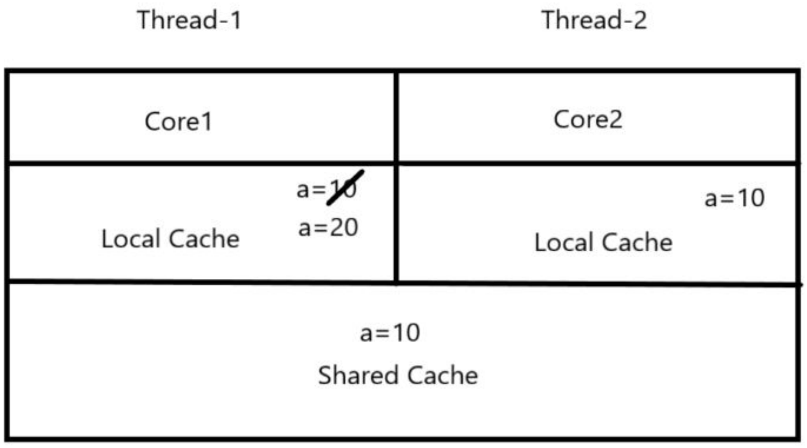
Volatile and Atomic Variable

What is the Visibility Problem?

As we already know the thread has its **own memory or local cache** so when any update or write operation is done then thread updates its local memory so the updated value is not visible to other threads in the **multi-cores** environment so this is called visibility problem.

Let’s understand from example.

- Initially, the value of **a is 10** and now thread 1 updated **value to 20** in its local cache.
- The value has been updated in the thread1 local cache is **not be visible to thread 2,** this causes visibility problem.



To fix this problem we use volatile keyword.

What is volatile?

Volatile variables are those variables which are not optimized by the compiler while parsing the code. Volatile indicates that this variable value may change every time.

- **Volatile** variable are not stored in cache.
- It will be accessed from the **shared cache(hardware memory)**, not from the cache.
- It **avoids optimization** every time it is accessed from the shared memory, requires more time to access.

Hardware Level working of Volatile:

Hardware-level any and all reads/writes to any and all memory addresses always occur in L1 and register first. Java abstracted out this low-level behavior from the programmer.

When we use the volatile keyword on a variable in Java, this simply tells the compiler to insert as a memory barrier on the reads/writes to this field.

The memory barrier effectively ensures two things;

1. Any threads reading this address will use the most up-to-date value, the barrier makes them wait until the most recent write makes it back to shared memory, and no reading threads can continue until this updated value makes it to their L1 cache.
2. No reads/writes to ANY fields can cross over the barrier, they are always written back before the other thread can continue, and the compiler cannot move them to a point after the barrier.

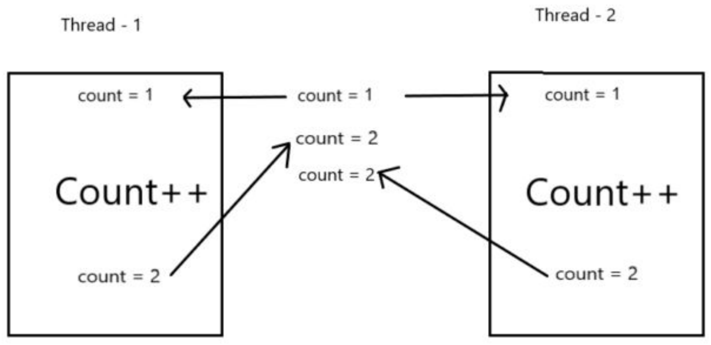
because of this volatile variable are slower than normal variables. Now we have understood the volatile variables.

Synchronization Problem with volatile variables:

Volatile variables are not atomic in so whenever a thread writes and at the same time other thread also write the value in shared memory this causes synchronization problem.

Let’s understand from the example.

- *Thread 1* read value 1 for the count variable. at the same time, *Thread 2* also read the same value of count.
- *Thread 1* updated value 2 for the count variable, at the same time, *Thread 2* also updated the same value of count.
- Due to no mutual locking of the critical block, this causes ***data integrity problems.***



- There are a couple of ways to fix this problem let’s see how we fix this using the atomic variables.

What are the atomic variables?

Atomic variables are the variable which non-blocking for concurrent environments. These variable uses low-level atomic machine instructions such as **compare-and-swap (CAS)**, to ensure data integrity.

- Atomic variable avoids deadlocks.
- The compound operations with such variables are performed automatically without the use of synchronization
- The memory effect is the same as a volatile variable

Let’s see some of the methods of atomic variables.

- ***incrementAndGet***(): Atomically increments by one the current value and returns the updated value.
- ***decrementAndGet***(): Atomically decrements by one the current value and returns the updated value.
- ***addAndGet(int delta):*** Atomically adds the given value to the current returns the updated value.
- ***compareAndSet():*** Atomically sets the value to the given updated value if the current value is equal to the expected value, **return true if** successful. False return indicates that the actual value was not equal to the expected value.