Concurrency : Basics

Spin lock – busy waiting – keep on checking.

Synchronize → boolean flag set on the object header by JVM.

JVM checks – Single core ? Thread Suspension
Multi core ? Do a brief spin.
Check for lock availability
Other wise OS call down to Thread Suspension.

LinkedTransfer Queue JDK 7 → logic is there.

→ JDK 6 Synchronized keyword has the above logic.

boolean checking at Object level for sync keyword
So 2 Objects will have 2 booleans. version

* Object on which you sync should be always final.
(bb) Sync @ class level . i.e this . class.

Class or Object level lock ?

→ Object one is more flexible, but chance of making mistake.

→ Class level – less error prone. But 2 methods of diff class then no mut exclu

→ Sync static – Should not choose
- because not polymorphic. (Not for OOP)
- Difficult to Unit Test.

```
        Class A {
t1 ⟶        static int i;
            sync static m1() {        locked at
                - - -                 class
            }
t2 ⟶        sync m2() {               locked at
                - - -                 Object.
            }
t3 ⟶        m3() {
                - - -
            }
        }
```

Reason to Sync a Get() :
→ Get should not look into a structure mid way of a multi step Operation.
→ Synchronization will always look at main memory and not the state data in caches.

Mutator method (add / remove)
→ lost data. Because both are (read, modify, write)

Wait / notify()
Jvm has a hidden queue (wait queue)

Notify → Ready To Run State
* Clearing of bit means Thread has been the notify().

① Check condition always after a notify.
i.e while vs if example
Best Practice is to use while for condition checking when you have wait() / notify here.

★ notify vs notifyAll() use case :
notify — Singular manner Ex : Printer case
notifyAll — Cache Clearing
So that objects dependent on this Cache Clearing will be woke up.

Thread.Stop Method is depricated now.

Instead Use Thread.interrupted. (Checks and clears the below flag)

```
Ex: run () {
      Sync () {  ← lock obtained
        ... → Something went wrong and in
      } lock release   in consistent state
    }
```

Instead

```
while (Thread.interrupted) {
  .....
}
```

Thread.interrupt flag

Synchronization :-

① Mutual Exclusion

Double Checking lock

```
if (resource == null) {
  Synchronized (constructor) {
    if (resource == null)
      new resource ();
  }
  return resource;
```

* Not when you say new() :

Address allocation by JVM happens in the Object class. then the constructor initialization is done.

JVM: (Comparisons)

1. Interpreter — Some time slower than native code
2. JIT Compiler (like C++, so 7 to 8 times slow)
3. Hot Spot Compiler
   ↳ Highly optimized machine dependent code on the fly.

Dynamic Compiler

→ Activated by -client command. Usually for GUI apps.

→ Activated by -Server option. (Don't waste time on start up)

→ -tiered Compilation. i.e best of both the above.

★ JDK 8 has -tiered Compilation by default.

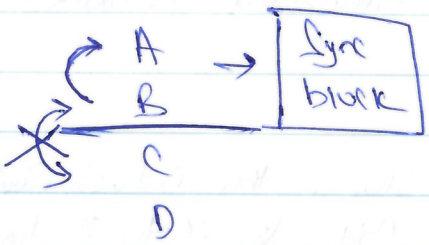
Program re ordering } JVM Processor ⇒ To keep all 6 ports busy
Instruction re ordering }

⇒ Helps in Performance.

★ int wraps around in Java.
i.e after the limit it will start from 2nn.

*Synchronization Advantage.

(2) Guaranteed against re ordering.



re ordering happens only here.

(3) Guarantee against caching

Any code, read inside sync code will be loaded from main memory irrespective of if that value is found on cache or not.

(If found in cache evict it)

(4) Guarantees happens before relationships.

Volatile!

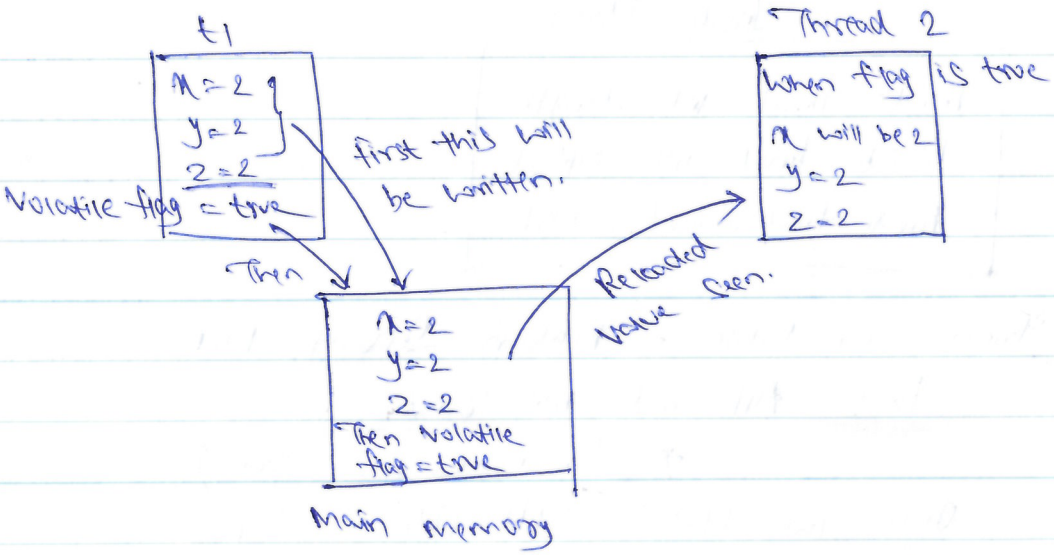
(From Java 5)

Old memory	New memory model
(1) Volatile reads from main memory and writes to main memory	* Same as the Old model.
(2) Volatile variable may get re ordered with a non-volatile but not get re ordered with volatile	* Volatile variable will not get re ordered.

(3) Volatile variable

Observes a happens before rela

* Happens before relationship.



* T1 & T2 should work on same volatile flags.
Other wise reload of non-volatile variables are not guaranteed.

* Volatile does not give mutual exclusion *

→ Rest all is same like Synchronization

Ex:
```
Counter class {                    volatile i;
Sync void increment {
    i++;
}
int getCount() {
    return i;
}
}
```

we need mutual exclusion here.

22-04-2015

Inside String

```
private final int length
private final int offset
private final char[] ref
```

Strings immutable – Reason Security Purpose

Ex: API and file reading

* (File names are String)

Only Two choices: ① Defensive Copies

② making immutable.

final Keyword:

JDK 5.

Old memory model

① final's definition is under specified allowing the JVM to re order or cache final.

② The run time (JVM) is not sure that the value of final that it sees is before or after initialization and hence has to reload it in certain cases.

New memory model.

* If a final field is initialized in a constructor, then the obj of the enclosing object is not allowed to escape unless the JVM has initialized the final fields and Pushed to main memory

→ Same thing applies when a constructor throws Exception. !!!

Point 2 : New model

* The JVM can always keep the final fields cached.

③ final fields observe a freezing effect for object references.

```
class CC {
    public final List t;
    Constructor() {
        add all 14 names
    }
    getCCSize() {
    }
    addLT(name) {
14 names
+ latest Trinee }
```

T1 →

T2

```
if (CC != null) {
    S.O.P (list.size());
}
```

14 guaranteed

late Trinee is might be

Immutable class.

```
class IP {
final x, y;
    IP (x, y) {
        this.x = x;
        this.y = y;
    }
}
```

**

```
class Point {
    volatile IP P;
    Set Coordinate (x,y) {
        P = new IP (x,y);
    }
```

Sync — Not required because its not a read/modify, write operation unlike int;

```
    int[] get Coordinate() {
        IP SP = P;
        return new int[] {SP.x, SP.getY()};
    }
}
```

* Read, modify, write always requires a sync.

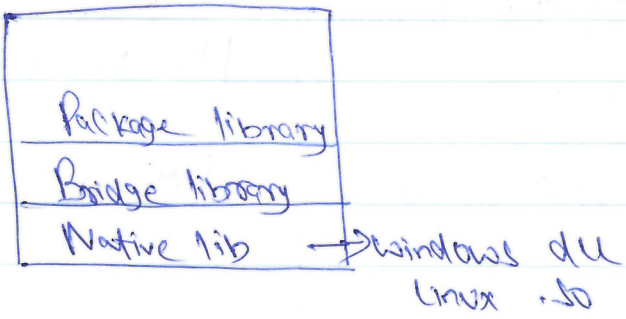
L3 Cache misses - Measuring Tool.

VTune intel - Paid version.

Solaris Studio analyzer

↳ Hardware | Software Measure.

JDK



Compare and Swap (CAS) - x86 Architecture.

Counter example :

⇒ Replace the sync with atomicInteger as i and use incrementAndGet()

↳ Internally has the call to CAS in JDK 7.

In JDK 8 it will call a different method in x86 Arch, so Performance boo

→ Then return ai.get();

Atomic Variables:

1. The re-ordering, caching, happens-before Semantics of Atomic Variables is identical to that of Volatile.
2. This Guarantee does not apply to the method weakCompareAndSet(). (i.e Should not be used)

The weakCompareAndSet() method ensures Ordering without Visibility.

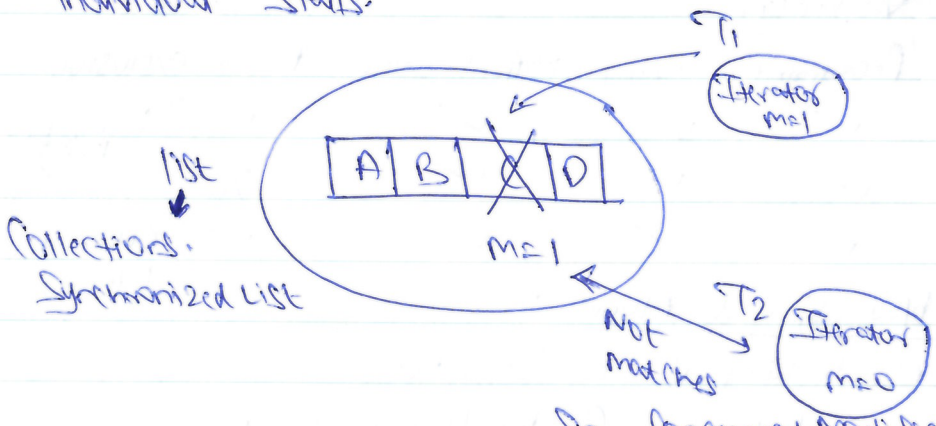
	Reordering	caching	happens-before	Mutual exclusion	
Sync	✓	✓	✓	✓	(Pessimistic lock)
Volatile	✓	✓	✓	X	
(AS) Atomic	✓	✓	✓	✓	Optimistic (lock)

* Difference b/w Sync & Atomic :-
  - → Sync locking is in software
    Atomic locking is @ hardware level
  - → Sync gives Compound Statement lock
    Atomic is individual elements
* Optimistic lock — read is allowed across Threads
  write is not allowed across
* Pessimistic lock — no read, write till lock is released.

"Wait free algorithm" → Using CAS (Atomic variables)

* CompareAndSet() Queue example demonstrates this. "Concurrent linked Queue".

Wait Free Algorithms are based on non-blocking (CAS) algorithms which Promises global Progress inSpite of individual Stalls.



So ConcurrentModificationException.
Even though list is Synchronized.

* All Data Structures except linked list should not have a no-arg constructor - Its a Anti Pattern, because it resizes over time.

Concurrent HashMap:

1. Put & remove use lock Striping.
2. Get does not have any Synchronization.

Because Value is Volatile → No Stale reads
next is final.
↳ No inconsistent Structure

JSR (Java Specification Request)

⇒ next is final and final value is always cached.

JDK 7 → ConcurrentHashMap → Array (Array (linked list))

* (3) ConcurrentHashMap does not allow null values

Iterator :

(1) Java.util.concurrent Collection classes return iterators which are weakly consistent.

(2) Weakly consistent iterators extend the core iterators contract with the promise of never throwing a ConcurrentModificationException.

* Do not use ConcurrentHashMap's size method. This is going to take 64 locks. (JDK 7)

⇒ Non-blocking Hash map [comes from Highly Scalable Package, not Java? library]

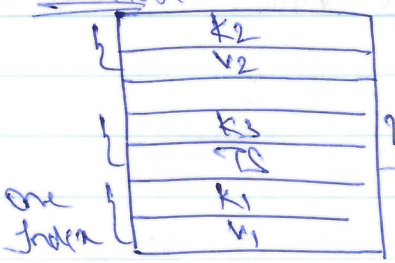
* Get, Put, remove and re-hashing is all wait free. No linkedlist. This uses a single array
  * Array is not synchronized as its an immutable one and (key, value) is CAS protected and there is a concept of TombStone for remove and then get.

⇒ Avoiding fm by not deleting the key from that place and having a "Tombstone value there.

if ("Tombstone in value)

return null;

Structure.



If slot is empty then key is not present.

* Hence no synchronization and no blocking.
* Re Probe limit has been set after which the array is re-hashed.

* If Get() is not that much, use non blocking hash map.

* Non blocking hash map scales linearly.
(Hadoop, Hbase, Cassandra use this)

* Re size limit (multiple Threads) collision scenario makes a non blocking Hash map to be a blocking Hash map for a brief time.

Issues with Sync :

1. One condition Per lock
2. NO lock Polling . i.e Thread which does not get a lock cant do anything else
3. NO Timed lock waits.
4. NO interrupted lock waits
5. Block Structured lock. (JVM is doing an intrinsic Ex: Calling an inner method lock) will still hold lock . i.e NO way to release lock.

JDK 5 locking (java.util.concurrent.locks.lock)

- tryLock() — i.e lock Polling
- tryLock(long time, TimeUnit unit)
- void lock() → Hold old Sync } Solves above BSL Problem
- void unlock() → Release the lock
- void await() → Identical to wait
- void await(long time, TimeUnit unit)

- long awaitNanos(long nanoTime) → Short time wait
- void signal() } Same as notify, notify all.
- void signalAll()

* Reentrant lock – Checks two attributes every time
  ↓ Owner and Reawester.
  Exclusive lock

Reentrant read write lock } Slightly better
Reentrant read lock } Concurrency here.
Multiple Threads can do read here
if there is no RRWL.

JDK8 { LongAdder
DoubleAdder
DoubleAccumulator
StampedReference - JDK 7.

* There are certain special methods in Java whose effects cannot be re ordered or ~~cached~~ cached and they always observe the happens – before relationship.
  Ex: Thread.Start
  Thread.Join ... Thread.*
  Object.wait / notify / notifyAll
  lock.*
  Condition.*

* Defensive copies are much much better than getting a lock.

Thread Local :

```
DistributedCounter {
    Connection conn; → have a ThreadLocal for connection
    ---
    ---

    increment() {
        Connection con = T.get();
        if (con != null) {
            // get Connection
            t.set(con);
        }
    }
```

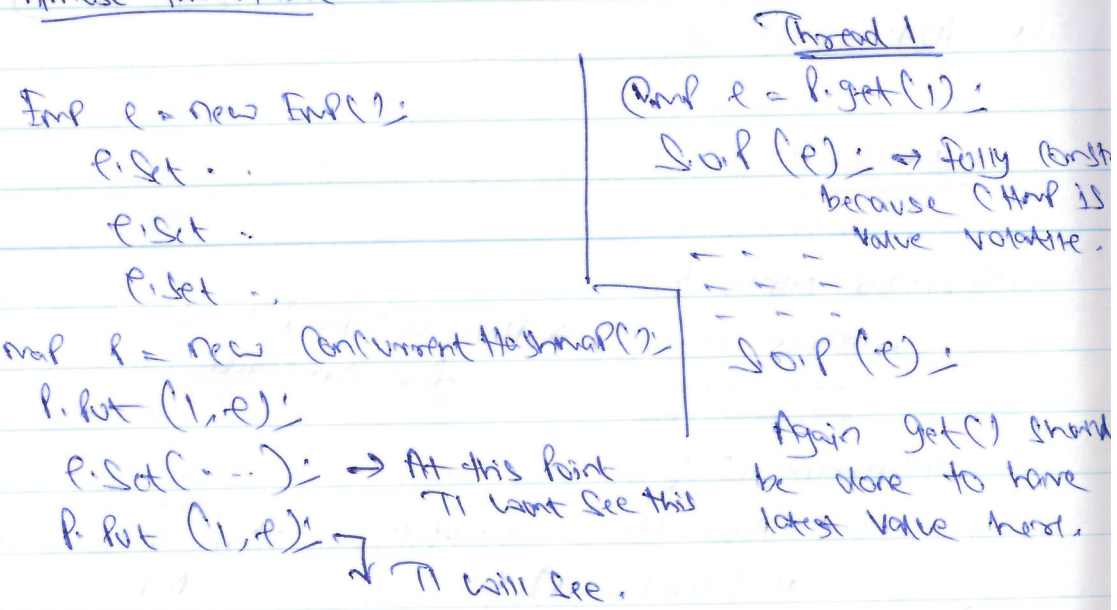
Thread 1 : → has a own ThreadLocal map.

Thread 2 : → Own ThreadLocal map

* So in above example, each Thread has its own connection, its not shared.
* Allows to have a local variable only owned by that thread. (In above example, not need to have per Thread basis connection code).

* Immutable : Don't make class as final, but make the constructor as Package Protected. So atleast You can use inheritance and not others.
  - Defensive copies for accessors. i.e get()
  - NO Setters
  - final fields.

Almost immutable:



* Class loader gives guarantee of concurrency on a static object. (i.e fully constructed object).

* From the concurrency perspective, the guarantees offered by static are identical to that of "final".

Future Interface: Not receiving the result immediately.

Vget(): retrieves result from the future object blocks if necessary if result not ready

* Async rendering.