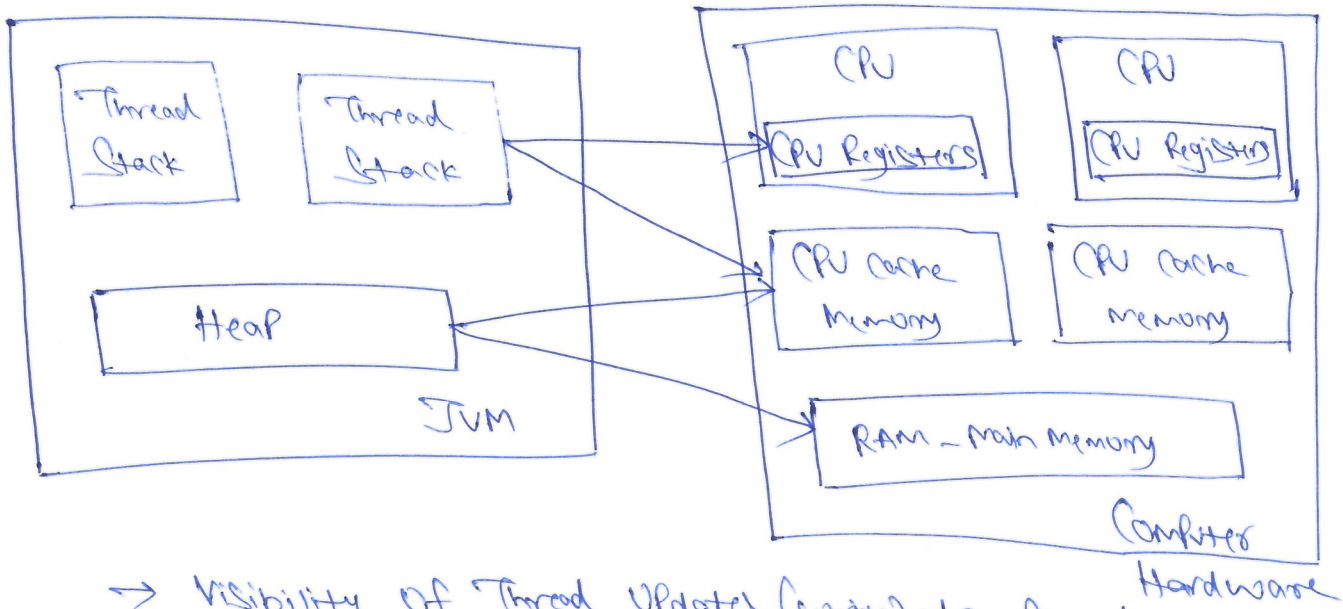
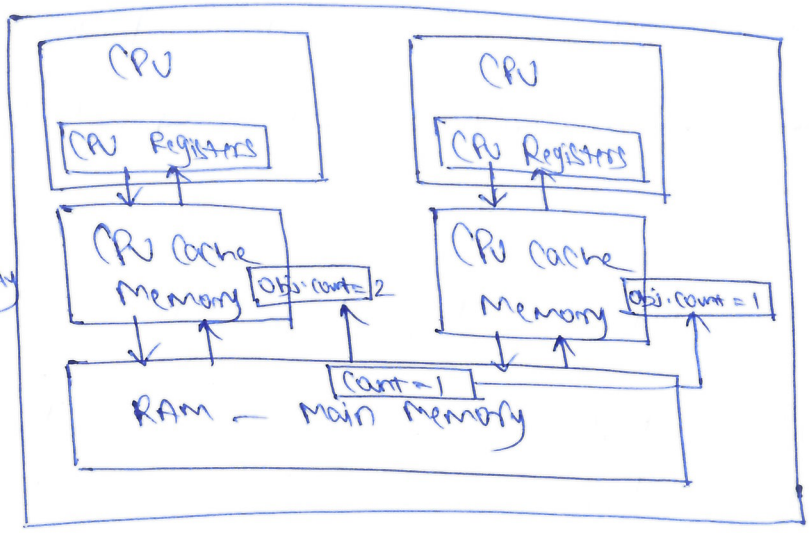


→ Visibility of Thread Updates (write) to Shared variables.

→ Race Conditions when reading, checking and writing Shared Variables.



⇒ Two increments carried out concurrently without Proper Synchronization.

⇒ To Solve above we need Volatile Variables.
Or Synchronization.

* Synchronization offers Mutual Exclusion. (i.e locking Critical Section of code)

* Check Synchronization Advantages in notes.

* Concurrency : Related to how an application handles multiple tasks it works on. i.e working on multiple tasks at a time

* Parallelism : Related to how an application handles each individual task. An application may process the task serially from start to end or split the task up into subtasks which completes in parallel.

Instruction Re-Ordering

Variables initialized to Zero.

T1	T2
x = 5;	if (y == 6)
y = 6;	SysOut("x = " + x);

With proper Synchronization, T2 will always print 5 or nothing.

* Instructions will be re ordered, either by Compiler, by the JVM at runtime, or even by Processor.

⇒ Happens before Relationship of JMM. → Check notes.

* Use volatile when having Single Writter – multiple readers. This will be faster than Synchronization.

Semaphore : Its a lock.

- → lock is binary → Semaphore is Counting.
- → lock is reentrant → Semaphore is not Reentrant.

* When there is no reentrant behaviour Use Semaphore Otherwise Semaphore on a single thread will deadlock

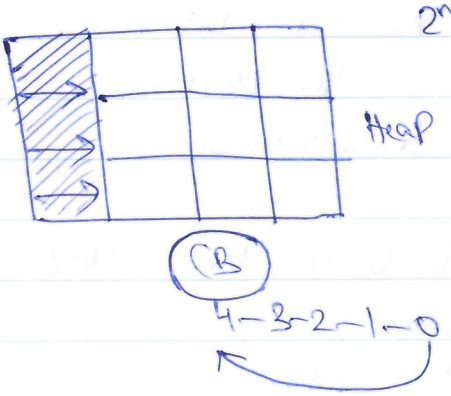
Barrier : Parrarel iterations become easy.

* Barriers are re-usable, and the moment count goes to zero, it set to 4 or (n).
* Number of threads on the barrier should be equal to barrier count.

"PHASER"

JDK 7. [redacted] - Per iteration basis threads.

Ex : 1st iteration - 4 Threads
2nd iteration - 2 Threads etc.



Latch : Event is external here.
Latch is not re-usable.

External Thread Opens the latch and then Other Threads get in. (ie locked from inside).

Stage wise Progression - Use latch.

* Count Down latch is not re-usable.
* Count on the latch is unrelated to the number of threads it's holding back. i.e the event is external.

(JDK 5 or 6)
Synchronous Queue: Don't use this on JDK 7.

Use TransferQueue extends BlockingQueue
- transfer() method is like a Synchronous Put method.
⇒ tryTransfer() - returns boolean i.e whether consumer is there or not
* If yes then Put
* else go back.

LinkedTransferQueue: implements TransferQueue.

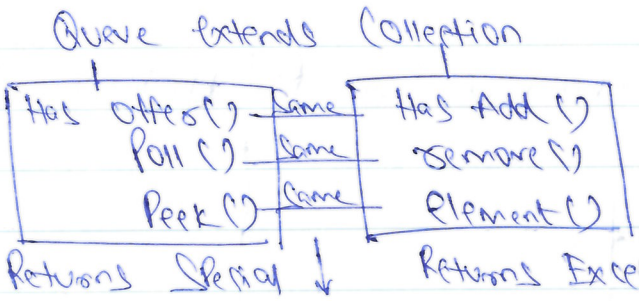
⇒ Object and Thread itself in Queue Hence called Dual Queue or blocking Queue.

transfer	Put	tryTransfer
Blocking	Normal	Depends if consumer present

* hasWaitingConsumerCount() → return blocking count
* hasWaitingConsumer → true/false.

Exchanger: Exchange values b/w 2 Threads.
Buffer copy in NIO class.

Queue Interfaces:
- Queue
- BlockingQueue
  - DeQueue (Pronounced as Deck)
  - BlockingDeQueue



* → Problem with Stack class Java
  - Extends Vector, So Concurrent
  - along with Push and Pop() it has remove() and add(): Coming from vectors.

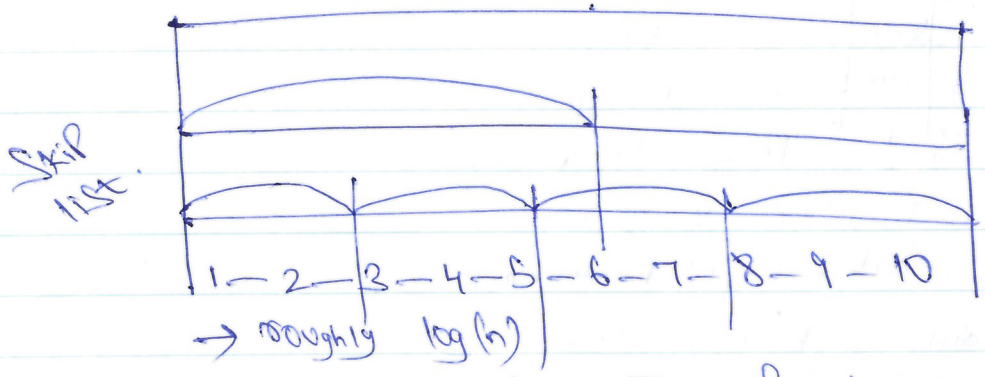
* "fastest blocking Queue" — linkedBlockingQueue (JDK 5 & 6)
  linkedTransferQueue (JDK 7)
* "fastest blocking Queue" — ArrayBlockingQueue()
  Bounded — linkedBlockingQueue() with Best size.
  multiple Producers
  multiple Consumers

ArrayBlockingQueue — Contiguous locations
↓ — Not good with multiple
[Producer & Consumer Producers and Consumers.
is one]

PriorityQueue - Heap Structure.
Data is Not Sorted, only Sorted when
You get().

ConcurrentSkipList (HBase uses this)

TreeMap, TreeSet — implemented Using Red Black Tree



→ Roughly log(n)
→ Not exact Binary Tree, Probabilistic Tree.
→ Real world multi-Thread Performance
Skip list is the winner (Hands Down)

ConcurrentSkipListMap
ConcurrentSkipListSet } Best in Concurrent env
(JDK 6)

Collections.newSetFromMap(new ConcurrentHashMap())
↳ will give a ConcurrentHashSet.
Non BlockingHashSet (Highly Scalable lib)

Executors: (Thread Pool)
- Asynchronous Task Execution
- Runnable or Callable Tasks

new Thread(aRunnable).start() → Never do this, its an anti Pattern.
↳ Use Executors:
- Executors factory class
- ThreadPoolExecutor

OpenHFT → library uses → Java Thread Affinity
↓ Make native commands like CAS.

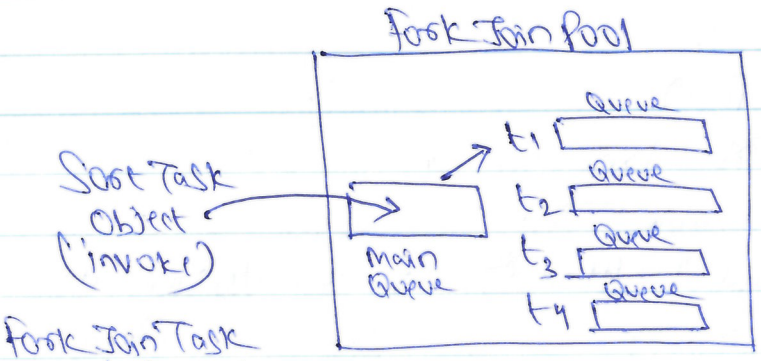
- Thread factory (creates new threads.
- Queue is used internally.

Fork and Join:

* Frame work for - for loops in Parallel.
Sorting in Parallel.
Iterations in Parallel.

JDK 8 - Triggers using lambda

- Divide individual task into multiple pieces.
(Divide & conquer)



Fork Join Task
↓ (compute() method)
Recursive Action
↓
Sort Task

→ ThreadPool.invoke(new SortTask(...))

Weak HashMap — No Concurrent version

(Custom) ConcurrentHashMap with weak keys.
↓
Weak keys
Strong
Soft
Identity

JSR 166 with soft keys
Class not in JDK.

IdentityHashMap

JSR 166 extra Jars downloadable

* CopyOnWriteArrayList
CopyOnWriteArraySet
] Special Purpose Class
↘ Use when Traversal are more and modifications are less.

☆ No non-blocking list — list has indexing methods like get(i), remove(i), Put(i)

⇒ Non blocking and indexing can't go together.

⇒ If you don't want indexing method then queues can be used i.e non-blocking ones.

Locking Mechanism

	CAS	Sync	Re entrant
①	* 90-96 %. this should be choosen i.e The non-blocking wait free Algorithm.	← → (4 to 10 %.) ↓ Compound Statements.	

② Simple Direct Use

Always choose CAS, Sync next and finally re-entrant

Books:

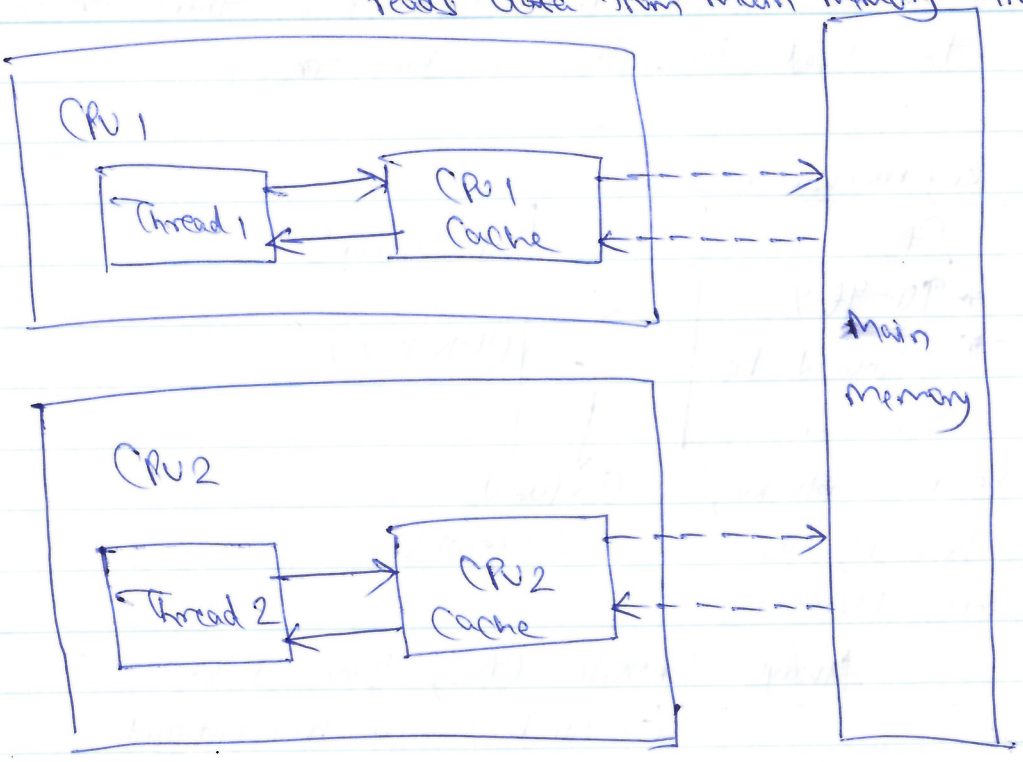
① Java Concurrency in Practice. (JDK 5)
② Concurrent Programming in Java (Doug Lea) — Avoid
③ Art of multi Processor Programming
- Most recent book.
- For experts.

Blogs:

① Jeremy manson
② Cliff click — Cliff c.org — Best out there

JVM - Dave Dice blogs.
Art of Concurrency.blogspot.in

Volatile Variables: With non-volatile variables there are no guarantees about when the JVM reads data from main memory into CPU caches



If your computer contains more than one CPU, each thread may run on a different CPU. That means, each thread may copy the variables into the CPU cache of different CPUs.

* Volatile Use When Single writer - multiple readers
Faster than Synchronization.

(14)

The reading and writing instructions of volatile variables cannot be re ordered by the JVM.

→ JVM may re order instructions for Performance reasons as long as the JVM detects no change in Program behaviour from re-ordering.

Volatile is Not always Enough:

As soon as a thread needs to first read the value of a volatile variable, and based on that value generate a new value for the shared volatile variable, a volatile variable is no longer enough to guarantee correct visibility. The short time gap in between the reading of the volatile variable and writing of its new values creates a race condition

Performance:

→ Reading & writing of volatile variables caused the variable to be read or written to main memory.
→ Reading from and writing to main memory is more expensive than accessing the CPU cache.
→ Accesing volatile variable prevent instruction re-ordering which is a normal Performance enhancement.

* Only use volatile when you need to enforce visibility of variables.

Thread local

Enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to a ThreadLocal variable, then the two threads cannot see each others' ThreadLocal variables.

⇒ For example: Check the example program which clearly shows that even though two threads work on the same shared runnable method, the values are always different.

wait(), notify() and notifyAll()

(1) wait and notify from a synchronized block. This is mandatory! A Thread cannot call wait(), notify(), notifyAll() without holding the lock on the object the method is called on. If it does, an IllegalMonitorStateException is thrown.

(2) Once a thread calls wait() it releases the lock it holds on the monitor object. This allows other threads to call wait() or notify() too.

(3) Once a thread is awakened it cannot exit the wait() call until the thread calling notify() has left its synchronized block.
i.e Awakened thread must re obtain the lock on the monitor object.

Spurious wake ups A.k.A Spin locks.

→ For some reasons it is possible for threads to wake up even if notify() and notifyAll() has not been called. This is called as Spurious wake ups.

→ To guard against this Spurious wakeups the Signal member variable is checked inside a while loop instead of for loops. Such a while loop is called Spin lock.

Dead lock:

① lock Ordering: All locks are always taken in the same order by any thread, dead locks cannot occur.
Example:

Thread 1: lock A
lock B
Thread 2: wait for A
lock C
Thread 3: wait for A
wait for B
wait for C

② Lock Time Out

③ Deadlock Detection: Every time a thread takes a lock it is noted in a data structure (map, graph) of threads & locks. Additionally, whenever a thread requests a lock this is also noted in this data struct

This way after a dead lock is detected:

i) either release all locks, back off, wait a random amount of time and retry.

ii) Better option is to determine or assign a priority of the threads so that only one (or a few) thread backs off.

Semaphore

→ Thread Sync construct used to send signals b/w threads or to avoid/guard critical sections.

→ When there is no need for a re-entrant behaviour we should use Semaphore otherwise Semaphore will cause a deadlock.

lock	vs	Semaphore
i) lock is binary		i) Semaphore is counting
ii) lock is re entrant		ii) Semaphore is not re-entrant.

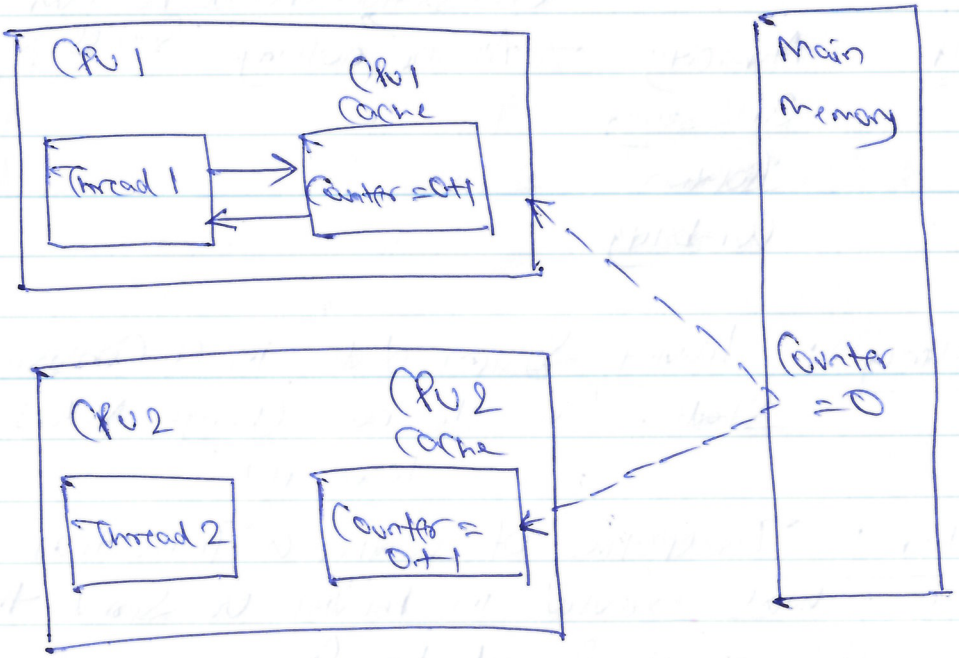
Using Semaphores as locks:

It is possible to use a bounded Semaphore as a lock. To do so, set the upper bound to 1, and have the call to take() and release() guard the critical section.

Ex: BoundedSemaphore Semaphore = new BoundedSemaphore(1);

```
Semaphore.take();
try {
    // Critical Section.
} finally {
    Semaphore.release();
}
```

When Volatile is Not always Enough :



Imagine Thread 1 reads a shared variable with the value 0 into its CPU cache, increment it to 1 and not write the changed value back into main memory.

Thread 2 could then read the same counter value from main memory i.e value 0 into its own CPU Cache. Thread 2 also can increment counter to 1 and not write it back to main memory.

Hence Thread 1 & Thread 2 are out of sync no

Spring Transactions.

ACID
- Atomicity — All or nothing. (Transaction should be completly sucess or no (rollback))
- Consistency
- Isolation
- Durability

Consistency : leaving system data in a consistent state. (ie do not change values & leave it)

Isolation : Irrespective of weather a Transaction was executed in Parallel or Serial the end result should be same.

Durability : Even if system crashes what ever transactions were done Previously should be saved Proberly i.e we should be able to recover to that state.

Dirty Read. Transaction reads a value which is updated by another transaction which has not yet completed. Eg : Account Transfer.

Solution : Have some kind of a lock on row.

This will be a problem if the ongoing transaction rolled back & then we are having stale values.

✗ Column level lock. ⟹ WRITE LOCK.

⇒ Can be solved using a read lock so that Transaction 2 has to wait.

Non-Repeatable reads: (Related to Updates)

id	name	age
1	Sunil	28

		Age
Select * from Users where id=1	Transaction 1 (Step 1)	28
Update Users Set age=21 where id=1	Transaction 2 (Step 1)	21
Select * from Users where id=1	Transaction 1 (Step 2)	21

** Problem: In the same Transaction 1 we are getting 2 different values for age with same Query which is difficult to handle.

Phantom reads: (Related to inserts)

Same as above but here instead of updated records, transaction got more rows due to inserts and hence its not desirable output.

* Range lock is created i.e a range lock for range of age b/w 10 & 30.

ISOLATION LEVEL

	Dirty	Non-Repeatable	Phantom
Read uncomitted	NO	NO	NO
Read Commited	SOLVED	NO	NO
Repeatable reads	SOLVED	SOLVED	NO
Serializable	SOLVED	SOLVED	SOLVED