

Programación multihilo IV

Programación de servicios y procesos

Contenidos:

- 1) Gestión y planificación de hilos.
- 2) Prioridad de hilos.
- 3) Hilos egoístas y programación expulsora.
- 4) Hilos demonio.
- 5) Pool de hilos.
- 6) Mecanismos de comunicación y sincronización de hilos. Semáforos.

Gestión y planificación de hilos.

La ejecución de hilos se puede realizar mediante:

- **Paralelismo.** En un sistema con múltiples CPU, cada CPU puede ejecutar un hilo diferente.
- **Pseudoparalelismo.** Si no es posible el paralelismo, una CPU es responsable de ejecutar múltiples hilos.

La ejecución de múltiples hilos en una sola CPU requiere la planificación de una secuencia de ejecución (sheduling).

El **planificador de hilos de Java** (Sheduler) utiliza un algoritmo de secuenciación de hilos denominado *fixed priority scheduling* que está basado en un sistema de prioridades relativas, de manera que el algoritmo secuencia la ejecución de hilos en base a la prioridad de cada uno de ellos.

Gestión y planificación de hilos.

El funcionamiento del algoritmo es el siguiente:

- ❑ El hilo elegido para ejecutarse, siempre es el hilo "Ejecutable" de prioridad más alta.
- ❑ Si hay más de un hilo con la misma prioridad, el orden de ejecución se maneja mediante un algoritmo por turnos (*round-rubin*) basado en una cola circular FIFO (Primero en entrar, primero en salir).
- ❑ Cuando el hilo que está "ejecutándose" pasa al estado de "No Ejecutable" o "Muerto", se selecciona otro hilo para su ejecución.
- ❑ La ejecución de un hilo se interrumpe, si otro hilo con prioridad más alta se vuelve "Ejecutable". El hecho de que un hilo con una prioridad más alta interrumpa a otro se denomina "planificación apropiativa" ('preemptive sheudling').

Gestión y planificación de hilos.

Pero la responsabilidad de ejecución de los hilos es del Sistemas Operativos sobre el que corre la JVM, y Sistemas Operativos distintos manejan los hilos de manera diferente:

- En un Sistema Operativo que implementa **time-slicing** (subdivisión de tiempo), el hilo que entra en ejecución, se mantiene en ella sólo un micro-intervalo de tiempo fijo o cuanto (*quantum*) de procesamiento, de manera que el hilo que está "ejecutándose" no solo es interrumpido si otro hilo con prioridad más alta se vuelve "Ejecutable", sino también cuando su "cuanto" de ejecución se acaba. Es el patrón seguido por Linux, y por todos los Windows a partir de Windows 95 y NT.
- En un Sistema Operativo que **no** implementa **time-slicing** el hilo que entra en ejecución, es ejecutado hasta su muerte; salvo que regrese a "No ejecutable", u otro hilo de prioridad más alta alcance el estado de "Ejecutable" (en cuyo caso, el primero regresa a "preparado" para que se ejecute el segundo). Es el patrón seguido en el Sistema Operativo Solaris.

Prioridad de hilos.

En Java, cada hilo tiene una prioridad representada por un valor de tipo entero entre 1 y 10. Cuanto mayor es el valor, mayor es la prioridad del hilo.

Por defecto, el hilo principal de cualquier programa, o sea, el que ejecuta su método `main()` siempre es creado con prioridad 5. El resto de hilos secundarios (creados desde el hilo principal, o desde cualquier otro hilo en funcionamiento), heredan la prioridad que tenga en ese momento su hilo padre.

En la clase `Thread` se definen 3 constantes para manejar estas prioridades:

- **MAX_PRIORITY** (= 10). Es el valor que simboliza la máxima prioridad.
- **MIN_PRIORITY** (=1). Es el valor que simboliza la mínima prioridad.
- **NORM_PRIORITY** (= 5). Es el valor que simboliza la prioridad normal, la que tiene por defecto el hilo donde corre el método `main()`.

Prioridad de hilos.

Además en cualquier momento se puede obtener y modificar la prioridad de un hilo, mediante los siguientes métodos de la clase `Thread`:

- **`getPriority()`**. Obtiene la prioridad de un hilo. Este método devuelve la prioridad del hilo.
- **`setPriority()`**. Modifica la prioridad de un hilo. Este método toma como argumento un entero entre 1 y 10, que indica la nueva prioridad del hilo.

Java tiene 10 niveles de prioridad que no tienen porqué coincidir con los del sistema operativo sobre el que está corriendo. Por ello, lo mejor es que utilices en tu código sólo las constantes `MAX_PRIORITY`, `NORM_PRIORITY` y `MIN_PRIORITY`.

Podemos conseguir aumentar el rendimiento de una aplicación multihilo gestionando adecuadamente las prioridades de los diferentes hilos, por ejemplo utilizando una prioridad alta para tareas de tiempo crítico y una prioridad baja para otras tareas menos importantes.

Hilos egoístas y programación expulsora.

¿De verdad existen los hilos egoístas? En un Sistema Operativo que no implemente time-slicing puede ocurrir que un hilo que entra en "ejecución" no salga de ella hasta su muerte, de manera que no dará ninguna posibilidad a que otros hilos "preparados" entren en "ejecución" hasta que él muera. Este hilo se habrá convertido en un **hilo egoísta**.

Según esto, un mismo programa Java se puede ejecutar de diferente manera según el Sistema Operativo donde corra. Entonces, ¿qué pasa con la portabilidad de Java? Java da solución a este problema mediante lo que se conoce como **programación expulsora** a través del método **yield()** de la clase `java.lang.thread`:

- ➔ **yield()** hace que un hilo que está "ejecutándose" pase a "preparado" para permitir que otros hilos de la misma prioridad puedan ejecutarse.

Hilos egoístas y programación expulsora.

Sobre el método **yield()** y el egoísmo de los threads debes tener en cuenta que:

- El funcionamiento de `yield()` no está garantizado, puede que después de que un hilo invoque a `yield()` y pase a "preparado", éste vuelva a ser elegido para ejecutarse.
- No debes asumir que la ejecución de una aplicación se realizará en un Sistema Operativo que implementa time-slicing.
- En la aplicación debes incluir adecuadamente llamadas al método `yield()`, incluso a `sleep()` o `wait()`, si el hilo no se bloquea por una Entrada/Salida.

Prioridad de hilos. Ejemplo

Dentro de la carpeta recursos multihilo IV, tenemos el proyecto UT3_4_PrioridadHilos. Tiene 2 clases, una que es el Hilo y otra Programa que lanza los hilos. Dentro de la clase Hilo, tenemos:

```
/**
 * ejecuta una tarea pesada
 */
@Override
public void run() {

    //cadena
    String strCadena = "";

    //agrega 30000 caracteres a una cadena vacía
    for (int i = 0; i < 20000; ++i) {
        //imprime el valor en la Salida
        strCadena += "A";
        this.yield();
        //yield() sugiere al planificador Java que puede seleccionar otro hilo,
    }

    System.out.println("Hilo de prioridad " + this.getPriority()
        + " termina ahora");
}
```

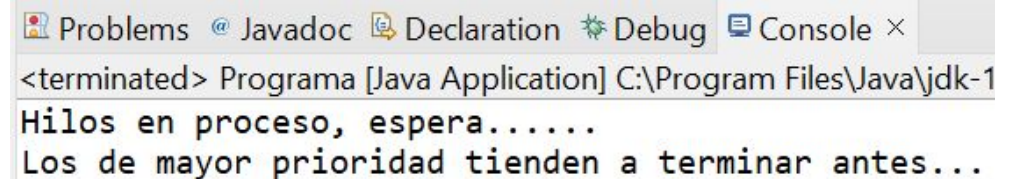
La tarea consiste en cada hilo en realidad sólo tiene que añadir 20.000 veces una letra a una variable. Pero después de añadir cada letra, “devuelve el control” al planificador para que determine qué hilo debe entrar al proceso.

Por cada hilo indicamos su prioridad y cuando acaba

Prioridad de hilos. Ejemplo

La ejecución de la clase simplemente implementa 15 hilos, 5 con cada prioridad max, min y estándar y podemos ver en la ejecución que acaban primero los más prioritarios

```
public static void main(String[] args) {  
    int contador = 5;  
  
    //vectores para hilos de distintas prioridades  
    Thread[] hiloMIN = new Thread[contador];  
    Thread[] hiloNORM = new Thread[contador];  
    Thread[] hiloMAX = new Thread[contador];  
  
    //crea los hilos de prioridad mínima  
    for (int i = 0; i < contador; i++) {  
        hiloMIN[i] = new Hilo(Thread.MIN_PRIORITY);  
    }  
  
    //crea los hilos de prioridad normal  
    for (int i = 0; i < contador; i++) {  
        hiloNORM[i] = new Hilo();  
    }  
  
    //crea los hilos de máxima prioridad  
    for (int i = 0; i < contador; i++) {  
        hiloMAX[i] = new Hilo(Thread.MAX_PRIORITY);  
    }  
  
    System.out.println("Hilos en proceso, espera.....\nLos de mayor "  
        + "prioridad tienden a terminar antes...\n");  
  
    //inicia los hilos  
    for (int i = 0; i < contador; i++) {  
        hiloMIN[i].start();  
        hiloNORM[i].start();  
        hiloMAX[i].start();  
    }  
}
```



The screenshot shows a Java IDE window with tabs for Problems, Javadoc, Declaration, Debug, and Console. The Console tab is active, displaying the output of the program. The output starts with a header line indicating the program is terminated and the file path. Below this, it shows the text 'Hilos en proceso, espera.....' and 'Los de mayor prioridad tienden a terminar antes...'. The rest of the output consists of 15 lines, each stating 'Hilo de prioridad X termina ahora', where X is the thread's priority level.

```
<terminated> Programa [Java Application] C:\Program Files\Java\jdk-1  
Hilos en proceso, espera.....  
Los de mayor prioridad tienden a terminar antes...
```

```
Hilo de prioridad 10 termina ahora  
Hilo de prioridad 10 termina ahora  
Hilo de prioridad 10 termina ahora  
Hilo de prioridad 10 termina ahora  
Hilo de prioridad 10 termina ahora  
Hilo de prioridad 5 termina ahora  
Hilo de prioridad 5 termina ahora  
Hilo de prioridad 5 termina ahora  
Hilo de prioridad 5 termina ahora  
Hilo de prioridad 5 termina ahora  
Hilo de prioridad 1 termina ahora  
Hilo de prioridad 1 termina ahora  
Hilo de prioridad 1 termina ahora  
Hilo de prioridad 1 termina ahora  
Hilo de prioridad 1 termina ahora
```

Hilos demonio.

Un demonio, a diferencia de los hilos tradicionales, no forma parte de la esencia del programa, sino de la máquina de Java. Los **demonios** son usados generalmente para **prestar servicios en un segundo plano** a todos los programas que puedan necesitar el tipo de servicio proporcionado. Si un hilo es un demonio, cualquier hilo que el cree será automáticamente un demonio.

Para crear un hilo demonio sólo hay que crear un hilo normal y enviarle el mensaje **setDaemon**: `hilo.setDaemon(true);`

Para saber si un hilo es un demonio, simplemente hay que enviar el mensaje **isDaemon**. El método que se ejecuta devolverá `true` si el hilo es un demonio, y `false` en caso contrario.

Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (garbage collector). Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Pool de hilos

Cuando trabajamos con aplicaciones tipo servidor, éstas tienen que atender un número masivo y concurrente de peticiones de usuario, en forma de tareas que deben ser procesadas lo antes posible. Al principio de la unidad ya te indicamos mediante un ejemplo la conveniencia de utilizar hilos en estas aplicaciones, pero si ejecutamos cada tarea en un hilo distinto, se pueden llegar a crear tantos hilos que el incremento de recursos utilizados puede comprometer la estabilidad del sistema. Los *pools de hilos* ofrecen una solución a este problema.

Y ¿qué es un pool de hilos? Un pool de hilos (thread pools) es básicamente **un contenedor** dentro del cual se crean y se inician **un número limitado de hilos**, para ejecutar **todas las tareas de una lista**.

Pool de hilos

Para declarar un pool, lo más habitual es hacerlo como un objeto del tipo `ExecutorService` utilizando alguno de los siguientes métodos de la clase estática `Executors`:

- ✓ **`newFixedThreadPool(int numeroHilos)`**: crea un pool con el número de hilos indicado. Dichos hilos son reutilizados cíclicamente hasta terminar con las tareas de la cola o lista.
- ✓ **`newCachedThreadPool()`**: crea un pool que va creando hilos conforme se van necesitando, pero que puede reutilizar los ya concluidos para no tener que crear demasiados. Los hilos que llevan mucho tiempo inactivos son terminados automáticamente por el pool.
- ✓ **`newSingleThreadExecutor()`**: crea un pool de un solo hilo. La ventaja que ofrece este esquema es que si ocurre una excepción durante la ejecución de una tarea, no se detiene la ejecución de las siguientes.
- ✓ **`newScheduledExecutor()`** : crea un pool que va a ejecutar tareas programadas cada cierto tiempo, ya sea una sola vez o de manera repetitiva. Es parecido a un objeto `Timer`, pero con la diferencia de que puede tener varios threads que irán realizando las tareas programadas conforme se desocupen.

Pool de hilos

Los objetos de tipo **ExecutorService** implementan la interfaz **Executor**. Esta interfaz define el método `execute(Runnable)`, al que hay que llamar una vez por cada tarea que deba ser ejecutada por el pool (la tarea se pasa como argumento del método).

La interface `ExecutorService` proporciona una serie de métodos para el control de la ejecución de las tareas, entre ellos el método `shutdown()`, para indicarle al pool que los hilos no se van a reutilizar para nuevas tareas y deben morir cuando finalicen su trabajo.

Ventajas del uso de `ExecutorService`?? https://oregoom.com/java/pool-de-hilos/#google_vignette

Lo cierto que es bastante habitual lanzar los hilos con esta interfaz, por lo que es recomendable hacerlo.

Pool de hilos. Ejemplo

```
6 public class Main {
7
8     /**
9      * ejecuta 30 veces la tarea NumerosAleatorios que imprime diez números
10     * aleatorios menores que cincuenta, mediante un pool de tan sólo dos hilos
11     */
12     public static void main(String[] args) {
13
14         //define un pool fijo de dos hilos
15         ExecutorService executor = Executors.newFixedThreadPool(2);
16
17         //pasa 30 tareas NumerosAleatorios al pool de 2 hilos
18         for (int i = 1; i <= 30; i++) {
19             executor.submit(new NumerosAleatorios());
20         }
21
22         //ordena la destrucción de los dos hilos del pool cuando hayan
23         //completado todas las tareas
24         executor.shutdown();
25     }
26 }
```


Pool de hilos. Ejemplo

```
public class NumerosAleatorios implements Runnable {  
    /**  
     * compone una cadena de diez números aleatorios menores que 50, separados  
     * por ','  
     */  
    public void run() {  
        String espacio = "";  
        Random random = new Random();  
  
        for (int i = 0; i < 10; i++) {  
            espacio += random.nextInt(50) + ", ";  
            Thread.yield();  
        }  
  
        System.out.println("Números aleatorio obtenidos por "  
            + Thread.currentThread().getName() + ": " + espacio);  
    }  
}
```

Sincronización de hilos. Semáforos

La clase **Semaphore** del paquete *java.util.concurrent*, permite definir un semáforo para controlar el acceso a un recurso compartido. Para crear y usar un objeto Semaphore haremos lo siguiente:

- Indicar al constructor **Semaphore (int permisos)** el total de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. Este valor coincide con el número de hilos que pueden acceder a la vez al recurso.
- Indicar al semáforo mediante el método **acquire()**, que queremos acceder al recurso, o bien mediante *acquire(int permisosAdquirir)* cuántos permisos se quieren consumir al mismo tiempo.
- Indicar al semáforo mediante el método **release()**, que libere el permiso, o bien mediante *release(int permisosLiberar)*, cuantos permisos se quieren liberar al mismo tiempo.
- Hay otro constructor **Semaphore (int permisos, boolean justo)** que mediante el parámetro justo permite garantizar que el primer hilo en invocar *acquire()* será el primero en adquirir un permiso cuando sea liberado. Esto es, garantiza el orden de adquisición de permisos, según el orden en que se solicitan.

Semáforos vs Sincronización

El uso de `synchronized` permite establecer secciones críticas a las que únicamente tiene acceso un hilo en un determinado momento. Pero existen otros escenarios en los que los recursos limitados permiten el acceso de más de un hilo.

- ❑ Cuando una sección crítica permite ser ejecutada por más de 1 hilo, pero el número de hilos que pueden acceder está limitado, es cuando es conveniente utilizar los semáforos.
- ❑ El semáforo proporciona, a través del constructor, la capacidad de definir el número de accesos concurrentes a un determinado recurso o bloque de código.
- ❑ Si todos los permisos están concedidos, los hilos se quedan a la espera de que los propietarios de los permisos los liberen

En determinadas circunstancias puede ser necesario utilizar ambos métodos..

Ejemplo Semáforo - Básico

Dentro de la carpeta recursos multihilo IV, tenemos el proyecto UT3_6_Semaforo.

```
public class Cola {
    Semaphore miSemaforo;

    public Cola(Semaphore miSemaforo) {
        super();
        this.miSemaforo = miSemaforo;
    }

    public void pasar(String nombre) {
        try {
            System.out.println("ESPERANDO semaforo hilo:"+nombre);
            this.miSemaforo.acquire();
            System.out.println("PASANDO semaforo hilo:"+nombre);
            System.out.println("Paso 1");
            Thread.sleep(1000);
            System.out.println("Paso 2");
            Thread.sleep(1000);
            System.out.println("Paso 3");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("SOLTANDO semaforo hilo:"+nombre);
        this.miSemaforo.release();
    }
}
```

Implementamos el semáforo

Pedimos un permiso. Si hay libre haremos
3 pasos de espera de 1 segundo y
soltaremos el permiso

Ejemplo Semáforo - Básico

Dentro de la carpeta recursos multihilo IV, tenemos el proyecto UT3_6_Semaforo.

```
public class Hilo extends Thread {  
  
    Cola miCola;  
  
    public Hilo(Cola miCola) {  
        super();  
        this.miCola = miCola;  
    }  
  
    public void run() {  
        miCola.pasar(this.getName());  
    }  
}
```

Los hilos como siempre reciben la cola que a su vez contiene el semáforo

Llaman al método pasar de la cola que es el que controla el semáforo. Pero podríamos tener el control del semáforo en los hilos si queremos porque el semáforo es el mismo para todos los hilos

Ejemplo Semáforo - Básico

Arrancamos los hilos pudiendo definir cuantos en paralelo y con el número de semáforos que queramos.

```
public class Principal {  
    public static void main (String args[]) {  
        Semaphore miSemaforo = new Semaphore(2);  
        Cola micola = new Cola(miSemaforo);  
  
        ExecutorService miExecutor = Executors.newFixedThreadPool(2);  
  
        for(int i=0; i<10; i++ ) {  
            miExecutor.submit(new Hilo(micola));  
        }  
  
        miExecutor.shutdown();  
    }  
}
```