Contenido

[1. INTRODUCCIÓN A LA PROGRAMACIÓN MULTIHILO 2](#_Toc85025356)

[1.1. Introducción 2](#_Toc85025357)

[1.2. Estados de un hilo 4](#_Toc85025358)

[1.3. Recursos compartidos y no compartidos de los hilos 4](#_Toc85025359)

[2. PROCESOS E HILOS 5](#_Toc85025360)

[3. CREAR HILOS EN JAVA 6](#_Toc85025361)

[3.1. La clase Thread 6](#_Toc85025362)

[3.2. La interface Runnable 9](#_Toc85025363)

[4. PARAR EN JAVA LA EJECUCIÓN DE LOS HILOS 10](#_Toc85025364)

[4.1. El método sleep de la clase Thread 10](#_Toc85025365)

[4.2. El método yield de la clase Thread 12](#_Toc85025366)

[5. CICLO DE VIDA DE UN HILO 13](#_Toc85025367)

[6. EJECUCIÓN CONCURRENTE Y PARALELA DE HILOS 15](#_Toc85025368)

[7. AFINIDAD DEL PROCESADOR 16](#_Toc85025369)

[7.1. Cambio de afinidad en sistemas operativos Windows 16](#_Toc85025370)

[7.2. Cambio de afinidad en sistemas operativos Linux 17](#_Toc85025371)

[7.3. Cambio de afinidad en sistemas operativos MacOS 17](#_Toc85025372)

[8. PRIORIDADES DE LOS HILOS 18](#_Toc85025373)

[9. HILOS Y LA PORTABILIDAD DE JAVA: DEBILIDADES 19](#_Toc85025374)

[9.1. Time slicing 19](#_Toc85025375)

[9.2. Hilos egoístas 19](#_Toc85025376)

[9.3. Acceso a datos compartidos 20](#_Toc85025377)

[10. BLOQUEO DE OBJETOS COMPARTIDOS 22](#_Toc85025378)

[10.1. Métodos declarados como synchronized 22](#_Toc85025379)

[10.2. Bloques de código declarados como synchronized 23](#_Toc85025380)

[10.3. DeadLock 23](#_Toc85025381)

[11. SINCRONIZACIÓN DE HILOS 25](#_Toc85025382)

[11.1. El modelo productor-consumidor 25](#_Toc85025383)

[11.2. Métodos wait, notify y notifyAll de la clase Object 28](#_Toc85025384)

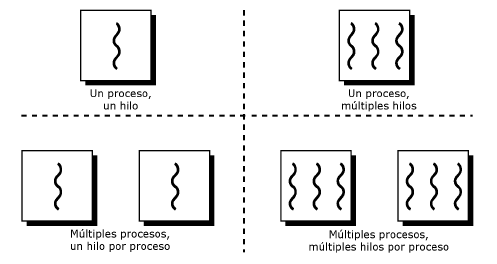
[11.3. Sincronización y bloqueo iterativo (spin lock) 31](#_Toc85025385)

# INTRODUCCIÓN A LA PROGRAMACIÓN MULTIHILO

## Introducción

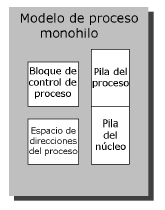
La idea fundamental es bien sencilla. En la programación tradicional hay un solo flujo de control, motivado fundamentalmente porque la máquina internamente suele tener un solo procesador (una sola "mente" que realiza las instrucciones una tras otra). La **programación multihilo** o multithreading permite la ocurrencia simultánea de varios flujos de control. Cada uno de ellos puede programarse independientemente y realizar un trabajo, distinto, idéntico o complementario, a otros flujos paralelos.

Los hilos son un concepto relativamente nuevo de los sistemas operativos. En este contexto, un proceso recibe el nombre de **proceso pesado**, mientras que un hilo recibe el nombre de **proceso ligero**. El término hilo (en inglés, thread) se refiere sintáctica y semánticamente a hilos de ejecución. El término **multihilo** hace referencia a la capacidad de un sistema operativo para mantener varios hilos de ejecución dentro del mismo proceso.



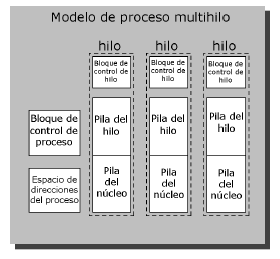
*Figura 1.1 – Procesos monohilo y multihilo*

En un sistema operativo con **procesos monohilo** (un solo hilo de ejecución por proceso), en el que no existe el concepto de hilo, la representación de un proceso incluye su BCP, un espacio de direcciones del proceso, una pila de proceso y una pila del núcleo.



*Figura 1.2 – Modelo de proceso monohilo*

En un sistema operativo con **procesos multihilo** hay un único BCP y espacio de direcciones para el proceso; sin embargo, existen pilas de hilo, pilas del núcleo y bloques de control separados para cada hilo.



*Figura 1.3 – Modelo de proceso multihilo*

Un **hilo** (**proceso ligero**) es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio de pila. Los hilos dentro de una misma aplicación comparten:

* La sección de código.
* La sección de datos.
* Los recursos del sistema operativo (archivos abiertos y señales).

Un **proceso** tradicional o **proceso pesado** es igual a una tarea con un solo hilo. Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, compartiendo un mismo espacio de direcciones y las mismas estructuras de datos del núcleo.

## Estados de un hilo

Los principales estados de un hilo son: **ejecución, preparado y bloqueado.** Además, existen 4 operaciones básicas relacionadas con el cambio de estado de los hilos:

* **Creación**: en general, cuando se crea un nuevo proceso se crea también un hilo para ese proceso. Posteriormente, ese hilo puede crear nuevos hilos dándoles un puntero de instrucción y algunos argumentos. Ese hilo se colocará en la cola de preparados.
* **Bloqueo:** cuando un hilo debe esperar por un suceso, se le bloquea guardando sus registros. Así el procesador pasará a ejecutar otro hilo preparado. Pasa a la pila de bloqueados.
* **Desbloqueo:** cuando se produce el suceso por el que un hilo se bloqueó, pasa a la cola de preparados.
* **Terminación:** cuando un hilo finaliza, se liberan su contexto y pilas.

Un punto importante es la posibilidad de que el bloqueo de un hilo lleve al bloqueo de todo el proceso al que pertenece; es decir, que el bloqueo de un hilo lleve al bloqueo de todos los hilos que lo componen, aun cuando el proceso esté en estado preparado.

## Recursos compartidos y no compartidos de los hilos

Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, compartiendo un mismo espacio de direcciones y las mismas estructuras de datos del núcleo.

**Recursos compartidos** entre los hilos:

* Código (instrucciones).
* Variables globales.
* Ficheros y dispositivos abiertos.

**Recursos no compartidos** entre los hilos:

* Contador del programa (cada hilo puede ejecutar una sección distinta de código).
* Registros de CPU.
* Pila para las variables locales de los procedimientos a los que invoca después de crear un hilo.
* Estado: en ejecución, preparado o bloqueado.

# PROCESOS E HILOS

**Semejanzas.** Los hilos operan en muchos sentidos como los procesos:

* Pueden estar en diferentes estados: preparado, bloqueado y en ejecución.
* Comparten la CPU.
* Sólo hay un hilo activo (en ejecución) en un instante dado.
* Un hilo dentro de un proceso se ejecuta secuencialmente.
* Cada hilo tiene su propia pila y contador de programa.
* Pueden crear sus propios hilos hijos.

**Diferencias.** Los hilos, a diferencia de los procesos, no son independientes entre sí:

* Como todos los hilos pueden acceder a todas las direcciones de la tarea, un hilo puede leer la pila de cualquier otro hilo o escribir sobre ella. Aunque pueda parecer lo contrario la protección no es necesaria ya que el diseño de una tarea con múltiples hilos tiene que ser de un único usuario.

**Ventajas** de los hilos sobre los procesos:

* Se tarda menos tiempo en crear un nuevo hilo en un proceso existente que en crear un nuevo proceso.
* Se tarda menos tiempo en terminar un hilo que un proceso.
* Se tarda menos tiempo en conmutar entre hilos de un mismo proceso que entre procesos.
* Los hilos hacen más rápida la comunicación entre procesos, ya que al compartir memoria y recursos, se pueden comunicar entre sí sin invocar al núcleo del sistema operativo.

**Ejemplos de uso de los hilos.** Hay miles de ejemplos en los que puede ser útil pensar en varios flujos de ejecución (hilos): la posibilidad de editar mientras seguimos cargando o salvando un gran fichero, la posibilidad de visualizar una página mientras se están buscando las siguientes, la visualización de varios procesos que ocurren a la vez de forma independiente, etc. Se pueden clasificar en:

* **Trabajo interactivo y en segundo plano**: en un programa de hoja de cálculo, un hilo podría estar leyendo la entrada del usuario y otro podría estar ejecutando las órdenes y actualizando la información, o mientras escribimos en un procesador de texto el sistema nos va corrigiendo.
* **Procesamiento asíncrono:** se podría implementar, con el fin de protegerse de cortes de energía, un hilo que se encargará de salvaguardar el buffer de un procesador de textos una vez por minuto.
* **Estructuración modular de los programas:** los programas que realizan una variedad de actividades se pueden diseñar e implementar mediante hilos.

# CREAR HILOS EN JAVA

Hay dos formas de crear un hilo en Java:

* Creando una subclase que herede de la **clase Thread** y sobrecargando el método *run()*.
* Implementando la **interface Runnable** y declarando el método *run()*.

## La clase Thread

Algunas de las características de la **clase Thread** son las siguientes:

**Atributos:**

* int MAX\_PRIORITY - Prioridad máxima que puede tener un hilo.
* int MIN\_PRIORITY - Prioridad máxima que puede tener un hilo.
* int NORM\_PRIORITY - La prioridad que se asigna por defecto a un hilo.

**Constructores:**

* public Thread() -> Crea un hilo con nombre "Thread-"+n (n es un entero secuencial).
* public Thread(String name) -> Como el anterior, pero con el nombre de thread name indicado.
* public Thread(Runnable target) -> Crea un hilo asociado al objeto target. El nombre es "Thread-"+n.
* public Thread(Runnable target, String name) -> Como el anterior, pero con nombre específico.

**Métodos de clase:**

* public static native Thread currentThread() -> Devuelve el hilo que está ejecutándose.
* public static native void yield() -> Causa que el hilo en curso haga una pausa temporal y permita a otros hilos ejecutarse.
* public static void sleep(long millis) throws InterruptedException -> Hace que el hilo activo se "duerma" (se detenga temporalmente) durante el tiempo indicado en milisegundos (millis).
* public static boolean interrupted() -> Chequea si el hilo activo ha sido interrumpido (ver isInterrupted).

**Métodos de instancia:**

* public native synchronized void start() -> Inicia la ejecución de un hilo. La máquina virtual de Java llama al método run() correspondiente a este objeto o al Runnable asociado. A partir de este momento los dos hilos, el que llama a start() y el nuevo, se ejecutan.
* public void run() -> Especificaremos en él lo que queremos que haga la ejecución de nuestro hilo.
* public final void stop() -> Detiene la ejecución del hilo. Es el método opuesto a start().
* public void interrupt() -> Interrumpe la ejecución del hilo.
* public boolean isInterrupted() -> Comprueba si el hilo ha sido interrumpido.
* public void destroy() -> Destruye el hilo.
* public final native boolean isAlive() -> Comprueba si el hilo está aún "vivo", es decir, ejecutándose.
* public final void suspend() -> Suspende el hilo y lo deja así hasta que se recupera (resume).
* public final void resume() -> Recupera la ejecución del hilo en el punto en el que estaba (ver suspend).
* public final void setPriority(int newPriority) -> Cambia la prioridad de este hilo. A mayor prioridad, más tiempo de procesador se concede con respecto a otros hilos.
* public final int getPriority() -> Devuelve la prioridad del hilo.
* public final void setName(String name) -> Cambia el nombre del hilo. Los nombres de hilos distintos no tienen por qué ser distintos, aunque obviamente es recomendable.
* public final String getName() -> Devuelve el nombre del hilo.
* public native int countStackFrames() -> Devuelve el número de registros de activación (stack frames) de este hilo. Ojo, que el hilo debe estar suspendido para poder llamar a este método.
* public final synchronized void join(long millis) throws InterruptedException -> Espera a que el hilo acabe (muera). Si se indica millis significa el tiempo máximo (en milisegundos) que se va a esperar. Un valor 0 significa que se espera hasta que acabe (o sea, lo mismo que no ponerlo).
* public final void setDaemon(boolean on) -> Marca el hilo como demonio o no (hilo de usuario). Lo normal es que los hilos no sean demonios. La máquina virtual Java se ejecuta hasta que no quedan hilos de usuario en curso. Por ejemplo, el garbage collector es un hilo de prioridad mínima que se ejecuta como un demonio. El método debe llamarse antes del start().
* public final boolean isDaemon() -> Consultor de lo anterior.

**Ejemplo 1**

En este ejemplo crearemos una clase (Hilos1) que será un hilo e instanciaremos dos objetos de dicha clase para lanzar dos ejecuciones paralelas y ver qué sucede.

//Hilos1.java

**public** **class** Hilos1 {

**public** **static** **void** main(String arg[]) {

Primero p=**new** Primero();

Segundo s=**new** Segundo();

p.start();

s.start();

System.***out***.print("Fin programa");

}

}

//Primero.java

**public** **class** Primero **extends** Thread {

**public** **void** run(){

**for** (**int** i=1;i<100;i++)

System.***out***.println("Primero " + i);

}

}

//Segundo.java

**public** **class** Segundo **extends** Thread {

**public** **void** run(){

**for** (**int** i=1;i<100;i++)

System.***out***.println("Segundo " + i);

}

}

**Resultado:**

Fin programa

Primero 1

Primero 2

Segundo 1

Segundo 2

Segundo 3

Segundo 4

……..

Cada vez que se ejecuta el programa, el orden de escritura de la información en la consola suele ser diferente. El **objeto p** ha ejecutado su método *run()* (eso es lo que hace el método *start()*, que es el iniciador de todo hilo) y ha escrito 100 veces "Primero" en la consola. Pero a la vez también se ha ejecutado el método *run()* del **objeto s**, que trataba de escribir 100 veces la palabra "Segundo" en la consola. Y obviamente, como no hay dos consolas, los dos hilos han usado la misma salida estándar, con lo que han ido entremezclándose las impresiones de las cadenas de caracteres "Primero" y "Segundo".

Para ello hemos hecho que las clases Primero y Segundo hereden de la clase *Thread*. Cualquier subclase de *Thread* permite crear objetos y ejecutar su método *run()* (que debe redefinirse, ya que por omisión no hace nada) en paralelo con otros.

## La interface Runnable

En el ejemplo anterior se ha visto que para crear un hilo, hay que heredar de la clase *Thread* y escribir en el método *run()* lo que se necesite que haga la ejecución del hilo. Pero, ¿qué sucede si se quiere heredar de otra clase diferente de la clase *Thread*? Como en Java no está soportada la herencia múltiple, ¿ya no se podrá crear un hilo de ejecución independiente? Para solucionar este problema está la interface *Runnable*, que requiere la definición de un método que tenga la forma *public void run()*, dentro de la clase en la que se está trabajando. A partir de ahí, cualquier objeto que lo haga, puede usarse para crear un hilo que ejecutará al iniciarse, el método *run()* de ese objeto. El siguiente ejemplo explica el funcionamiento de esta interface:

**Ejemplo 2**

// Hilos2.java

**public** **class** Hilos2 {

**public** **static** **void** main(String arg[]) {

Thread p = **new** Thread(**new** Primero());

Thread s = **new** Thread(**new** Segundo());

p.start();

s.start();

System.***out***.println( "Fin programa ");

}

}

// Primero.java

**public** **class** Primero **implements** Runnable {

**public** **void** run() {

**for** (**int** i=1;i<100;i++)

System.***out***.println( "Primero " + i );

}

}

// Segundo.java

**public** **class** Segundo **implements** Runnable {

**public** **void** run() {

**for** (**int** i=1;i<100;i++)

System.***out***.println( "Segundo " + i );

}

}

**Resultado:**

Análogo al del Ejemplo 1.

# PARAR EN JAVA LA EJECUCIÓN DE LOS HILOS

## El método sleep de la clase Thread

**Función:** detener la ejecución del hilo actual por la cantidad de milisegundos indicada en el parámetro, siempre sujeto a la precisión de los temporizadores y planificadores del sistema.

**Sintaxis**:

public static void **sleep** (long milis) throws InterruptedException;

**Ejemplo 3**

Implementando la clase *Thread*, detener la ejecución del hilo actual durante 1 segundo (1000 milisegundos).

// Hilos4.java

**public** **class** Hilos4 {

**public** **static** **void** main(String arg[]) {

Primero p=**new** Primero();

Segundo s=**new** Segundo();

p.start();

s.start();

System.***out***.print("Fin programa \n");

}

}

// Primero.java

**public** **class** Primero **extends** Thread {

**public** **void** run(){

**for** (**int** i=1;i<10;i++){

System.***out***.println("Primero " + i);

**try** {

*sleep* (1000);

}

**catch** (InterruptedException e) {

}

}

}

}

// Segundo.java

**public** **class** Segundo **extends** Thread {

**public** **void** run(){

**for** (**int** i=1;i<10;i++){

System.***out***.println("Segundo " + i);

**try** {

*sleep* (1000);

}

**catch** (InterruptedException e) {

}

}

}

}

**Ejemplo 4**

Implementando la interface *Runnable*, detener la ejecución del hilo actual invocando al método *Thread.sleep().* En el primer hilo, poner un retardo de 100 milisegundos antes de que se ejecute. En el segundo hilo, poner para que aleatoriamente se ejecute un retraso de entre 0 y 200 milisegundos.

// Hilos5.java

**public** **class** Hilos5 {

**public** **static** **void** main(String arg[]) {

Thread p = **new** Thread(**new** Primero());

Thread s = **new** Thread(**new** Segundo());

p.start();

s.start();

System.***out***.println( "Fin programa ");

}

}

// Primero.java

**public** **class** Primero **implements** Runnable {

**public** **void** run() {

**for** (**int** i=1;i<10;i++){

System.***out***.println( "Primero " + i );

**try**{

Thread.*sleep*(100);

} **catch** (InterruptedException e){}

}

}

}

// Segundo.java

**public** **class** Segundo **implements** Runnable {

**public** **void** run() {

**for** (**int** i=1;i<10;i++){

System.***out***.println( "Segundo " + i );

**try**{

Thread.*sleep*((**long**) ( Math.*random*() \* 200));

}**catch** (InterruptedException e){}

}

}

}

## El método yield de la clase Thread

**Función:** detener la ejecución del hilo actual y permitir que otros hilos se puedan ejecutar.

**Sintaxis:**

Public static void **yield**()

**Ejemplo 5**

Implementando la interface *Runnable*, detener la ejecución del hilo actual invocando al método *Thread.yield().*

// Hilos5bis.java

**public** **class** Hilos5bis {

**public** **static** **void** main(String arg[]) {

Thread p = **new** Thread(**new** Primero());

Thread s = **new** Thread(**new** Segundo());

p.start();

s.start();

System.***out***.println( "Fin programa ");

}

}

// Primero.java

**public** **class** Primero **implements** Runnable {

**public** **void** run() {

**for** (**int** i=1;i<10;i++)

System.***out***.println( "Primero " + i );

Thread.*yield*();

}

}

// Segundo.java

**public** **class** Segundo **implements** Runnable {

**public** **void** run() {

**for** (**int** i=1;i<10;i++)

System.***out***.println( "Segundo " + i );

Thread.*yield*();

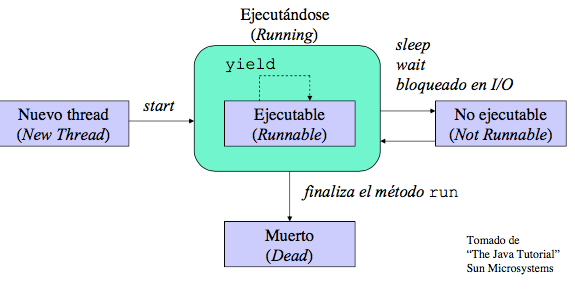
}

}

Posiblemente el resultado no sea el esperado, debido a que el micro de tu equipo puede tener varios núcleos y el sistema operativo ejecuta los hilos en paralelo. Más adelante se verá cómo ver este resultado correctamente mediante la utilización de lo que se conoce como afinidad de procesador.

# CICLO DE VIDA DE UN HILO

En la siguiente figura se pueden ver los 4 posibles estados de los hilos en Java así como las transiciones entre ellos a través de métodos de la clase *Thread*:

**

*Figura 4.1 – Ciclo de vida de un hilo*

Los **estados de los hilos** son los siguientes:

* *Nuevo thread*: es el estado cuando se crea un objeto hilo con el operador new. En este estado el hilo aún no se ejecuta; es decir, el programa no ha comenzado la ejecución del código del método **run** del hilo.
* *Ejecutable*: cuando se invoca al método **start**, el hilo pasa a este estado. El sistema operativo tiene que asignar tiempo de CPU al hilo para que se ejecute; por tanto, el hilo puede estar o no en ejecución.
* *No ejecutable*: cuando el hilo no puede ejecutarse, ya que hay algo que lo impide.
* *Muerto*: un hilo puede morir por varias razones. En primer lugar, por muerte natural (cuando el método **run** finaliza con normalidad) o por muerte no natural (debido a alguna excepción no capturada en el método **run**).

Las posibles **transiciones** entre los estados de los hilos son las siguientes:

* Transición de *Nuevo thread* a *Ejecutable*: cuando el hilo invoca al método **start**.
* Transición de *Ejecuta*ble a *No ejecutable.* Se produce por alguna de las siguientes situaciones:
* El hilo invoca el método **sleep**.
* El hilo invoca el método **wait**.
* El hilo se bloquea en una **operación de I/O** (entrada/salida).
* Transición de *No ejecutable* a *Ejecutable*. Se produce por alguna de las siguientes situaciones:
* Si el hilo había invocado el método **sleep** y el número de milisegundos de pausa ya ha transcurrido.
* Si el hilo había invocado el método **wait** de un objeto y otro hilo le pide que continúe llamando a los métodos **notify** o **notifyAll** del mismo objeto.
* Si el hilo estaba bloqueado en **operación de I/O** y la operación de entrada/salida se ha completado.
* Transición de *Ejecutable* a *Muerto*: cuando termina el método **run**.

El método **isAlive** ayuda a conocer el estado de un hilo. Los valores que devuelve son:

* **True:** si el hilo ha sido lanzado (método start invocado) y no detenido.
* **False:** si el hilo está en estado *Nuevo thread* (no ha sido lanzado) o en estado *Muerto* (método run terminado).

# EJECUCIÓN CONCURRENTE Y PARALELA DE HILOS

**Paralelismo**: se produce en sistemas con múltiples CPUs o CPUs con varios núcleos; cada CPU puede ejecutar un hilo distinto.

Hilo A Hilo B

Hilo A Hilo B

Hilo A Hilo B

**CPU 1** **CPU 2**

**Concurrencia:** cuando el paralelismo no es posible, una CPU es responsable de ejecutar múltiples hilos.

Hilo A

Hilo B

Hilo A

Hilo B

**CPU**

# AFINIDAD DEL PROCESADOR

En aquellos equipos que disponen de CPUs con varios procesadores o de procesadores con varios núcleos, la afinidad del procesador permite asignar un determinado procesador o núcleo a un programa dado.

En Java es posible conocer el número de procesadores que tiene el equipo de la siguiente forma:

**Ejemplo 6**

**public** **class** Procesadores {

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

System.***out***.println("Número de procesadores "+ Runtime.*getRuntime*().availableProcessors());

}

}

## Cambio de afinidad en sistemas operativos Windows

**Windows Modo Gráfico**

* Pulsar simultáneamente las teclas Ctrl+Alt+Supr para poder acceder al Administrador de tareas.
* Dentro del Administrador de tareas, seleccionar la aplicación a la que queremos asignarle una afinidad, pinchar en el botón derecho del ratón e ir a su proceso. Si no se trata de una aplicación podemos ir directamente a la pestaña de procesos.
* En el caso de la máquina virtual de Java, el proceso a seleccionar sería uno llamado *javaw.exe*, el cual se encarga de ejecutar los programas Java. Seleccionar el proceso *javaw.exe* y, con el botón derecho, asignar su afinidad. Seleccionar el procesador que se le quiere asignar al proceso *javaw.exe*.
* En el caso del IDE Eclipse, el proceso a seleccionar sería uno llamado *eclipse.exe*, el cual se encarga de compilar y ejecutar los programas Java. Seleccionar el proceso *eclipse.exe* y, con el botón derecho, asignar su afinidad. Seleccionar el procesador que se le quiere asignar al proceso *eclipse.exe*.

**Windows modo comando**

Por ejemplo, para asignar una determinada CPU a la aplicación *Bloc de notas*, se puede usar el siguiente comando:

c:\windows\system32\cmd.exe /C start /affinity 1 notepad.exe

Se puede ver en el administrador de tareas que el proceso asociado a la aplicación *Bloc de Notas* sólo tiene asignada la ***CPU 0***

Para iniciar un proceso en la ***CPU 0***, habría que usar el siguiente modificador de comando:

/affinity 1

Para iniciar un proceso en la ***CPU 1***, habría que utilizar el siguiente modificador de comando:

/affinity 2

Se puede utilizar un modificador de comando hasta el número de núcleos de CPU del equipo. La afinidad es esencialmente ***CPU core # + 1***, por lo que ***/affinity 5*** usaría el ***CPU 4***.

## Cambio de afinidad en sistemas operativos Linux

La afinidad de CPU no es más que una propiedad del planificador del sistema operativo que enlace un proceso con una determinada CPU. El planificador de Linux respetará la afinidad de CPU establecida y el proceso no se ejecutará en ninguna otra CPU.

El planificador de Linux funciona en base a la llamada *afinidad natural de CPU*:

* El planificador intenta mantener a los procesos en la misma CPU hasta que sea posible por cuestiones de rendimiento.
* Por lo tanto, forzar una afinidad de CPU específica es sólo útil con ciertas aplicaciones.

Para establecer la afinidad para una tarea o proceso concretos, es preciso usar el comando *taskset*, el cual no viene instalado por defecto. Para instalarlo, es necesario instalar el paquete *schedutils*.

Taskset –cp 1,3 45678

## Cambio de afinidad en sistemas operativos MacOS

El sistema operativo OS X soporta una API de afinidad de hilos desde la versión 10.5. Un conjunto de afinidad es una colección de hilos que comparten recursos de memoria y que quieren compartir una cache L2.

Los conjuntos de afinidad se identifican con una etiqueta. A los hilos se les asigna a un determinado conjunto de afinidad mediante dicha etiqueta. Un hilo puede pertenecer como mucho a un conjunto de afinidad; es decir, tiene una etiqueta de afinidad.

Efecto de establecer distintas etiquetas de afinidad. Por ejemplo: una aplicación que quiera ejecutar 2 hilos en cachés L2 diferentes, deberá configurar los hilos con diferentes etiquetas de afinidad.

Ejemplo de utilización. Una aplicación que quiera ubicar un hilo en cada procesador disponible, haría lo siguiente:

* Obtener el número de procesadores del sistema, usando sysctl(3).
* Crear ese número de hilos.
* Asignar a cada hilo una etiqueta de afinidad diferente.
* Iniciar todos los hilos.

# PRIORIDADES DE LOS HILOS

En Java cada hilo tiene una prioridad. Por defecto, un hilo hereda la prioridad del hilo padre que lo crea. La prioridad se puede aumentar o disminuir mediante el método **setPriority()**. El método **getPriority()** devuelve la prioridad del hilo.

La clase Thread declara tres atributos:

* public static final int **MAX\_PRIORITY** 10
* public static final int **MIN\_PRIORITY** 1
* public static final int **NORM\_PRIORITY** 5

La prioridad de los hilos es un valor entero que va desde 1 hasta 10, siendo el valor 1 la mínima prioridad (MIN\_PRIORITY) y el valor 10 la máxima prioridad (MAX\_PRIORITY). El atributo NORM\_PRIORITY de la clase *Thread* asigna por defecto una prioridad de valor 5 a los hilos.

La ejecución de múltiples hilos en una sola CPU requiere la determinación de una secuencia de ejecución (“***scheduling***”).

Java soporta un algoritmo de secuenciación de hilos simple denominado “***fixed priority scheduling***”. Este algoritmo secuencia la ejecución de hilos en base a la “***prioridad relativa***” que les ha sido asignada.

**El “fixed priority scheduling” de Java**

* Entre todos los hilos en estado **“ejecutable”** es escogido el hilo con la prioridad más alta.
* Si hay dos hilos con la misma prioridad, es escogido uno de ellos mediante el algoritmo *Round-Robin (*seleccionar todos los elementos en un grupo de manera equitativa y en un orden racional, normalmente comenzando por el primer elemento de la lista hasta llegar al último y empezando de nuevo desde el primer elemento*).*
* Cuando el hilo en ejecución pasa al estado **“no ejecutable”** o **“muerto”** otro hilo es seleccionado para su ejecución.
* La ejecución de un hilo es interrumpida si otro hilo con prioridad más alta pasa al estado **“ejecutable”**.

# HILOS Y LA PORTABILIDAD DE JAVA: DEBILIDADES

## Time slicing

Los responsables de la ejecución de los hilos son los sistemas operativos. Pero no todos los sistemas operativos manejan los hilos de la misma forma. Por ejemplo: los sistemas operativos NT y Solaris tienen diferentes niveles de prioridades (incluso diferentes respecto de las definidas por Java).

**Time slicing** (subdivisión de tiempo): cuando el sistema operativo asigna una porción de tiempo a la ejecución de cada hilo. En este caso, la ejecución de un hilo es interrumpida no sólo por otro hilo de mayor prioridad que pasa a estado “ejecutable”, sino también cuando su tiempo asignado de ejecución se acaba. No todos los sistemas operativos implementan el concepto de *time slicing*.

## Hilos egoístas

Este concepto aparece en los sistemas operativos que no implementan el concepto de *time slicing,*  cuando un determinado hilo entra en estado *Ejecutable y* continua su ejecución hasta morir. Durante ese período de tiempo ningún otro hilo podrá ser ejecutado.

Por lo tanto:

* No se debe asumir que la ejecución de una aplicación Java se vaya a producir siempre en sistemas que soporten el concepto de *time slicing*.
* En consecuencia, para evitar el egoísmo de los hilos, es necesario incluir adecuadamente invocaciones a los métodos **yield, sleep y wait**, si los hilos no se bloquean en operaciones de entrada/salida.

## Acceso a datos compartidos

Es común que dos o más hilos tengan acceso a objetos comunes.

**Ejemplo 7 – Aplicación con 2 hilos (Primero y Segundo) que actualizan un objeto compartido (mensajes) de la clase Historial**

// Hilos.java

**public** **class** Hilos **extends** Thread {

String mensaje;

Historial historial;

**public** Hilos (String msg, Historial h){

mensaje = msg;

historial = h;

}

**public** **void** run() {

**for** (**int** i=1;i<=20;i++){

historial.agregar( mensaje );

*yield*();

}

}

}

**//**Historial.java

**public** **class** Historial {

String[] mensajes = **new** String[1000];

**int** pos = 0;

**public** **void** agregar(String msg) {

mensajes[pos] = msg;

pos++;

}

}

**//**Ejemplo7.java

**public** **class** Ejemplo7 {

**public** **static** **void** main(String arg[]) {

Historial historial= **new** Historial();

Hilos p = **new** Hilos("Primero", historial);

Hilos s = **new** Hilos("Segundo", historial);

p.start();

s.start();

}

}

Se espera que ocurra lo siguiente:

pos=0 **Hilo Primero Hilo Segundo**

mensaje[pos]=msg; mensaje[0]=”Primero”

pos=pos+1 pos=1

mensaje[pos]=msg; mensaje[1]=”Segundo”

pos=pos+1 pos=2

mensaje[pos]=msg; mensaje[2]=”Primero”;

pos=pos+1 pos=3

Pero podría ocurrir lo siguiente:

pos=0 **Hilo Primero Hilo Segundo**

mensaje[pos]=msg; mensaje[0]=”Primero”

mensaje[pos]=msg; mensaje[0]=”Segundo”

pos=pos+1 pos=1

pos=pos+1 pos=2

mensaje[pos]=msg; mensaje[2]=”Primero”;

pos=pos+1 pos=3

Es decir, en ocasiones se puede machacar la posición si el segundo hilo hace la asignación antes de que el primero incremente la posición.

# BLOQUEO DE OBJETOS COMPARTIDOS

## Métodos declarados como synchronized

La sincronización para el acceso a objetos compartidos se basa en el concepto de “**monitor**”, desarrollado por Charles Antony Richard Hoare. Un monitor es una porción de código protegida por un “**mutex**” (“*mutual exclusion semaphore*”).

Sólo un hilo puede tener el mutex de un objeto en un momento dado. Si un segundo hilo trata de obtener un mutex ya adquirido por otro hilo, se bloquea hasta que el primero libere el mutex. En el momento de liberarse un mutex, todos los hilos en espera de él se “despertarán” (en base a algún criterio) y el mutex se asignará a uno de ellos.

**Analogía de los conceptos de monitor y mutex**

Para entender el funcionamiento del monitor y del mutex, se va a pensar en **un edificio** en el cual algunas oficinas tienen llave y otras tienen libre acceso. El monitor es el conjunto de oficinas cuyo acceso requiere la llave. Los hilos son las personas que quieren acceder a las oficinas.

Para entrar a una oficina con llave, una persona tiene que obtener el manojo con las llaves de la oficina. El manojo con las llaves es el **mutex** del edificio: solo la persona que tiene el manojo puede entrar a las oficinas con llave.

En Java el bloqueo de un objeto ocurre cuando un hilo entra a un método declarado como **synchronized** (de un objeto compartido). Por ejemplo, el método *agregar()* de la clase Historial del **Ejemplo 7** de este documento:

**//**Historial.java

**public** **class** Historial {

String[] mensajes = **new** String[1000];

**int** pos = 0;

**public** **synchronized void** agregar(String msg) {

mensajes[pos] = msg;

pos++;

}

}

En el momento de que un hilo entra a un método declarado como **synchronized** de un objeto compartido, el hilo se encontrará con una de las siguientes situaciones:

* Mutex libre: el hilo tomará el mutex, ejecutará el método y lo liberará al terminar la ejecución del método.
* Mutex tomado por otro hilo: el hilo se bloqueará en espera de que el primero lo libere.

Por lo tanto, sólo un hilo a la vez podrá ejecutar un método **synchronized** sobre un objeto. Si por lo que fuera, se interrumpe la ejecución del hilo que ha tomado el mutex, el paso se dará a otro hilo que no requiera el mutex sobre tal objeto. Una vez liberado el mutex por el hilo que lo tiene tomado, los hilos bloqueados en espera de él se vuelven “ejecutables”, aunque sólo a uno de ellos se le asignará el mutex.

En el caso del Ejemplo 7, mientras un hilo se encuentra ejecutando el método **agregar()** de la clase Historial, ningún otro hilo puede ejecutar dicho método, ya que la ejecución del método **agregar()** requiere el mutex del objeto **historial**.

La invocación al método **sleep** de laclase Thread por parte de un hilo, no libera el mutex de los objetos que eventualmente pudiera tener bloqueados.

Los i**nconvenientes** de declarar métodos como **synchronized** son los siguientes:

* Mayor lentitud en la ejecución de dichos métodos, ya que el hilo que toma el mutex bloquea la posible ejecución de dicho método para otros hilos.
* Peligro de deadlock (abrazo mortal): bloqueo mutuo de dos hilos que esperan adquirir mutex intercambiados.

## Bloques de código declarados como synchronized

Además de declarar como synchronized todo un método de una determinada clase, también es posible hacer declaración para sólo porciones de código dentro de un método. Esto permite implementar exclusiones mutuas sobre bloques de instrucciones.

Su formato es el siguiente:

**synchronized** (objeto) {

// Instrucciones

}

Cuando un hilo entra en un bloque de código declarado como synchronized, el hilo se bloquea en espera de la adquisición del mutex asociado al objeto (o arreglo) declarado en su encabezado. El mutex es liberado a la salida del bloque de código.

## DeadLock

El DeadLock es el bloqueo mutuo de dos hilos que esperan adquirir mutex intercambiados. Para ilustrar este tipo de bloqueo se incluye el siguiente ejemplo.

**Ejemplo 8 – Los hilos h1 y h2 generan un DeadLock sobre los recursos “a” y “b”**

**//Ejemplo8.java**

**public** **class** Ejemplo8 **extends** Thread {

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

Recurso a = **new** Recurso();

Recurso b = **new** Recurso();

Hilo h1 = **new** Hilo(a, b, "uno");

Hilo h2 = **new** Hilo(b, a, "dos");

h1.start();

h2.start();

}

}

**//Hilo.java**

**class** Hilo **extends** Thread {

Recurso a;

Recurso b;

**public** Hilo(Recurso a, Recurso b,String nombre) {

**super**(nombre);

**this**.a = a;

**this**.b = b;

}

**public** **void** run(){

System.***out***.println("Hilo " + **this**.getName() + " comienza");

**synchronized**(a) {

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**synchronized**(b) {

}

System.***out***.println("Hilo " + **this**.getName() + " ha terminado");

}

}

}

**class** Recurso{

}

**Resultado obtenido:**

Hilo uno comienza

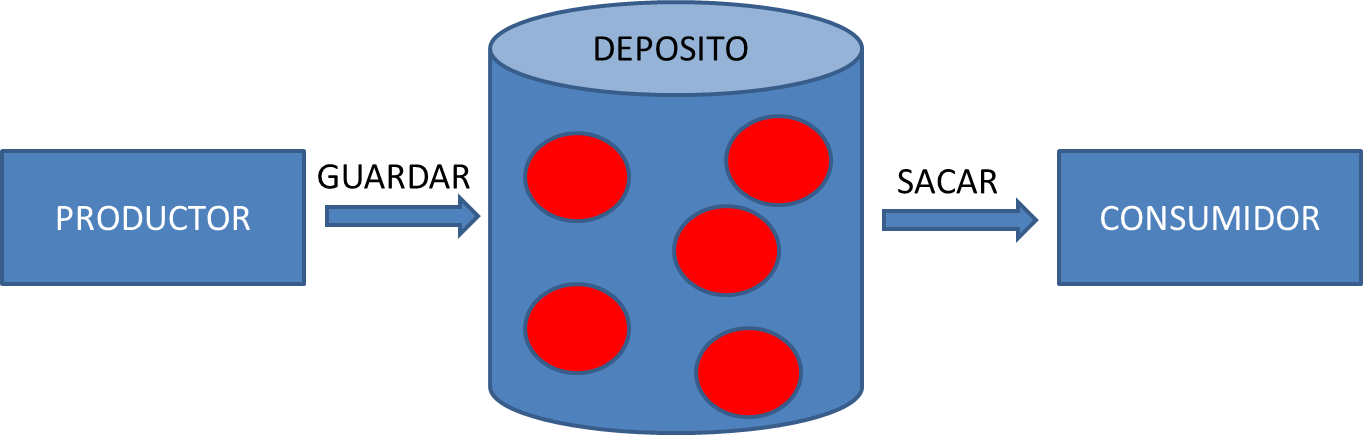
Hilo dos comienza

En este ejemplo no se imprimen los mensajes indicando que los hilos *uno* y *dos* han terminado, porque dichos hilos se quedan para siempre esperando en la sentencia synchronized(b).

Lo que sucede es que el hilo "h1" bloquea el recurso "a", se duerme durante 100 milisegundos (invocando al método sleep) y de esa forma al hilo "h2" le da tiempo a bloquear el recurso "b". Después de que el hilo “h1” haya terminado su pequeña siesta de 100 milisegundos, cada uno de los hilos tratará de obtener el recurso que el otro hilo tiene bloqueado; es decir, el hilo “h1” querrá bloquear el recurso “b” y el hilo “h2” querrá bloquear el recurso “a”.

# SINCRONIZACIÓN DE HILOS

## El modelo productor-consumidor

**

*Figura 11.1 – Modelo productor-consumidor*

Este modelo se basa en lo siguiente:

* Un hilo (el productor) genera un elemento que es agregado a un depósito.
* El elemento agregado es consumido por otro hilo (consumidor).
* El depósito tiene capacidad limitada y cuando está lleno, el productor debe esperar hasta que haya espacio.
* El consumidor debe esperar hasta que haya elementos en el depósito para poder retirarlos.

Supongamos que la capacidad del depósito es de un elemento. El depósito no solo debe soportar acceso concurrente, sino que el productor y el consumidor tienen que actuar de forma sincronizada.

**Ejemplo 9 – Implementación de un hilo productor y otro consumidor**

**//Productor.java**

**public** **class** Productor **extends** Thread {

**private** Deposito deposito;

**public** Productor(Deposito d) {

deposito = d;

}

**public** **void** run() {

**for** (**int** i=1;i<20 ;i++ )

deposito.guardar();

}

}

**//Consumidor.java**

**public** **class** Consumidor **extends** Thread{

**private** Deposito deposito;

**public** Consumidor(Deposito d) {

deposito = d;

}

**public** **void** run() {

**for** (**int** i=1;i<20 ;i++ )

deposito.sacar();

}

}

Una solución simple sería que el productor verificase si hay espacio en el depósito antes de agregar un elemento al mismo y que el consumidor verificase si hay elementos en el depósito antes de sacarlos.

**//Deposito.java**

**public** **class** Deposito {

**private** **int** elementos = 0;

**public** **synchronized** **void** guardar() {

**if** ( elementos == 0 )

elementos++;

System.***out***.println("Guardar:" + elementos);

**return**;

}

**public** **synchronized** **void** sacar() {

**if**( elementos > 0 )

elementos--;

System.***out***.println("Sacar:" + elementos);

**return**;

}

}

El programa Java completo que daría solución a esta necesidad estaría compuesto por cuatro clases (Ejemplo9, Deposito, Productor y Consumidor) y el detalle de la clase principal (clase Ejemplo9) sería el siguiente:

**//Ejemplo9.java**

**public** **class** Ejemplo9 {

**public** **static** **void** main( String[] arg ) {

Deposito deposito = **new** Deposito();

Productor productor = **new** Productor( deposito );

Consumidor consumidor = **new** Consumidor( deposito );

productor.start();

consumidor.start();

}

}

El inconveniente que tiene esta solución es que existe un alto consumo de recursos de CPU en procesos improductivos. Una solución podría ser detener los hilos hasta que se den las condiciones adecuadas para que actúen. Dicha detención se explicará en el siguiente apartado de este documento con unos métodos de la clase Object: wait, notify y notifyAll.

## Métodos wait, notify y notifyAll de la clase Object

* La clase Objectprovee el método **wait()** para detener la ejecución de un hilo hasta que le sea notificada la posibilidad de continuar.
* El método **wait()** debe ser invocado sobre un objeto compartido por los hilos a sincronizar; por ejemplo, en el modelo productor-consumidor, el objeto deposito.
* Para poder invocar el método **wait()** es necesario que el hilo tenga el mutex del objeto compartido.
* La invocación del método **wait()** detiene el hilo, lo pone en una lista de espera asociada al objeto y libera su mutex.
* El hilo saldrá́ de la lista de espera cuando otro hilo invoque el método **notify()** sobre el objeto compartido. Al salir de la lista de espera, se bloqueará en espera del mutex del objeto para continuar su ejecución.
* Una vez recuperado el mutex del objeto, el hilo que salió de la lista de espera continuará la ejecución en la instrucción siguiente a la llamada al método **wait()**.
* Si hay más de un hilo en la lista de espera, **notify()** reactivará sólo uno de ellos. El criterio de selección del hilo a reactivar depende de la implementación de Java.
* El método **wait()** puede generar una **InterruptedException**.
* El hilo que invoca al método **notify()** no tiene ninguna referencia sobre el hilo que está en espera (la notificación actúa sobre un objeto compartido y no sobre un hilo).
* El método **notifyAll()** permite reactivar todos los hilos bloqueados en la lista de espera de un objeto; es decir, todos los hilos bloqueados se vuelven ejecutables, aunque sólo uno de ellos podrá tomar el mutex a la vez.

**Ejemplo 10 – Implementación del objeto compartido deposito, un productor y un consumidor con la ayuda de los métodos wait, notify**

**//Ejemplo10.java**

**public** **class** Ejemplo10 {

**public** **static** **void** main( String[] arg ) {

Deposito deposito = **new** Deposito();

Productor productor = **new** Productor( deposito );

Consumidor consumidor = **new** Consumidor( deposito );

productor.start();

consumidor.start();

}

}

**//Deposito.java**

**class** Deposito{

**private** **int** elementos = 0;

**public** **synchronized** **void** guardar() {

**try**{

**if**( elementos > 0) // Mas adelante se verá que no es correcto

**this**.wait();

} **catch**( InterruptedException e ){}

elementos++;

System.***out***.println( "Guardar - numero elementos: " + elementos );

**this**.notify();

}

**public** **synchronized** **void** sacar() {

**try**{

**if**( elementos == 0) // Mas adelante se verá que no es correcto

**this**.wait();

} **catch**( InterruptedException e ){}

elementos--;

System.***out***.println( "Sacar - numero elementos: " + elementos );

**this**.notify();

}

}

**//Productor.java**

**class** Productor **extends** Thread {

**private** Deposito deposito;

**public** Productor(Deposito d) {

deposito = d;

}

**public** **void** run() {

**for** (**int** i=1;i<20 ;i++ )

deposito.guardar();

}

}

**//Consumidor.java**

**class** Consumidor **extends** Thread{

**private** Deposito deposito;

**public** Consumidor(Deposito d) {

deposito=d;

}

**public** **void** run() {

**for** (**int** i=1;i<20 ;i++ )

deposito.sacar();

}

}

## Sincronización y bloqueo iterativo (spin lock)

De todas formas, en la implementación del objeto compartido *deposito* aún existe un problema, el cual aparece en el supuesto de que haya más de un hilo Consumidor, el deposito esté vacío y se produzca la siguiente secuencia de eventos:

1.- Un hilo Consumidor toma el mutex del deposito y verifica la existencia de un elemento en él. Dado que el depósito está vacío, invoca el método wait (sobre el deposito), se bloquea en la lista de espera (de deposito) y libera su mutex.

**if**( elementos > 0)

**this**.wait();

2.- El hilo Productor es reactivado, adquiere el mutex de depósito, comprueba que éste está vacío, le agrega un elemento e invoca notify (sobre el objeto deposito).

3.- El hilo Consumidor que estaba en la lista de espera de deposito es notificado (sacado de dicha lista), y puesto en espera del mutex (de deposito).

4.- El hilo Productor libera el mutex de deposito.

5.- Otro hilo Consumidor, que no estaba en la lista de espera de deposito, adquiere su mutex, comprueba que el deposito tiene un elemento, lo saca del deposito, invoca notify y libera el mutex de deposito.

6.- El primer hilo Consumidor (que en el paso 3 había sido sacado de la lista de espera y bloqueado en espera del mutex) adquiere el mutex de deposito (antes de que el hilo Productor trate de agregar un elemento), y continúa su ejecución en el punto en que estaba: trata de sacar un elemento, pero el depósito está vacío: ERROR.

Este problema nace del hecho que la especificación de Java no establece que la salida de la lista de espera y adquisición del mutex sean implementados como una operación atómica (distintas JVM se pueden comportar de distinto modo).

Problemas análogos se presentan cuando:

* Hay más de un *Productor*.
* La notificación ocurre con *notifyAll* en vez de con *notify*.

La solución a estos problemas se resuelve sustituyendo la siguiente estructura:

**if**( condición de detención )

wait();

Por la siguiente estructura:

**while**( condición de detención )

wait();

Por tanto, el código Java de la clase Deposito sería el siguiente:

**//Deposito.java**

**class** Deposito{

**private** **int** elementos = 0;

**public** **synchronized** **void** guardar() {

**try**{

**while** ( elementos > 0)

**this**.wait();

} **catch**( InterruptedException e ){}

elementos++;

System.***out***.println( "Guardar - numero elementos: " + elementos );

**this**.notify();

}

**public** **synchronized** **void** sacar() {

**try**{

**while**( elementos == 0)

**this**.wait();

} **catch**( InterruptedException e ){}

elementos--;

System.***out***.println( "Sacar - numero elementos: " + elementos );

**this**.notify();

}

}