejecuta de forma directa este método, sino que se llama al método *start()* de dicha clase para arrancar el hilo. En este caso se heredan los métodos y variables de la clase padre.

```
public class NuevoThread extends Thread {

    public void run() {
        // Código a ejecutar por el hilo
    }
}
```

El siguiente ejemplo muestra el proceso:



EJEMPLO 2.3

Creación de un hilo 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.");  
    }  
}
```

A la hora de decantarse por una u otra opción hay que saber que la utilización de la interfaz *Runnable* es más general, ya que el objeto puede ser una subclase de una clase distinta de *Thread*, pero no tiene ninguna otra funcionalidad además de *run()* que la incluida por el programador. La segunda opción es más fácil de utilizar, ya que está definida una serie de métodos útiles para la administración de hilos, pero está limitada porque las clases creadas como hilos deben ser descendientes únicamente de la clase *Thread*. Con la implementación de la interfaz *Runnable*, se podría extender la clase hilo creada, si fuese necesario. Hay que recordar que siempre se arrancan los hilos ejecutando *start()*, la cual llamará al método *run()* correspondiente.

ACTIVIDADES 2.2



- 💡 Crea un hilo que realice el cálculo de los primeros N números de la sucesión de Fibonacci. La sucesión de Fibonacci comienza con los números 1 y 1 y el siguiente elemento es la suma de los dos elementos anteriores. Así la sucesión de Fibonacci sería 1, 1, 2, 3, 5, 8, 11, 19, 30, 49... El parámetro N será indicado cuando se llame al constructor de la clase *Thread* correspondiente.

2.4.1.2 Espera de hilos (operaciones *join* y *sleep*)

Si todo fue bien en la operación *create*, el objeto de la clase debería contener un hilo cuya ejecución depende del método *run()* especificado para ese objeto.

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 para el resto de hilos del sistema. La ejecución del hilo se puede suspender mediante la operación *join* esperando hasta que el hilo correspondiente finalice su ejecución.

Además se puede dormir un hilo mediante la operación *sleep* por un período especificado, teniendo en cuenta que el tiempo especificado puede no ser preciso, ya que depende de los recursos proporcionados por el sistema operativo subyacente.

2..4.1.2.1 interrupción

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. El hilo interrumpido recibe la excepción *InterruptedException* si están ejecutando un método que la lance (por ejemplo, los que implementan las operaciones *join* y *sleep*), pero podrían haber sido interrumpidos por *interrupt()* mientras tanto. En este sentido se debe invocar periódicamente el método *interrupted()* para saber si se ha recibido una interrupción.

Una vez comprobado si el hilo ha sido interrumpido se puede o bien finalizar su ejecución, o lanzar *InterruptedException* para manejárla en una sentencia *catch* centralizada en aplicaciones complejas.



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;
        }
        // Gestionar dato i
    }
    System.out.println("Hilo finalizando correctamente.");
}
```

Queda a criterio del programador decidir exactamente cómo un hilo responde a una interrupción, pero en muchos de los casos lo que se hace es finalizar su ejecución.

Por último, como no se puede controlar cuándo un *thread* finaliza su ejecución, al depender de su código en el método *run()*, se puede utilizar el método *isAlive()* para comprobar si el método no ha finalizado su ejecución antes de trabajar con él.



¿SABÍAS QUE...?

Los métodos `stop()`, `suspend()` y `resume()` de la clase `Thread` que se usaban para controlar hilos están deprecados desde la versión Java 1.4 porque pueden tener un comportamiento imprevisto. La parada de un *thread* causa que todos los cerros que había bloqueados se desbloqueen, pudiendo provocar un comportamiento imprevisto en el resto de *threads* que los estén utilizando. En este sentido debe evitarse su utilización utilizando **variables globales** para indicar la finalización del hilo.

Por ejemplo:

```
private Thread ejemplo;

public void start() {
    ejemplo = new Thread(this);
    ejemplo.start();
}

public void stop() {
    ejemplo.stop(); // INSEGURO!
}

public void run() {
    while (true) {
        try {
            Thread.sleep(intervalo);
        } catch (InterruptedException e) {
        }
        ...
    }
}
```

Podría modificarse de la siguiente forma:

```
private volatile Thread ejemplo;

public void stop() {
    ejemplo = null;
}

public void run() {
    Thread thisThread = Thread.currentThread();
    while (ejemplo == thisThread) {
        try {
            Thread.sleep(intervalo);
        } catch (InterruptedException e) {
        }
        ...
    }
}
```

El `volatile` le indica al compilador que es posible que el objeto vaya a ser modificado simultáneamente por varios hilos, y que y asíncronas no queremos guardar una copia local en cache por cada hilo, sino que queremos que los valores de todos los hilos estén sincronizados a través de un paso de rendimiento.

2.4.1.3 Clase Thread

A la hora de utilizar hilos en Java se utiliza la clase *Thread*. Se presenta, por simplicidad, una tabla con los métodos recogidos más importantes explicados a lo largo del capítulo:

Método	Tipo de retorno	Descripción
<i>start()</i>	<i>void</i>	Implementa la operación <i>create</i> . Comienza la ejecución del hilo de la clase correspondiente. Llama al método <i>run()</i>
<i>run()</i>	<i>void</i>	Si el hilo se construyó implementando la interfaz <i>Runnable</i> , entonces se ejecuta el método <i>run()</i> de ese objeto. En caso contrario, no hace nada
<i>currentThread()</i>	<i>static Thread</i>	Devuelve una referencia al objeto hilo que se está ejecutando actualmente
<i>join()</i>	<i>void</i>	Implementa la operación <i>join</i> para hilos
<i>sleep(long milis)</i>	<i>static void</i>	El hilo que se está ejecutando se suspende durante el número de milisegundos especificado
<i>interrupt()</i>	<i>void</i>	Interrumpe el hilo del objeto
<i>interrupted()</i>	<i>static boolean</i>	Comprueba si el hilo ha sido interrumpido
<i>isAlive()</i>	<i>boolean</i>	Devuelve <i>true</i> en caso de que el hilo esté vivo, es decir, no haya terminado el método <i>run()</i> en su ejecución

2.4.2 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. Por ejemplo, si se utiliza uno basado en prioridades, el valor de prioridad de un hilo indica que en caso de no disponibilidad de núcleos para la ejecución de todos los hilos del proceso en ejecución a la vez, se ejecute antes el que tenga mayor prioridad sobre uno que tenga una valor de prioridad menor.

Como algoritmos de planificación de hilos se pueden entender los mismos algoritmos vistos para procesos (ver *Planificación de procesos*, Capítulo 1). Esto implica que, en función del algoritmo de planificación, por ejemplo en planificación cooperativa, se pueda ejecutar completamente un hilo de prioridad superior, hasta que cambie su estado a *Bloqueado* o *Terminado*, antes que un hilo de prioridad inferior en estado *Runnable* pueda pasar a ejecutarse.

Java, por defecto, utiliza un planificador apropiativo. Si en un momento dado un hilo que tiene una prioridad mayor a cualquier otro hilo que se está ejecutando, pasa al estado *Runnable*, entonces el sistema elige a este nuevo hilo para su ejecución. 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, en caso de que el sistema operativo subyacente lo permita.

La prioridad de los hilos se puede establecer utilizando el método `setPriority()` de la clase `Thread`. Las prioridades de un hilo varían en un rango de enteros comprendido entre `MIN_PRIORITY` y `MAX_PRIORITY` (definidas en la misma clase y habitualmente 1 y 10, donde 1 significa mínima y 10 significa máxima prioridad). Cuando se crea un hilo, este hereda la prioridad del proceso que lo creó, pero puede modificarse dicha prioridad en cualquier momento. Los procesos en Java no pueden cambiar de prioridad, ya que Java le deja esa planificación al sistema operativo, pero en cambio sí permite que cambie la prioridad de los *threads* que se ejecutan.



EJEMPLO 2.5

Utilización de prioridades para gestionar hilos:

```
class CounterThread extends Thread {  
  
    String name;  
  
    public CounterThread(String name) {  
        super();  
        this.name = name;  
    }  
  
    public void run() {  
        int count = 0;  
        while (true) {  
            try {  
                sleep(10);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            if (count == 1000)  
                count = 0;  
            System.out.println(name + ":" + count++);  
        }  
    }  
}  
  
public class Prioridad{  
    public static void main(String[] args) {  
        CounterThread thread1 = new CounterThread("thread1");  
        thread1.setPriority(10);  
        CounterThread thread2 = new CounterThread("thread2");  
        thread2.setPriority(1);  
        thread2.start();  
        thread1.start();  
    }  
}
```

Eso sí, los sistemas operativos no están obligados a tener en cuenta la prioridad de hilo ya que trabajan a nivel de procesos en sus algoritmos de planificación.

chrt -f 6 java Prioridad Cambiar la política de planificación a SCHED-FIFO con prioridad de S.O. de 6



- Comprobad el funcionamiento del ejemplo anterior en vuestro propio ordenador cambiando la prioridad de los hilos. ¿El cambio de prioridades está afectando significativamente al resultado? ¿A qué se debe ese comportamiento?

2.5 SINCRONIZACIÓN DE HILOS

Los *threads* se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria. Como todos los *threads* pertenecen al mismo proceso, pueden acceder a toda la memoria asignada a dicho proceso y utilizar las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente. Sin embargo, cuando varios hilos manipulan concurrentemente objetos conjuntos, puede llevar a resultados erróneos o a la paralización de la ejecución. La solución es la sincronización.

2.5.1 PROBLEMAS DE SINCRONIZACIÓN

2.5.1.1 Condición de carrera

Se dice que existe una **condición de carrera** si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria. Veamos un ejemplo:



EJEMPLO 2.6

Dos hilos, *sumador* y *restador*, se ejecutan al mismo tiempo sobre la misma variable:

- *Sumador*: cuenta++;
- *Restador*: cuenta--;

En código maquina eso se representa por:

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

Supongamos que *cuenta* vale 10, y los dos hilos están ejecutándose. Puede suceder que ambos lleguen a la instrucción que modifica *cuenta* a la vez y se ejecute lo siguiente:

- | | |
|-----------------------|--|
| • 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} |

Se ha llegado al valor de *cuenta* = 9, el cual es incorrecto. En función del orden en que se ejecuten cada una de las sentencias del código máquina el resultado podría ser tanto 9 como 10 u 11. Si el resultado depende del orden en la ejecución en concreto realizada, existirá una condición de carrera. Debido a que el resultado es impredecible y podría devolver el resultado esperado (en este caso 10) las condiciones de carrera son complicadas de detectar y de solucionar.

2.5.1.2 Inconsistencia de memoria

Una **inconsistencia de memoria** se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato. Las causas de los errores de coherencia son complejas y van desde la no liberación de datos obsoletos hasta el desbordamiento del *buffer* (*buffer overflow*, en inglés), entre otros muchos.

ACTIVIDADES 2.4



- El problema conocido como *buffer overflow* es una vulnerabilidad muy conocida por los *hackers* en Internet. Se aprovechan del fallo de coherencia de memoria para tomar el control de un ordenador.

Busca información acerca del problema y comprende en qué se basa.

Hay que tener especial cuidado al programar para evitar posibles inconsistencias en memoria. Veamos un ejemplo:



EJEMPLO 2.7

Dos hilos, *sumador* e *impresor*, realizan el siguiente código partiendo de cuenta=0

- *Sumador*: cuenta++;
- *Impresor*: System.out.println(cuenta).

Si las dos sentencias se ejecutaran por el mismo hilo, se podría asumir que el valor impreso sería 1. Pero si se ejecutan en hilos separados, el valor impreso podría ser 0, ya que no hay garantía para el *impresor* de que el *sumador* haya realizado ya su operación a menos que el programador haya establecido una relación de ocurrencia entre estas dos sentencias.

2.5.1.3 Inanición

Se conoce como *inanición* al fenómeno que tiene lugar cuando a un proceso o hilo se le deniega continuamente el acceso a un recurso compartido. Este problema se produce cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto de procesos o hilos siempre toman el control antes que él por diferentes motivos. Por ejemplo, esto ocurriría si un proceso tiene baja prioridad y existen procesos más prioritarios que él en el sistema. La inanición puede ocurrir tanto por prioridad como por la propia estructuración del código. Es un problema muy complicado de encontrar y diagnosticar ya que no tiene por qué ocurrir en todas las ejecuciones que se realicen, por lo que hay que tener especial cuidado para evitarlo.

2.5.1.4 Interbloqueo

A pesar de las ventajas que proporcionan la multiprogramación y la multitarea, hay que tener especial cuidado para evitar un problema muy grave conocido como “interbloqueo”. Un *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. Dicho error no tiene por qué aparecer en todas las ejecuciones realizadas, sino que puede ocurrir únicamente en casos muy concretos que dependen del orden específico en que se ejecuten los hilos.



EJEMPLO 2.8

Un ejemplo de un interbloqueo sería:

H0	H1
bloquear (S);	bloquear (Q);
bloquear (Q);	bloquear (S);
...	...
desbloquear (S);	desbloquear (Q);
desbloquear (Q);	desbloquear (S);

2.5.1.5 Bloqueo activo

Un *bloqueo 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. Frente al interbloqueo, donde los procesos se encuentran bloqueados, en un bloqueo activo no lo están, sino que es una forma de inanición debido a que un proceso no deja avanzar al otro.



EJEMPLO 2.9

En un ejemplo del mundo real, un bloqueo activo ocurre, por ejemplo, cuando dos personas se encuentran en un pasillo avanzando en sentidos opuestos y cada una trata de ser amable moviéndose a un lado para dejar a la otra persona pasar. Si se mueven ambas de lado a lado, encontrándose siempre en el mismo lado, se están moviendo, pero ninguna podrá avanzar.

2.5.2 MECANISMOS DE SINCRONIZACIÓN

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**”.

2.5.2.1 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:

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.

El resto del código que no cumpla las dependencias de Bernstein debería ejecutarse de forma síncrona. La utilización de operaciones atómicas y secciones críticas permite separar la ejecución de los segmentos de código que pueden ejecutarse de forma asíncrona de aquellos que deben ejecutarse de forma ordenada.

2.5.2.2 Operación atómica

Una *operación atómica* es una operación que sucede toda al mismo tiempo. Es decir, siempre se ejecutará de forma continuada sin ser interrumpida, por lo que ningún otro proceso o hilo puede leer o modificar datos relacionados mientras se esté realizando la operación. Es como si se realizará en un único paso.



EJEMPLO 2.10

Las operaciones de sacar dinero a través de un cajero automático suelen ser atómicas. Una vez realizada la operación, siempre que se obtenga el dinero quedará reflejado en la cuenta correspondiente. En caso de fallo, no se obtendrá el dinero y no se modificará la cuenta corriente. Una operación (sacar dinero) no puede realizarse sin que se realice en el mismo paso la otra (apuntar en la cuenta).

En un sistema con un único procesador, si una operación se ejecuta en una sola instrucción de la CPU, será atómica. Si una operación requiere de múltiples instrucciones de la CPU, entonces puede ser interrumpida, produciéndose el correspondiente cambio de contexto. Esto indica que no hay atomicidad. Las escrituras en memoria u operaciones matemáticas como $a++$ no son atómicas.

En sistemas multiprocesador, con múltiples hilos de ejecución sobre el mismo proceso, asegurar atomicidad es mucho más complicado. Los hilos podrían modificar datos a la vez sobre los que se esté operando. Una forma de asegurar atomicidad es declarando las variables como *volatile*. Dicha declaración indica al sistema que la actualización de la variable no se realiza en un registro, sino directamente en memoria, teniendo que realizarla en un único paso. Sin embargo, aunque esto permite solucionar el problema específico de accesos a variables, existen muchos casos en los cuales se desea aportar atomicidad a parte del código. Esto se puede realizar mediante la creación de una sección crítica.

2.5.2.3 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 comparten datos o recursos.

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

- Si otro proceso quiere ejecutar su sección crítica, se bloqueará hasta que el primer proceso finalice la ejecución de su sección crítica.
- Se establece una relación antes-después en el orden de ejecución de la sección crítica. Esto garantiza que los cambios en los datos son visibles a todos los procesos.

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:

- **Exclusión mutua:** si un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su sección crítica. Es la característica principal.
- **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. Esta decisión no puede posponerse indefinidamente, esperando a un proceso más prioritario, por ejemplo.
- **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. En caso contrario se produciría inanición.

En definitiva, el código correspondiente a la sección crítica debe terminar en un tiempo determinado y deben existir mecanismos internos del sistema que eviten la inanición, para que el resto de procesos solo tengan que esperar un período determinado de tiempo para entrar a ejecutar sus correspondientes secciones críticas.

Para su implementación se necesita un mecanismo de sincronización tanto que actúe tanto antes de entrar en la sección crítica como después de salir de ejecutarla. Esto asegura la utilización en exclusiva de los datos o recursos compartidos. La manera de implementar las secciones críticas puede variar de un sistema operativo a otro. Java oculta estas diferencias, proporcionando formas sencillas de implementar las secciones críticas: los semáforos y monitores.

2.5.2.4 Semáforos

Los semáforos se pueden utilizar para controlar el acceso a un determinado recurso formado por un número finito de instancias. Un semáforo se representa como una variable entera donde su valor representa el número de instancias libres o disponibles en el recurso compartido y una cola donde se almacenan los procesos o hilos bloqueados esperando para usar el recurso. En la fase de **inicialización**, se proporciona un valor inicial al semáforo igual al número de recursos inicialmente disponibles. Posteriormente, se puede acceder y modificar el valor del semáforo mediante dos operaciones atómicas:

- *wait* (espera). Un proceso que ejecuta esta operación disminuye el número de instancias disponibles en uno, ya que se supone que la va a utilizar. Si el valor es menor que 0, significa que no hay instancias disponibles. En ese sentido, el proceso queda en estado Bloqueado hasta que el semáforo se libere cuando haya instancias. El valor negativo del semáforo especifica cuántos procesos están bloqueados esperando por el recurso.

```

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

```

- *signal* (señal). Un proceso, cuando termina de usar la instancia del recurso compartido correspondiente, avisa de su liberación mediante la operación *signal*. Para ello aumenta el valor de instancias disponibles en el semáforo. Si el valor es negativo o menor que 0, significará que hay procesos en estado Bloqueado por lo que despertará a uno de ellos obteniéndolo de la cola. Si existen varios procesos esperando, solamente uno de ellos pasará a estado *Runnable*. Esto ocurre cuando el número de recursos es limitado, por ejemplo, una plaza de *parking*. Si un coche abandona el *parking*, solamente otro puede ocupar la plaza dejada sin ser necesario avisar a todos los coches. El hilo que se despierta cuando se hace un *signal* es aleatorio y depende de la implementación de los semáforos y del sistema operativo subyacente.

```

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

```

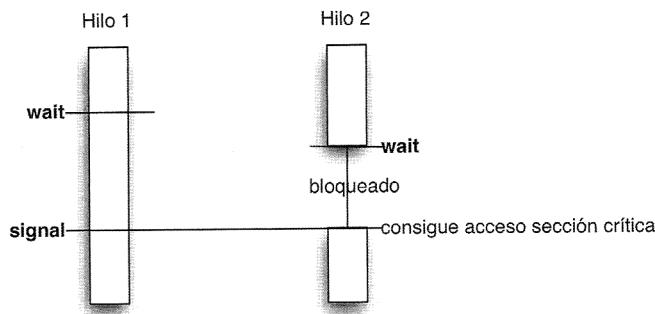


Figura 2.3. Utilización de *wait* y *signal* para acceder a la sección crítica

Las operaciones *wait* y *signal* deben ser atómicas para evitar los problemas anteriormente vistos. Para ello, en ordenadores con un único procesador se utiliza internamente la inhibición de interrupciones. En sistemas multiprocesador, donde las instrucciones de cada núcleo se pueden entrelazar de cualquier forma, la solución pasa por utilizar instrucciones hardware especiales, como *TestAndSet* o *Swap*, o soluciones software, como el algoritmo de Peterson.

Existen varios mecanismos para proporcionar una sección crítica dentro de los diferentes sistemas operativos. Busca información sobre los métodos clásicos (algoritmo de Peterson [1981], *TestAndSet* y *Swap*) para comprender su funcionamiento.

Como podrás ver, estas propuestas son complicadas de usar por parte del programador, además de ser dependientes de la arquitectura del ordenador subyacente. En su lugar se utilizan herramientas que permiten abstraerse de esos problemas, como semáforos, mutex, monitores, etc. en función del sistema operativo y del lenguaje de programación.

Un semáforo binario, también denominado *mutex* (*MUTual EXclusion*, “exclusión mutua” en español) es un indicador de condición que registra si un único recurso está disponible o no. Un *mutex* solo puede tomar los valores 0 y 1. Si el semáforo vale 1, entonces el recurso está disponible y se accede a la zona de compartición del recurso mientras que si el semáforo es 0, el hilo debe esperar.

Los *mutex* son un mecanismo liviano de sincronización idóneo para hilos, ya que posibilitan tanto la exclusión mutua en los accesos a los recursos como la ordenación en el acceso a los mismos pudiendo establecer relaciones antes-después. Para ello, un *mutex* representa un cerrojo sobre una parte de código. Este cerrojo se puede ver como si tuviéramos una cerradura cerrada con una llave puesta para acceder a esa sección de código. Cuando un hilo accede a esa sección, adquiere el semáforo binario, abre la cerradura, la cierra por dentro y se lleva la llave. Hasta que el hilo no finaliza la ejecución de la sección de código, no vuelve a dejar la llave en su sitio, imposibilitando que ningún otro hilo pueda acceder tanto a esa sección como a cualquier otra que requiera esa llave específica (*mutex*). Si la sección de código es aquella que utiliza los recursos compartidos, los semáforos binarios permiten entonces resolver de forma sencilla el problema de la sección crítica. Esta solución fue propuesta por Dijkstra en 1968.



¿SABÍAS QUE...?

Dijkstra es uno de los investigadores más importantes dentro del mundo de la informática, destacando principalmente por sus aportaciones en el campo de la computación distribuida. Como solución a problemas de coordinación de información, propuso la utilización de semáforos. Además propuso soluciones a múltiples problemas relacionados, como el problema del camino más corto (conocido como “algoritmo de Dijkstra”), la cena de filósofos y el algoritmo del banquero.

En Java, la utilización de semáforos se realiza mediante el paquete *java.util.concurrent* y su clase *Semaphore* correspondiente.



ACTIVIDADES 2.6

- En este ejemplo se utiliza un único semáforo con valor 1 para crear una sección crítica.

```
import java.util.concurrent.Semaphore;

class Acumula {
    public static int acumulador = 0;
}

class Sumador extends Thread {

    private int cuenta;
    private Semaphore sem;

    Sumador(int hasta, int id, Semaphore sem) {
        this.cuenta = hasta;
        this.sem = sem;
    }

    public void sumar () {
        Acumula.acumulador++;
    }

    public void run() {
        for (int i=0; i< cuenta; i++){
            try {
                sem.acquire();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            sumar();
            sem.release();
        }
    }
}

public class SeccionCriticaSemaforos {
    private static Sumador sumadores[];
    private static Semaphore semaphore = new Semaphore(1);

    public static void main (String[] args) {
        int n_sum = Integer.parseInt (args[0]);
        sumadores = new Sumador[n_sum];
        for (int i= 0; i < n_sum; i++) {
            sumadores[i] = new Sumador(100000000,i, semaphore);
        }
    }
}
```

```

        sumadores[i].start();
    }

    for (int i= 0; i < n_sum; i++) {
        try {
            sumadores[i].join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    System.out.println ("Acumulador: " + Acumula.acumulador);
}
}

```

Sin la utilización del semáforo, el resultado acumulador depende de la ejecución en concreto que realicen los hijos, pudiéndose obtener resultados erróneos en algunas ejecuciones. Comprueba que esto sucede comentando el semáforo y analiza a qué se debe.

2.5.2.4.1 Clase Semaphore

A la hora de utilizar semáforos en Java se utiliza la clase *Semaphore*. Se presenta, por simplicidad, una tabla con los métodos recogidos más importantes:

Método	Tipo de retorno	Descripción
<i>Semaphore(int valor)</i>	<i>void</i>	Inicialización del semáforo. Indica el valor inicial del semáforo antes de comenzar su ejecución
<i>acquire()</i>	<i>void</i>	Implementa la operación <i>wait</i>
<i>release()</i>	<i>void</i>	Implementa la operación <i>signal</i> . Desbloquea un hilo que esté esperando en el método <i>wait()</i>

2.5.2.5 Monitores

Un *monitor* es un 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.

Funciona de forma similar a los semáforos binarios, pero proporciona una mayor simplicidad ya que lo único que tiene que hacer el programador es ejecutar una entrada 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 utilizar la palabra clave *synchronized*: los métodos y las sentencias sincronizadas.

2.5.2.5.1 Métodos sincronizados

Los métodos sincronizados son un 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, cumpliendo así con los requisitos de las secciones críticas.

Cuando un hilo invoca un método sincronizado, adquiere automáticamente el monitor que el sistema crea específicamente para todo el objeto que contiene ese método. Solo lo libera cuando el método finaliza o si lanza una excepción no capturada. Ningún otro hilo podrá ejecutar ningún método sincronizado del mismo objeto mientras el monitor de ese objeto no sea liberado, es decir, el cerrojo está cerrado. Esto implica que se bloqueen los métodos que afectan a los recursos compartidos del objeto (indicados con *synchronized*).

ACTIVIDADES 2.7

- 💡 Los métodos *static* están asociados a una clase, en vez de al objeto propiamente dicho. Comprueba qué sucede cuando un método sincronizado está definido como *static*.

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 (y no pueden ser marcados como *synchronized*) ya que solo el hilo que lo llama tiene acceso a ese objeto mientras lo está creando.



EJEMPLO 2.11

Creación de métodos sincronizados:

```
public class Contador {  
    private int c = 0;  
  
    public void Contador(int num) {  
        this.c = num;  
    }  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```



ACTIVIDADES 2.8

- » Crea una clase Java que utilice 5 hilos para contar el número de vocales que hay en un determinado texto. Cada hilo se encargará de contar una vocal diferente, actualizando todos los hilos la misma variable común que representa el número de vocales totales. Para evitar condiciones de carrera se deben utilizar métodos sincronizados.

Así, si todos los métodos de lecturas y modificación sobre un objeto al cual accedan varios hilos están sincronizados, se evitan condiciones de carrera e inconsistencias de memoria. Sin embargo, hay que tener en cuenta que la sincronización en este caso no permite la ejecución de otro método sincronizado del mismo objeto al mismo tiempo mientras un método sincronizado de ese objeto se esté ejecutando.

ACTIVIDADES 2.9

- » Las variables definidas como *final*, que no se pueden modificar después de construir el objeto, se pueden leer de forma segura (es decir, sin provocar condiciones de carrera) a través de métodos no sincronizados.

Prueba mediante un ejemplo que no se necesita crear métodos sincronizados para evitar condiciones de carrera al acceder a las mismas.

2.5.2.5.2 Sentencias sincronizadas

Los métodos sincronizados utilizan un monitor que afecta a todo el objeto correspondiente, bloqueando todos los métodos sincronizados del mismo. Esto provoca, por ejemplo, que la ejecución de métodos de solo lectura (sin modificación de datos compartidos) no pueda paralelizarse si existe algún método sincronizado que sí modifique los datos y se quiera mantener el orden entre modificaciones y lecturas.

La utilización de *synchronized* en una sentencia o región específica de código es mucho más versátil y permite una sincronización de grano fino. 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. De esta forma, se pueden crear nuevos objetos que se compartirán entre los hilos. Al bloquearse únicamente los hilos al acceder a esos nuevos objetos, lo que se consigue es bloquear a los hilos únicamente en las secciones de código especificadas por el programador (secciones críticas).



EJEMPLO 2.12

B.S. 250 DÍAS INTENSIVOS

Utilización de sentencias sincronizadas.

```
class GlobalVar {
    public static int c1 = 0;
    public static int c2 = 0;
}

class TwoMutex extends Thread{
    //static (obj)
    private Object mutex1 = new Object();
    private Object mutex2 = new Object();

    public void inc1() {
        synchronized(mutex1) {
            GlobalVar.c1++;
        }
    }

    public void inc2() {
        synchronized(mutex2) {
            GlobalVar.c2++;
        }
    }

    public void run()
    {
        inc1();
        inc2();
    }
}

public class MutualExclusion {

    public static void main(String[] args) throws InterruptedException {
        int N = Integer.parseInt(args[0]);
        TwoMutex hilos[];
        System.out.println ("Creando " + N + " hilos");

        hilos= new TwoMutex[N];
        for (int i= 0; i < N; i++)
        {
            hilos[i] = new TwoMutex();
            hilos[i].start();
        }
        for (int i= 0; i < N; i++) {
            hilos[i].join();
        }
        System.out.println ("C1 = " + GlobalVar.c1);
        System.out.println ("C2 = " + GlobalVar.c2);
    }
}
```

Recordemos que un hilo no puede acceder a una sección protegida por un monitor que ha conseguido otro hilo para ejecutar. Sin embargo, un hilo puede acceder a una sección de código de un monitor que ya posee. Permitir que un hilo pueda adquirir un monitor que ya tiene es lo que se denomina **sincronización reentrante**. Esto describe una situación en la cual un 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.



¿SABÍAS QUE...?

Se puede utilizar también JNI (*Java Native Interface*) para programar aplicaciones multihilo, además de la comunicación multiproceso vista con anterioridad. En este caso, hay que tener especial cuidado en su realización ya que los métodos nativos se deben programar como si fueran métodos sincronizados. En este sentido, los métodos nativos, no deben modificar variables globales sensibles de forma no protegida y los accesos a las variables deben ser coordinados mediante el método correspondiente que permita el lenguaje de programación nativo para resolver el problema de la sección crítica.

2.5.2.6 Condiciones

En concreto, a veces el hilo que se está ejecutando dentro de una sección crítica no puede continuar, porque no se cumple cierta *condición* que solo podría cambiar otro hilo desde dentro de su correspondiente sección crítica. En este caso es preciso 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. Este proceso debe ser atómico y cuando el hilo retoma la ejecución lo hace en el mismo punto donde lo dejó (dentro de la sección crítica). En definitiva, una condición es una variable que se utiliza como mecanismo de sincronización en un monitor.

Para implementar condiciones se pueden utilizar operaciones conocidas. Llamar a la operación *wait* libera automáticamente el cerrojo sobre la sección crítica que se está ejecutando. Para avisar de la ocurrencia de la condición por la que espera, se puede utilizar *signal*. Sin embargo, esta operación no provoca que los hilos notificados empiecen a ejecutarse en ese preciso instante, sino que el hilo notificado no puede ejecutarse hasta que el hilo que lo notificó no libere el cerrojo de la sección crítica por la cual el hilo notificado está esperando.

Si no se tiene en cuenta que el hilo que esperaba no empieza a ejecutarse inmediatamente cuando recibe la notificación se pueden provocar comportamientos anómalos. Como se ve en la Figura 2.4, el hilo no se pone en ejecución hasta que consigue el cerrojo correspondiente, por el cual pelea en igualdad de condiciones con el resto de hilos. Esto podría provocar que otro hilo ejecute antes y le robe los recursos necesarios (condición) por los que el otro hilo esperó. Por eso mismo, la comprobación de la condición de espera no se debe realizar en una sentencia *if*, sino en un *while*. Cuando el hilo vuelva a ejecutar después de realizar la operación *wait* siempre debe comprobar si la condición por la que esperó y le notificaron se cumple a su vuelta para poder seguir con la ejecución.

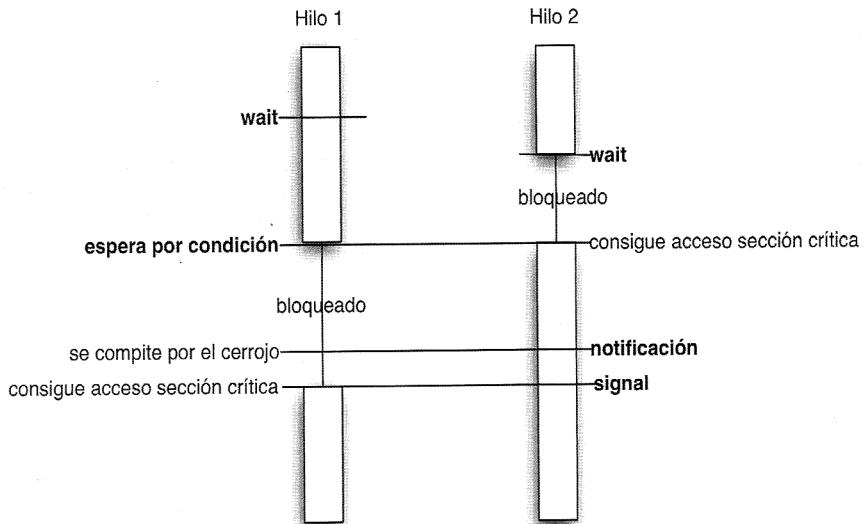


Figura 2.4. Utilización de varios hilos con condiciones

La implementación de condiciones en Java se realiza mediante la utilización de la clase *Object*. Cualquier hilo puede utilizar la operación *signal* en cualquier objeto en el cual otro hilo ejecutó la operación *wait*. Es decir los *wait* y *signal* tienen relación únicamente sobre el mismo objeto. Un hilo que espere con un *wait* sobre un objeto no se despertará por *signal* de otros hilos en otros objetos. Los métodos que corresponden a la implementación de las operaciones *wait* y *signal* para condiciones son *wait()* y *notify()*. Dichos métodos siempre se deben ejecutar sobre un bloque sincronizado, es decir, dentro de un monitor.

```

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
}

```

2.5.2.6.1 Clase Object

A la hora de utilizar condiciones en Java se utiliza la clase *Object*. Se presenta, por simplicidad, una tabla con los métodos recogidos más importantes explicados a lo largo del capítulo sabiendo que deben ejecutarse desde bloques sincronizados:

Método	Tipo de retorno	Descripción
<code>wait()</code>	<code>void</code>	Implementa la operación <i>wait</i> . El hilo espera hasta que otro hilo invoque <i>notify</i> o <i>notifyAll()</i>
<code>notify()</code>	<code>void</code>	Implementa la operación <i>signal</i> . Desbloquea un hilo que esté esperando en el método <i>wait()</i>
<code>notifyAll()</code>	<code>void</code>	Despierta a todos los hilos que estén esperando para que continúen con su ejecución. Todos los hilos esperando por <i>wait</i> reanudan su ejecución. Por ejemplo, si hay un proceso escritor, escribiendo datos, cuando finalice puede avisar a todos los procesos lectores para que continúen su ejecución a la vez (ya que pueden operar todos a la vez)



EJEMPLO 2.13

www.orientacion.com

Utilización de condiciones. La condición de *clase comenzada* no sería necesaria ya que se podría gestionar fácilmente con *wait* y *notifyAll* de forma sencilla:

```
class Bienvenida {  
  
    boolean clase_comenzada;  
  
    public Bienvenida(){  
        this.clase_comenzada = false;  
    }  
  
    // Hasta que el profesor no salude no empieza la clase,  
    // por lo que los alumnos esperan con un wait  
    public synchronized void saludarProfesor(){  
        try {  
            while (clase_comenzada == false){  
                wait();  
            }  
            System.out.println("Buenos días, profesor.");  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```



EJEMPLO 2.13 (cont.)

```
// Cuando el profesor saluda avisa a los alumnos con notifyAll de su llegada
public synchronized void llegadaProfesor(String nombre){
    System.out.println("Buenos días a todos. Soy el profesor " + nombre);
    this.clase_comenzada = true;
    notifyAll();
}

class Alumno extends Thread{
    Bienvenida saludo;

    public Alumno(Bienvenida bienvenida){
        this.saludo = bienvenida;
    }

    public void run(){
        System.out.println("Alumno llegó.");
        try {
            Thread.sleep(1000);
            saludo.saludarProfesor();
        } catch (InterruptedException ex) {
            System.err.println("Thread alumno interrumpido!");
            System.exit(-1);
        }
    }
}

class Profesor extends Thread{
    String nombre;
    Bienvenida saludo;

    public Profesor(String nombre, Bienvenida bienvenida){
        this.nombre = nombre;
        this.saludo = bienvenida;
    }

    public void run(){
        System.out.println(nombre + " llegó.");
        try {
            Thread.sleep(1000);
            saludo.llegadaProfesor(nombre);
        } catch (InterruptedException ex) {
            System.err.println("Thread profesor interrumpido!");
            System.exit(-1);
        }
    }
}
```



```
public class ComienzoClase {  
  
    public static void main(String[] args) {  
  
        // Objeto compartido  
        Bienvenida b = new Bienvenida();  
  
        int n_alumnos = Integer.parseInt(args[0]);  
        for (int i=0; i< n_alumnos; i++) {  
            new Alumno(b).start();  
        }  
        Profesor profesor = new Profesor("Osvaldo Ramirez",b);  
        profesor.start();  
    }  
}
```

2.6 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 mismas fases que para la programación de aplicaciones multiproceso, aunque su comprensión difiere por las diferencias existentes entre hilos y procesos. Los pasos serían los siguientes:

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. Hay que tener en cuenta que la existencia de secciones críticas bloquea a los procesos para su sincronización, provocando una pérdida del rendimiento que se puede conseguir. Es conveniente minimizar la dependencia de sincronización existente entre los hilos, para aumentar la ejecución paralela o ejecución asíncrona.

Implementación. Se utiliza la clase *Thread* o la interfaz *Runnable* como punto de partida.



- Este libro ha contado los conceptos de bajo nivel para entender en profundidad los hilos y cómo operan. A partir de este funcionamiento básico, se puede implementar cualquier solución a problemas multihilo. Para facilitar la programación de aplicaciones complejas, a partir de la versión 5 de Java se creó un nuevo paquete de datos `java.util.concurrent` donde aparecen clases de más alto nivel, que se abstraen del bajo nivel mostrado por las clases vistas en el capítulo.

Lee el siguiente tutorial, donde se muestran las nuevas clases <http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html> para comprender los nuevos conceptos probando su funcionalidad. La clase `Semaphore` pertenece a este paquete.

2.7 CASO PRÁCTICO

En este caso práctico se va a desarrollar una solución multitarea al problema clásico de la cena de filósofos.

En una mesa redonda hay N filósofos sentados. En total tiene N palillos para comer arroz, estando cada palillo compartido por dos filósofos, uno a la izquierda y otro a la derecha. Como buenos filósofos, se dedican a pensar, aunque de vez en cuando les entra hambre y quieren comer. Para poder comer, un filósofo necesita utilizar los dos palillos que hay a sus lados.

Para implementar este problema se debe crear un programa principal que cree N hilos ejecutando el mismo código. Cada hilo representa un filósofo. Una vez creado, se realiza un bucle infinito de espera. Cada una de los hilos tendrá que realizar los siguientes pasos:

1. Imprimir un mensaje por pantalla “Filósofo *i* pensando”, siendo *i* el identificador del filósofo.
2. Pensar durante un cierto tiempo aleatorio.
3. Imprimir un mensaje por pantalla “Filósofo *i* quiere comer”.
4. Intentar coger los palillos que necesita para comer. El filósofo 0 necesitará los palillos 0 y 1, el filósofo 1, los palillos 1 y 2, y así sucesivamente.
5. Cuando tenga el control de los palillos, imprimirá un mensaje en pantalla “Filósofo *i* comiendo”.
6. El filósofo estará comiendo durante un tiempo aleatorio.
7. Una vez que ha finalizado de comer, dejará los palillos en su sitio.
8. Volver al paso 1.

Sin embargo, se pueden producir interbloqueos si por ejemplo todos los filósofos quieren comer a la vez. Si todos consiguen coger el palillo de su izquierda ninguno podrá coger el de su derecha. Para ello se plantean varias soluciones:

- Permitir que como máximo haya N-1 filósofos sentados a la mesa.
- Permitir a cada filósofo coger sus palillos solamente si ambos palillos están libres.
- Solución asimétrica: un filósofo impar coge primero el palillo de la izquierda y luego el de la derecha. Un filósofo par los coge en el orden inverso.

Implementar una solución al problema de los filósofos, solución que no presente un problema de interbloqueo. Por sencillez, se recomienda utilizar el método propuesto de solución asimétrica.



RESUMEN DEL CAPÍTULO



El capítulo ha mostrado el funcionamiento de los sistemas operativos en lo relativo a la ejecución de tareas. Se ha definido el concepto de tarea o hilo como una secuencia de código en ejecución que se puede ejecutar en paralelo con otras tareas siempre que sea dentro del contexto del mismo proceso. La utilización de hilos permite aprovechar procesadores que tienen varios núcleos, ya que permite que cada uno de los núcleos ejecute al mismo tiempo una instrucción de uno de los hilos que pertenecen al mismo proceso (ya que en un momento determinado solo puede haber un único proceso en ejecución). Los hilos perteneciente al mismo proceso comparten entre sí tanto código como memoria (variables) y otros recursos que están asignados al proceso.

Los hilos se pueden estar ejecutando siempre que el proceso esté en ejecución. Sin embargo, como el sistema operativo gestiona procesos, no hilos, no se sabe si en un momento determinado el *thread* tiene recursos hardware suficientes (un núcleo disponible) para su ejecución. Eso hace que su estado habitual sea *Runnable* indicando que puede estar ejecutando en función de los recursos disponibles. Además de eso, los hilos pueden bloquearse por motivos de sincronización.

Al igual que los procesos, durante la ejecución de un programa se pueden arrancar nuevos hilos (que tienen que estar previamente programados) en el espacio de código del proceso. Cualquier proceso tiene un hilo principal que representa la ejecución básica del proceso. La operación *create* permite arrancar nuevos hilos y *join* esperar la finalización de los mismos. Esto permite realizar múltiples tareas de forma eficiente al mismo tiempo (no es necesario realizar cambios de contexto, como sí lo sería si estuvieran expresadas mediante procesos, y su ejecución es en paralelo, y no de concurrente) pero hay que tener especial cuidado con su gestión. Al compartir recursos, un hilo podría afectar la ejecución de otro hilo existiendo problemas de comunicación y sincronización. Dichos problemas van desde que el resultado de la ejecución de un programa dependa del orden en concreto en que se realicen los accesos a memoria (condiciones de carrera o inconsistencias de memoria), hasta el bloqueo o inanición de hilos que trabajan conjuntamente en función del código generado.

El acceso compartido a memoria suele generar los problemas anteriores, por lo que acceder en exclusión mutua a las zonas compartidas puede solucionar el problema. Cuando un hilo está ejecutando su sección crítica, ningún otro hilo puede ejecutar su correspondiente sección crítica, ordenando de esta forma la ejecución. La implementación de una sección crítica puede variar de un sistema operativo a otro, utilizando diversos mecanismos como semáforos, *mutex* y monitores. En general, para su implementación, se necesita un mecanismo de sincronización que indique la entrada a la sección crítica (operación *wait*) y su salida (operación *signal*). Sin embargo, a veces el hilo que está ejecutando dentro de una sección crítica no puede continuar porque no se cumple cierta condición que solo podría cambiar otro hilo desde su correspondiente sección crítica. En este caso, es preciso que el hilo que no pueda continuar libere temporalmente la sección crítica y espere hasta la ocurrencia de dicha condición. Mediante el uso de secciones críticas y condiciones se puede gestionar la programación multihilo.



EJERCICIOS PROUESTOS

- 1. Escribe una clase llamada *Orden* que cree dos hilos y fuerce que la escritura del segundo sea siempre anterior a la escritura por pantalla del primero.

Ejemplo de ejecución:

Hola, soy el thread número 2

Hola, soy el thread número 1

- 2. Escribe una clase llamada *Check* que cree dos *threads* que accedan simultáneamente a un *buffer* de 10.000 enteros. Uno de ellos lee en el *buffer* y el otro escribe en el mismo. El *thread* escritor debe escribir el mismo valor en todos los elementos del *buffer* incrementando en uno el valor en cada pasada. El *thread* lector debe ir comprobando que todos los números del *buffer* son iguales, mostrando un mensaje de error en caso contrario o un mensaje de correcto si la condición se cumple. El código a realizar utilizará un monitor para acceder al *buffer* si se indica un parámetro al ejecutar el programa. En caso contrario, los *threads* accederán al *buffer* sin hacer uso del monitor.

- 3. Escribe una clase llamada *Relevos* que simule una carrera de relevos de la siguiente forma:

- Cree 4 *threads*, que se quedarán a la espera de recibir alguna señal para comenzar a correr. Una vez creados los *threads*, se indicará que comience la carrera, con lo que uno de los *threads* deberá empezar a correr.
- Cuando un *thread* termina de correr pone algún mensaje en pantalla y espera un par de segundos, pasando el testigo a otro de los hilos para que comience a correr, y terminando su ejecución (la suya propia).
- Cuando el último *thread* termine de correr, el padre mostrará un mensaje indicando que todos los hijos han terminado.

Ejemplo de ejecución:

Todos los hilos creados.

Doy la salida!

Soy el thread 1, corriendo . . .

Terminé. Paso el testigo al hijo 2

Soy el thread 2, corriendo . . .

Terminé. Paso el testigo al hijo 3

Soy el thread 3, corriendo . . .

Terminé. Paso el testigo al hijo 4

Soy el thread 4, corriendo . . .

Terminé!

Todos los hilos terminaron.

- 4. Escribe una clase llamada *SuperMarket* que implemente el funcionamiento de N cajas de un supermercado. Los M clientes del supermercado estarán un tiempo aleatorio comprando y con posterioridad seleccionarán de forma aleatoria en qué caja posicionarse para situarse en su cola correspondiente. Cuando les toque el turno serán atendidos procediendo al pago correspondiente e ingresando en la variable Resultados del supermercado. Se deben crear tantos *threads* como clientes haya y los parámetros M y N se deben pasar como argumentos al programa. Para simplificar la implementación, el valor de pago de cada cliente puede ser aleatorio en el momento de su pago.

- 5. Escribe una clase llamada *ModernSuperMarket* que implemente el funcionamiento de N cajas de supermercado. Los mismos M clientes del supermercado realizarán el mismo proceso que en el ejercicio anterior, situándose cuando han realizado la compra, en este caso, en una única cola. Cuando cualquier caja esté disponible, el primero de la cola será atendido en la caja correspondiente. Calcula el tiempo medio de espera por cliente y compáralo con el tiempo medio que se obtendría en el ejercicio anterior.

¿Cuál de las dos alternativas es más eficiente? ¿Cuál elegirías si tú tuvieras un supermercado? Razona la respuesta.

6. Escribe una clase llamada *Parking* que reciba el número de plazas del *parking* y el número de coches existentes en el sistema. Se deben crear tantos *threads* como coches haya. El *parking* dispondrá de una única entrada y una única salida. En la entrada de vehículos habrá un dispositivo de control que permita o impida el acceso de los mismos al *parking*, dependiendo del estado actual del mismo (plazas de aparcamiento disponibles). Los tiempos de espera de los vehículos dentro del *parking* son aleatorios. En el momento en el que un vehículo sale del *parking*, notifica al dispositivo de control el número de la plaza que tenía asignada y se libera la plaza que estuviera ocupando, quedando así estas nuevamente disponibles. Un vehículo que ha salido del *parking* esperará un tiempo aleatorio para volver a entrar nuevamente en el mismo. Por tanto, los vehículos estarán entrando y saliendo indefinidamente del *parking*. Es importante que se diseñe el programa de tal forma que se asegure que, antes o después, un vehículo que permanece esperando a la entrada del *parking* entrará en el mismo (no se produzca inanición).

Ejemplo de ejecución:

ENTRADA: Coche 1 aparca en 0.

Plazas libre: 5

Parking: [1] [2] [3] [0] [0] [0]

ENTRADA: Coche 2 aparca en 1.

Plazas libre: 4

Parking: [1] [2] [3] [0] [0] [0]

ENTRADA: Coche 3 aparca en 2.

Plazas libre: 3

Parking: [1] [2] [3] [0] [0] [0]

ENTRADA: Coche 4 aparca en 3.

Plazas libre: 2

Parking: [1] [2] [3] [4] [0] [0]

ENTRADA: Coche 5 aparca en 4.

Plazas libre: 1

Parking: [1] [2] [3] [4] [5] [0]

SALIDA: Coche 2 saliendo.

Plazas libre: 2

7. Escribe una clase llamada *ParkingCamion* que reciba el número de plazas del *parking*, el número de coches y el número de camiones existentes en el sistema. Dicha clase debe realizar lo mismo que la clase *Parking* pero debe permitir a su vez aparcar camiones. Mientras un automóvil ocupa una plaza de aparcamiento dentro del *parking*, un camión ocupa dos plazas contiguas de aparcamiento. Hay que tener especial cuidado con la inanición de camiones, que puede producirse si están saliendo coches indefinidamente y asignando la nueva plaza a los coches que esperan en vez de esperar a que haya un hueco para el camión (un camión solo podrá acceder al *parking* si hay, al menos, dos plazas contiguas de aparcamiento libre).

Ejemplo de ejecución:

ENTRADA: Coche 1 aparca en 0.

Plazas libre: 5

Parking: [1] [2] [3] [0] [0] [0]

ENTRADA: Camión 101 aparca en 2.

Plazas libre: 3

Parking: [1] [101] [101] [0] [0] [0]

ENTRADA: Coche 2 aparca en 4.

Plazas libre: 2

Parking: [1] [101] [101] [2] [0] [0]

ENTRADA: Coche 3 aparca en 5.

Plazas libre: 1

Parking: [1] [101] [101] [2] [3] [0]

SALIDA: Coche 1 saliendo.

Plazas libre: 2

Parking: [0] [101] [101] [2] [3] [0]



TEST DE CONOCIMIENTOS

1 Cuando varios hilos acceden a datos compartidos, y el resultado de la ejecución depende del orden concreto en que se accede a los datos compartidos se dice que:

- a) Hay una sección crítica.
- b) Hay una condición de carrera.
- c) Eso no puede suceder, los hilos no pueden compartir datos.
- d) Ninguna de las anteriores.

2 Cualquier solución al problema de la sección crítica debe cumplir las condiciones de:

- a) Exclusión mutua, progreso y espera limitada.
- b) Exclusión mutua, espera limitada y retención.
- c) Envejecimiento e inanición.
- d) Exclusión mutua, aislamiento y espera limitada.

3 Indica cuál de las siguientes afirmaciones sobre los monitores es FALSA:

- a) Permiten resolver el problema de la sección crítica.
- b) Pueden ser binarios o contadores.
- c) Se pueden utilizar para garantizar el orden de ejecución entre procesos.
- d) Son un mecanismo de sincronización.

4 Dado el siguiente fragmento de código sobre un objeto en particular para realizar una sección crítica:

```
notify();  
SECCIÓN CRÍTICA  
wait();
```

¿Cuál de las siguientes afirmaciones es cierta?

- a) El código mostrado no asegura retención.
- b) El código mostrado no asegura exclusión mutua.

- c) Es una solución válida para el problema de la sección crítica cuando el código se ejecuta en un único procesador.
- d) El código mostrado permite resolver el problema de la sección crítica.

5 Indica cuál de las siguientes afirmaciones sobre los semáforos es FALSA:

- a) Permiten resolver el problema de la sección crítica.
- b) Pueden ser binarios o contadores.
- c) No se pueden utilizar para garantizar el orden de ejecución entre procesos.
- d) Son un mecanismo de sincronización.

6 Dado el siguiente fragmento de código, y suponiendo que el objeto *Semaphore S* tiene como valor inicial 2:

```
S.release(S);  
Funcion();  
S.acquire(S);
```

¿Cuántos procesos pueden ejecutar al mismo tiempo la tarea Función?

- a) 0
- b) 2
- c) 3
- d) Ninguna de las anteriores. Depende de la ejecución.

7 Dado el siguiente fragmento de código ejecutado por un hilo:

```
synchronyzed(Object) {  
...  
if (<<se cumple condicion>>)  
    wait();  
FUNCIÓN}
```

¿Cuál de las siguientes afirmaciones es cierta?

- a) El *thread* solo ejecutará FUNCIÓN cuando no se cumpla la condición.
- b) El *thread* podría ejecutar FUNCIÓN aunque esta se cumpla, debido a la espera del hilo por conseguir el monitor.
- c) El *thread* podría ejecutar FUNCIÓN aunque esta se cumpla, debido que es un problema inherente al uso paralelo de *threads*.
- d) El *thread* nunca conseguirá ejecutar FUNCIÓN.

8

Una condición de carrera:

- a) Sigue por permitir que varios procesos manipulen variables compartidas de forma concurrente.
- b) Existe cuando varios procesos acceden a los mismos datos en memoria y el resultado de la ejecución depende del orden concreto en que se realicen los accesos.
- c) No necesita de mecanismos software para ser impedida, ya que siempre que se ejecuta el código de procesos concurrentes con los mismos argumentos, estos se ejecutarán de la misma forma.
- d) a y b son correctas.

9 Dado el siguiente fragmento de código suponiendo que el objeto *Semaphore sem* tiene como valor inicial 2:

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

¿Qué sucederá?

- a) El proceso que ejecuta la operación se bloquea hasta que otro ejecute una operación *sem.release()*;
- b) El proceso continuará adelante sin bloquearse, y si previamente existían procesos bloqueados a causa del semáforo, se desbloqueará uno de ellos.
- c) Tras hacer la operación, el proceso continuará adelante sin bloquearse.
- d) Un semáforo jamás podrá tener el valor 2, si su valor inicial era 0 (cero) y se ha operado correctamente con él.

10

Si se usa un bloque sincronizado para lograr la sincronización de procesos:

- a) Siempre se deben incluir variables de condición, pues el bloque únicamente proporciona exclusión mutua.
- b) Las operaciones *wait* y *notify* se utilizan dentro de un mismo objeto.
- c) Las operaciones *wait* y *notify* se utilizan en objetos separados.
- d) La variable de condición se especifica siempre con una condición *if*.