CESUR

UT2. Programación de Hilos

PSP - DAM

Francisco Gallego Perona

Introducción a los hilos

La ejecución de un proceso comienza con un único hilo, pero se pueden crear más sobre la marcha.

Los distintos hilos de un mismo proceso comparten:

- El **espacio de memoria** asociado al proceso.
- La información de **acceso a ficheros** (almacenamiento de datos y E/S).

Cada hilo tiene sus propios valores para:

- **Registros del procesador**.
- El estado de su **pila** (stack). En la pila se guarda información acerca de las **llamadas en curso de ejecución a métodos** de diversos objetos. Para cada llamada se guardan, entre otras cosas, los **datos locales** (en variables internas del método).

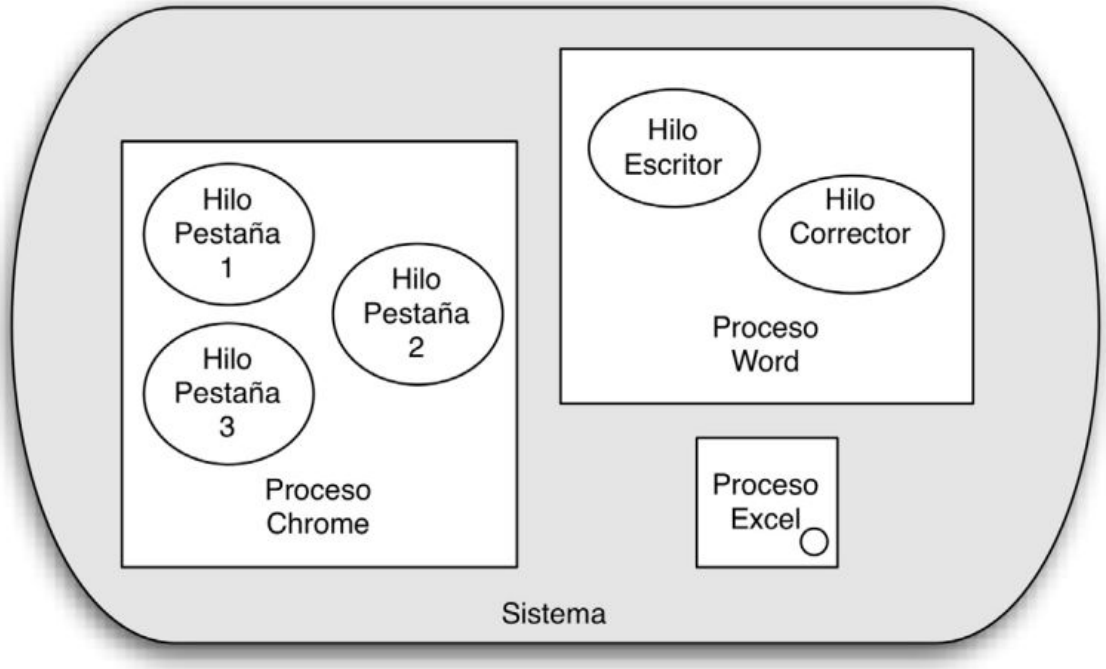
Hilos vs Procesos

Hilo (Thread):

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

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

Hilos vs Procesos



Multitarea

Multitarea: ejecución simultánea de varios hilos:

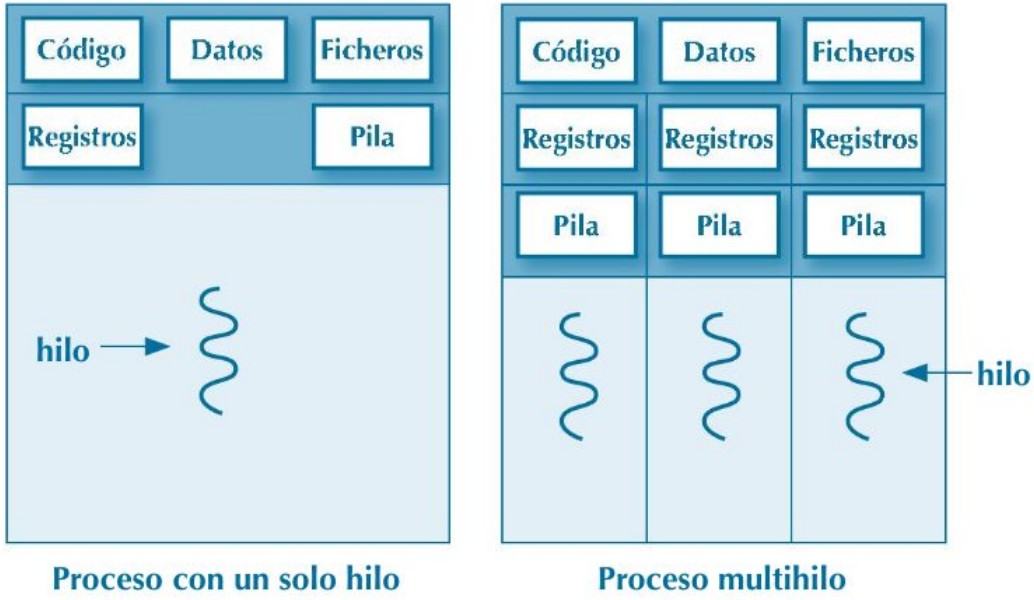
- **Capacidad de respuesta**. Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas (hilo) que esté realizando el programa sea muy larga.
- **Compartición de recursos**. Por defecto, los threads comparten la memoria y todos los recursos del proceso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

Recursos compartidos por hilos

Los procesos mantienen su propio espacio de direcciones y recursos de ejecución mientras que los hilos dependen del proceso.

- Comparten con otros hilos la sección de código, datos y otros recursos.
- Cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.

Introducción a los hilos



Estados de un hilo

Los hilos pueden cambiar de estado a lo largo de su ejecución.

Se definen los siguientes estados:

- **Nuevo**: el hilo está preparado para su ejecución pero todavía no se ha realizado la llamada correspondiente en la ejecución del código del programa.
- **Listo**: el hilo no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
- **Runnable**: el hilo está preparado para ejecutarse y puede estar ejecutándose.
- **Bloqueado**: el hilo está bloqueado por diversos motivos, esperando un suceso que lo devuelva a estado Runnable.
- **Terminado**: el hilo ha finalizado su ejecución.

Introducción a los hilos

El intercambio de información entre hilos es sencillo, dado que los distintos hilos de un mismo proceso comparten la memoria asignada al proceso por el sistema operativo. Pero los distintos hilos de un mismo proceso pueden coordinarse para el acceso a los contenidos de la memoria y a los ficheros que se utilizan para controlar las operaciones de E/S.

Creación de hilos en Java

Un programa en Java → Se inicia con **un solo hilo**, que ejecuta el método main() de una clase.

Cómo crear un hilo en Java → implementando la **Interfaz Runnable**

\+ programando el **método run()**

Clase Thread

Para añadir la funcionalidad de hilo a una clase, debe heredar de la clase Thread.

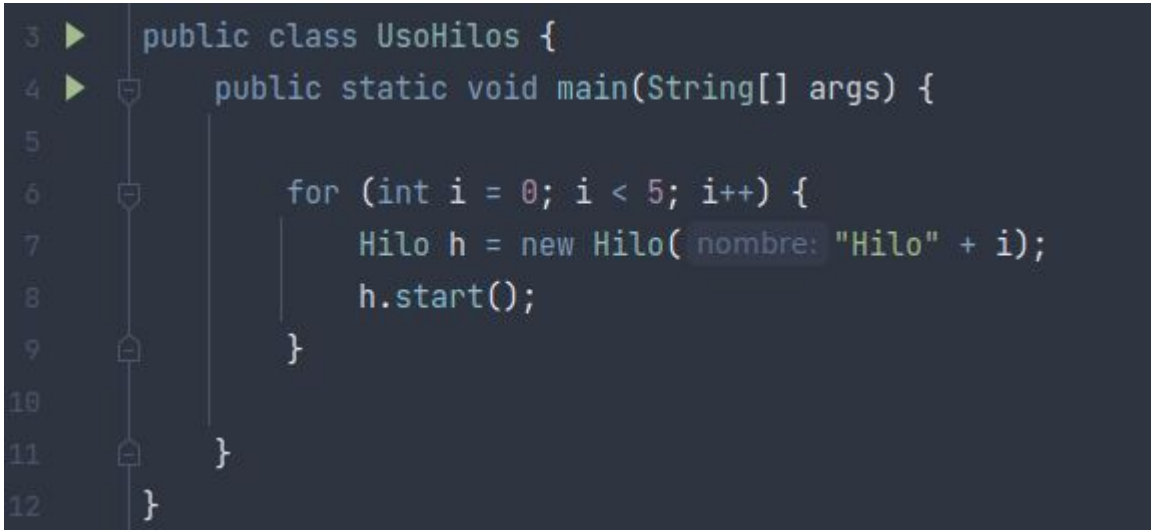
Debe sobreescribir el método run(). También cuenta con los métodos start y stop.

```java
public class Hilo extends Thread {
    1 usage
    public Hilo(String nombre) {
        super(nombre);
        System.out.println("Creando hilo: " + getName());
    }

    public void run() {
        System.out.println("Ejecutando hilo: " + getName());
    }
}
```

Clase Thread

Usamos el Hilo creado en una clase UsoHilos con un método principal.



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

        for (int i = 0; i < 5; i++) {
            Hilo h = new Hilo( nombre: "Hilo" + i);
            h.start();
        }

    }
}
```

Métodos de la clase Thread

MÉTODOS	MISIÓN
start()	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método **run()** de este hilo.
boolean isAlive()	Comprueba si el hilo está vivo
sleep(long mils)	Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción *InterruptedException*.
run()	Constituye el cuerpo del hilo. Es llamado por el método **start()** después de que el hilo apropiado del sistema se haya inicializado. Si el método **run()** devuelve el control, el hilo se detiene. Es el único método de la interfaz **Runnable**.
String toString()	Devuelve una representación en formato cadena de este hilo, incluyendo el nombre del hilo, la prioridad, y el grupo de hilos. Ejemplo: Thread[HILO1,2,main]
long getId()	Devuelve el identificador del hilo.
void yield()	Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
String getName()	Devuelve el nombre del hilo.
setName(String name)	Cambia el nombre de este hilo, asignándole el especificado como argumento.

Ejemplo de utilización de sleep(milisegundos)

Modificamos el ejemplo anterior para que se detenga la ejecución de uno de los hilos creados (el hilo cuyo nombre es Hilo3) durante 2 segundos (2000 milisegundos).

```java
public class Hilo extends Thread {
    1 usage
    public Hilo(String nombre) {
        super(nombre);
        System.out.println("Creando hilo: " + getName());
    }

    public void run() {

        if (this.getName().equals("Hilo3")){
            try {
                sleep( millis: 2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Ejecutando hilo: " + this.getName() + " con Id: " + this.getId());
    }
}
```

Ejemplo de código de hilos

Ver código de paquete **finish**

Viendo el estado de un hilo

```java
public class Hilo extends Thread {

    public Hilo(String nombre) {
        super(nombre);
        System.out.println("Creando hilo: " + getName());
        System.out.println("Estado al construir el hilo: " + this.getState());
    }

    public void run() {

        if (this.getName().equals("Hilo3")){
            try {
                sleep( millis: 2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Estado al ejecutar el hilo: " + this.getState());
        System.out.println("Ejecutando hilo: " + this.getName() + " con Id: " + this.getId());
    }
}
```

Más métodos de un hilo

Método	Funcionalidad
`void run()`	Se ejecuta cuando se lanza el hilo. Es el punto de entrada del hilo, como el método `main()` es el punto de entrada del proceso.
`void start()`	Lanza el hilo. La JVM crea el hilo y ejecuta su método `run()`.
`static void sleep(long ms)` `static void sleep(long ms, long ns)`	Detiene la ejecución del hilo actualmente en ejecución durante un tiempo, que se puede indicar en microsegundos o en una combinación de microsegundos y nanosegundos.

`void join()`
`void join(long ms)`
`void join(long ms, long ns)`
	Espera a que termine el hilo. Se puede indicar un tiempo máximo de espera, bien en milisegundos, bien en una combinación de milisegundos y nanosegundos.

Más métodos de un hilo

`public void interrupt()`
`public boolean isInterrupted()`
`public static boolean interrupted()`
	El primer método interrumpe la ejecución de un hilo. El segundo método verifica si se ha interrumpido un hilo. El tercer método (estático) verifica si se ha interrumpido la ejecución del hilo actual, y borra el estado de interrupción, de manera que una llamada posterior devolvería `false`, a menos que se vuelva a interrumpir.

`boolean isAlive()`	Comprueba si el hilo está vivo. Un hilo está vivo cuando se ha iniciado y no ha terminado su ejecución.
`int getPriority()` `void setPriority(int nuevaPrior)`	Se puede asignar una prioridad a un hilo, y se puede obtener la prioridad de un hilo.
`static Thread currentThread()`	Devuelve un objeto de clase `Thread` correspondiente al hilo en ejecución actualmente.
`long getId()`	Devuelve el identificador del hilo.
`String getName()` `void setName(String nombre)`	Se puede asignar un nombre a un hilo, y se puede recuperar el nombre del hilo.
`Thread.State getState()`	Devuelve el estado del hilo.
`boolean isDaemon()` `void setDaemon(boolean on)`	Un hilo puede ser de tipo *daemon*. La distinción es importante, porque la JVM termina su ejecución cuando no queda ningún hilo activo o cuando solo quedan hilos de tipo *daemon*.

Sincronización de hilos

- Un programa puede lanzar múltiples hilos que **colaboren** entre sí para la realización de una tarea. **Deben** utilizarse mecanismos de sincronización para evitar problemas que se puedan dar en determinadas situaciones.

VER EJERCICIO DE PAQUETE **coop**.

Exclusión mutua

El código del ejercicio anterior no funciona correctamente. El motivo es que las operaciones del método **increment** no son atómicas.

Todas las operaciones aritméticas se realizan en **registros del procesador**. Para realizarlas sobre datos que están en memoria, antes hay que pasar estos datos a un registro del procesador y, una vez realizadas, hay que pasar el resultado a memoria.

LAS OPERACIONES DE LOS HILOS SE PUEDEN INTERCALAR → DANDO LUGAR A **RESULTADOS NO DESEADOS**.

Exclusión mutua

Ejemplo de ejecución de 2 hilos modificando valores de una variable en común:

H1	`registro ← [cuenta]`	Se guarda valor 86 en **`registro`** para H1
H2	`registro ← [cuenta]`	Se guarda valor 86 en **`registro`** para H2
H1	`incrementar registro`	El valor de **`registro`** pasa a ser 87 para H1
H2	`incrementar registro`	El valor de **`registro`** pasa a ser 87 para H2
H1	`registro → [cuenta]`	El valor en la variable **`cuenta`** pasa a ser 87
H2	`registro → [cuenta]`	El valor en la variable **`cuenta`** pasa a ser 87. Se ha perdido el incremento hecho por H1

Bloques de código sincronizado

Las tres operaciones anteriores para incrementar el valor de la variable se deberían ejecutar sin que entre ellas se intercalen las mismas operaciones de otro hilo → **SECCIÓN CRÍTICA**

La sección crítica de diferentes hilos debería ejecutarse en **EXCLUSIÓN MUTUA**

Una forma de conseguir la exclusión mutua en Java es utilizar la palabra **synchronized**

Synchronized

Este modificador en un método hace que no este método no se pueda ejecutar concurrentemente para un objeto en más de un hilo.

```
1 usage
public synchronized int increment() {
    this.count++;
    return count;
}
```

Synchronized en bloques Java

- **En métodos no estáticos**: Se añade synchronized a la declaración del método. Dos hilos distintos no pueden estar a la vez ejecutando cada uno un método no estático sincronizado sobre el mismo objeto.
- **En métodos estáticos**: Bloqueo intrínseco sobre la clase a la que pertenece el método.
- **Bloques de código cualquiera**: Después de la palabra clave synchronized, y entre paréntesis, se indica el objeto sobre el que se realiza el bloqueo

Synchronized sobre bloques de código

```java
public void incrementar1() {
    synchronized (lock1) {
        cont1++;
    }
}

public long getContador1() {
    synchronized (lock1) {
        return cont1;
    }
}

public void incrementar2() {
    synchronized (lock2) {
        cont2++;
    }
}

public long getContador2() {
    synchronized (lock2) {
        return cont2;
    }
}
```

```java
private long cont1 = 0;
2 usages
private long cont2 = 0;
2 usages
private final Object lock1 = new Object();
2 usages
private final Object lock2 = new Object();
```

Compartición de recursos. Interbloqueo

Para realizar operaciones, los hilos necesitan **obtener el acceso en exclusiva** a un conjunto de recursos. Una vez terminada la operación, **liberan los recursos**, que pueden ser usados por otros hilos.

Los recursos se representan mediante **objetos**, y se obtiene acceso en exclusiva a ellos mediante bloques **synchronized**. Si se utiliza más de uno, se utilizan bloques **synchronized anidados**.

```
synchronized(r1) {
  synchronized(r2) {
    (realizar operaciones con r1 y r2)
  }
}
```

Compartición de recursos. Interbloqueo

Puede producirse un interbloqueo (deadlock) cuando dos hilos quedan mutuamente bloqueados, cada uno a la espera de que el otro desbloquee un objeto que ha bloqueado.