

Programación multihilo II

Programación de servicios y procesos

Contenidos:

- 1) Introducción a los problemas de concurrencia.
- 2) Sincronización de hilos. Exclusión mutua. Condiciones de sincronización.
- 3) Problemas. Inanición, interbloqueos.
- 4) Mecanismos de comunicación y sincronización de hilos. Monitores. Synchronized

Introducción a los problemas de concurrencia.

En programación concurrente las dificultades aparecen cuando los distintos hilos acceden a un recurso compartido y limitado. Si en el ejemplo de los ratones (el físico, no el programático), estos comiesen a través de un dispositivo al que solo pudiesen acceder de uno en uno, la concurrencia sería imposible. En el ejemplo programático el problema es el mismo, pero no existe la restricción física, por lo que nada impide que los hilos accedan al recurso compartido y es en ese momento donde aparecerán los problemas.

El acceso a los recursos compartidos limitados, por lo tanto, se debe gestionar correctamente. Por suerte, existen técnicas, clases y librerías que implementan soluciones, por lo que únicamente hay que conocerlas y saberlas utilizar en el momento adecuado.

Un aspecto importante referente a los problemas de concurrencia es que no siempre provocan un error, o el mismo error, en tiempo de ejecución. Esto significa que en sucesivas ejecuciones del programa multihilo el error se producirá en algunas de ellas y en otras no (en la programación secuencial por norma general si repites los pasos, repites los errores).

Los problemas de la programación concurrente se resuelven, en ciertos casos, convirtiendo en secuenciales determinadas partes del código. Si no se programa correctamente se corre el riesgo de convertir todo el programa en secuencial, evitando los problemas de concurrencia pero perdiendo las ventajas que da la programación concurrente.

Sincronización de hilos.

Los threads se comunican principalmente mediante el *intercambio de información a través de variables y objetos* en memoria. Recordamos lo visto en el tema anterior, las interacciones entre procesos concurrentes pueden ser:

- **Independientes:** Sólo interfieren en el uso de la CPU.
 - ✓ No necesitan comunicarse.
 - ✓ No necesitan sincronizarse
- **Cooperantes:** Un proceso genera la información o proporciona un servicio que otro necesita.
 - ✓ Pueden comunicarse entre sí.
 - ✓ Pueden sincronizarse
- **Competidores:** Procesos que necesitan usar los mismos recursos de forma exclusiva

Si trabajamos con procesos pertenecientes al último caso, tendremos que establecer mecanismos de sincronización y comunicación.

Sincronización de hilos.

Antes de empezar a trabajar con este tipo de procesos, recordemos los siguientes conceptos:

- ❑ **Región crítica:** Conjunto de instrucciones en la que el proceso utiliza un recurso y que se deben ejecutar de forma exclusiva con respecto a otros procesos competidores por ese mismo recurso.
- ❑ **Lock, bloqueo:** Para evitar que el recurso anterior sea accedido por múltiples procesos competidores, se bloqueará mientras haya un proceso que lo esté utilizando.
- ❑ **Deadlock, interbloqueo:** Se produce cuando los procesos no pueden obtener nunca los recursos necesarios para terminar su ejecución. Hay que evitar a toda costa darse este tipo de casos.

Sincronización de hilos. Acción atómica

Es una operación que sucede completa sin interrupciones, por lo que ningún otro hilo puede leer o modificar datos relacionados mientras se esté realizando la operación. Cualquier efecto de la acción sólo es visible al finalizar la misma. En Java, algunas acciones sencillas son atómicas:

- **Leer y escribir** variables de tipos primitivos, excepto *long* y *double*.
- **Leer y escribir** variables declaradas ***volatile***, incluidas *long* y *double*.

En sistemas multiprocesador, con múltiples hilos de ejecución, asegurar atomicidad es muy complicado. Cuando varios hilos comparten la misma variable, si esta se almacena en las cachés de los núcleos, puede ser que los hilos vean copias distintas de la misma variable lo que puede provocar *inconsistencias de memoria*.

Declarando una variable como ***volatile*** sólo existirá una copia en el procesador. Esto no resuelve todos los problemas de concurrencia, especialmente si hay varios procesos modificando la misma variable, pero si es útil cuando hay un único hilo que modifica el valor de la variable y varios hilos que sólo la consultan.

Sincronización de hilos. Sección crítica

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:

- ❑ **Exclusión mutua:** si un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su sección crítica.
- ❑ **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.
- ❑ **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.

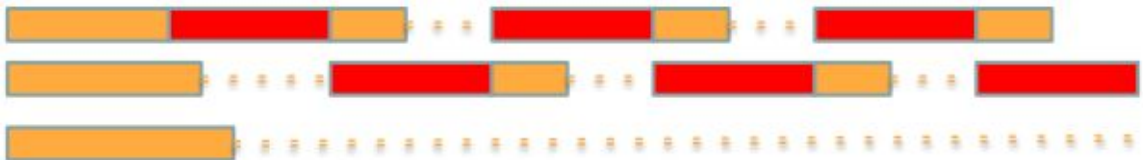
Sincronización de hilos.

Para solucionar estas situaciones, utilizaremos los siguientes mecanismos de interacción entre procesos:

- ❑ **Sincronización:** Un proceso puede conocer el punto de ejecución en el que se encuentra otro en ese determinado instante.
- ❑ **Exclusión mutua:** Mientras que un proceso accede a un recurso, ningún otro proceso accede al mismo recurso o variable compartida.
- ❑ **Sincronización condicional:** Sólo se accede a un recurso cuando se encuentra en un determinado estado interno.

Problemas de sincronización.

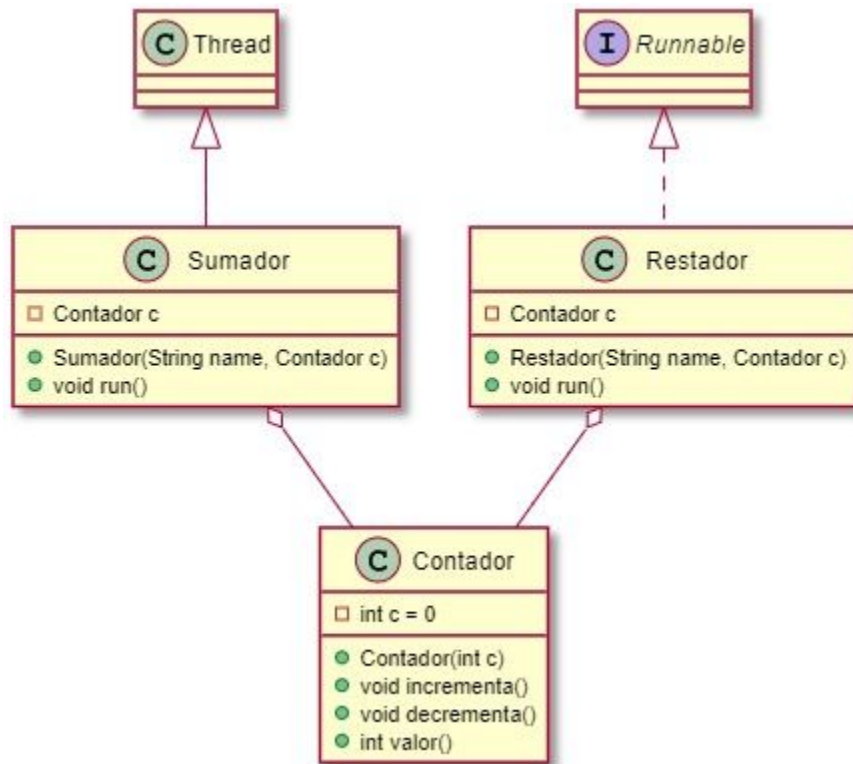
Cuando varios hilos manipulan a la vez objetos compartidos, pueden ocurrir diferentes problemas:

- 1) **CONDICIONES DE CARRERA** : Si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.
- 2) **INCONSISTENCIA DE MEMORIA**: Se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato (Ejemplo 0_Bomba.java de los recursos multihilo II)
- 3) **INANICIÓN**: Cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto siempre toman el control antes que él por diferentes motivos.
- 4) **INTERBLOQUEO**: Se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado.
- 5) **BLOQUE ACTIVO**: Es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro.

Memoria compartida.

A menudo los hilos necesitan comunicarse unos con otros. La forma que tienen de hacerlo consiste en compartir un objeto.

Vamos a desarrollar un ejemplo en el que dos hilos comparten un objeto de la clase Contador.



Para probar el objeto compartido, en una cuarta clase que contiene el main se crea un objeto Contador que se inicializa a 100 y se crean y lanzan dos threads, uno de tipo Sumador y otro de tipo Restador.

En la clase Sumador se usa el método del objeto Contador que incrementa en uno su valor mientras que en la clase Restador se usa el método que decrementa en uno su valor.

Cada una va a realizar la acción 300 veces, esperando entre cada acción un tiempo de 10ms.

Es muy importante asegurarse que pasamos el mismo objeto Contador como parámetro al constructor de Sumador y de Restador, para que ambos trabajen con la misma instancia.

Memoria compartida.

El código anterior no funciona porque las operaciones que se realizan en los métodos incrementa y decrementa no son atómicas, sino que se descomponen en operaciones más simples que se ejecutan una tras otra.

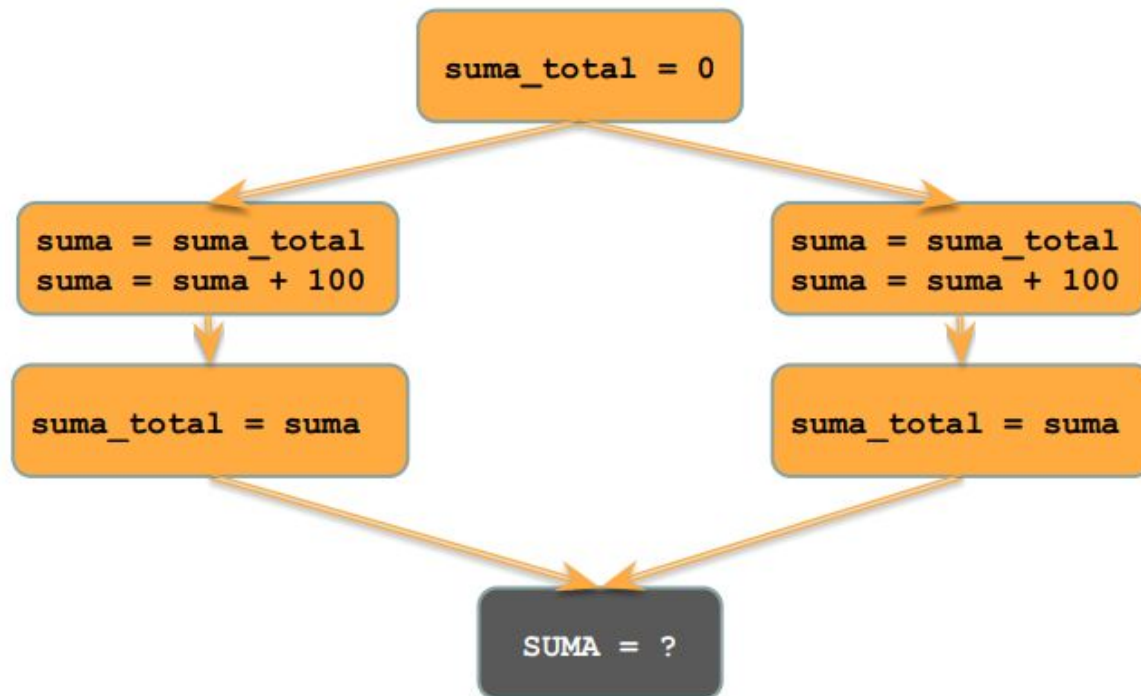
Cuando estas operaciones se ejecutan en un hilo, la ejecución del hilo se puede interrumpir y se pueden intercalar entre ellas operaciones de otros hilos. Según cómo se intercalen las operaciones y los datos a los que accedan, se pueden obtener resultados no esperados. Esto es lo que se conoce como una condición de carrera.

Como vimos en la primera parte del tema, la comunicación entre threads se produce principalmente mediante el acceso compartido a objetos y sus propiedades. Este mecanismo de comunicación es muy eficiente pero presenta dos tipos de errores:

- Interferencia entre threads
- Errores de consistencia de la información en memoria.

Problemas de sincronización.

- 1) **CONDICIONES DE CARRERA** : Se da si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.



El funcionamiento de un proceso y su resultado debe ser independiente de su velocidad relativa de ejecución con respecto a otros procesos. Es necesario garantizar que el orden de ejecución no afecte al resultado.

La forma de proteger las secciones críticas es mediante sincronización. La sincronización se consigue mediante:

- Exclusión mutua. Asegurar que un hilo tiene acceso a la sección crítica de forma exclusiva y por un tiempo finito.
- Por condición. Asegurar que un hilo no progrese hasta que se cumpla una determinada condición.

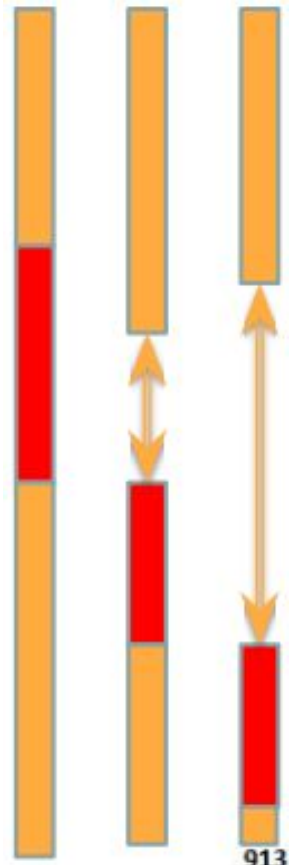
En Java, la sincronización para el acceso a recursos compartidos se basa en el concepto de monitor.

Ejemplo: UT3_CompartirInfo de los recursos multihilo II

Problemas de sincronización. Mutex

Condiciones para la exclusión mutua (mutex):

- Solamente se permite un proceso puede estar simultáneamente en la sección crítica de un recurso.
- No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente.
- Cuando ningún proceso esté en una sección crítica, cualquier proceso que solicite su entrada lo hará sin demora.
- No se puede hacer suposiciones sobre la velocidad relativa de los procesos ni el número de procesadores.
- Un proceso permanece en su sección crítica durante un tiempo finito



913

Sincronización de hilos. “Synchronized”

Uno de los mecanismos proporcionados por Java para sincronizar segmentos de código consiste en utilizar la palabra clave “synchronized”:

- Mediante su uso se puede limitar el acceso a un segmento de código a un único hilo concurrentemente, logrando así la exclusión mutua o mutex.
- Se puede usar a nivel de método o de segmento de código, permitiendo delimitar la sección de código con más precisión.
- Al declararlo establecemos un **monitor a nivel de instancia del objeto**, esto es que para todos los hilos que utilicen la instancia de un objeto, no se permitirá el acceso de forma concurrente al método o bloque de código en el que se haya declarado.
- Si la exclusión que establecemos es muy genérica puede que sea poco eficiente.

Sincronización de hilos. “Synchronized”

Para utilizar una comunicación coordinada entre los diversos threads se utiliza el mecanismo de sincronización:

```
public synchronized void metodo_seccion_critica()  
{  
    // Código sección crítica  
}
```

```
synchronized (Object)  
{  
    // Código sección crítica  
}
```

Ejemplo (en recursos multihilo II):

- UT3_CompartirInfo2: Se sincroniza el objeto contador, con todo lo que contiene
- UT3_CompartirInfo3: Se sincronizan los métodos que incrementan y decrementan la variable

En la programación multihilo, y dentro de la sincronización, un bloque protegido por un monitor nos garantiza que:

- ❑ Cuando un hilo entra en un bloque synchronized se actualizará el valor de todas las variables visibles para el hilo.
- ❑ Cuando un hilo salga de un bloque synchronized todos los cambios realizados por el hilo se actualizarán en la memoria principal.

Ejercicio

Crea un programa Java que lance cinco hilos, cada uno incrementará una variable contador de tipo entero, compartida por todos, 5000 veces y luego saldrá. Comprobar el resultado final de la variable. ¿Se obtiene el resultado correcto?

Ahora sincroniza el acceso a dicha variable. Lanza los hilos primero mediante la clase Thread y luego mediante el interfaz Runnable. Comprueba el resultado.

Sincronización de hilos. “Synchronized”



Un objeto, un monitor

He recalcado en varias ocasiones que los bloques de código a los que se accede en exclusión mutua son aquellos que están protegidos por el mismo monitor. Esto es lo mismo que decir a aquellos que se realizan sobre el mismo objeto. Cada objeto tiene asociado un monitor y la exclusión mutua y la sincronización de memoria tiene sentido si varios threads usan el mismo monitor para su sincronización.

Cada objeto gestiona una cola de hilos que quieren conseguir el bloqueo del mismo. Como suele ser habitual, la elección del proceso de la cola que conseguirá el bloqueo es totalmente indeterminista, depende de múltiples factores y no sigue ningún orden preestablecido.

Ejemplo uso synchronized. Relevos

En el ejercicio 4 que hicimos en Programación Multihilo I, usamos el método join para conseguir que los corredores de relevos esperaran a que el anterior hubiera acabado para entrar.

El método join nos daba una sincronización básica, ya que nos permite que un hilo espere a que otros se completen antes de ejecutarse.

Vamos a modificar ese ejercicio para no tener que usar join, lanzando todos los hilos a la vez y sincronizando el acceso al método que simulaba el relevo de cada corredor.

Y OJO: la sincronización se hace a nivel de objeto. Haremos 2 objetos para que puedan correr los relevos y cada uno sincronizará los hilos que compartan ese objeto.

La clase completa está dentro de recursos multihilo II: UT3_1_1_Relevos_Sync

Ejemplo uso synchronized. Relevos

```
public class CorrerRelevoEquipo {
```

```
    private String equipo;
```

```
    public CorrerRelevoEquipo(String equipo) {  
        this.equipo = equipo;  
    }
```

Generamos una clase donde “correrá” cada equipo, con un constructor y el método sincronizado que simula el relevo del corredor.

```
    public synchronized void carrera(String corredor) {  
        try {  
            Integer numero = (int)(Math.random()*(1050-950+1)+950);  
            System.out.printf(equipo+ ":"+corredor+" comienza su relevo%n") ;  
            for (int i=0; i<10; i++) {  
                Thread.sleep(numero);  
            }  
            Float tiempo = numero.floatValue()/100;  
            System.out.printf(equipo+ ":"+corredor+" ha acabado su relevo - "+  
                "Ha tardado: "+ tiempo + " segundos%n") ;  
        } catch (InterruptedException e) {  
            e.printStackTrace () ;  
        }  
    }  
}
```

Ejemplo uso synchronized. Relevos

```
public class Corredor extends Thread{  
  
    private String corredor;  
    private CorrerRelevoEquipo miEquipoRelevo;  
  
    public Corredor(String corredor, CorrerRelevoEquipo miEquipoRelevo ) {  
  
        super ();  
        this.corredor = corredor;  
        this.miEquipoRelevo = miEquipoRelevo;  
    }  
  
    @Override  
    public void run() {  
  
        this.miEquipoRelevo.carrera(corredor);  
    }  
}
```

En la clase Corredor tenemos el constructor, donde además del corredor recibimos el OBJETO relevoEquipo que debe ser único para todos los hilos que lo compartan de forma que “funcione” la sincronización.

Ejemplo uso synchronized. Relevos

```
public class Principal {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        CorrerRelevoEquipo miRelevoEspaña = new CorrerRelevoEquipo("España");  
        CorrerRelevoEquipo miRelevoEEUU = new CorrerRelevoEquipo("EEUU");  
        Corredor hiloEsp[] = new Corredor[4];  
        Corredor hiloEEUU[] = new Corredor[4];  
  
        String Equipo[] = {"Pepe", "Maria", "Juan", "Marta"};  
        String Equipo2[] = {"John", "Kate", "Larry", "Sarah"};  
  
        //Creamos objetos en cada posicion  
        for(int i=0; i<Equipo.length; i++) {  
            hiloEsp[i] = new Corredor(Equipo[i], miRelevoEspaña);  
            hiloEsp[i].start();  
            hiloEEUU[i] = new Corredor(Equipo2[i], miRelevoEEUU);  
            hiloEEUU[i].start();  
            try {  
                hilo[i].join();  
            } catch (InterruptedException e) { }  
        }  
  
        System.out.println("FINAL DE PROGRAMA");  
    }  
}
```

En el main, instanciamos el objeto RelevoEquipo una vez para cada equipo que va a correr y arrancamos los 4 hilos de los 4 corredores para cada equipo y reciben la clase común con la que sincronizaremos los hilos.

Comentamos el join porque ya no necesitamos que los hilos “esperen”... de hecho, arrancan los hilos y se ejecuta la sentencia de final de programa mientras los hilos se van “relevando” aunque el orden de los corredores no se puede determinar (podemos probar a poner prioridades a los hilos a ver si hay suerte)...

Sincronización de hilos. “Synchronized”

La sincronización no tiene porqué realizarse sobre todo un método. A veces es preferible sincronizar sólo una parte de un método. Otras no es posible sincronizar el método completo. Para sincronizar un bloque de código usamos la palabra reservada `synchronized` seguida, entre paréntesis, del objeto del que usaremos el monitor. El código protegido se ubicará entre un par de llaves.

```
1 public void add(int value){  
2     synchronized(this){  
3         this.count += value;  
4     }  
5 }
```

java

En el ejemplo se ha utilizado “this” como objeto para el monitor. Decimos que el código está sincronizado con el monitor del objeto que pongamos entre paréntesis.

Sincronización de hilos. “Synchronized”

Un método sincronizado usa el objeto al que pertenece como monitor, es decir, también usa this.

```
1  public class MyClass {
2      public synchronized void log1(String msg1, String msg2){
3          log.writeln(msg1);
4          log.writeln(msg2);
5      }
6
7      public void log2(String msg1, String msg2){
8          synchronized(this){
9              log.writeln(msg1);
10             log.writeln(msg2);
11         }
12     }
13 }
```

java

En términos de sincronización, ambos bloques son totalmente equivalentes.

Sincronización de hilos. “Synchronized”



¿Qué objetos se pueden usar como monitores

Oracle dice que se puede usar cualquier objeto como monitor de sincronización, sin embargo recomiendan que no se sincronice sobre **String**, o cualquier objeto envoltorio (wrapper) de los tipos de datos primitivos (Integer, Double, Boolean, ...).

Para estar seguros, lo mejor es sincronizar sobre `**this**`, sobre una instancia de un objeto o, en su defecto sobre un nuevo objeto de tipo `object`, aunque sea un objeto vacío sin propiedades ni funcionalidad.

Sincronización de hilos. “Synchronized”



Usar final con objetos de tipo monitor

Un objeto usado como monitor, o como memoria compartida entre hilos, debería ser de tipo **final**, porque si se le asigna un nuevo valor quedan sin efecto todos los bloqueos que existan sobre dicho objeto. Un objeto de tipo final una vez que se ha creado y se le ha asignado un valor no se le puede asignar un nuevo valor.