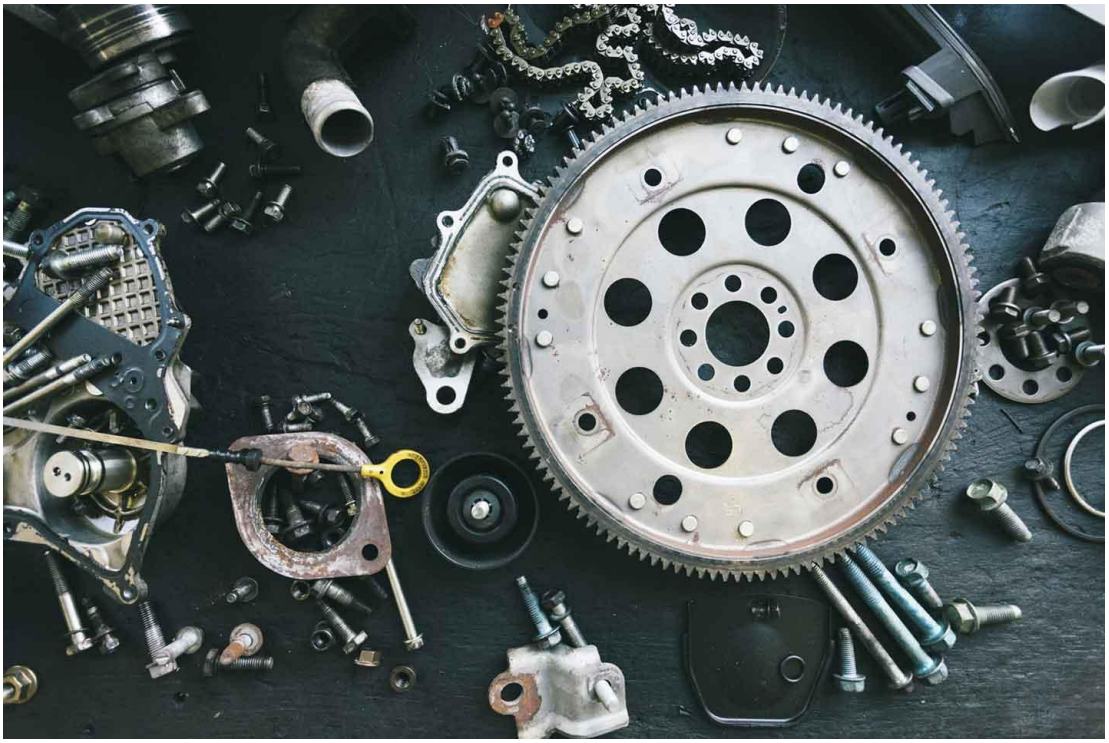
Programación de hilos

Programación de servicios y procesos



Progresa
Una formación de futuro

ÍNDICE

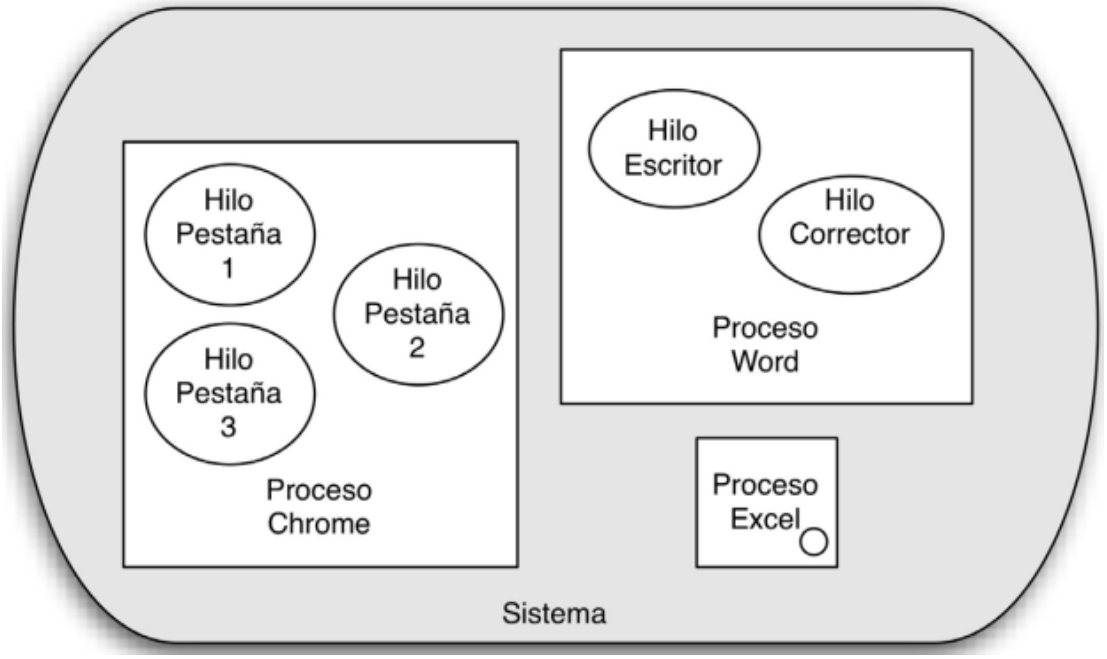
CONCEPTOS BÁSICOS

HILO

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

Diferencia con procesos

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



MULTITAREA

Ejecución simultánea de varios hilos:

- Capacidad de respuesta. Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas (hilo) que esté realizando el programa sea muy larga.
- Compartición de recursos. Por defecto, los threads comparten la memoria y todos los recursos del proceso al que pertenecen.

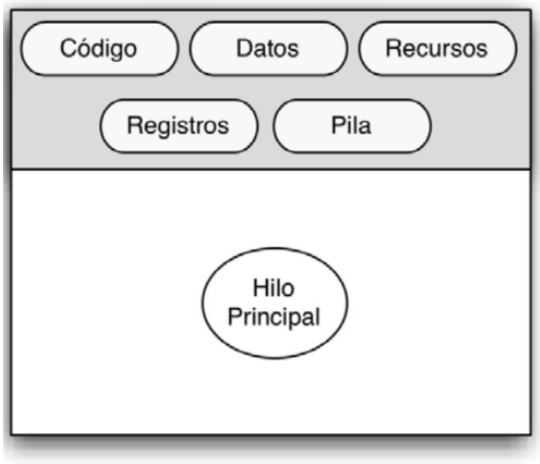
- La creación de nuevos hilos no supone ninguna reserva adicional de memoria por parte del sistema operativo.
- Paralelismo real. La utilización de threads permite aprovechar la existencia de más de un núcleo en el sistema en arquitecturas multicore.

Recursos compartidos por hilos

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

Comparten con otros hilos la sección de código, datos y otros recursos.

Cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.



Proceso con un único thread

Proceso con varios threads

CREACIÓN DE HILOS EN JAVA

Extendiendo de la clase Thread mediante la creación de una subclase. La clase Thread es responsable de producir hilos funcionales para otras clases e implementa la interfaz Runnable.

El método run() implementa la operación create conteniendo el código a ejecutar por el hilo. Dicho método contendrá el hilo de ejecución.

La clase Thread define también el método start() para implementar la operación create. Este método es el que comienza la ejecución del hilo de la clase correspondiente.

```
public class HiloBase extends Thread{
   public HiloBase(String name) {
        super(name);
   }
   @Override
   public void run() {
        System.out.println("Hola soy el hilo "+this.getName());
   }
}
```

Y la invocación desde el programa principal:

```
public static void main(String[] args) {
Thread hiloA = new HiloBase("A");
Thread hiloB = new HiloBase("B");

hiloA.start();
hiloB.start();
}
```

OPERACIONES

Join: La ejecución del hilo puede suspenderse esperando hasta que el hilo correspondiente por el que espera finalice su ejecución.

Sleep: duerme un hilo por un período especificado.

Ambas operaciones de espera pueden ser interrumpidas, si otro hilo interrumpe al hilo actual mientras está suspendido por dichas llamadas.

Una interrupción es una indicación a un hilo que debería dejar de hacer lo que esté haciendo para hacer otra cosa.

interrupt(): Envía una interrupción mediante la invocación del método en el objeto del hilo que se quiere interrumpir.

isAlive(): comprueba si el hilo no ha finalizado su ejecución antes de trabajar con él.

```
public class Ejemplo2 {

   public static void main(String[] args) {

        Thread hiloA = new Hilo2("A",1);

        Thread hiloB = new Hilo2("B",10);


        try {

            hiloB.join();

            hiloA.join();

        } catch (InterruptedException e) {

            throw new RuntimeException(e);

        }

        System.out.println("Soy el hilo principal y he terminado.");

                System.out.println("La variable global vale:" +
Hilo2.global);

   }

}
```

Y en la clase del hilo:

```
public class Hilo2 extends Thread{

    static int global = 0;

    public Hilo2(String name, int prioridad) {

        super(name);

        this.setPriority(prioridad);

        start();

    }

    @Override

    public void run() {

        for (int i = 0; i < 100; i++) {

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

        }

    }

}
```

Planificación de hilos

Cuando se trabaja con varios hilos, a veces es necesario pensar en la planificación de threads, para asegurarse de que cada hilo tiene una oportunidad justa de ejecutarse.

El planificador del sistema operativo determina qué proceso es el que se ejecuta en un determinado momento en el procesador (uno y solamente uno).

Dentro de ese proceso, el hilo que se ejecutará estará en función del número de núcleos disponibles y del algoritmo de planificación que se esté utilizando.

Java, por defecto, utiliza un planificador apropiativo cuando un hilo que se está ejecutando pasa al estado Runnable. Si los hilos tienen la misma prioridad, será el planificador el que asigne a uno u otro el núcleo correspondiente para su ejecución utilizando tiempo compartido.

SINCRONIZACIÓN DE HILOS

Los threads se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria.

Los threads pertenecen al mismo proceso, y pueden acceder a toda la memoria asignada a dicho proceso utilizando las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente.

Cuando varios hilos manipulan a la vez objetos compartidos, pueden ocurrir diferentes problemas:

- Condición de carrera
- Inconsistencia de memoria
- Inanición
- Interbloqueo
- Bloqueo activo

CONDICIONES DE CARRERA

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

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.

```
public class HiloCC extends Thread{
   // Variable global que comparten los diferentes hilos
      static int global = 0;


      public HiloCC(String name) {
        super(name);
        start();
      }
```

```
        @Override
        public void run() {
          for (int i = 0; i < 100; i++) {
              int j = global +1;
              global = j;
          }
          for (int i = 0; i < 100; i++) {
              int j = global -1;
              global = j;
          }
    }
}
```

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.

SECCIÓN CRÍTICA

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

```
 for (int i = 0; i < 100; i++) {
              int j = global +1; // ZONA CRÍTICA
              global = j;// ZONA CRÍTICA
          }
```

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

El problema de la sección crítica consiste en diseñar un protocolo que permita a los procesos cooperar. Cualquier solución al problema de la sección crítica debe cumplir:

1. Exclusión mutua: si un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su sección crítica.
2. Progreso: si ningún proceso está ejecutando su sección crítica y hay varios procesos que quieren entrar en su sección crítica, solo aquellos procesos que están esperando para entrar pueden participar en la decisión de quién entra definitivamente.
3. Espera limitada: debe existir un número limitado de veces que se permite a otros procesos entrar en su sección crítica después de que otro proceso haya solicitado entrar en la suya y antes de que se le conceda.

Para la implementación de la sección crítica se necesita un mecanismo de sincronización tanto que actúe tanto antes de entrar en la sección crítica como después de salir de ejecutarla.

SEMÁFOROS

Un semáforo se representa como:

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

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

public class HiloCC extends Thread{

// Definimos el valor inicial del semáforo que representa

// el número total de hilos que podrán pasar inicialmente

   private static Semaphore sem = new Semaphore(1);

   static int global = 0;

   public HiloCC(String name) {

       super(name);

       start();

   }
```

```
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                semaphore.acquire(); //semaphore -1
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            int j = global +1; // ZONA CRÍTICA
            global = j; // ZONA CRÍTICA
            semaphore.release(); // semaphore +1
        }
        for (int i = 0; i < 100; i++) {
            try {
                semaphore.acquire();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            int j = global -1;// ZONA CRÍTICA
            global = j;// ZONA CRÍTICA
            semaphore.release();
        }
    }
}
```