# How ConcurrentHashMap is implemented in Java

ConcurrentHashMap allows multiple readers to read concurrently without any blocking by partitioning Map into different parts based on concurrency level and locking only a portion of Map during updates. Default concurrency level is 16, and accordingly Map is divided into 16 part and each part is governed with different lock. This means, 16 thread can operate on Map simultaneously, until they are operating on different part of Map. This makes ConcurrentHashMap high performance despite keeping thread-safety intact. Though, it comes with caveat. Since update operations like put(), remove(), putAll() or clear() is not synchronized, concurrent retrieval may not reflect most recent change on Map.

In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries. Another important point to remember is iteration over CHM, Iterator returned by keySet of ConcurrentHashMap are weekly consistent and they only reflect state of ConcurrentHashMap and certain point and may not reflect any recent change. Iterator of ConcurrentHashMap's keySet area also fail-safe and doesn’t throw ConcurrentModificationExceptoin..

Default concurrency level is 16 and can be changed, by providing a number which make sense and work for you while creating ConcurrentHashMap. Since concurrency level is used for internal sizing and indicate number of concurrent update without contention, so, if you just have few writers or thread to update Map keeping it low is much better. ConcurrentHashMap also uses ReentrantLock to internally lock its segments.

# Summary

1. **ConcurrentHashMap allows concurrent read and thread-safe update operation**.

2. **During update operation, ConcurrentHashMap only lock a portion of Map instead of whole Map**.

3. Concurrent update is achieved by internally dividing Map into small portion which is defined by concurrency level.

4. All operations of ConcurrentHashMap are thread-safe.

5. Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.

6. Iterator returned by ConcurrentHashMap is weekly consistent, fail safe and never throw ConcurrentModificationException. Iterator does not support remove() method ConcurrentHashMap.

8. ConcurrentHashMap doesn't allow null as key or value.

9. You can use ConcurrentHashMap in place of Hashtable but with caution as CHM doesn't lock whole Map.

10. During putAll() and clear() operations, concurrent read may only reflect insertion or deletion of some entries.

**ThreadPoolExecutor**

Thread pools address two different problems: **they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead**, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.

**How AtomicInteger or AtomicLong works in Java**

First consider this code:

public class MyApp {

private volatile int count = 0;

public void upateVisitors() {

++count; //increment the visitors count

}

}

Actually the problem is simply **marking count as volatile does not guarantee atomicity and ++count is not an atomic operations**. Can we solve this problem if we mark the method itself synchronized as shown below:

public class MyApp {

private int count = 0;

public **synchronized** void upateVisitors() {

++count; //increment the visitors count

}

}

Although the above problem solves the atomicity and visibility but it makes use of locking and that introduces lot of delay and overhead. This is very expensive way of making things work. To overcome these problems atomic constructs were introduced. If we make use of an AtomicInteger to track the count it will work.

public class MyApp {

private **AtomicInteger count = new AtomicInteger(0);**

public void upateVisitors() {

count.incrementAndGet(); //increment the visitors count

}

}

Check the following code in AtomicLong class:

public final long incrementAndGet() {

for (;;) {

long current = get();

long next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

In JDK 8 the above code has been changed to a single intrinsic:

**public final long incrementAndGet() {**

**return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L;**

**}**

**public** **final** **boolean** [http://grepcode.com/static/app/images/1x1.gif](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java)compareAndSet(**long** expect, **long** update) { //As for JDK 7

**return** [**unsafe**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java#AtomicLong.0unsafe)**.**[**compareAndSwapLong**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/sun/misc/Unsafe.java#Unsafe.compareAndSwapLong%28java.lang.Object%2Clong%2Clong%2Clong%29)**(this,**[**valueOffset**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java#AtomicLong.0valueOffset)**, expect, update);**

}

[Asynchronous VS Multithreading](https://stackoverflow.com/questions/34680985/what-is-the-difference-between-asynchronous-programming-and-multithreading)

<https://stackoverflow.com/questions/34680985/what-is-the-difference-between-asynchronous-programming-and-multithreading>

Many people are taught that multithreading and asynchrony are the same thing, but they are not.

An analogy usually helps. You are cooking in a restaurant. An order comes in for eggs and toast.

* Synchronous: you cook the eggs, then you cook the toast.
* Asynchronous, single threaded: you start the eggs cooking and set a timer. You start the toast cooking, and set a timer. While they are both cooking, you clean the kitchen. When the timers go off you take the eggs off the heat and the toast out of the toaster and serve them.
* Asynchronous, multithreaded: you hire two more cooks, one to cook eggs and one to cook toast. Now you have the problem of coordinating the cooks so that they do not conflict with each other in the kitchen when sharing resources. And you have to pay them.

**Threading is about workers; asynchrony is about tasks**. In multithreaded workflows you assign tasks to workers.

<https://blog.slaks.net/2014-12-23/parallelism-async-threading-explained/>

Multi-threading means running more than one thread of execution at a time. In this model, all operations are still synchronous, but the CPU will execute multiple threads of synchronous operations at the same time.

Multi-threading makes most sense when calling multiple (and independent) CPU-bound operations, on a multi-core processor. For example, a program that independently analyzes every pixel in an image could divide the image into one strip for each CPU core, then analyze each strip in its own thread at the same time.

If you have strictly CPU-bound work, multi-threading is your only option. The whole point of asynchrony is to leave the CPU (and your thread(s)) free, so that you can run other code while waiting for the asynchronous operations to finish. If your work is CPU-bound, running it asynchronously does not make sense.

<https://stackoverflow.com/questions/4844637/what-is-the-difference-between-concurrency-parallelism-and-asynchronous-methods>

*Concurrency means multiple tasks which start, run, and complete in overlapping time periods, in no specific order. Parallelism is when multiple tasks OR several part of a unique task literally run at the same time, e.g. on a multi-core processor.*

Remember that Concurrency and parallelism are NOT the same thing.

**Differences between concurrency vs. parallelism**

Concurrency is when two tasks can start, run, and complete in overlapping time periods. Parallelism is when tasks literally run at the same time, eg. on a multi-core processor.

Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.

An application can be concurrent – but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at same time instant.

An application can be parallel – but not concurrent, which means that it processes multiple sub-tasks of a task in multi-core CPU at same time.

An application can be neither parallel – nor concurrent, which means that it processes all tasks one at a time, sequentially.

An application can be both parallel – and concurrent, which means that it processes multiple tasks concurrently in multi-core CPU at same time.

<http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html>

**Parallelism**

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time. To achieve true parallelism your application must have more than one thread running, or at least be able to schedule tasks for execution in other threads, processes, CPUs, graphics cards etc.

**Concurrency**

Concurrency is related to how an application handles multiple tasks it works on. An application may process one task at at time (sequentially) or work on multiple tasks at the same time (concurrently).

Parallelism on the other hand, is 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 can be completed in parallel.

An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel.

**Concurrency & Parallelism**

**Concurrency**

Consider you are given a task of singing and eating at the same time. At a given instance of time either you would sing or you would eat as in both cases your mouth is involved. So in order to do this, you would eat for some time and then sing and repeat this until your food is finished or song is over. So you performed your tasks concurrently.

Concurrency means executing multiple tasks at the same time but not necessarily simultaneously. In a concurrent application, two tasks can start, run, and complete in overlapping time periods i.e Task-2 can start even before Task-1 gets completed.

In the computer science world, the way how concurrency is achieved in various processors is different. In a single core environment (i.e your processor is having a single core), concurrency is achieved via a process called [context-switching](http://www.linfo.org/context_switch.html). If its a multi-core environment, concurrency can be achieved through parallelism.

**Parallelism**

Consider you are given two tasks of cooking and speaking to your friend over the phone. You could do these two things simultaneously. You could cook as well as speak over the phone. Now you are doing your tasks parallelly.

Parallelism means performing two or more tasks simultaneously. Parallel computing in computer science refers to the process of performing multiple calculations simultaneously.

**How is concurrency related to parallelism?**

* Concurrency and Parallelism refer to computer architectures which focus on how our tasks or computations are performed.
* ***In a single core environment, concurrency happens*** with tasks executing over same time period via context switching i.e at a particular time period, only a single task gets executed.
* ***In a multi-core environment, concurrency can be achieved via parallelism***in which multiple tasks are executed simultaneously.

**Threads & Processes**

**Threads**

Threads are a sequence of execution of code which can be executed independently of one another. ***It is the smallest unit of tasks that can be executed by an OS. A program can be single threaded or multi-threaded.***

**Process**

A process is an instance of a running program. ***A program can have multiple processes. A process usually starts with a single thread i.e a primary thread but later down the line of execution it can create multiple threads.***

**Synchronous and Asynchronous**

**Synchronous**

Imagine you were given to write two letters one to your mom and another to your best friend. You cannot at the same time write two letters unless you are a pro ambidextrous.

In a synchronous programming model, tasks are executed one after another. Each task waits for any previous task to complete and then gets executed.

**Asynchronous**

Imagine you were given to make a sandwich and wash your clothes in a washing machine. You could put your clothes in the washing machine and without waiting for it to be done, you could go and make the sandwich. Here you performed these two tasks asynchronously.

In an asynchronous programming model, when one task gets executed, you could switch to a different task without waiting for the previous to get completed.

**Is Volatile keyword required in case of Reentrant Lock?**

<https://stackoverflow.com/questions/1570589/is-the-volatile-keyword-required-for-fields-accessed-via-a-reentrantlock>

Answer by **Jon Skeet: It's safe without volatility**. ReentrantLock implements Lock.

**Is it necessary to use volatile with AtomicInteger?**

<https://stackoverflow.com/questions/14338533/atomicinteger-and-volatile>

**Answer by Jon Skeet**

**Atomic\* actually gives *both* atomicity and volatility.** **So when you call (say) AtomicInteger.get(), you're guaranteed to get the *latest* value. This is documented in the java.util.concurrent.atomic**[**package documentation**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html)**:**

**get has the memory effects of reading a volatile variable.**

**set has the memory effects of writing (assigning) a volatile variable.**

**Difference between Atomic, Volatile and Synchronized**

<https://stackoverflow.com/questions/9749746/what-is-the-difference-between-atomic-volatile-synchronized>

**No synchronization**

private int counter;

public int getNextUniqueIndex() {

return counter++;

}

It basically reads value from memory, increments it and puts back to memory. This works in single thread but nowadays, in the era of multi-core, multi-CPU, multi-level caches it won't work correctly. **First of all it introduces race condition (several threads can read the value at the same time), but also visibility problems.** **The value might only be stored in "*local*" CPU memory (some cache) and not be visible for other CPUs/cores (and thus - threads).** This is why many refer to *local copy* of a variable in a thread. It is very unsafe. Consider this popular but broken thread-stopping code:

private boolean stopped;

public void run() {

while(!stopped) {

//do some work

}

}

public void pleaseStop() {

stopped = true;

}

Add volatile to stopped variable and it works fine - if any other thread modifies stopped variable via pleaseStop() method, you are guaranteed to see that change immediately in working thread's while(!stopped) loop. BTW this is not a good way to interrupt a thread either, see: [How to stop a thread that is running forever without any use](https://stackoverflow.com/questions/6410721) and [Stopping a specific java thread](https://stackoverflow.com/questions/7786305).

**AtomicInteger**

private AtomicInteger counter = new AtomicInteger();

public int getNextUniqueIndex() {

return counter.getAndIncrement();

}

**The AtomicInteger class uses CAS (**[**compare-and-swap**](http://en.wikipedia.org/wiki/Compare-and-swap)**) low-level CPU operations (no synchronization needed!)** They allow you to modify a particular variable only if the present value is equal to something else (and is returned successfully). So when you execute **getAndIncrement()** it actually runs in a loop (simplified real implementation):

**int current;**

**do {**

**current = get();**

**} while(!compareAndSet(current, current + 1));**

So basically: read; try to store incremented value; if not successful (the value is no longer equal to current), read and try again. **The compareAndSet() is implemented in native code (assembly).**

**Volatile without synchronization**

private volatile int counter;

public int getNextUniqueIndex() {

return counter++;

}

**This code is not correct. It fixes the visibility issue (volatile makes sure other threads can see change made to counter) but still has a race condition.** This has been [explained](https://stackoverflow.com/questions/25168062/why-is-i-not-atomic) multiple times: pre/post-incrementation is not atomic. **The only side effect of volatile is "*flushing*" caches so that all other parties see the freshest version of the data.** This is too strict in most situations; that is why volatile is not default.

**Volatile without synchronization (2)**

**volatile int i = 0;**

**void incIBy5() {**

**i += 5;**

**}**

The same problem as above, but even worse because i is not private. The race condition is still present. Why is it a problem? **If, say, two threads run this code simultaneously, the output might be + 5 or + 10**. **However, you are guaranteed to see the change.**