Sincronización de Hilos

Los hilos se comunican principalmente compartiendo el acceso a campos y objetos referenciados. Esta forma de comunicación es extremadamente eficiente, pero conlleva a dos posibles errores:

- **Interferencia de hilos.**

- **Errores de consistencia de memoria**.

Para prevenir estos errores, necesitamos de la herramienta de sincronización.

Sin embargo, la sincronización puede introducir problemas de **contención de hilos** (e.g. **starvation** y **livelock**), los cuales ocurren cuando 2 o más hilos intentan acceder a un mismo recurso de forma simultánea, ocasionando que el Java runtime ejecute uno o más hilos de forma más lenta o incluso suspenda la ejecución.

Interferencia de Hilos

La interferencia ocurre cuando dos o más operaciones ejecutándose en hilos distintos, pero actuando sobre la misma data, se **entrelazan**. Esto quiere decir que las dos operaciones consistan de múltiples pasos, y las secuencias de estos se sobreponga.

Consideremos la siguiente clase con dos simples operaciones:

```
class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

Notemos que una simple operaciones C++ puede ser descompuesta en 3 pasos:

1. Recuperar el valor de C.
2. Incrementar el valor de C en 1.
3. Almacenar el valor incrementado de nuevo en C.

Evidenciamos la posible interferencia (tomando un valor inicial de C igual a 0) con la siguiente secuencia:

1. Hilo A: Recupera C.
2. Hilo B: Recupera C.
3. Hilo A: Incrementa el valor recuperado; el resultado es 1.
4. Hilo B: Decrementa el valor recuperado; el resultado es -1.
5. Hilo A: Almacena el resultado en C; C ahora es 1.
6. Hilo B: Almacena el resultado en C; C ahora es -1.

El resultado del hilo A se pierde al ser sobrescrito por el hilo B.

Errores de Consistencia de Memoria

Los errores de consistencia de memoria ocurren cuando diferentes hilos tienen una vista inconsistente de lo que debería ser la misma data.

La clave para evitar errores de consistencia de memoria está en entender la relación **happens-before**. Esta relación es simplemente una garantía que las escrituras de memoria por una sentencia en específico son visibles por otra sentencia en específico.

Consideremos las siguientes acciones en hilos distintos:

- ✓Una variable compartida: `int counter = 0;`
- ✓Hilo A: `System.out.println(counter);`
- ✓Hilo B: `counter++;`

El valor impreso por el hilo B podría ser tanto 0 como 1 debido a que no hay garantía que los cambios de A sean visibles para B, a menos que se haya establecido una relación happens-before entre las dos sentencias.

Hay varias maneras de establecer relaciones happens-before. Hasta el momento se han visto 2 sentencias que crean relaciones happens-before:

- ✓Thread.start( ) : cada sentencia que guarde una relación happens-before con esta sentencia, también guardará una relación happens-before con cada sentencia ejecutada en el mismo hilo.

- ✓Thread.join( ) : todas las sentencias ejecutadas por el hilo terminado tienen una relación happens-before con toda sentencia que siga a la sentencia join exitosa.

Métodos Sincronizados

Para crear un método sincronizado basta con incluir la palabra clave **synchronized** en su declaración.

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

El uso de métodos sincronizados tiene 2 efectos:

1. No es posible que dos invocaciones de métodos sincronizados sobre un mismo objeto se interpongan. Cuando un hilo está ejecutando un método sincronizado para un objeto, todos los otros hilos que hayan invocado métodos sincronizados para el mismo objeto se bloquean hasta que el primer hilo termine con el objeto.

2. Cuando un método sincronizado termina, automáticamente establece una relación happens-before con cualquier invocación subsecuente de un método sincronizado para el mismo objeto.

Bloqueo Intrínseco

- La sincronización se crea entorno a una entidad interna conocida como **bloqueo intrínseco** o **monitor de bloqueo** (a veces se le suele referir simplemente como monitor).

- Cada objeto tiene un bloqueo intrínseco asociado a él. Un hilo que necesite acceso exclusivo y consistente a los campos de un objeto debe adquirir el bloqueo intrínseco del objeto y liberarlo una ves haya terminado.

- Mientras un hilo posea el bloqueo intrínseco de un objeto, ningún otro hilo podrá adquirir el mismo bloqueo.

- Liberar el bloqueo establece una relación happens-before entre esa acción y cualquier adquisición subsecuente del mismo bloqueo.

Sentencias Sincronizadas

A diferencia de los métodos sincronizados, las sentencias sincronizadas deben especificar el objeto que proporcione el bloqueo intrínseco.

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

SINCRONIZACIÓN REENTRANTE

- Un hilo no puede adquirir un bloqueo en posesión de otro hilo, pero puede adquirir un bloqueo que ya tenga. Permitir que un hilo adquiera el mismo bloqueo más de una vez habilita la **sincronización reentrante**. Esto permite que un código sincronizado invoque algún método con código sincronizado, y ambos conjuntos de código pueden usar el mismo bloqueo. Gracias a ello no es necesario tomar precauciones para evitar que un hilo se bloquee a sí mismo.

Acceso Atómico

Una acción atómica es una que efectivamente ocurre una a la vez. Una acción atómica siempre se completa del todo o no sucede. Ningún efecto secundario de una acción atómica es visible hasta que la acción este completa.

- Toda escritura y lectura son atómicas para toda variable que haya sido declarada como **volatile**.

Usar variables volátiles reduce el riesgo de errores de consistencia de memoria debido a que cualquier escritura a una variable volátil establece una relación happens-before con las lecturas subsecuentes de la misma variable.

Problemas de la sincronización de hilos

Deadlock

El **Deadlock** describe una situación donde dos o más hilos están bloqueados por siempre, esperando al otro. Consideremos la siguiente clase:

```
static class Friend {
    private final String name;
    public Friend(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public synchronized void bow(Friend bower) {
        System.out.format("%s: %s" + "  has bowed to me!%n",
            this.name, bower.getName());
        bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s" + " has bowed back to me!%n",
            this.name, bower.getName());
    }
}
```

Observando la siguiente secuencia evidenciamos un problema de deadlock:

1. Hilo A: Invoca el método bow sobre Amigo1, adquiere el bloqueo de Amigo1.
2. Hilo B: Invoca el método bow sobre Amigo2, adquiere el bloqueo de Amigo2.
3. Hilo A: Invoca el método bowBack sobre Amigo2, se bloquea pues el hilo B tiene el bloqueo de Amigo2.
4. Hilo B: Invoca el método bowBack sobre Amigo1, se bloquea pues el hilo A tiene el bloqueo de Amigo1.

Ambos hilo se bloquean para siempre pues ninguno soltará el bloqueo hasta que halla podido adquirir el bloqueo del otro amigo para responder al saludo.

Starvation y Livelock

STARVATION

- Starvation describe una situación donde un hilo es incapaz de ganar acceso regular a recursos compartidos y es incapaz de hacer progresos. Esto ocurre cuando los recursos compartidos son inaccesibles por largos periodos debido a hilos codiciosos.

LIVELOCK

- Un hilo usualmente actúa en respuesta a la acción de otro hilo. Si las acciones del otro hilo son también en respuesta a las acciones de un tercer hilo, entonces un livelock podría ocurrir. Los hilos con livelock son incapaces de hacer progreso debido a que están ocupados respondiéndose entre ellos para reanudar su trabajo. Es similar a la situación en la que dos personas quieren pasar por un corredor estrecho, una se mueve a la izquierda y la otra a la derecha, pero siguen bloqueándose mutuamente y no pueden pasar.

Comunicación entre hilos

La comunicación entre hilos es un mecanismo en el que un hilo es pausado durante la ejecución de una sección crítica y otro hilo tiene permitido entrar o bloquear la misma sección crítica para ser ejecutada. Se implementa haciendo uso de los siguientes métodos de la clase Objeto:

- wait( )
- notify( )
- notifyAll( )

Método wait( )

Ocasiona que el hilo actual libere el bloqueo y espere a que algún otro hilo invoque un método notify( ) o notifyAll( ) para este objeto, o una cantidad de tiempo especificado haya transcurrido.

El hilo actual debe tener propiedad del monitor del objeto, por lo que este método debe ser llamado desde un método sincronizado o arrojará una excepción.

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Método notify( )

- El método **notify( )** despierta uno de los hilos que este esperando en monitor de este objeto. Si algún hilo está esperando en este objeto, uno de ellos es escogido para ser despertado de forma arbitraria.

- El hilo despertado no será capaz de proceder hasta que el hilo actual libere el bloqueo del objeto. Los hilos despertados no cuentan con privilegios o desventajas en ser los próximos hilos en bloquear este objeto.

- Este método sólo puede ser llamado por un hilo que tenga propiedad del monitor del objeto, por tanto solo puede ser llamado dentro de un método sincronizado o dentro de un conjunto de sentencias sincronizadas.

- Arroja un **IllegalMonitorStateException** si el hilo actual no es dueño del monitor del objeto.

Método notifyAll( )

- El método **notify( )** despierta todos los hilos que esten esperando en monitor de este objeto.
- El hilo despertado no será capaz de proceder hasta que el hilo actual libere el bloqueo del objeto.
- Los hilos despertados serán terminados de la forma usual con cualquier otro hilo que esté compitiendo por el bloqueo del monitor de este objeto, i.e. los hilos despertados no cuentan con privilegios o desventajas en ser los próximos hilos en bloquear el objeto.
- Este método sólo puede ser llamado por un hilo que tenga propiedad del monitor del objeto .
- Arroja un **IllegalMonitorStateException** si el hilo actual no es dueño del monitor del objeto.

Java.util.concurrent.locks

Lock

- Las interfaces **Lock** proporcionan operaciones de bloqueo más extensivas a las que se pueden obtener a través de métodos sincronizados.
- Permiten un estructuración más flexible, puede tener distintas propiedades, y puede soportar la asociación de múltiples objetos de la clase **Condition**.

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this
lock
} finally {
    l.unlock();
}
```

Cuando el bloqueo y desbloqueo ocurre en distintos scopes, debemos asegurarnos que todo el código ejecutado durante el bloqueo esté protegido por un try-finally o try-catch para asegurarnos que el bloqueo es liberado cuando sea necesario.

Lock vs. Synchronized

- El uso de métodos sincronizados fuerza que todas las adquisiciones y liberaciones de bloqueos ocurran de forma estructurada por bloques; cuando se adquieren múltiples bloqueos, estos deben ser liberados en el orden opuesto, y todos los bloqueos deben ser liberados en el mismo scope en el que fueron adquiridos.

- El uso de Locks permite adquirir y liberar distintos bloqueos en cualquier orden e incluso en distintos scopes.

- Las implementaciones lock proporcionan una mayor funcionalidad sobre el uso de métodos sincronizados proporcionando un intento de adquirir bloqueos sin necesidad de bloquearse a sí mismo hasta lograrlo, y un intento de adquirir un bloqueo que puede ser interrumpido.

Métodos de Lock

- **lock( )** : adquiere el bloqueo.
- **unlock( )** : libera el bloqueo.
- **lockInterruptibly( )** : adquiere el bloqueo a menos que el hilo actual sea interrumpido.
- **newCondition( )** : retorna una instancia de la clase Condition que está vinculada a esta instancia de Lock.
- **tryLock( )** : adquiere el bloqueo sólo si se encuentra libre al momento de la invocación. Retorna un boolean.
- **tryLock(long time, TimeUnit unit)** : adquiere el bloqueo si se encuentra libre durante el tiempo de espera dado y el hilo actual no ha sido interrumpido. Retorna un boolean.

Condition

- Las condiciones proporcionan medios para que un hilo suspenda su ejecución hasta que sea notificado por algún otro hilo que alguna condición de estado podría ya no ser verdadera.

- Debido a que el acceso a este estado compartido ocurre en distintos hilos, debe ser protegido, por lo que un bloqueo está de cierta forma asociado con la condición.

- El uso de condiciones permite reemplazar el uso de un objeto monitor y las funciones wait, notify y notifyAll al igual que el uso de Locks permite reemplazar el uso de métodos sincronizados.

- Para conseguir una instancia de Condition para un Lock en particular, se usa su método newCondition( ) ya visto.

Métodos de Condition

- **await( )** : causa que el hilo actual espere hasta que sea “señalado” o interrumpido.

- **await(long time, TimeUnit unit)** : causa que el hilo actual espere hasta que sea “señalado”, interrumpido, o el tiempo de espera especificado termine. Retorna un boolean dependiendo si el tiempo de espera se completo (false) o no (true).

- **awaitUninterrumptibly( )** : causa que el hilo actual espere hasta que sea “señalado”.

- **signal( )** : despierta uno de los hilos en espera.

- **signalAll( )** : despierta todos los hilos en espera.