Multihilo

Hablamos de **multihilo** cuando se ejecutan varias tareas relacionadas o no entre sí dentro de una misma aplicación. En este caso no son procesos diferentes, sino que dichas tareas se ejecutan dentro del mismo proceso del Sistema Operativo. A cada una de estas tareas se le conoce como hilo o thread (en algunos contextos también como procesos ligeros).

Por tanto, podemos definir los **Hilos** (threads, también llamados hebras):

- Ejecuciones simultáneas dentro del contexto de un mismo proceso (podemos considerarlos como "procesos ligeros")
- Comparten el espacio de memoria del proceso donde han sido creados.

Recursos compartidos por hilos

Cuando creamos un objeto de una clase, puede ocurrir que varios hilos de ejecución accedan al mismo objeto. Es importante recordar que **todos los campos del objeto son compartidos entre todos los hilos**.

Si varios hilos ejecutan sin querer el método incrementar en teoría debería ocurrir que el número se incrementase en este caso 1000 veces por cada hilo. Sin embargo, **es muy probable que eso no ocurra** si el método incrementar no está sincronizado.

Ejemplo (ejecutarlo, con el método incrementar sincronizado y sin sincronizar):

```
class Contador {
    private int cuenta = 0;

    // Método sincronizado para incrementar la cuenta
    public synchronized void incrementar() {
        cuenta++;
    }

    // Método para obtener el valor de la cuenta
    public synchronized int getCuenta() {
        return cuenta;
    }
}
```

```
public class Tarea extends Thread {

	Contador contador;
	public Tarea(Contador contador) {
		this.contador = contador;
	}

	@Override
	public void run() {
	  for (int i = 0; i < 1000; i++) {
	      contador.incrementar();
	  }
	}
}
```

```
public class UsarTarea {

	public static void main(String[] args) throws Exception {

		//recurso compartido
		Contador contador = new Contador();

		// Crear dos hilos que comparten contador y hacen lo mismo
		Tarea hilo1 = new Tarea(contador);
		Tarea hilo2 = new Tarea(contador);

		// Iniciar los hilos
		hilo1.start();
		hilo2.start();

		// Esperar a que ambos hilos terminen
		hilo1.join();
		hilo2.join();

		// Mostrar el valor final del contador
		System.out.println("Cuenta final: " + contador.getCuenta());

	}

}
```

Sincronización de hilos

Cuando existen diferentes hilos accediendo al mismo tiempo a la misma zona del programa, se pueden producir situaciones inesperadas que lleven a errores en el resultado esperado.

El hecho de que varios hilos compartan el mismo espacio de memoria puede causar dos problemas:

- Interferencias
- Inconsistencias

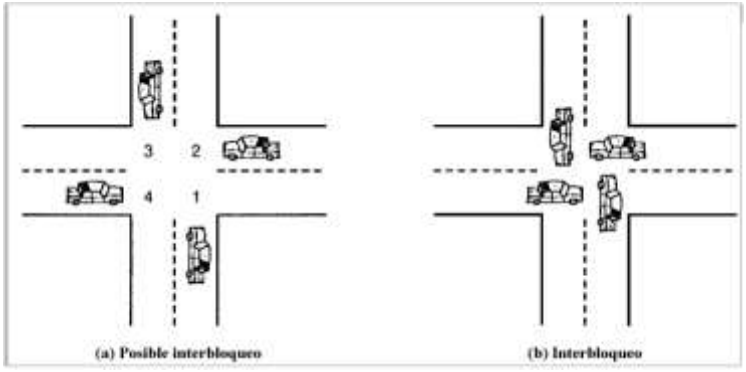
Algunos hilos necesitan esperar a otros para disponer de datos importantes o evitar problemas de acceso simultáneo a recursos. Es necesario proteger y sincronizar el acceso a esas zonas de código.

A estas zonas del código fuente que van a funcionar como recurso compartido, se denominan **secciones críticas**.

Cuando un método acceda a una variable miembro que esté compartida deberemos proteger dicha sección crítica, usando **synchronized**.

La sincronización y gestión de las secciones críticas se tienen que hacer con mucho cuidado ya que se pueden producir errores aleatorios en tiempo de ejecución complicados de detectar:

- Se conoce como **inanición** (starvation) al fenómeno que tiene lugar cuando a un hilo se le deniega continuamente el acceso a un recurso compartido. Esto ocurre porque otros hilos siempre obtienen el recurso compartido antes que él por diferentes motivos.
- Se conoce como **interbloqueo** (deadlock) cuando dos o mas hilos están esperando indefinidamente por un evento que solo puede generar un hilo bloqueado. Este error se puede producir solo en algunas ejecuciones del programa dependiendo del orden específico de ejecución de los hilos.



- Un **bloqueo activo** (livelock) Los hilos se estorban mutuamente por evitar un interbloqueo. En el interbloqueo los hilos estaban detenidos, en el bloqueo activo los hilos no están detenidos, cambian de estado continuamente.

Condición de carrera

Una condición de carrera se puede producir cuando 2 o más hilos acceden (lectura o modificación) a un recurso compartido a la vez.

El comportamiento de un programa con una condición de carrera es imprevisible, son difíciles de detectar, difíciles de reproducir y difíciles de depurar.

Existe una condición de carrera si el resultado de la ejecución del programa depende del orden concreto en que se realicen los accesos a memoria, pudiéndose provocar resultados incorrectos.

En Java puede ocurrir una condición de carrera de dos formas:

- Con referencias al mismo objeto compartidas entre hilos
  - En Java, cualquier objeto o array se pasa por referencia
  - Si se comparte una referencia entre varios hilos, pueden estar accediendo al mismo objeto
- Variables estáticas:
  - Si el código de dos o más hilos accede (lectura o escritura) a la misma variable estática.
  - Si hay dos instancias de la misma clase thread, que accede a la misma variable estática.

Cuando un hilo accede a un recurso compartido, decimos que está accediendo a una **sección crítica**. Debemos tener en cuenta que un mismo hilo puede acceder a mas de una sección crítica.

Es necesario un protocolo para controlar el acceso a la sección crítica:

- Es necesario algún mecanismo para pedir permiso para entrar en la sección crítica
- Es necesario avisar que hemos salido para que otros puedan entrar.

Los mecanismos que pueden evitar que se produzca este inconveniente conocido como condición de carrera son:

- Exclusión mutua (uso de métodos sincronizados)
- Sincronización condicional (uso de wait, notify, nofityAll)

La **exclusión mutua**, es la comunicación requerida entre dos o mas hilos que necesitan acceder a la vez a un recurso compartido para que únicamente acceda uno de los hilos a dicho recurso:

- Asegura que una secuencia de instrucciones sea tratada como una operación indivisible.
- Permite el acceso seguro a una sección crítica.

La **sincronización condicional**, consiste en retrasar un proceso hasta que una de las variables compartidas cumpla las condiciones necesarias para ejecutar la siguiente operación:

- Asegura que se cumplen las condiciones necesarias para poder llevar a cabo una determinada acción.
- En el caso que no se cumplan estas condiciones, se retrasa la acción, por ejemplo, antes de sacar elementos, es necesario que haya elementos.
- No se garantiza que, cuando un hilo despierta, lo hace porque ya se cumple la condición a la que estaba esperando, por lo que la llamada a **wait** se debe hacer en un **while**, en vez de en un **if**

```
while(!condicion) {
try {
     [this.]wait();
} catch (InterruptedException e) {}
}
```

Toda solución concurrente a un problema puede calificarse según las siguientes propiedades:

- safety (seguridad): Asegurar que no ocurre nada malo. Contrarias a la seguridad son las condiciones de carrera.
- liveness (vivacidad): antes o después, ocurre algo bueno. Contrarios a la vivacidad son el interbloqueo [deadlock] y el bloqueo activo (livelock).
- fairness (equidad): los recursos se reparten con justicia, tal que, si hay varios hilos intentando entrar en la sección crítica, estos deberán hacerlo con frecuencias similares. Contrario a la equidad es la inanición [starvation].

Opciones en entornos concurrentes

Java ha creado en su paquete ***java.concurrent*** un conjunto de posibles opciones denominadas **thread-safe**.

Se dice que una clase es **thread-safe** cuando los objetos de dicha clase se pueden utilizar desde varios hilos sin violar el principio de seguridad.

Por ejemplo, a partir de java 1.5 una alternativa al uso de wait y notify es el uso de la clase **Semaphore**, donde *acquire()* funciona de manera similar a wait() y *release()* funciona de manera similar a notify().