Thread-safe maps

Thread-safe maps are quite popular. Imagine you implement a cache based on a map. You have two writer threads and one reader thread.

If you use HashMap, there is no guarantee that the reader thread will see updates made by a writer thread right after these changes were made.

Another problem is that some of writer thread's updates may be lost, or that the changed data will turn out in an inconsistent state.

So if your map is meant to be used by several threads, there are two paths for you: ConcurrentHashMap and Collections.synchronizedMap. Let's start with the firstborn!

Collections.synchronizedMap

Collections.synchronizedMap first appeared in Java 2 and it is a static method of the standard java.util.Collections class.
The synchronizedMap method inputs a classic Java map and returns a thread-safe map. The returned map is backed by the input map and contains the same elements:

```
Map<String, String> map = new HashMap<>();
map.put("a", "Apple");
map.put("b", "Banana");

Map<String, String> synchronizedMap = Collections.synchronizedMap(map); // contains the same as the input map
```

Use only the returned map, otherwise, you won't be able to ensure thread safety.

Let's consider the following example. Here we have two threads: writer which adds non-negative numbers to the map, and the main thread which adds negative numbers to the map.

```
public static void main(String[] args) throws InterruptedException {
    Map<Integer, String> synchronizedNumbers =
Collections.synchronizedMap(new HashMap<>());

    Thread writer = new Thread(() ->
addPositiveNumbers(synchronizedNumbers));
    writer.start();

    addNegativeNumbers(synchronizedNumbers); // add negative numbers from the main thread
```

```
    writer.join(); // wait for the writer thread

    System.out.println(synchronizedNumbers.size()); // the size is always
200_000
}

private static void addPositiveNumbers(Map<Integer, String> target) {
    for (int i = 0; i < 100_000; i++) {
        target.put(i, "Number is " + i);
    }
}

private static void addNegativeNumbers(Map<Integer, String> target) {
    for (int i = -100_000; i < 0; i++) {
        target.put(i, "Number is " + i);
    }
}
```

Everything works as intended and the resulting size of the map is always the same. However, if you experiment and use a classic non-synchronized map instead of a synchronizedMap , the size of the returned map can be totally unpredictable.
Here are the main features of the Collections.synchronizedMap to keep in mind:

- The synchronization is performed by an object.
- The methods (and as a result, all operations) of a synchronizedMap are protected by a lock, which provides thread-safe access.
- To iterate over a synchronizedMap you need to use a **synchronized block**:

```
synchronized (synchronizedMap) {
    Iterator<String> iterator = synchronizedMap.keySet().iterator();
    while (iterator.hasNext()) {
        // do important things
        iterator.next();
    }
}
```

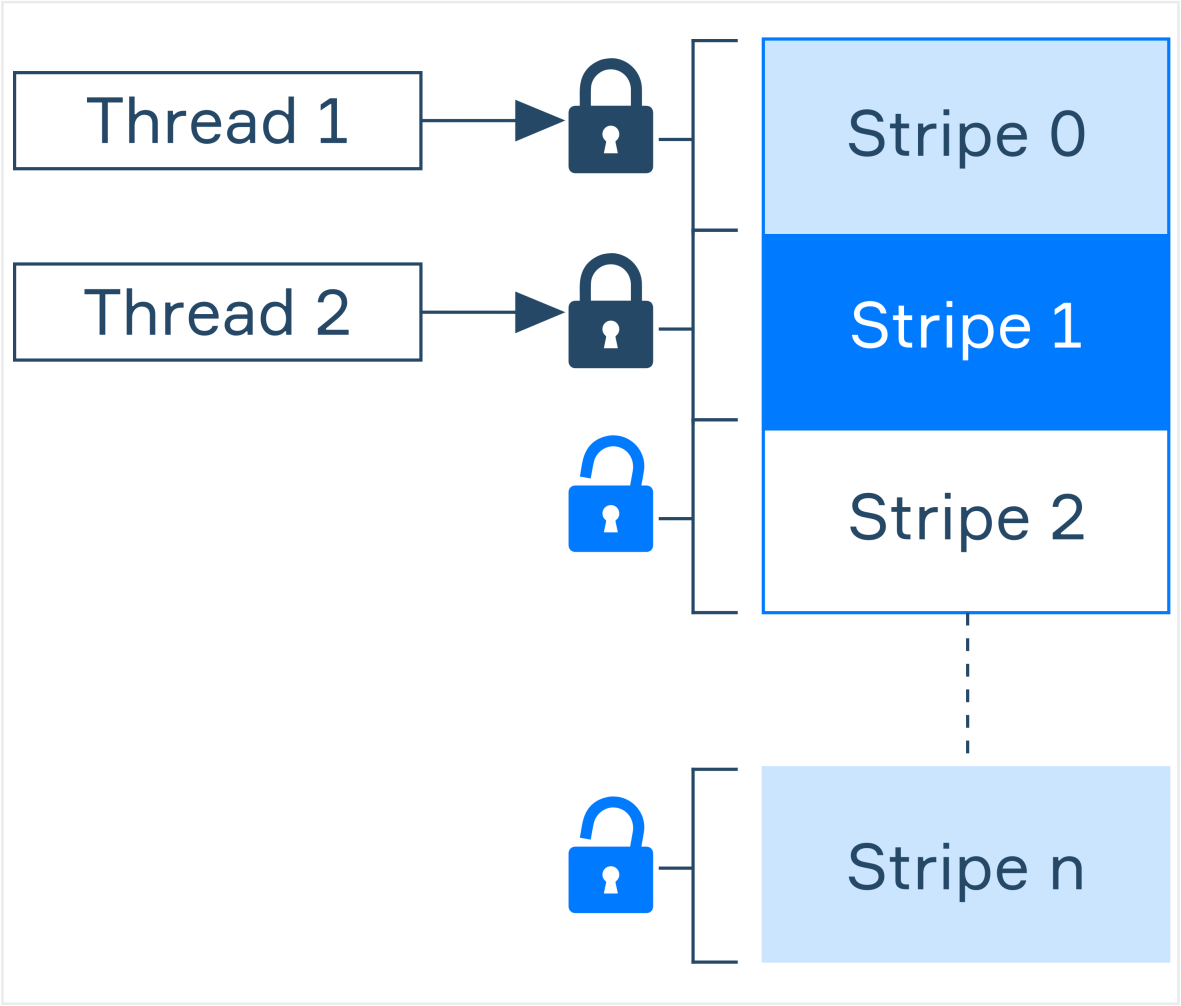
Without the synchronized block the iterator may throw ConcurrentModificationException. This will happen if one thread is iterating over a map and another thread is trying to modify our map.

- Only one thread at a time has access to the map, other threads are blocked.

ConcurrentHashMap

The ConcurrentHashMap class was created to allow multiple threads access to the same map without blocking each other and, as a consequence, to increase

their performance. You can find ConcurrentHashMap in the java.util.concurrent package. At this point, you may wonder: how come the ConcurrentHashMap class was made to be so amazing? The secret is in **Lock Striping**. This technique means that the lock mechanism occurs only on separate stripes (or buckets), but not on the whole data structure. When a thread accesses the stripe, it locks only that stripe, leaving other stripes available. You can see what it looks like in the diagram below.



Lock Striping technique

In the history of Java, the internal realization of the ConcurrentHashMap class has changed quite a lot. It has been the way we know it since Java 8. Let's quickly look into what it was like until Java 7 and how it has changed later on.

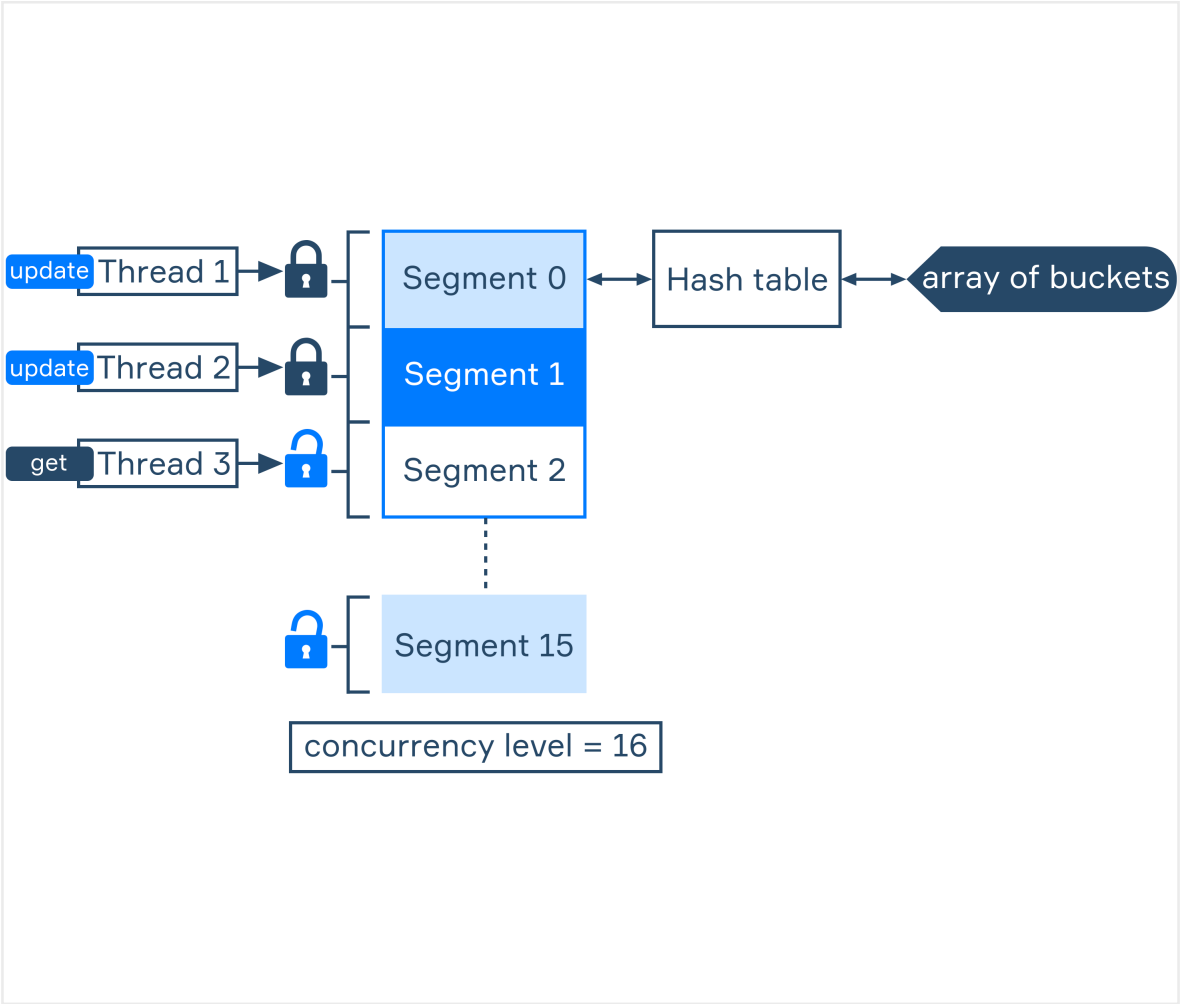
Early implementation

First, there is something both of these versions have in common: **retrieval operations** (such as get) do not block and give the results of the most recent update operations.

Until Java 7, ConcurrentHashMap consisted of several Segments, where one Segment is a specialized and concurrently readable hash table. The number of Segments is given in a constructor of ConcurrentHashMap and called concurrencyLevel. By default, concurrencyLevel is 16.

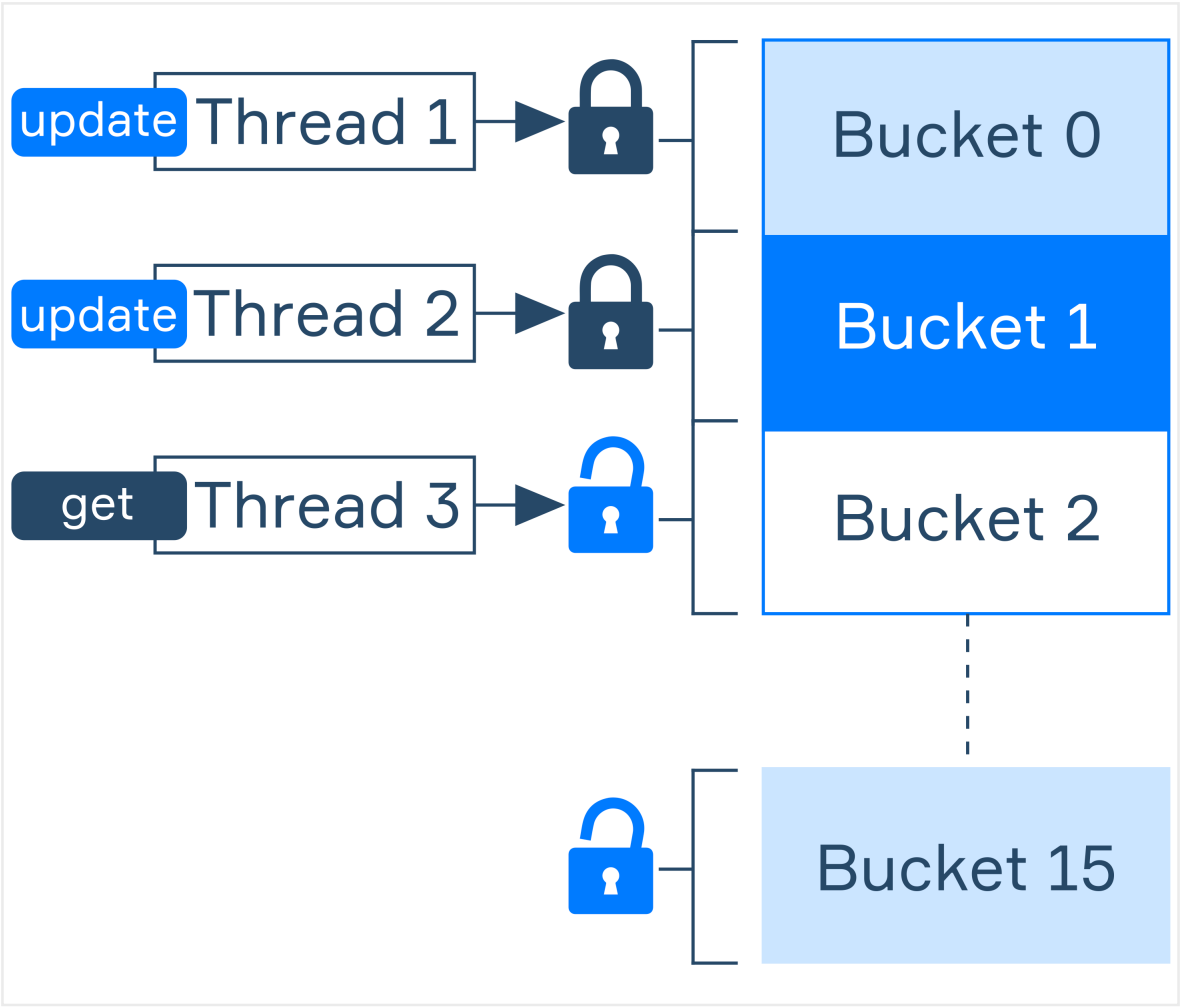
As you can guess from the name, concurrencyLevel represents how many threads can update data at the same time. Also, concurrencyLevel is fixed, so it is recommended to choose it to accommodate the exact number of threads that will ever modify the ConcurrentHashMap at the same time.

concurrencyLevel and Segments are still with us but we only use them for combining with the previous versions of ConcurrentHashMap.



Modern implementation

Since Java 8 ConcurrentHashMap contains 16 buckets by default, the same number as in the HashMap. There every bucket is a list with key-value nodes. As you can see, the structure looks more compact:



Here are the main features of ConcurrentHashMap:

- The synchronization is performed at the bucket level.
- Retrieval operations don't require a block.
- The iterator won't throw ConcurrentModificationException when one thread is iterating over a map and another thread is trying to modify it. However, there is no guarantee that the iterator will see changes made by another thread.
- The number of threads working simultaneously with the ConcurrentHashMap can increase if the size of the ConcurrentHashMap has been increased.

Note that ConcurrentHashMap doesn't allow using null as a key or a value

ConcurrentHashMap example

Here is an example of ConcurrentHashMap usage:

```
public static void main(String[] args) throws InterruptedException {

    Map<Integer, String> map = new ConcurrentHashMap<>();
```

```
    Thread writer1 = new Thread(() -> addPositiveNumbers(map));
    Thread writer2 = new Thread(() -> addNegativeNumbers(map));

    writer1.start();
    writer2.start();

    // Here, in the main thread, we can use Iterator, retrieve values or update the
map

    writer1.join(); // wait for writer1 thread
    writer2.join(); // wait for writer2 thread

    System.out.println(map.size()); // the result is always the same
}

private static void addPositiveNumbers(Map<Integer, String> target) {
    for (int i = 0; i < 100_000; i++) {
        target.put(i, "Number is " + i);
    }
}

private static void addNegativeNumbers(Map<Integer, String> target) {
    for (int i = -100_000; i < 0; i++) {
        target.put(i, "Number is " + i);
    }
}
```

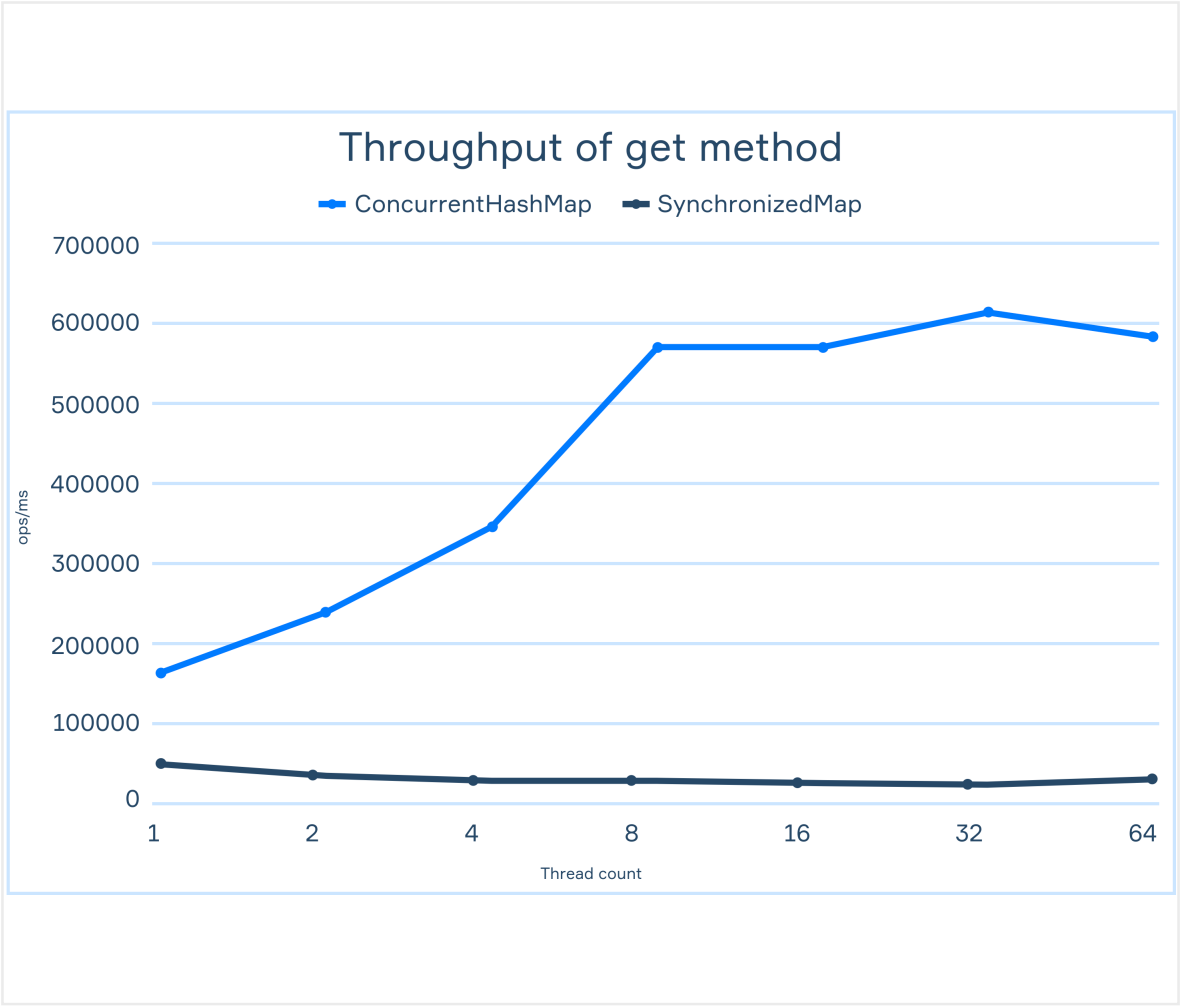
Performance comparison
This all sounds great, but facts speak louder than words, right? Let's compare the throughput of put and get methods
for ConcurrentHashMap and synchronizedMap!
To make it more interesting, we will test the throughput for JDK 11 with random keys. Both maps will have the same number of elements — 1000.

Disclaimer: the values of the charts shown are approximate. They just compare the two approaches. Do not focus on numbers, it is the trend that matters!

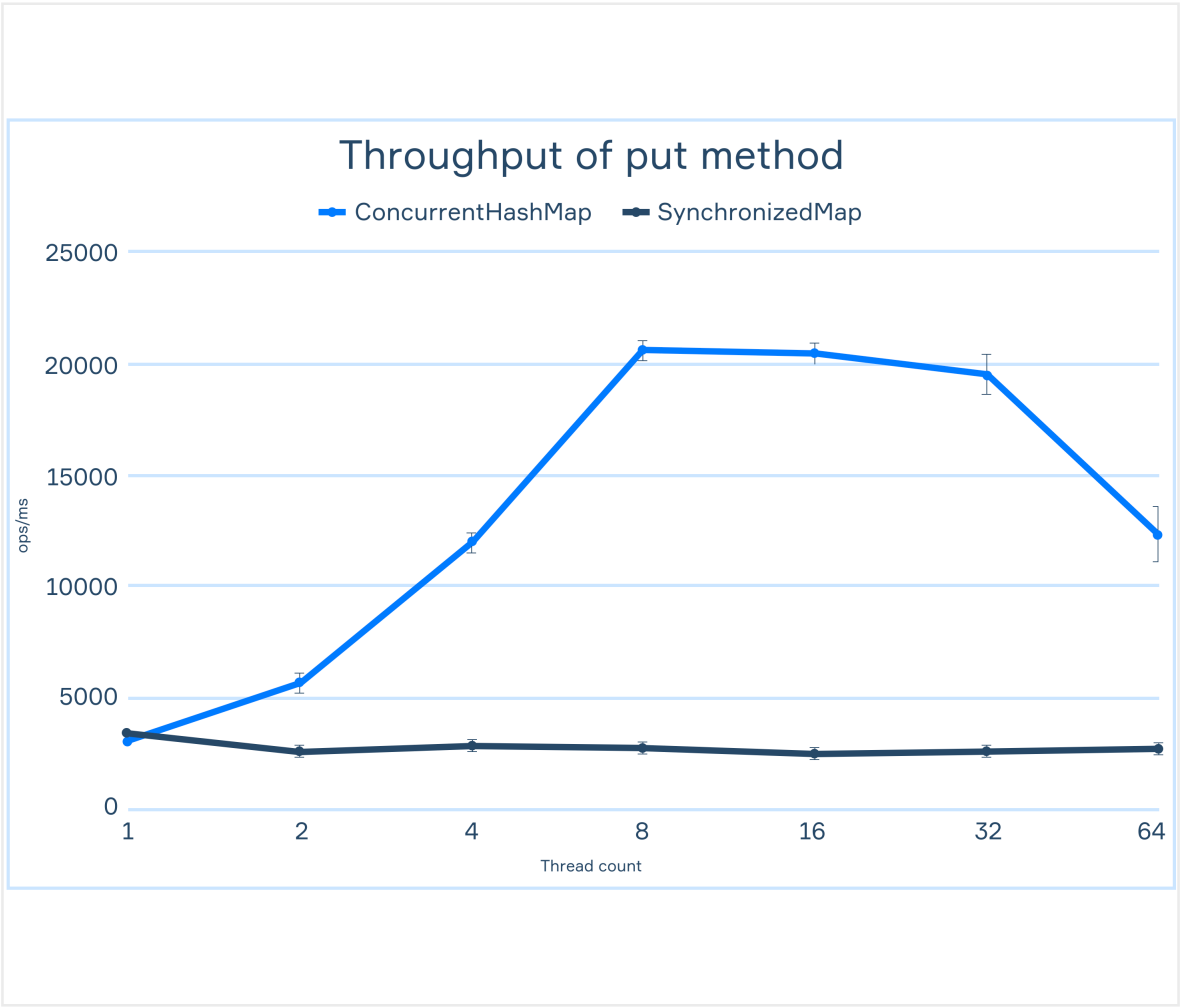
The representative range here is [1 - 8] threads. After that, other effects come into play.

As you see, ConcurrentHashMap has a much better performance in both

scenarios. The non-blocking get operation makes ConcurrentHashMap faster even for a single thread. Then the throughput gradually increases, while the values of SynchronizedMap remain unchanged.



The put operation in the case of a single thread shows the same throughput, but then the situation looks similar to the get operation.



Conclusion

There are two thread-safe implementations of a Map interface in the standard library: ConcurrentHashMap and Collections.synchronizedMap. They use totally different ideas to achieve the same result. Collections.synchronizedMap inputs a classic Java map and wraps its methods by synchronized blocks. ConcurrentHashMap relies on a lock-striping approach. It allows multiple threads to access different parts of the same map without blocking each other. In practice, ConcurrentHashMap is a more preferred choice because of better performance. It performs both the get and put operations faster than Collections.synchronizedMap