Programación Concurrente
Programación Concurrente Multihilos - Java
Parte I

Programación Orientada a Objetos Concurrente

"hacer mas de una cosa a la vez"

Propiedades de concurrencia

Hay que considerar dos items de correctitud/ propiedades de concurrencia:
Seguridad (safety): Se refiere a comportamiento inesperado en tiempo de ejecución (atributos y restricciones, restricciones de representación). La seguridad en multihilos tiene la particularidad de que la mayoría de los puntos a tener en cuenta no pueden ser chequeados automáticamente y esta muy ligado a la disciplina del programador. Se agrega una dimensión temporal a las técnicas de diseño y programación.
El control de concurrencia introduce la desabilitación del acceso de forma pasajera/temporal, basándose en consideraciones de las acciones que están siendo ejecutadas por otros hilos, asegurar que los objetos en el sistema mantienen estados consistentes, en los cuales todos los campos/variables-instancia propias y las de los objetos de los que depende posean valores legales y significativos. Cada método público en cada clase debe llevar al objeto de un estado consistente a otro. Para asegurar consistencia se emplean las técnicas de ***exclusión mutua***, garantizando la atomicidad de las acciones públicas.
Ejemplos: 1) el balance de una cuenta bancaria es incorrecto despues de un intento por retirar dinero en el medio de una transferencia automática. 2) seguir el puntero next de una lista vinculada puede llevarnos a un nodo que aún no este en la lista. 3) dos actualizaciones concurrentes a un sensor causa que un controlador de tiempo real lleve a cabo una acción de efecto incorrecta.

Viveza (liveness): "Eventualmente algo ocurrirá en una actividad", es decir progresará. En un sistema "vivo" cada actividad eventualmente progresa hasta completarse, cada método invocado eventualmente se ejecuta. Sin embargo una actividad puede fallar en progresar por varias razones: bloqueo (un método sincronizado bloquea a un hilo mientras otro hilo mantiene su lock), espera (un método se bloquea esperando por un evento, un mensaje o una condición que todavia tiene que producirse), entrada (un método basado en entrada/salida se bloquea esperando por una entrada que aun no ha llegado de otro proceso o dispositivo), contención de CPU (un hilo no logra obtener tiempo de CPU aunque esta en condiciones de ejecutarse), falla (se produce una excepción, error o falta).

Fallas de progreso permanente:

a) deadlock, dependencias circulares entre locks,
b) señales perdidas, un hilo permanece dormido porque empezó a esperar después de que una notificación se produjo,
c) cierre de monitores anidados, un hilo en espera mantiene un lock que otro hilo que debe despertarlo esta esperando,
d) livelock, una acción que se intenta continuamente, contínuamente falla,

e) starvation, la máquina virtual falla siempre en asignar tiempo de CPU a un hilo.
f) agotamiento de recursos, un grupo de hilos mantienen todos los recursos,
g) falla distribuida, una máquina remota conectada por un socket se hace inaccesible o se rompe.

Ademas debe tenerse en cuenta la Performance: se requiere que la ejecución sea pronto y rápida. El soporte de concurrencia introduce cierta sobrecarga: locks, monitores, hilos, cambio de contexto, planificación, localidad, algoritmia.

Exclusion Mutua

Las situaciones que pueden darse en programación concurrente son:

- los procesos o hilos no se conocen, y no comparten recursos
- los hilos no se conocen pero comparten recursos
- los hilos se conocen y comparten recursos, y cooperan mediante el intercambio de información.

Cuando dos hilos o procesos acceden a un recurso compartido pueden generarse problemas, por lo tanto es necesario utilizar algún mecanismo para asegurar que los datos compartidos esten en un estado consistente antes de que cualquier hilo comience a usarlos para una tarea particular. Para ello una de las soluciones es la ***exclusión mutua***, que garantiza el cumplimiento permanente de los invariantes del objeto, es decir que el objeto se mantiene en estados consistentes. Solamente mientras se este en la etapa de modificación del objeto (por ejemplo por el incremento de un atributo), este puede encontrarse en un estado de inconsistencia.
Los recursos que quieren ser accedidos por varios hilos o procesos son ***recursos críticos,*** y el sector de programa que desea usarlos es la ***sección crítica***.

Para asegurar la exclusión mutua:

- solo un proceso debe poder acceder a la vez a la sección crítica
- un proceso detenido fuera de su sección crítica no debe afectar a los demás procesos
- un proceso no puede ser retrasado indefinidamente para entrar en su sección crítica
- si ningún proceso está en su sección crítica, el que quiera entrar debe poder hacerlo sin demoras
- un proceso no puede permanecer indefinidamente en su sección crítica
- **no se debe asumir nada respecto a la cantidad de procesos ni a la velocidad relativa de los mismos.**

Si nos contextualizamos en orientación a objetos, si un hilo esta intentando acceder a los atributos de un objeto se debe lograr que no haya mas de un hilo accediendo a los atributos del mismo objeto a la vez. Además utilizando correctamente los conceptos de orientación a objetos, respecto al ocultamiento de la información y encapsulación, el acceso se realizará por medio de métodos de acceso, así lo que se debe controlar es que los métodos no sean accedidos simultáneamente. Para ello una gran mayoría de los lenguajes propone trabajar de forma sincronizada, es decir métodos que garantizan que si un hilo penetra en uno de ellos, ningún otro hilo lo hará, en el mismo método o en otro método sincronizado para el mismo objeto. Como clientes de una clase, para utilizarla en un contexto concurrente, se debe asegurar que tenga implementada la exclusión mutua. Y si no es así, se debe trabajar sobre una subclase propia.

En un sistema seguro cada objeto se protege a si mismo de violaciones de integridad. Esto requiere la cooperación de otros objetos y sus métodos. Las "técnicas de exclusión" preservan la consistencia de los objetos y evitan los efectos que resultarían de actuar sobre estados inconsistentes. Las técnicas de programación logran la exclusión evitando que múltiples hilos modifiquen o actúen sobre las representaciones de objetos.

Se requieren algunas técnicas para compartir y publicar objetos de forma que puedan ser accedidos por multiples hilos de forma segura. La sincronización no se trata solamente de atomicidad y demarcación de secciones críticas, tambien tiene que ver con visibilidad de memoria (aspecto fundamental). Se desea prevenir que un hilo modifique el estado de un objeto mientras otro lo esta utilizando, pero además asegurar que cuando un hilo modifica el estado de un objeto, los otros hilos ven el cambio realizado. Sin sincronización esto no sucede, no se puede asegurar.

Las estrategias básicas son: eliminar la necesidad del control de exclusión al no permitir la modificación de la representación de objetos; asegurar dinámicamente que sólo un hilo por vez pueda acceder al estado del objeto, utilizando locks y construcciones relacionadas; y asegurar en la estructura que solamente un hilo puede utilizar un objeto dado. El uso de estas técnicas es fundamental y una diferencia importante entre la programación secuencial y la programación concurrente

Ejemplo: considere 2 hilos generadores de numeros, que generan numeros aleatorios, y cuentan la cantidad de numeros generados que verifican cierta condicion, por ejemplo estar entre 5 y 6. Se desea saber cual es el total de numeros logrados entre los 2 hilos.
Lo primero que se nos podría ocurrir es tener una clase *Contador* y que los hilos compartan la instancia de *Contador.* La clase Contador debe tener métodos para setear su valor, obtener su valor, incrementar su valor en 1 e incrementar su valor en n.

En Java sería

```
public class Contador {
        int valor;
        public Contador (){
                valor = 0;
        }

        public int getValor(){
                return valor;
        }

        public void setValor(int nro){
                valor = nro;
        }

        public void incrementar(){
                valor = valor + 1;
```

```
	}

	public void incrementar (int incremento){
		valor = valor + incremento;
	}
}
```

Y la clase que utilizamos como Runnable para luego crear los hilos generadores es:

```
public class GeneradorNros implements Runnable{
	private String nombre;
	private Contador cuenta;
	private int cantidad;

	public GeneradorNros(Contador cont, String cadena, int maximo){
		cuenta = cont;
		nombre = cadena;
		cantidad = maximo;
	}

	public void run(){
		int totalNros = 0;
		double nroAleatorio;
		for (int i=1; i< cantidad; i++) {
			nroAleatorio = Math.random() * 10;
			if (nroAleatorio > 5 && nroAleatorio < 6){
				totalNros ++;
			}
		}
		cuenta.incrementar(totalNros);
		System.out.println ("hilo " + nombre + " genero " + totalNros );
	}
}
```

Y finalmente consideramos la clase TestGenerador:

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

		Contador elContador = new Contador();
		GeneradorNros genera1, genera2;
		Thread hilo1, hilo2;

		genera1 = new GeneradorNros(elContador, "soyGen_1 ", 100 );
```

```
                hilo1 = new Thread(genera1);
                genera2 = new GeneradorNros(elContador, "soyGen_2 ", 130 );
                hilo2 = new Thread(genera2);

                hilo1.start();
                hilo2.start();

                try {
                        hilo1.join();
                        hilo2.join();
                } catch (InterruptedException e){...}

                System.out.println( "total de numeros generados por los hilos "
                                        + elContador.getValor());
        }
}
```

Al ejecutarlo puede obtenerse un resultado inesperado. Cada hilo al ejecutar el método *run* genera números aleatorios y cuenta cuantos de los números generados respetan la característica de estar entre 5 y 6, utilizando la variable local al método, *totalNros*. A continuación cada hilo ejecuta el método *incrementar* sobre el objeto compartido *elContador*, y es en esta ejecución donde pueden producirse las inconsistencias, dado que la operación valor = valor + incremento no se realiza de forma atómica. Puede darse que el hilo corriente (hilo1) obtenga el valor, y lo sume a incremento pero antes de guardar el resultado nuevamente en valor pierda el tiempo de CPU. Al otro hilo (hilo2) le toca el turno para ejecutarse y logra realizar completamente la operación de incremento. En este escenario cuando el hilo 1 retoma su ejecución y logra completar la operación de incremento que habia quedado inconclusa, modifica valor y se pierde la modificación realizada por el hilo 2, es decir el hilo 1 no vería el cambio realizado por el hilo 2.

Sin embargo puede ocurrir que se ejecute el test repetidas veces y no se produzca la situación planteada. pero… esto no asegura que nunca ocurra.

Para garantizar la seguridad en un sistema concurrente se debe asegurar que todos los objetos accesibles desde muchos hilos son INMUTABLES o emplean SINCRONIZACIÓN apropiada; y también se debe asegurar que ningun objeto se hará accesible de forma concurrente por escaparse fuera de su dominio de propiedad:

- Inmutabilidad: Si un objeto no puede cambiar su estado nunca puede encontrarse en situación de inconsistencia o conflicto cuando multiples actividades intentan cambiar su estado en formas incompatibles. El uso selectivo de inmutabilidad es fundamental en concurrencia. Los objetos inmutables mas sencillos no poseen campos internos, y sus métodos son intrínsecamente sin estado. Tambien son inmutables las clases con sus atributos final.
- Sincronización: Los bloqueos protegen contra conflictos de almacenamiento de bajo nivel y las fallas de invariante de alto nivel. Las violaciones de seguridad pueden ser raras y

difíciles de testear, pero pueden tener efectos devastadores. Son importantes las prácticas de diseño conservativas que se utilizan en programas concurrentes confiables. El uso de bloqueos serializa la ejecución de métodos sincronizados.

Sincronización

Cuando dos o mas hilos necesitan utilizar el mismo recurso se da la posibilidad de que los operaciones se entrelacen provocando inconsistencias en el recurso compartido. Los hilos compitan por el acceso a eso recurso. En este caso hablamos de "sincronizacion por competencia". Si la sincronización se hace de forma correcta, los hilos no realizarán acciones que puedan generar inconsistencias en los datos

Bloqueos intrínsecos (implícitos)

Java provee un mecanismo incorporado para reafirmar la atomicidad y la exclusion mutua: "Synchronized". La propuesta es trabajar con una combinación de la palabra clave synchronized y la llave de bloqueo o lock implícito. De esta manera se permite el acceso exclusivo al código que afecta a los datos compartidos. Todos los métodos que acceden a los datos compartidos deben sincronizarse sobre la misma llave de bloqueo o lock. Todo objeto tiene un lock, heredado de la clase Object.
Hay 2 opciones: trabajar con métodos sincronizados y con bloques sincronizados. El mecanismo de sincronización funciona solo si todos los aceesos a los datos compartidos y delicados ocurren dentro de métodos o bloques sincronizados. Todos los datos delicados deben ser tratados con buenas prácticas de orientación a objetos, es decir deben ser privados, porque en caso contrario pueden generarse problemas.

Un *bloque sincronizado* tiene 2 partes:

- una referencia a un objeto que servirá como lock (llave), en muchas ocasiones es el objeto corriente (this), indicado explícitamente.
- un bloque de código que será guardado por el lock

Un *método sincronizado* es como un bloque sincronizado que toma TODO el codigo del método y cuyo lock (llave) corresponde al objeto sobre el cual se ejecuta el método. Los métodos estáticos o de clase utilizan la clase como objeto llave.

Cada objeto Java puede implícitamente actuar como un lock para propósitos de sincronización. Este tipo de locks son locks intrínsecos o locks de monitores.
El lock (o llave de bloqueo) es adquirido automáticamente por el hilo que se está ejecutando ANTES de entrar a un bloque/método sincronizado, y es automáticamente liberado cuando el control sale del bloque/método sincronizado, ya sea de forma normal o por una excepción. La UNICA forma de adquirir un lock intrínseco es entrando a un bloque sincronizado o a un método sincronizado guardado (o protegido) por el lock.

Los locks intrínsecos en Java son locks de exclusión mutua, que significa que como máximo un hilo puede tener el lock por vez. Cuando el hilo A intenta adquirir un lock mantenido por el hilo

B, A debe esperar o bloquearse haste que B libere el lock. Si B nunca libera el lock, A esperará por siempre...

Dado que SOLO un hilo por vez puede ejecutar un bloque/método guardado por la sincronización sobre un mismo lock, ese bloque o método se ejecuta de "forma atómica" con respecto a los otros bloques/métodos guardados por el mismo lock. **En el contexto de concurrencia la atomicidad significa que un conjunto de sentencias se ven como si se ejecutaran como una unidad indivisible.**

Volviendo al ejemplo de los hilos generadores de números, para asegurar que los hilos actúen sobre el contador siempre en un estado consistente, hay que hacer que la clase Contador sea ***Thread-safe***, es decir ***segura para ser utilizada en un contexto concurrente***. Para lograrlo se puede utilizar sincronización, así las operaciones en secciones críticas se ejecutarán en exclusión mutua y como si fueran atómicas.
Ahora el método *incrementar*() es sincronizado, lo que significa que cuando un hilo lo ejecuta tiene *exclusión mutua* sobre el código ya que es propietario del lock del objeto contador hasta que termine la ejecución del método, aun cuando se le quitara le CPU antes de terminar dicha ejecución. En el método *incrementar(...)* con parámetros se utiliza bloque sincronizado sobre el objeto this, que en este caso se refiere al contador, o sea que tiene el mismo efecto que al sincronizar el método.

```
public synchronized void incrementar(){
        valor = valor + 1;
}

public void incrementar (int incremento){
        synchronized (this) {
                valor = valor + incremento;
        }
}
```

Al utilizar un método sincronizado el lock que se adquiere es el que corresponde al objeto actual sobre el que se está ejecutando el método. La llave de bloqueo se adquiere al comenzar la ejecución del método y se libera al terminar la ejecución del mismo.
Por el contrario un bloque sincronizado permite adquirir la llave de bloqueo sobre cualquier objeto, incluso el objeto actual. La llave de bloqueo se adquiere al comenzar a ejecutar las sentencias del bloque y se libera al final del bloque.
En ambos casos la adquisición y liberación de la llave de bloqueo es implicita.

La decisión entre utilizar métodos sincronizados o bloques sincronizados tiene relación con el tamaño del código que es necesario sincronizar. Siempre hay que tener en cuenta la regla general de la programación concurrente: "se deben mantener los bloqueos tan poco tiempo como sea posible" . Entonces si se puede lograr que los métodos sean pequeños y solo tengan en su código una sección crítica no hay problemas con sincronizar el método, pero si el método ademas

realizara otras tareas no críticas, será mas apropiado sincronizar por bloques.

Mecánica básica de sincronización en Java:

- cada objeto tiene una bandera asociada con él (llave de bloqueo o lock implícito)
- se pueden sincronizar métodos y bloques
- el bloque tiene un argumento que indica sobre que objeto bloquear
- cualquier método puede bloquear sobre cualquier objeto la sincronización no se hereda
- los bloqueos obedecen a un protocolo adquirir/liberar
- los locks operan en base "por hilo", no por invocación.
- sincronizado NO es equivalente a atómico, pero puede utilizarse con esa intención. El acceso a los métodos que no están sincronizados no es afectado.
- no se garantiza justicia respecto a que hilo será beneficiado con la adquisición del lock.
- el bloqueo de un objeto NO protege el acceso a variables estáticas. Para ello hay que sincronizar utilizando el lock que posee el objeto Class asociado con la clase en la que se definen los métodos estáticos (evitar la sincronización sobre objetos Class)

Objetos completamente sincronizados

- todos los métodos están sincronizados
- no hay campos publicos ni violaciones de encapsulación
- todos los campos son inicializados en un estado consistente en el constructor
- **el estado del objeto es consistente al inicio y fin de cada método, aun con excepciones.**

Cuando la sincronización es utilizada de forma apropiada, todos los cambios realizados en un método o bloque sincronizado son atómicos y visibles con respecto a otros métodos/bloques sincronizados sobre el mismo lock.

Modelo de memoria de Java

En un lenguaje secuencial no debe importar el orden en que compiladores, sistemas de tiempo de ejecución, y hardware puedan reordenar las instrucciones para lograr optimizaciones, dado que la ejecución del programa obedece a una semántica "as-if-serial". Un programa secuencial no puede depender de los detalles internos del procesamiento de sentencias dentro de bloques de codigo simple. En la programación concurrente, en cambio, las ejecuciones además de ser entrelazadas, pueden ser reordenadas y manipuladas con propósitos de optimización. Se pueden obtener resultados inesperados si no se tiene sumo cuidado al trabajar con concurrencia.
El modelo de memoria define una relación abstracta entre los hilos y la memoria. Cada hilo es definido teniendo una memoria de trabajo en la cual almacena valores.

Bloqueos transitorios y bloqueos problemáticos

En general, todo sistema debe progresar, de modo tal de terminar en un tiempo finito. Sin

embargo es posible que transitoriamente un hilo/proceso quede detenido.
Los hilos pueden bloquearse por varios razones: esperar E/S, esperar para adquirir un lock, esperar ser despertado de un sleep, esperar el transcurso del tiempo indicado, o esperar por el resultado de una computación. Cuando un hilo se bloquea es suspendido y puesto en un estado BLOCKED, WAITING o TIMED-WAITING.

En ocasiones los hilos pueden quedar total y permanentemente bloqueados:

- señales perdidas: ocurre cuando un hilo recibe la notificación para despertarse antes de quedar inactivo y luego no la recibe mas.
- bloqueo anidado: cuando un hilo espera que se libere el acceso a un recurso a la vez que éste está siendo esperado por otro objeto que necesita antes despertarlo.
- recursos insuficientes: cuando un grupo de hilos se apropia de un número finito de recursos sin cederlos a nadie mas.
- Starvation/inanición: cuando un proceso se queda esperando indefinidamente un recurso, debido a las políticas de planificación del sistema operativo, por asignación de una prioridad muy baja o por negación de servicio.
- Interbloqueo pasivo/deadlock: se da en un escenario de exclusión mutua, con 2 o mas recursos críticos, cuando un proceso o hilo está bloqueando un recurso y esperando para reservar otro, mientras hay otro hilo bloqueando el segundo recurso y esperando el primero. Como el sistema no puede quitar un recurso a un proceso que lo tiene reservado, estamos en un escenario de espera circular.

Reentrada (reentrancy)
Los locks operan en una base de por-hilo, NO por invocación.
Cuando un hilo requiere un lock que ya lo tiene otro hilo, el requerimiento se bloquea. Pero, debido a que los locks intrínsecos son reentrantes, si un hilo trata de adquirir un lock que ya tiene, el requerimiento tiene éxito. El comportamiento reentrante permite que un método sincronizado realice una autollamada a otro método sincronizado sobre el mismo objeto sin bloquearse.

Las acciones compuestas sobre datos compartidos, tales como incrementar un contador deben ser atómicas para evitar las condiciones de carrera. Sin embargo, hacer un conjunto de operaciones atómicas utilizando un bloque sincronizado no es suficiente; si la sincronización es utilizada para coordinar el acceso a una variable, es necesaria en cada lugar donde se accede a esa variable, es decir en cada sección crítica. Además, cuando se usan locks para coordinar el acceso a una variable, el mismo lock debe ser usado cada vez que la variable es accedida.

Entonces, para cada variable de estado mutable (es decir que pueda cambiar su estado) que pueda ser accedida por mas de un hilo, TODOS los accesos a la variable deben llevarse a cabo manteniendo el mismo lock (o llave de bloqueo), y decimos que **la variable es guardada por ese lock**. Cuando una variable es guardada por un lock, cada acceso a esa variable se realiza manteniendo el lock, y se asegura que solamente un hilo por vez puede acceder la variable.

¿Cómo diseñar la sincronización?

Diseñar la sincronización apropiada es difícil. Podriamos pensar en una relación cliente-servidor donde el servidor es el objeto compartido, y los clientes los que utilizan ese recurso compartido. En este escenario se puede dar la responsabilidad de la sincronización al cliente o al servidor.
Si le damos la responsabilidad al cliente, es necesario que TODOS los clientes que utilicen ese recurso compartido se ocupen de considerar las sentencias necesarias para lograr una buenea sincronización, y para que funcione debemos confiar en que todos los clientes de ese recurso se comportaran de forma correcta.
Si le damos la responsabilidad al servidor, o sea al recurso compartido, logramos que el recurso se proteja a si mismo de inconsistencias. Lo puede hacer con métodos o bloques sincronizados (luego agregaremos mas opciones). Esto impide que algun cliente utilice el objeto sin sincronización, es decir independiza al cliente de la responsabilidad de contemplar la sincronización.

Objetos sincronizadores

Existen situaciones en las que trabajar con bloques sincronizados permite mejorar el diseño y el desempeño al aumentar el nivel de concurrencia.
Por ejemplo, si una clase tiene varios métodos que requieren exclusión mutua pero actuan sobre distinto grupo de variables, utilizar métodos sincronizados no sería la mejor opcion.

```
Clase Muestra
        var uno
        var dos
        var tres
        var cuatro

Metodo sincronizado unoDosPrimero ()     // actua sobre uno y dos
        ...

Metodo sincronizado unoDosSegundo ()   // actua sobre uno y dos
        ...

Metodo sincronizado tresCuatroPrimero ()   // actua sobre tres y cuatro
        ...

Metodo sincronizado tresCuatroSegundo ()   // actua sobre tres y cuatro
        ...

Metodo sincronizado unoDosTercero ()   // acta sobre uno y dos
        ...
```

Si una instancia de la clase Muestra es compartida por el hilo1, hilo2 y hilo3, cada vez que un hilo ejecute alguno de estos métodos, tendrá la llave en su poder hasta que termine esa ejecución, y mientras tanto los otros hilos no podrán acceder a ninguno de los métodos, ya que para ello

necesitan la llave. En esto caso, considerando que hay 3 métodos que actuan sobre las variables uno y dos solamente, y los otros 2 métodos actuan sobre las variables tres y cuatro solamente, se podría mejorar la sincronización trabajando con bloques sincronizados. Esos bloques se sincronizarian sobre objetos especialmente creados para lograr esa sincronización.

… continuará

Bibliografia

• **Concurrent Programming in JAVA: design principles and patterns** - Doug Lea
• **Thinking in JAVA** - Bruce Eckel - President, MindView Inc. - 2008
• **Seven Concurrency Models in Seven Weeks**. When Threads Unravel - Paul Butcher - The Pragmatic Programmers Inc. - 2014
• **Java Concurrency in Practice** - Brian Goetz - et all - Addison Wesley 2006.
• **Documentacion Oracle**: http://docs.oracle.com/javase/tutorial/essential/concurrency/ y el paquete java.util.concurrent.
• **Orientación a Objetos con Java y UML** - Carlos Fontella - nueva libreria nl 2004