# What is “non-blocking” concurrency and how is it different than normal concurrency?

1. What is "non-blocking" concurrency and how is it different than normal concurrency using threads? Why don't we use non-blocking concurrency in all the scenarios where concurrency is required? Is there overhead for using non-blocking concurrency?
2. I have heard that non-blocking concurrency is available in Java. Are there any particular scenarios where we should use this feature?
3. Is there a difference or advantage to using one of these methods with a collection? What are the trade-offs?

Example for Q3:

```
class List
{
    private final ArrayList<String> list = new ArrayList<String>();

    void add(String newValue)
    {
        synchronized (list)
        {
            list.add(newValue);
        }
    }
}
```

vs.

```
private final ArrayList<String> list = Collections.synchronizedList();
```

The questions are more from a learning/understanding point of view. Thanks for attention.

java concurrency

edited Jun 29 '12 at 18:09
Gray
80.8k 14 174 241

asked May 13 '10 at 3:14
codeObserver
2,463 11 51 96

1 some info here en.wikipedia.org/wiki/Non-blocking_synchronization – Dimitar May 13 '10 at 3:17

What do you mean by your first question, what exactly are you comparing non-blocking concurrency to? – Matti Virkkunen May 13 '10 at 3:18

private final ArrayList<String> list = Collections.synchronizedMap() will not compile for two reasons… – harschware May 13 '10 at 3:30

## 8 Answers

> What is Non-blocking Concurrency and how is it different.

Formal:

> In computer science, non-blocking synchronization ensures that threads competing for a

> shared resource do not have their execution indefinitely postponed by mutual exclusion. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress; wait-free if there is also guaranteed per-thread progress. (wikipedia)

Informal: One of the most advantageous feature of non-blocking vs. blocking is that, threads does not have to be suspended/waken up by the OS. Such overhead can amount to 1ms to a few 10ms, so removing this can be a big performance gain. In java, it also means that you can choose to use non-fair locking, which can have much more system throughput than fair-locking.

> I have heard that this is available in Java. Are there any particular scenarios we should use this feature

Yes, from Java5. In fact, in Java you should basically try to meet your needs with java.util.concurrent as much as possible (which happen to use non-blocking concurrency a lot, but you don't have to explicitly worry in most cases). Only if you have no other option, you should use synchronized wrappers (.synchronizedList() etc.) or manual `synchronize` keyword. That way, you end up most of the time with more maintainable, better performing apps.

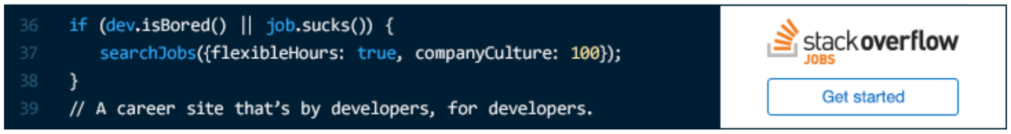
Non-blocking concurrency is particularly advantageous when there is a lot of contention. You can't use it when you need blocking (fair-locking, event-driven stuff, queue with maximum length etc.), but if you don't need that, non-blocking concurrency tends to perform better in most conditions.

> Is there a difference/advantage of using one of these methods for a collection. What are the trade offs

Both have the same behavior (the byte code should be equal). But I suggest to use `Collections.synchronized` because it's shorter = smaller room to screw up!

edited Sep 16 '13 at 7:44

answered May 13 '10 at 4:02
Enno Shioji
16.9k 6 51 95


```
if (dev.isBored() || job.sucks()) {
   searchJobs({flexibleHours: true, companyCulture: 100});
}
// A career site that's by developers, for developers.
```

> 1) What is Non-blocking Concurrency and how is it different.

A task (thread) is non-blocking when it doesn't cause other tasks (threads) to wait until the task is finished.

> 2) I have heard that this is available in Java. Are there any particular scenarios we should use this feature

Java supports at its own multithreading. You can take benefit of it to run multiple tasks concurrently. When well written and implemented, this may speed up the program when running at a machine with multiple logical CPU's. To start, there are `java.lang.Runnable` and `java.lang.Thread` as low level concurrency implementations. Then there is high level concurrency in flavor of the `java.util.concurrent` API.

> 3) Is there a difference/advantage of using one of these methods for a collection. What are the trade offs

I would use `Collections#synchronizedList()` , not only because that's more robust and convenient, but also because a `Map` isn't a `List` . There's no need to attempt to homegrow one when the API already offers the facility.

That said, there's a Sun tutorial about Concurrency. I recommend you to get yourself through it.

answered May 13 '10 at 3:30
BalusC
706k 227 2593 2793

> What is Non-blocking Concurrency and how is it different than Normal Concurrency using

> Threads.

Non-blocking concurrency is a different way to coordinate access between threads from blocking concurrency. There is a lot of background (theoretical) material out there, but the simplest explanation (as it seems that you're looking for a simple, hands-on answer), is that non-blocking concurrency does not make use of locks.

> Why dont we use "non-blocking" Concurrency in all the scenarios where concurrency is required.

We do. I'll show you in a bit. But it is true that there aren't always efficient non-blocking algorithms for every concurrency problem.

> are there any overheads for "non-blocking"

Well, there's overhead for any type of sharing information between threads that goes all the way down to how the CPU is structured, especially when you get what we call "contention", i.e. synchronizing more than one thread that are attempting to write to the same memory location at the same time. But in general, non-blocking is faster than blocking (lock-based) concurrency in many cases, especially all the cases where there are well known, simple, lock-free implementation of a given algorithm/data-structure. It is these good solutions that are provided with Java.

> I have heard that this is available in Java.

Absolutely. For starters, all the classes in java.util.concurrent.atomic provide lock-free maintenance of shared variables. In addition, all the classes in java.util.concurrent whose names start with ConcurrentLinked or ConcurrentSkipList, provide lock-free implementation of lists, maps and sets.

> Are there any particular scenarios we should use this feature.

You would want to use the lock-free queue and deque in all cases where you would otherwise (prior to JDK 1.5) use Collections.synchronizedlist, as they provide better performance under most conditions. i.e., you would use them whenever more than one thread is concurrently modifying the collection, or when one thread is modifying the collection and other threads are attempting to read it. Note that the very popular ConcurrentHashMap does actually use locks internally, but it is more popular than ConcurrentSkipListMap because I think it provides better performance in most scenarios. However, I think that Java 8 would include a lock-free implementation of ConcurrentHashMap.

> Is there a difference/advantage of using one of these methods for a collection. What are the trade offs

Well, in this short example, they are exactly the same. Note, however, that when you have concurrent readers and writers, you must synchronize the reads as well as the writes, and Collections.synchronizedList() does that. You might want to try the lock-free ConcurrentLinkedQueue as an alternative. It might give you better performance in some scenarios.

**General Note**

While concurrency is a very important topic to learn, bear in mind that it is also a very tricky subject, where even very experienced developers often err. What's worse, you might discover concurrency bugs only when your system is under heavy load. So I would always recommend using as many ready-made concurrent classes and libraries as possible rather than rolling out your own.

edited Jan 19 '12 at 12:30

answered Jan 18 '12 at 21:58

pron
**1,009** 12 23

---

1) **Non-blocking synchronization** means that the risk of deadlocks is removed. No one thread will wait to get a lock held by another thread "forever".

2) More on non-blocking synchronization algorithms in java.

edited May 13 '10 at 3:29

answered May 13 '10 at 3:22

Lars Andren
**4,866** 4 24 47

> **1]What is Non-blocking Concurrency and how is it different.**

As others have mentioned, Non-blocking is a way of saying deadlock-free (meaning we shouldn't have a condition where progress halts entirely while threads are blocked, waiting for access).

What is meant by 'concurrency' is just that multiple computations are happening at the same time (concurrently).

> **2] I have heard that this is available in Java. Are there any particular scenarios we should use this feature**

You want to use non-blocking algorithms when it is important that multiple threads can access the same resources concurrently, but we aren't as concerned with the order of access or the possible ramifications of interleaving action (more on this below).

> **3] Is there a difference/advantage of using one of these methods for a collection. What are the trade offs**

.

Using the synchronized(list) block ensures that all of the actions performed within the block are seen as atomic. That is to say, as long as we only access the list from synchronized(list) blocks, all updates to the list will appear as if they happened at the same time within the block.

A synchronizedList (or synchronizedMap) object only ensures that individual operations are thread-safe. This means that two inserts will not occur concurrently. Consider the following loop:

```
for(int i=0; i < 4; i++){
    list.add(Integer.toString(i));
}
```

If the list in use was a synchronizedList and this loop was executed on two different threads, then we may end up with {0,0,1,2,1,3,2,3} in our list, or some other permutation.

Why? Well, we are guaranteed that thread 1 will add 0-3 in that order and we are guaranteed the same of thread 2, however we have no guarantee of how they will interleave.

If, however, we wrapped this list in a synchronized(list) block:

```
synchronized(list){
    for(int i=0; i < 4; i++){
        list.add(Integer.toString(i));
    }
}
```

We are guaranteed that the inserts from thread 1 and thread 2 will not interleave, but they will occur all at once. Our list will contain {0,1,2,3,0,1,2,3}. The other thread will block, waiting on list, until the first thread completes. We have no guarantee which thread will be first, but we are guaranteed it will finish before the other begins.

So, some trade-offs are:

- With a syncrhonizedList, you can insert without explicitly using a synchronized block.
- A syncrhonizedList might give you a false sense of security, since you may naively believe successive opertaions on one thread to be atomic, when only individual operations are atomic
- Using a syncrhonized(list) block must be done with care, because we are in a position to create deadlock (more below).

We can create a deadlock when two (or more) threads are each waiting for a subset of resources held by another. If, for example, you had two lists: userList and movieList.

If thread 1 first acquires the lock to userList, then movieList, but thread two performs these steps in reverse (acquires the lock to movieList before userList), then we have opened ourself up for deadlock. Consider the following course of events:

1. Thread 1 gets lock to userList
2. Thread 2 gets lock to movieList
3. Thread 1 tries to get lock to movieList, waits on Thread 2 to release
4. Thread 2 tries to get lock to userList, waits on Thread 1 to release

Both threads are waiting for the other and neither can move forward. This is a blocking scenario, and since neither will relinquish its resource, we are deadlocked.

edited May 13 '10 at 17:25 answered May 13 '10 at 11:24
Ed Gonzalez
**1,616** 10 15

Your enumeration of events on the two lists is not accurate. should be #1 userlist, #2 movieList, #1 movieList (block), #2 userList (block) – harschware May 13 '10 at 17:01

Thanks harschware! Fixed. – Ed Gonzalez May 13 '10 at 17:26

Thanks Ed for the answers. I specifically assumed only one add operation inside a synchronized block. If we compare a synchronized link Vs a List which can add multiple elements synchronized in a block then it is not apple to apples. In general if both perform same functionality of adding one element at a time as in my question, would you prefer one over the other ? – codeObserver Jun 21 '10 at 2:54

I'd probably prefer a synchronized(list) block over the synchronizedList, simply because I feel it makes the intent more explicit where we use it. – Ed Gonzalez Jun 22 '10 at 14:08

1. Wikipedia is a great resource for any Computer Science student. Here is an article on Non-blocking synchronization - http://en.wikipedia.org/wiki/Non-blocking_synchronization
2. Non-blocking synchronization is available in any language, it is how the programmer defines their multi-threaded application.
3. You should only use locking (i.e. synchronized (list) ) when necessary due to the time it takes to acquire the lock. In Java, the Vector is a thread safe data structure that is very similar to Java's ArrayList.

answered May 13 '10 at 3:23
Eric U.
**985** 2 15 27

**1: What is "non-blocking" concurrency and how is it different than normal concurrency using threads? Why don't we use non-blocking concurrency in all the scenarios where concurrency is required? Is there overhead for using non-blocking concurrency?**

Non blocking algorithms do not use specific object locking schemes to control concurrent access to memory (synchronized and standard object locks are examples that use object/function level locks to reduce concurrent access problems in Java. Instead these use some form of low level instruction to perform (on some level) a simulataneous compare and swap on a memory location; if this fails it just returns false and does not error out, if it works then it was successful and you move on. Generally, this is attempted in a loop until it works, since there will only be small periods of time (hopefully) when this would fail, it just loops a few extra times until it can set the memory it needs to.

This is not always used because it is much more complex from a code perspective even for relatively trivial use cases than the standard Java synchronization. Moreover, for most uses the performance impact of the locking is trivial compared to other sources in the system. In most cases, the performance requirements are not nearly high enough to warrant even looking at this.

Finally, as the JDK/JRE evolves, the core designers are improving the internal language implementations to attempt to incorporate the most efficient means of achieving these ends in the core constructs. As you move away from the core constructs, you lose the automatic implementation of those improvements since you are using less standard implementations (for instance jaxb/jibx; jaxb used to grossly underperform jibx, but now is equal if not faster in most cases that I've tested as of java 7) when you bump your java version.

if you look at the code example below, you can see the 'overhead' locations. It's not really overhead per se, but the code must be extremely efficient in order to work non-locking and actually perform better than a standard synchronized version due to the looping. Even slight modifications can lead to code that will go from performing several times better than standard to code that is several times worse (for instance object instantiations that don't need to be there or even quick conditional checks; you're talking about saving cycles here, so the difference between success and failure is very slim).

**2: I have heard that non-blocking concurrency is available in Java. Are there any particular scenarios where we should use this feature?**

In my opinion you should only use this if you A) have a proven performance problem in your running system in production, on its production hardware; and B) if you can prove that the only inefficiency left in the critical section is locking related; C) you have firm buy in from your stakeholders that they are willing to have non-standard less maintainable code in return for the performance improvement which you must D) prove numerically on your production hardware to be certain it will even help at all.

**3: Is there a difference or advantage to using one of these methods with a collection?**

**What are the trade-offs?**

The advantage is performance, the trade off is first that it's more specialized code (so many developers don't know what to make of it;making it harder for a new team or new hire to come up to speed, remember that the majority of the cost of software is labor, so you have to watch the total cost of ownership that you impose through design decisions), and that any modifications should be tested again to ensure that the construct is still actually faster. Generally in a system that would require this some performance or load and throughput testing would be required for any changes. If you aren't doing these tests then I would argue that you almost certainly don't need to even think about these approaches, and would almost definitely not see any value for the increased complexity (if you got it all to work right).

Again, I just have to restate all the standard warnings against optimization generally, as many of these arguments are the same that I would use against this as a design. Many of the drawbacks to this are the same as any optimization, for instance, whenever you change the code you have to ensure that your 'fix' doesn't introduce inefficiency in some construct that was only placed there to improve performance, and deal with that (meaning up to refactoring the entire section to potentially remove the optimizations) if the fix is critical and it reduces performance.

It is really, really easy to mess this up in ways that are very difficult to debug, so if you don't have to do it (which I have only found a few scenarios where you ever would; and to me those were pretty questionable and I would have preferred to not do it) do not do it. use the standard stuff and everyone will be happier!

***Discussion/Code***

Non blocking or lock free concurrency avoids the use of specific object locks to control shared memory access (like synchronized blocks or specific locks). There is a performance advantage when the code section is non-locking; however, the code in the CAS loop(if this is the way you go, there are other methods in Java) must be very, very efficient or this will end up costing you more performance than you gain.

Like all performance optimizations, the extra complexity is not worth the effect for most use cases. Cleanly written Java using standard constructs will work as well if not better than most optimizations (and actually allow your organization to maintain the software more easily once you're gone). To my mind this only makes sense in very high performance sections with proven performance issues where the locking is the only source of inefficiency. If you do not definitely have a known and quantified performance problem, I would avoid the use of any technique like this until you have proven the problem is actually there because of the locking and not do to other issues with the efficiency of the code. Once you have a proven locking based performance problem I would ensure that you have some type of metric in place to ensure that this type of setup is actually going to run faster for you than just using standard Java concurrency.

The implementation that I have done for this use CAS operations and the Atomic family of variables. This basic code has never locked up or raised any errors for me in this use case (random sampling input and output for offline testing from a high throughput translation system). It basically works like this:

You have some object that is shared between threads, and this is declared as either an AtomicXXX or an AtomicReference (for most non-trivial use cases you'll run with the AtomicReference version).

when the given value/object is referenced, you retrieve it from the Atomic wrapper, this gets you a local copy on which you perform some modification. From here you use a compareAndSwap as the condition of a while loop to attempt to set this Atomic from your thread, if this fails it returns false as opposed to locking up. This will iterate until it works (the code in this loop must be very efficient and simple).

You can look up CAS operations to see how they work, it's basically intended to be implemented as a single instruction set with a comparison at the end to see if the value is what you tried to set it to.

If the compareAndSwap fails, you get your object again from the Atomic wrapper, perform any modifications again, and then try the compare and swap again until it works. There is no specific lock, you're just trying to set the object back into memory and if it fails you just try again whenever your thread gets control again.

Code for this is below for a simple case with a List:

```
/* field declaration*/
//Note that I have an initialization block which ensures that the object in this
//reference is never null, this was required to remove null checks and ensure the CAS
//loop was efficient enough to improve performance in my use case
private AtomicReference<List<SampleRuleMessage>> specialSamplingRulesAtomic = new
AtomicReference<List<SampleRuleMessage>>();


/*start of interesting code section*/
```

```
    List<SampleRuleMessage> list = specialSamplingRulesAtomic.get();
    list.add(message);
    while(!specialSamplingRulesAtomic.compareAndSet(specialSamplingRulesAtomic.get(),
list)){
        list = specialSamplingRulesAtomic.get();
        list.add(message);
    };
 /* end of interesting code section*/
```

edited Feb 27 '14 at 15:00

answered Feb 27 '14 at 14:55

Scott Taylor
**56** 2

Non-blocking synchronization is the same a blocking synchronization, both are kind of synchronization, the only difference is that non-blocking synchronization is faster overall.

For starters you want to use synchronization only when multiple threads access the same resource in RAM. You can't use synchronization when trying to access things on disk, or better said, you need to use locks on disk.

That said, how can you synchronize if no thread ever blocks?

The answer is optimistic locking. This idea has existed for at least 20 years. Maybe more.

You maybe have heard of the Lisp language. As it turns out functional languages never modify its parameters, only return new values, so they never need to synchronize.

In Lisp you can have shared state, but it gets tricky. So most programs can run in parallel and never worry about synchronization.

The idea of optimistic locking is that all threads modify shared valued willingly, but they have a local area to modify the values, and only apply the modification at the end, with one instruction, atomically, using CAS. Cas stands for Compare And Swap,it performs in just one CPU cycle, and it has been implemented in CPUs for at least 20 years.

An excellent explanation of what CAS is:
https://www.cs.umd.edu/class/fall2010/cmsc433/lectures/nonBlocking.pdf

So if there is a conflict in the modification, it will only affect one of the writers, the rest will be done with it.

Moreover, if there is no contention whatsoever, non-blocking algorithms perform a lot faster than their blocking counterparts.

Tutorial on non-blocking algorithms in Java with code examples you can use in real life:
http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html

answered Jun 22 '15 at 8:17

bill schwarz
**31** 1