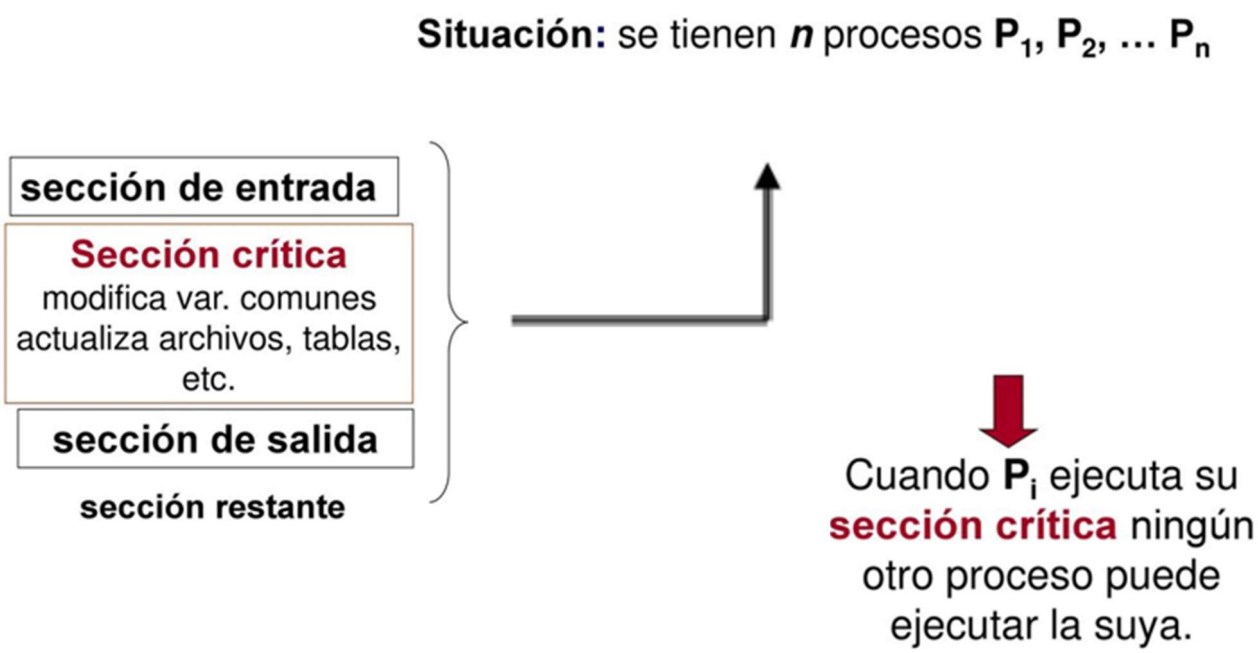
Problemas al trabajar con hilos

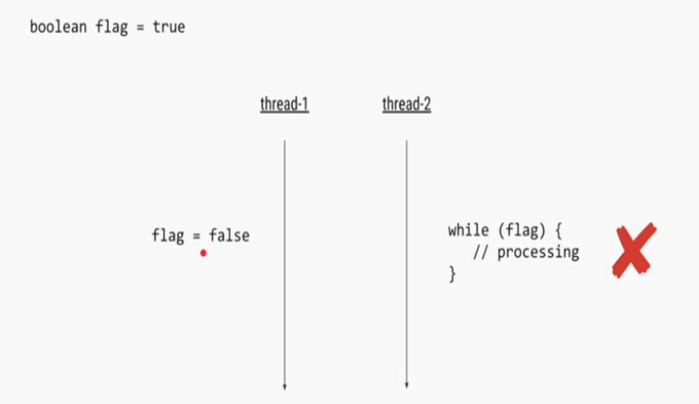
Cuando trabajamos con hilos que intervienen en partes de un programa y se ejecutan de forma independiente sin que ninguno de los hilos comparta recursos no tendremos ningún problema más allá de la lógica que cada uno de los hilos ha de tener para resolver su tarea.

Sin embargo, cuando nos encontramos en problemas donde los procesos han de compartir recursos en la memoria podemos encontrarnos con dos problemas:

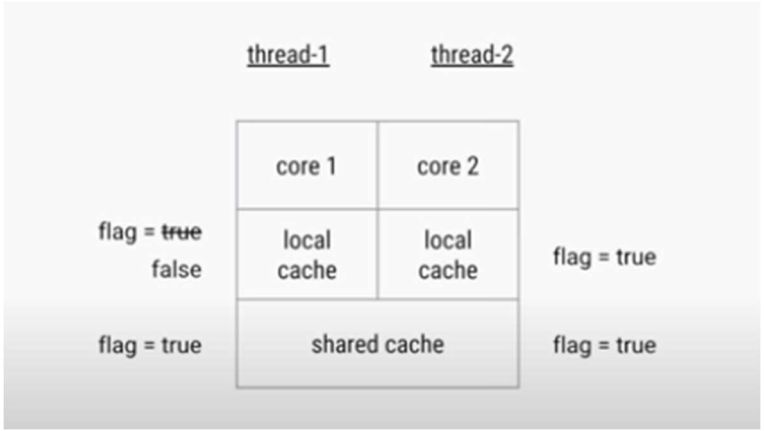
- Visibilidad: En este caso el problema es que cuando un hilo cambia el valor de una variable otro hilo puede no ver ese cambio por como se estructura la memoria.
- Sincronización: Este caso se produce cuando varios procesos quieren acceder de forma concurrente a la memoria de un recurso con lo que debemos de proteger este acceso ya que no se podría garantizar la coherencia del estado de ese recurso si se accediera a él de forma indiscriminada.



Modificador volatile en java



Lo primero que tenemos que entender, es que es un problema de visibilidad. Como ya hemos comentado un hilo puede no ver el cambio que hace otro en una variable. En la imagen podemos apreciar como un hilo depende del cambio que haga otro para para su ejecución, pero como veremos en el ejemplo no podremos conseguir que esto funcione bien sin el operador volatile. Esto es por lo que hemos dicho de la memoria, en concreto de la memoria cache. Cuando una cpu cambia el dato en su cache, en el caso de no usar volatile, el dato se queda en la cpu que realiza el cambio y no la refresca en la memoria compartida, pero si utilizamos volatile lo que ocurre es que se fuerza inmediatamente después del cambio a que la memoria compartida tenga ese valor y por lo tanto podamos ver ese valor desde todas las CPU.



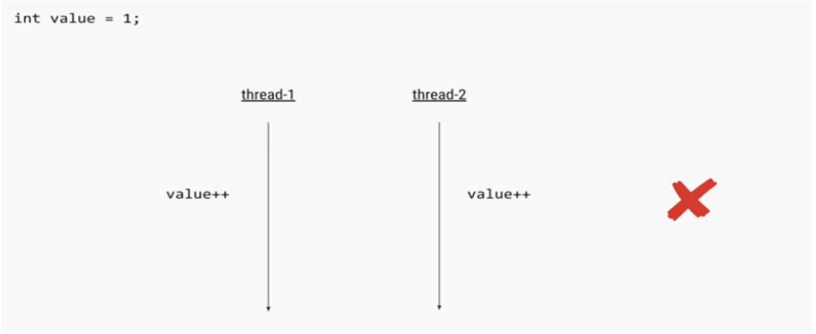
Operaciones Atómicas

```
volatile int value = 1;
```

Even with volatile

#	Thread-1	Thread-2
1	Read value (=1)	
2		Read value (=1)
3	Add 1 and write (=2)	
4		Add 1 and write (=2)

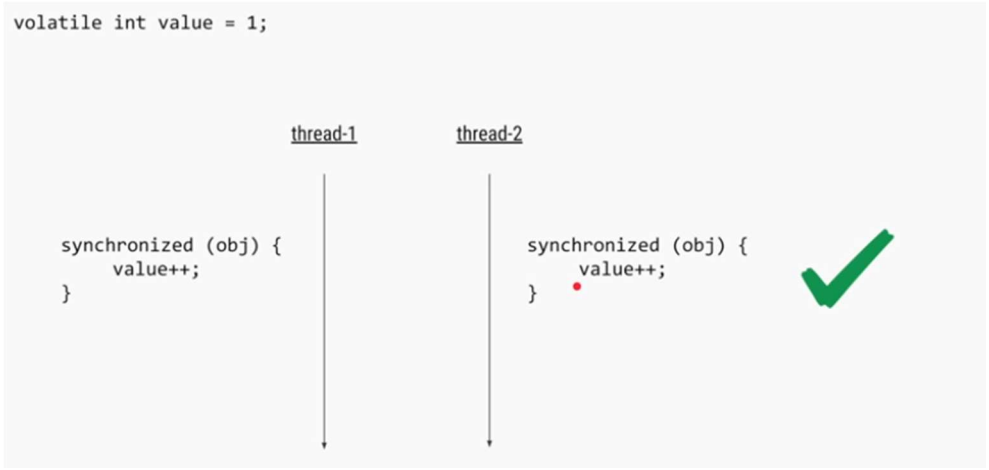
Una operación atómica es aquella que no puede divirse en ninguna otra operación más. Cuando hemos visto volatile hemos operado sobre la variable para hacer una única operación, cambiar su valor. En el caso de que la operación que pongamos en una instrucción se subidiva en alguna operación más, no podremos considerarla atómica, por ejemplo, i++. Esa expresión se traduce en la suma del valor contenido y luego en la asignación de nuevo a la variable, con lo que aunque sea una instrucción, está dividida en dos operaciones.

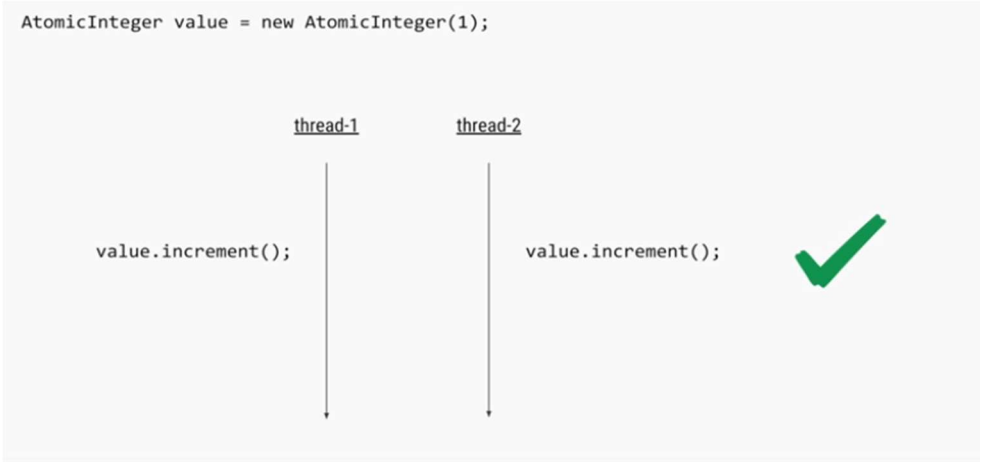


En este caso la operación de lectura se ejecuta en dos ciclos, cuando el hilo 1 cambie la variable, la cpu ya tiene el valor antiguo leído, con lo que la CPU lanzará el valor dos a la memoria compartida y lo mismo hara la CPU que maneje el hilo 2, habiendo perdido un número en la secuencia. Este es el caso del contador.

Para solucionar esto tenemos unos objetos especiales que nos suministrar de herramientas de sincronización, recordad que si no es un problema de visibilidad lo es de sincronización, con lo que esos objetos tienen unos mecanismo de sincronización para realizar las operaciones de este tipo como si fueran atómicas.

Podríamos resolver este caso con otros mecanismos y utilizando el modificado volatile, tan solo tenemos que proteger de una forma sincronizada el incremento de la variable. De esta forma, un único hilo podrá acceder a esa parte del código, con lo que las operaciones que haga un hilo en ese bloque serán vistas por todos cuando ese hilo haya acabado sus instrucciones y salga del bloque.





También podríamos y sería mejor solución para este caso utilizar las variables atómicas que tenemos disponibles para estos casos y utilizar sus métodos especialmente diseñados para estas ocasiones.

- incrementAndGet
- decrementAndGet
- addAndGet (int delta)
- compareAndSet (int expectedValue, int newValue)

Type	Use Case
volatile	Flags
AtomicInteger AtomicLong	Counters
AtomicReference	Caches (building new cache in background and replacing atomically) Used by some internal classes Non-blocking algorithms

OTROS MECANISMOS DE SINCRONIZACION

La comunicación o **cooperación entre subprocesos** se trata de permitir que los subprocesos sincronizados se comuniquen entre sí.

La cooperación (comunicación entre subprocesos) es un mecanismo en el que un subproceso se detiene en ejecución en su sección crítica y se permite que otro subproceso entre (o bloquee) en la misma sección crítica que se ejecutará. Se implementa siguiendo los métodos de la **clase Object** :

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

wait, notify, notifyAll()

1) método wait ()

Hace que el subproceso actual libere el bloqueo y espere hasta que otro subproceso invoque el método notify () o el método notifyAll () para este objeto, o haya transcurrido un período de tiempo especificado.

El subproceso actual debe poseer el monitor de este objeto, por lo que debe llamarse desde el método sincronizado solo de lo contrario arrojará una excepción.

Método	Descripción
public final void wait () throws InterruptedException	espera hasta que se notifique el objeto.
public final void wait (long time) throws InterruptedException	espera la cantidad de tiempo especificada.

---

2) método de notificación ()

Despierta un solo hilo que está esperando en el monitor de este objeto. Si hay hilos esperando este objeto, se elige uno de ellos para ser despertado. La elección es arbitraria y ocurre a discreción de la implementación. Sintaxis:

public final void notify()

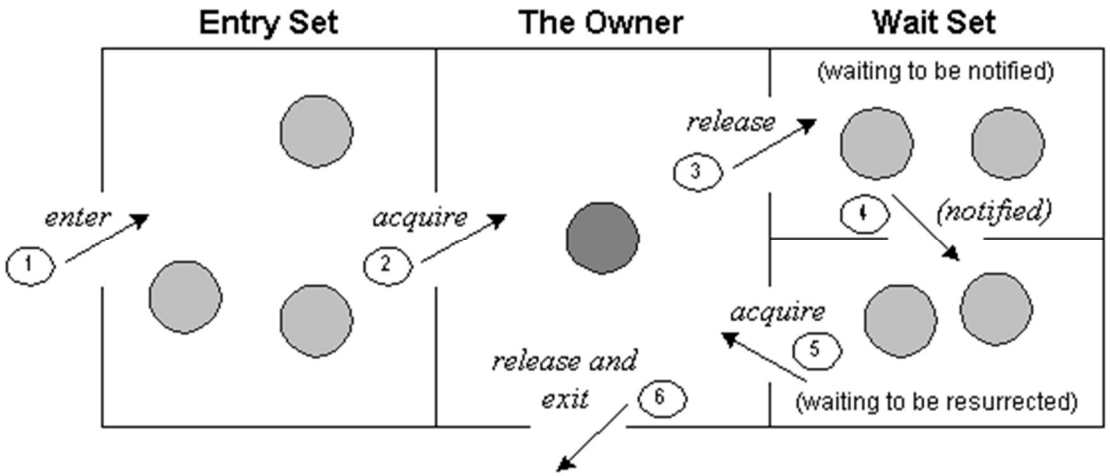
3) método notifyAll ()

Despierta todos los hilos que están esperando en el monitor de este objeto. Sintaxis:

public final void notifyAll()

Comprender el proceso de comunicación entre hilos

La explicación punto a punto del diagrama es la siguiente:



1. Los hilos entran para adquirir el bloqueo.
2. El bloqueo se adquiere en el hilo.
3. Ahora el hilo pasa al estado de espera si llama al método wait () en el objeto. De lo contrario, libera el bloqueo y sale.
4. Si llama al método notify () o notifyAll (), el subproceso se mueve al estado notificado (estado ejecutable).
5. Ahora el hilo está disponible para adquirir el bloqueo.
6. Después de completar la tarea, el hilo libera el bloqueo y sale del estado del monitor del objeto.

¿Por qué los métodos wait (), notify () y notifyAll () se definen en la clase Object y no en la clase Thread?

Es porque están relacionados con el bloqueo y el objeto tiene un bloqueo.

¿Diferencia entre esperar y dormir?

Veamos las diferencias importantes entre los métodos de espera y de sueño.

wait()	sleep()
el método wait () libera el bloqueo	El método sleep () no libera el bloqueo.
es método de la clase Object	es método de la clase Thread
es método no estático	es método estático
es método no estático	es método estático
debe notificarse mediante los métodos notify () o notifyAll ()	después de la cantidad de tiempo especificada, se completa el "sueño".

Palabra reservada synchronized

Como ya hemos dicho la palabra synchronized nos va a permitir bloquear una determinada sección del código de forma que no pueda ser accedida por ningún otro proceso mientras un proceso lo tenga bloqueado

Donde podemos utilizar esta palabra reservada

La palabra synchronized puede ser utilizada en diferentes niveles:

- Métodos de instancia
- Métodos estáticos
- Bloques dentro de un método

Todos estos métodos utilizan lo que se conocen como monitores que nos proveen de una garantía de acceso concurrente al código.

1) método de instancia

Solamente hay que añadir la palabra synchronized al método que vayamos a escribir.

```
public synchronized void synchronisedCalculate() {
  setSum(getSum() + 1);
}
```

2) método estático ()

añadirEstos métodos se sincronizan a nivel de clase y puesto que solo existe un objeto de esa clase solamente un hilo se podrá ejecutar dentro de un método estático sincronizada, un hilo por clase, independientemente del número de instancias que tenga la clase

3) método bloque sincronizado

El bloque sincronizado se puede usar para realizar la sincronización en cualquier recurso específico del método. Supongamos que tiene 50 líneas de código en su método, pero desea sincronizar solo 5 líneas, puede usar el bloque sincronizado. Si coloca todos los códigos del método en el bloque sincronizado, funcionará igual que el método sincronizado.

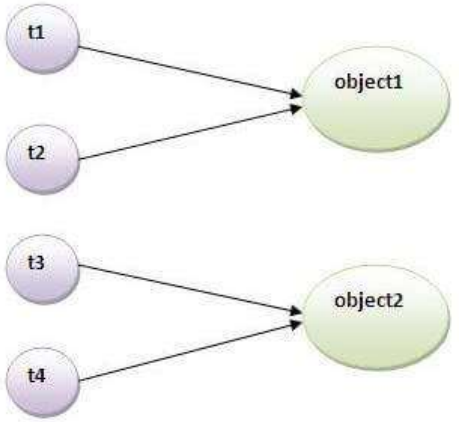
Puntos a recordar para el bloque sincronizado

- El bloqueo sincronizado se usa para bloquear un objeto para cualquier recurso compartido.
- El alcance del bloque sincronizado es menor que el método.

Sintaxis para usar bloque sincronizado

```
1. synchronized  (expresión de referencia del objeto) {
2.  // bloque de código
3. }
```

Sincronización estática



Si realiza algún método estático como sincronizado, el bloqueo estará en la clase, no en el objeto.

Problema sin sincronización estática

Supongamos que hay dos objetos de una clase compartida (por ejemplo, Tabla) llamados object1 y object2. En el caso del método sincronizado y el bloque sincronizado no puede haber interferencia entre t1 y t2 o t3 y t4 porque t1 y t2 se refieren a un objeto común que tiene un solo bloqueo. Pero puede haber interferencia entre t1 y t3 o t2 y t4 porque t1 adquiere otro bloqueo y t3 adquiere otro bloqueo. No quiero interferencia entre t1 y t3 o t2 y t4. La sincronización estática resuelve este problema.

Deadlock

El deadlock en Java es una parte del subprocesamiento múltiple. El punto muerto puede ocurrir en una situación en la que un subproceso está esperando un bloqueo de objeto, que es adquirido por otro subproceso y el segundo subproceso está esperando un bloqueo de objeto adquirido por el primer subproceso. Dado que ambos subprocesos están esperando entre sí para liberar el bloqueo, la condición se llama punto muerto.