Programación Concurrente – 2do. C 2015
Programación Concurrente Multihilos - Java
Parte I

Programación Orientada a objetos concurrente

"hacer mas de una cosa a la vez"

Cuande se habla de concurrencia en programación, se habla de la técnica de hacer que un programa haga mas de una cosa a la vez.

Si bien esquemas mas distribuidos generan una mejor distribución del trabajo, también provocan mayor necesidad de comunicación.

Concurrencia Vs paralelismo

Un sistema se dice *concurrente* si puede soportar 2 o mas actividades *en progreso* a la vez. Un programa concurrente tiene múltiples hilos de control lógicos que pueden o no ejecutarse en paralelo.
Un sistema se dice *paralelo* si puede soportar 2 o mas actividades *ejecutándose simultaneamente*. Un programa paralelo puede ejecutarse mas rápidamente que un programa secuencial ejecutando diferentes partes simultáneamente. Puede o no tener mas de un hilo de control lógico.

Se podría pensar que la "concurrencia" es un aspecto relacionado al dominio del problema y el "paralelismo" es un aspecto del dominio de la solución. En un caso se necesita manejar situaciones eventos simultáneamente, y en el otro se quiere hacer un programa mas rápido procesando diferentes porciones del problema en paralelo.

> *Concurrencia no es paralelismo, es mejor. ... Concurrencia es programar como la composición de procesos que se ejecutan de forma independiente. Paralelismo es programar como la ejecución simultánea de computaciones, posiblemente relacionadas.*
> *Concurrencia es sobre tratar con muchas cosas a la vez*
> *Paralelismo es sobre hacer muchas cosas a la vez.*
> *Concurrencia es sobre estructura, y paralelismo es sobre ejecución.*
> *La concurrencia provee una forma de estructurar una solución para resolver un problema que puede ser paralelizable (pero no necesariamente lo es).*
> *La comunicación se usa para coordinar las ejecuciones independientes.*

Rob Pike, http://concur.rspace.googlecode.com/concur.html ver slides14..25

Consideremos la siguiente situación:

> Un docente, un grupo de alumnos, una clase. El docente es un "master en multitarea", en cada instante esta realizando una única tarea pero tiene que tratar con varias de forma concurrente: mientras espera que el proyector reconozca a la notebook, escribe un ejercicio en el pizarrón, contesta a la pregunta de un alumno, registra que otro alumno la esta llamando, etc. Esto es concurrente, pero no es paralelo. En caso de que haya 2 auxiliares en la clase entonces sí seria además de concurrente, paralelo.

Una aplicación concurrente puede tener 2 o mas hilos de ejecución en progreso al mismo tiempo. Esto puede significar que la aplicación tiene 2 hilos que estan siendo intercambiados in/out por el sistema operativo en un procesador uni-core. Estos hilos estarán en progreso en el medio de su ejecución. En cambio si pensamos en la ejecución en paralelo debe haber multiples cores disponibles en la plataforma. Asi a cada hilo se le puede asignar un core separado y se ejecutarán efectivamente de forma simultánea.

Escribir programas correctos es dificil, escribir programas concurrentes correctos es mas dificil.

El modelo de programación secuencial es intuitivo y natural, modela la forma en que trabajan los humanos: hacer una cosa por vez, mayormente en secuencia. Levantarse de la cama, ponerse la bata de baño, bajar la escalera, tomar el te. Cada una de estas acciones del mundo real es una abstracción para una secuencia de acciones mas finas (abrir la caja de te, seleccionar un saquito de te, poner el saquito de te en la taza, ver si hay suficiente agua en la pava, encender la cocina, poner la pava con agua a hervir, etc.) Mientras se espera que el agua hierva (o este lo suficientemente caliente) se puede hacer algo mas: solo esperar, o hacer otras tareas como preparar una tostada, exprimir naranjas para hacer jugo, encender la computadora para leer el diario, encender el televisor para ver algo, ... mientras se mantiene la atención en la pava que pronto estará lista. Los fabricantes de pavas y tostadoras saben de esta situación, por lo que las producen para que cuando la tarea esta completa (el agua lista o la tostada lista) emitan un sonido para señalar el hecho. Estas tareas envuelven un grado de asincronismo.

Uno de los desafíos de desarrollar programas concurrentes en Java es la discordancia entre las características de concurrencia ofrecidas por la plataforma y como los desarrolladores necesitan pensar acerca de la concurrencia.

Los hilos (procesos livianos) permiten que varias corrientes de flujo de control de programas coexistan dentro de un proceso. Los hilos comparten los recursos del proceso (memoria, etc), pero tienen su propio contador, pila y variables locales.
SI NO EXISTE coordinación explícita entre los hilos, se ejecutan de forma simultánea y asíncrona con respecto a los otros hilos.

Hay tres aspectos de la concurrencia que requieren atención:

- distintas tareas compiten por los recursos (se entiende por recurso todo lo que necesita un proceso para operar: espacio de memoria, dispositivos de E/S, etc). En este sentido se debe garantizar que todos los procesos puedan acceder a los recursos necesarios para seguir procesando en un tiempo finito. Es decir se debe asegurar el progreso de todos los procesos: **liveness**
- respecto a los tipos de métodos, cuando hay concurrencia las operaciones de consulta provenientes de distintas tareas pueden hacerse en cualquier orden sobre el mismo objeto. En cambio, la intervención de operaciones de modificación puede alterar los resultados.
- respecto a los invariantes, puede ser bastante mas difícil de garantizar su cumplimiento cuando varios procesos o hilos pueden acceder simultáneamente a un mismo objeto y modificarlo.

Construir programas concurrentes correctos requiere el uso correcto de hilos y bloqueos. Es necesario proteger los datos de un acceso concurrente descontrolado. Importa COMO es usado el objeto, no lo que hace.

"Cada vez que mas de un hilo accede a una variable de estado dada, y uno de ellos puede escribir en ella, TODOS deben coordinar su acceso a ella utilizando sincronización..."

"Debe evitarse la tentación de pensar que hay situaciones especiales en las cuales la regla no se aplica. Un programa que omite la sincronización necesaria podría parecer que trabaja, pasar las pruebas, y ejecutarse bien durante años, pero ... aún esta roto y puede fallar en cualquier momento"

Condición de carrera

El tipo mas comun es "*chequear y luego actuar"*
Ejemplo: supongamos que planeamos encontrarnos con un amigo, al mediodía en XXX, en la Avenida Argentina, pero cuando usted llega alli descubre que hay 2 XXX en esa avenida, una de cada lado de la calle y no está seguro en cual quedaron en encontrarse. A las 12.10 no ve a su amigo en el XXX A, entonces camina hasta el XXX B para ver si está su amigo, pero tampoco está allí.
Hay algunas posibilidades:
- su amigo se retraso, y no está en ningún XXX todavía,
- su amigo llego al XXX A después que usted se fue
- su amigo estaba en el XXX B pero salió a buscarlo y ahora esta rumbo al XXX A

En el caso de la última opción:
Ahora son las 12.15 y ambos están pensando que hacer, volver a ir al otro XXX, ¿cuántas veces ir y venir? A menos que hayan acordado un protocolo, podrán pasar el resto del día caminando a lo largo de la avenida, frustados y con mucho café encima.

El problema: la frase "Yo cruzaré la calle para ver si él está en el otro XXX" es que mientras usted está caminando, cruzando la calle, su amigo puede haberse movido. Usted observa alrededor en XXX A y piensa "él no está aquí", y se va buscándolo, y hace lo mismo con XXX B... ***pero no al mismo tiempo***. Toma unos minutos cruzar la calle y durante ese tiempo ***el estado del sistema puede haber cambiado***.
Este ejemplo ilustra una condición de carrera, porque alcanzar la salida deseada (encontrarse con su amigo) depende del temporizado relativo de los eventos (cuando usted arriva a XXX A, cuanto espera alli antes de cambiar?). La observación de que “él no esta en XXX A” se torna potencialmente inválida tan pronto como usted sale de XXX A. El podria haber ingresado por otra puerta sin que usted lo sepa.
Esta invalidación de la observación es lo que caracteriza la mayoría de las condiciones de carrera:
- observar algo que sea T (archivo X no existe), y tomar una acción basándose en esa observación (crear X), pero la observación puede ser invalidada entre el tiempo que se hizo la observación, y el tiempo en que se actuó sobre ella (alquien mas creo el archivo X), causando un problema (excepción inesperada, archivo corrupto, datos sobreescritos, etc.).

Otro tipo de condición de carrera: operaciones "*lee-modifica-escribe*", como incrementar un contador, define una transformación del estado del objeto en términos de su estado previo. Para incrementar un contador se debe conocer su valor previo y asegurarse que nadie mas lo cambie o

use mientras uno está en medio de su actualización.

Como la mayoría de los errores de concurrencia, las condiciones de carrera no siempre resultan en fallas, se requiere tambien un poco de malasuerte, pero estas pueden causar problemas graves

Aplicaciones concurrentes: servidores web, cálculos numéricos pesados, procesamiento de E/S, simulación, aplicaciones basadas en GUI, software basado en componentes, código móvil, sistemas embebidos de tiempo real.

Escenarios concurrentes
Se puede tener 2 vistas complementarias de un sistema orientado a objetos: una vista centrada en los objetos (colección de objetos interconectados, que se agrupan formando componentes mayores o subsistemas) y otra vista centrada en las actividades (colección de actividades posiblemente concurrentes, que comprenden mensajes individuales, cadenas de llamadas, secuencias de eventos, tareas, sesiones, transacciones, e hilos). Una actividad lógica puede comprender varios hilos.
Entonces un sistema Orientado a Objetos es "objetos + actividades". En este contexto un objeto puede estar envuelto en múltiples actividades y una actividad puede incluir muchos objetos.

Cuando se produce la comunicación entre objetos se puede identificar un objeto cliente (el emisor del mensaje), y un objeto servidor (el receptor del mensaje), y en algunos casos un objeto agente o intermediario. En un escenario concurrente los objetos clientes son los activos y los servidores son pasivos. Esto es porque un servidor no hace nada a menos que se le indique, permanece a la espera de recibir un mensaje de parte de otro objeto.

Desde el punto de vista de la forma de sincronización los mensajes se pueden clasificar en :

- simple: sólo el objeto que envía el mensaje está activo,
- síncrono: el hilo emisor del mensaje queda esperando la respuesta del receptor,
- esperado: el mensaje desencadena una operación solo si está siendo esperado por el otro objeto,
- cronometrado: síncrono en el cual el hilo emisor da un tiempo máximo al receptor para atenderlo,
- asíncrono: el hilo emisor envía el mensaje y sigue ejecutándose sin importarle que ocurre con el mensaje. El hilo receptor atiende el mensaje cuando puede, y eventualmente envía una notificación o respuesta, asíncrona.

Planificación de hilos

Puede haber varios hilos en estado listo (runnable), es decir listos para su ejecución, pero sólo 1 se encuentra en ejecución. El hilo que se encuentra en ejecución puede cambiar su estado a bloqueado o a listo. El cambio a listo se produce cuando termina su tiempo de CPU. El cambio a bloqueado se produce cuando el hilo es interrumpido por otro hilo con mayor prioridad, y entonces es pasado a estado preemptivo, o el mismo hilo puede voluntaria y explícitamente solicitar que se detenga la ejecución por el período de tiempo indicado.

Los hilos que están es estado listo se mantienen en una colección, organizados segun su prioridad. Cuando un hilo es desbloqueado vuelve a esa colección de listos.
Luego, un hilo durante su tiempo de vida puede estar en uno de los siguientes estados:

- *creado* esperando pasar a estado listo
- *habilitado* para ser ejecutado (listo/runnable)
- *en ejecución*, hasta que termina o es bloqueado.
- *bloqueado,* esperando ser desbloqueado.
- *muerto,* cuando termina su ejecución

Algunas características de hilos en Java

- Un objeto Thread mantiene el control de estas actividades.
- Los hilos pueden compartir el acceso a memoria, archivos abiertos, y otros recursos. Se podria decir que son procesos "livianos".
- un hilo NO puede ser reiniciado.
- las prioridades asignadas (entre 1 y 10) a los hilos afectan al planificador.
- El main tiene prioridad 5. El lenguaje no asegura planificación o justicia, ni siquiera garantizan que los hilos progresen. LA POLITICA EXACTA APLICADA POR EL PLANIFICADOR PUEDE VARIAR SEGUN LA PLATAFORMA, ALGUNAS MAQUINAS VIRTUALES SIEMPRE SELECCIONAN EL HILO CON MAYOR PRIORIDAD CORRIENTE. EXISTEN VARIAS POSIBILIDADES. EL SETEO DE PRIORIDADES NO DEBE SUSTITUIR AL BLOQUEO.
- Para determinar la prioridad actual de un hilo se utiliza el método *getPriority*
- los métodos *sleep, yield, interrupted* y *currentThread* son "estáticos" ya que se aplican sobre el hilo en ejecución.
- El método *yield* tiene sentido efectivo en presencia de un planificador que no es "time-sliced" preemptivo. Coloca al hilo en ejecución (el llamador) en el pool de hilos listos y da lugar para que otro hilo listo se ejecute.
- Al momento de la implementación debemos independizarnos de la política de planificación que sea utilizada por la JVM.
- Se puede utilizar el método *isAlive* para determinar si un hilo todavia esta disponible.

En el ejemplo siguiente se utiliza una bandera para detener al hilo. Notese que esto se logra actuando sobre el "runnable".

```
public class Trabajador implements Runnable{
    private boolean tiempoDeTerminar = false;

    public void run(){
        while (!tiempoDeTerminar){
            System.out.println("hola");
        }
    }

    public void terminarEjecucion(){
        tiempoDeTerminar = true;
        System.out.println(Thread.currentThread() + "termina");
    }

}
```

```
public class Controlador {
    public static void main(String[] arg){

        Trabajador r1 = new Trabajador();
        Thread hilo = new Thread(r1);

        hilo.start();
        try {
            System.out.println ("el main se va a dormir");
            Thread.sleep(2000);
        } catch (InterruptedException e){}

        r1.terminarEjecucion();
        System.out.println ("se fini");
    }
}
```

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 dejar 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.

Exclusión 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. 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 de la cantidad de procesos ni de 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 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 los invariantes de 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 ademas asegurar que cuando un hilo modifica el estado de un objeto, los otros hilos ven el cambio realizado. Sin sincronización esto no sucede.

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 numeros aleatorios y cuenta cuantos de los numeros 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 retome 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 hara 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.

Mecánica básica de sincronización en Java:

- cada objeto tiene una bandera asociada con él (bandera 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.

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.

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.

Bloqueos intrínsecos

Java provee un mecanismo incorporado para reafirmar la atomicidad y la exclusion mutua: el bloque sincronizado. La propuesta es trabajar con una combinación de la palabra clave *synchronized* y la *bandera de bloqueo.* 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 el mismo lock. Todo objeto tiene un lock, heredado de la clase *Object*.
El mecanismo de sincronización funciona solo si todos los aceesos a los datos compartidos y delicados ocurren dentro de 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.
Cuando todo el código del método esta sincronizado, se puede utilizar ***synchronized*** como modificador del método.

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 es adquirido automáticamente por el hilo que se está ejecutando ANTES de entrar a un bloque sincronizado, y es automáticamente liberado cuando el control sale del bloque 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 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 utilizarla 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 elmétodo ***incrementar(...)*** con parámetro 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;
  }
}
```

Bloqueos transitorios y bloqueos patológicos

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/****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 o inicializar un singletón 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. 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 que pueda ser accedida por mas de un hilo, TODOS los accesos a la variable deben llevarse a cabo manteniendo el mismo lock, 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. Cuando una clase tiene invariantes que involucran mas de una variable de estado, hay un requerimiento adicional: cada variable que participa en el invariante debe ser guardada por el mismo lock

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