Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Non-blocking synchronization

vorgelegt von

Joshua Krüger

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik
Matrikelnummer: 6946236

Betreuer: Eugen Betke

Hamburg, 2019-02-26

Abstract

The modern era of computing makes extensive use of parallel computing, and non-blocking synchronization is a way to most optimally utilize the capabilities of modern multi-core machines. This elaboration will first discuss the downsides of blocking synchronization and subsequently ways to overcome them using non-blocking synchronization. Additionally it will provide a look into lock-free and wait-free algorithms, elaborate on their advantages and limitations. To further illustrate that performance experiments with a lock-free data structure will be conducted and it's results examined. Furthermore it will provide an in-depth look at the inner workings of said data-structure and compare it with it's blocking and not-synchronized counterparts.

Contents

1. Introduction

In this chapter, the use of non-blocking synchronization techniques will be further motivated or rather the use of blocking techniques further discouraged.

Blocking synchronization has the commonality that threads are paused until a resource is available. What the resource ends up being can greatly differ. It may be access to a file/data structure or atomic access to a critical section of code. In network communication even a message from a remote computer, can be considered a resource. In any of these examples blocking the thread can or will cause a number of issues.

Before elaborating on all the disadvantages of blocking synchronization however, there is a need to discuss it's one advantage: It is very simple.
Whenever a resource is required the execution simply holds until said resource is available. For the programmer this creates concise semantics. They will simply put a *lock* before using the resource and a corresponding *unlock* after they are done using the resource. In the line after the *lock* call, the programmer can assume atomic access to the resource and use it like they are the only process on the planet. Concise and simple, both syntactically and semantically. It seems the step from single process to concurrent environment only takes two lines of code.

However that simplicity does not automatically entail that it is *not* also very easy to get wrong. When someone first starts with concurrent programming they will often run into the issue that their program compiles, runs without errors, but nonetheless never completes. When they then look at the states of all of their threads, a majority of them remains in the WAIT-state. They have inadvertently created a deadlock. Meaning that their locking mechanism is somehow faulty and created a state in which all threads are waiting, possibly for each other and are not able to continue computation. Indefinitely. The reason maybe as simple as forgetting the *unlock* after finishing to use a resource, but the reasons for a deadlock are vast and some of the most complex bugs out there.

To shortly summarize why they are so difficult:

- There is no error message.
  The code fails silently. If the in-progress operation is represented in the form of a spinning wheel, then this wheel will just spin indefinitely. Until something or someone external decides that it must be a deadlock.

- You cannot always decide whether it actually is a deadlock.
  It is possible that the saving unlock call is right around the corner. This is especially an issue in distributed systems, there it is known as a 'phantom deadlock'.

- It may not always occur.
  If the deadlock is caused by a rare race condition, then it may not even occur until way into production. To be fair this is more of race condition and concurrency problem in general, but locking synchronization is especially prone to this type of bug.

Now a *very smart* programmer may argue something along the lines of: "Well, anything can be done wrong.. So just don't do that!"

One problem with that hypothetical statement is that there are other downsides that are not avoidable using blocking synchronization. Even if the programmer using it is *very smart*.

A deadlock can even occur when the programmer did not make any mistakes, as threads holding a resource are still subject to the scheduler. If, by chance or priority-scheduling the thread holding the resource is paused for a long time then all the other threads cannot continue execution either. Even worse: if the thread is interrupted without releasing the lock, then all other threads are halted indefinitely and a deadlock occurs. A more detailed analysis of that behavior can be found in Appendix A. It is rare, but especially in low-level, high-performance programming it can become an issue.

Even if the code computes the correct result, anyone doing concurrent programming will want to see the merits of their efforts at some point. They won't want to run their code on a single core. What good are many threads if they are eventually sequentialized by some scheduler. In that case one could just write ordinary, sequential code in the first place. Therefore most concurrent code will eventually be run on multiple cores, multiple machines even. Best case: Each thread executes on it's own core[1]. With any lock, for any waiting thread the resources of that thread are wasted. One way to mitigate that problem is to use Futures (see Appendix D).

Threads are expensive, they do have overhead. Both to create and to keep up. On a smaller linux machine the maximum number of threads per process is 61573 [2]. To be fair, most programs do not require that number of threads. In webserver-programming however that number can quickly become a limiting factor with regards to scalability.

[1] With hyper-threading this principle can fall. The important thing is that there is no theoretical need for there to be more threads than the hardware can execute in parallel.

[2] obtained by executing 'cat /proc/sys/kernel/threads-max' in a shell on a linux machine with 2 cores

2. Concurrency guarantees

In this chapter, the different guarantees that can be assigned to a concurrent algorithm will be discussed.

2.1. Lock Freedom

In the introduction it was shown that algorithms employing blocking mechanisms have a number of disadvantages. Lock-Free algorithms do not have these disadvantages. By definition.

Lock-Freedom is essentially just the trait of an algorithm, guaranteeing that it is free of a number of potential problems. A Lock-Free algorithm guarantees global process in a concurrent system [HHCP18]. This may seem like a necessity for any system, but with deadlocks and the interrupted thread scenario detailed in Appendix A it can become a complex guarantee to make or prove. As both of those issues are common and complicated. The trait of Lock Freedom is commonly associated to concurrent data structures or the algorithms mutating the data structure.

While Lock Freedom guarantees globabl progress, it does not guarantee, and this differentiates it from Wait-Freedom(Section 2.2), is that any single thread may fail or take an infinite number of steps [HHCP18]. As long as there is still progress at a global level. While Lock-Free algorithms do not exhibit the same issues that blocking algorithms do, they only do so by definition. How to actually design and implement such an algorithm is not easily derived from it. Depending on the problem the algorithm aims to solve, it may not even be possible. For example computing data with multiple threads, then evaluating the combined results is an algorithm that requires waiting and cannot handle thread-failure. It would therefore not be possible to implement a lock-free version of it.

Later chapters will explore an implementation of a lock-free data-structure, analyze how it works and do some experiments with it (see Section 3.1 and Chapter 4).

2.2. Wait Freedom

The next level of behavior guarantee in a concurrent environment is Wait-Freedom.

It does not actually guarantee no wait for any single thread as the name suggests, but only guarantees that any single thread will *not wait indefinitely* and eventually make progress. As that automatically entails global progress, every Wait-Free algorithm is

also a Lock-Free algorithm [HHCP18]. It solves the remaining problem in Lock-Free algorithms by making a guarantee for every single thread.

As with Lock-Freedom, Wait-Freedom is a very difficult property to actually implement and prove. Furthermore it is not possible find a wait-free algorithm for every problem. Even algorithms that have a lock-free implementation, do not necessarily have a wait-free implementation. i.e not every problem can be solved with a wait-free algorithm. [HHCP18].

Due to it being a very strong guarantee Wait-Freedom is hard to implement and therefore typically only implemented at a very low level and/or for very simple data-structures. For example there is an implementation of a Wait-Free Queue that employs a custom scheduling algorithm and helper threads [KE11]. Most commonly however Wait-Freedom is encountered in the form of CPU-primitives. These are of the utmost importance, because they are in turn used to construct most lock-free and wait-free algorithms. Most notably the lock-free algorithm that will be discussed in a later chapter.

The most relevant primitive is compare-and-swap[1], commonly referred to by it's acronym CAS. It is the one used by the algorithm in a later chapter. CAS is wait-free([Her91]) and can even be wait-free when operating on multiple words [DFL13].

The following showcases how CAS works[2]:

> This primitive takes two values: *old* and *new*. If the register's current value is equals to *old*, it is replaced by *new*; otherwise it is left unchanged. The register's old value is returned.
>
> ([DFL13])

At a programming language level one typically has to additionally supply the memory address to 'compare and swap'. Compare Listing 2.1, showing an implementation for cas on integers. Instead of the old value, this versions returns whether the swap has occurred. Which will be helpful when implementing a stack in the next chapter.

```
1 \\atomic
2 boolean compare_and_swap(int* pointer_to_val,        int
    ↪ expected_val, int new_val) {
3    int current_val = *pointer_to_val;  //obtain value
4    if(current_val == expected_value) { //compare value
5       *pointer_to_val = new_val;       //set value
6       return true;
7    }
8    return false;
9 }
```

Listing 2.1: Compare and Swap (CAS - Pseudo C Code)

[1] contemporarily also known as compare-and-set

[2] Shown only for integers, but the generic version works equivalently

3. Concurrent Mutation

In this chapter, the ways one can concurrently mutate certain data structures without using locks will be explored. Most notably the implementation of a lock free stack is presented.

3.1. Lock Free Stack

A simple and concise data structure taught in every CS-101 class is the 'stack'. It is easy to describe and has, in it's simplest form, only three operations:

- *Push* - puts a new element on top of the stack.
- *Pop* - removes and returns the most-recently pushed, but not popped element. i.e. removes and returns the top of the stack[1].
- *Peek* - returns the TOS, without mutating the stack.

For the following experiments four version of the stack data-structure have been implemented for this elaboration. The complete code for each of them can be found in Chapter Appendix B.
One version is the linked stack(see Listing B.3), that ignores the side-effects of concurrent access entirely. It's approach to concurrency is to hope that it causes no issues. As the experiments will show this does not always work and results in a loss of data. Why data is lost can be best shown at the push method of the unsynchronized stack, seen in Listing 3.1.
The method itself is easily understood. The TOS is redefined as a new node that contains the added element e and points to the old TOS as the next, now second element on the stack. This method can be even shorter as demonstrated in Listing B.3. The data loss problem occurs if a thread A reads the head(currently value x) and puts it into oldHead. Then a thread B concurrently does the same. For both of them the local variable oldHead now contains x. Now thread A sets the head to $a->x$, having amended a to the stack. Afterwards thread B set the head to $b->x$, having amended b to the stack. Thread-B had no way of knowing that thread A had changed the stack and that x was not the real old head anymore. This is known as the lost update problem.

```
1 public void push(E e) {
2     Node oldHead = head;
```

[1] from here on referred to as TOS

```
3       Node newHead = new Node<>(e, oldHead);
4       head = newHead;
5   }
```

Listing 3.1: Unsynchronized Stack - Push Method

Two other version's employ blocking locking to solve the lost update problem. Before every read and write operation the entire stack is locked. One version uses java.util.concurrent.locks.ReentrantLock(see Listing B.4), the other uses the java-keyword 'synchronized'(see Listing B.5). Using the keyword is advantageous, because it allows the Java Virtual Machine to optimize scheduling. Experiments will show that this version is the best performing for many use cases. Yet it still employs locks, is not Lock-Free and therefore retains the disadvantages of blocking synchronization (As shown in Appendix A). This will be further illustrated by the experiment in Section 4.2.4.

Finally the lock free version(see Listing B.6). It heavily employs the CAS-primitive(described in detail above, see Listing 2.1).

In the following, once again only the push method of the Stack is discussed in detail. The pop method functions similarly(compare Listing B.6) and the peek method requires no synchronization at all. The reason is examined at the end of this chapter.

```
1   public void push(E e) {
2       Node<E> oldHead;
3       Node<E> newHead;
4       do {
5           oldHead = head.get();
6           newHead = new Node<>(e, oldHead);
7       } while(!head.compareAndSet(oldHead, newHead));
8   }
```

Listing 3.2: Lock Free Stack - Push Method

Listing 3.2 shows the push method of the lock-free stack in it's entirety. When comparing it to the unsynchronized push in Listing 3.1 a lot of similarities can be discovered and only a few differences. It now declares the variables oldHead and newHead up front, which is required by do-while loop syntax. Furthermore it uses a 'get' on head to obtain it's value, which is required because normal Java objects do not support compareAndSet. Instead it has to be wrapped in a java.util.concurrent.atomic.AtomicReference, which unwraps using said 'get' method.
The interesting and critical new part is the do-while loop and the compareAndSet[2].
Combined with the loop this is can be viewed as a try and error loop. The algorithm attempts to do what the unsynchronized push(see Listing 3.1) did, however if another

[2]CAS - contemporary name for compareAndSwap

thread altered the real head since oldHead was queried(using 'get'), then it does not overwrite that update and instead just try's again. The significant advantage, and what makes this a lock-free stack implementation[3], is the fact that a thread A can be interrupted in the middle of this method(for example line 5) and a thread B will can nonetheless set oldHead to newHead and continue to make progress. Without any knowledge of thread A's failure. Global progress is therefore independent of any single thread.

Since there are no complex operations between query and the set attempt it is likely that the operation completes successfully at some point, however it is not guaranteed. There is a theoretical non-zero chance that in a busy system a thread never successfully sets it's newHead to head, because some other thread always alters it before. It is unlikely, but possible. This scenario is known as starvation and the reason this algorithm is not wait-free. No per thread progress can be guaranteed.
There is a wait-free implementation for the stack data-structure [GAS15].

Another upside is that readers do not require any synchronization. The head a reader obtains is always valid(once again obtained using get) and cannot be altered by other threads because it's reference is assigned to a local variable and nodes itself are immutable. From that copy of the head it's value or lack thereof can be easily determined. As would be done in the non-concurrent case.

3.2. Copy On Write

Another common pattern in highly concurrent data-structures is Copy-On-Write
Note up front that the use of Copy-On-Write does in and of itself not make any guarantees regarding Lock- or Wait-Freedom. It does not even necessarily need to be used with non-blocking techniques, though it often is. The lock free stack previously discussed (see Section 3.1) can be considered a Copy-On-Write algorithm, though the only thing copied is a reference to the current head.
In a Copy-On-Write data structure on every mutation the data[4] is copied, mutated and atomically set. Until the mutation is complete, any read on the data occurs on the old, un-mutated and therefore consistent data. The major advantage here is that reader's do not require a synchronization mechanism. Neither during a read nor a mutating operation. [5].
The major disadvantage of Copy-On-Write is that mutation requires at least twice the memory of the area to be mutated and that locking is still required(see Listing 3.4 for an example). Alternatively a lock free approach using the cas primitive is feasible. This however increases the maximum space complexity to infinity, if there is no set maximum of concurrent modifiers (see Listing 3.5 for an example).
Copy-On-Write nonetheless has many use cases, one example from the Java world is the

[3]by definition, not just in the obvious way of not containing any locks
[4]many modern algorithms only copy a partition of the data, but the principle remains the same
[5]Why readers need to be locked without Copy-On-Write can be understood when looking at Listing 3.3

CopyOnWriteArrayList. It is a commonly used list implementation for concurrent Java programs.
Generally speaking Copy-On-Write is optimal when a data-structure is expected to have many readers, few writers and few mutating-operations in general. Otherwise a simple ReadWriteLock may offer both better performance and scalability. A read-write-lock is a blocking-technique of fine grained locking that allows multiple concurrent readers, but locks the entire list on a write operation.

```
1  //Readers require locking.
2  int find_index(array, element) {
3      for(int i=0; i < array.length; i++) {
4          //if the thread is paused here, and another threads
               ↪ 'clears' the list, the next line will yield a
               ↪ IndexOutOfBoundsException. And not return
               ↪ DOES_NOT_EXIST.
5          if(array[i] == element) {
6              return i;
7          }
8      }
9      return DOES_NOT_EXIST;
10 }
```

Listing 3.3: List - Safe Replace Operation (Pseudo Java)

```
1  //A Copy-On-Write list using locks
2  void replace(array, old_element, new_element) {
3    //lock (writers only)
4    Array array_new = array.clone();
5
6    int old_element_index =
       ↪ find_index(array_new, old_element);
7    array_new[old_element_index] = new_element;
8    array = array_new;
9    //unlock (writers only)
10 }
```

Listing 3.4: List - Safe Replace Operation (Pseudo Java)

```
1  //A Lock-Free Copy-On-Write list
2  void replace(array, old_element, new_element) {
3    //'array' vom Java-Typ AtomicReference
4    Array array_old;
```

```
5     Array array_new;
6     do {
7       array_old = array.load();
8       array_new = array_old.clone();
9
10      int old_element_index =
          ↪ find_index(array_new, old_element);
11      array_new[old_element_index] = new_element;
12
13    } while(! array.cas(array_old, array_new));
14  }
```

Listing 3.5: List - Lock-Free Replace (Pseudo Java)

4. Experiment

In this chapter, a performance experiment is conducted to compare the performance and correctness of the different stack implementations shown in Appendix B

4.1. Methodology

The results presented in Section 4.2 were obtained by running each of the four test methods shown in Appendix C with each of the four stack implementations shown in Appendix B. The experiments where repeated multiple times within the same JVM-instance as is best-practice when running benchmarks. The elapsed time of one run was measured using the Java function 'System.nanoTime'. For the assertions JUnit4 was used. If such an assertion failed or another kind of exception was thrown the attempt was not counted, otherwise it's time was measured and added to the results set.
Additionally the complete first run of each test was omitted from the results, to first allow JIT optimization to occur on every implementation. The Tests where run on a Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, with 2 cores. The machine was running the linux kernel 4.19.23-1-MANJARO [1] and an openjdk JVM version "1.8.0-202" [2].
The different run-times of each test cannot be compared, but the different stack implementation's results within one test environment can. To best compare the results visually, bloxplots where chosen. Additionally outliers in the data are presented as points or rectangles next to the boxplots.

[1] obtained using 'uname -r'
[2] obtained using 'java -version'

4.2. Results

4.2.1. Single Thread Comparison

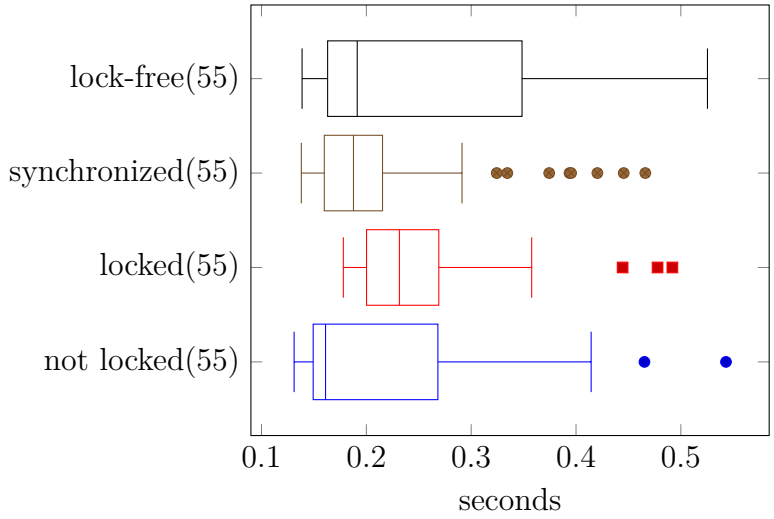

Figure 4.1.: Test results for multiple executions of Listing C.1

Each version successfully completed 55 times.
Unsurprisingly the version with no synchronization performed best in a single threaded environment, due to the fact that it has no synchronization overhead. That overhead is visible in the other even when there is no synchronization required.

4.2.2. Single Writer - Multiple Readers Comparison

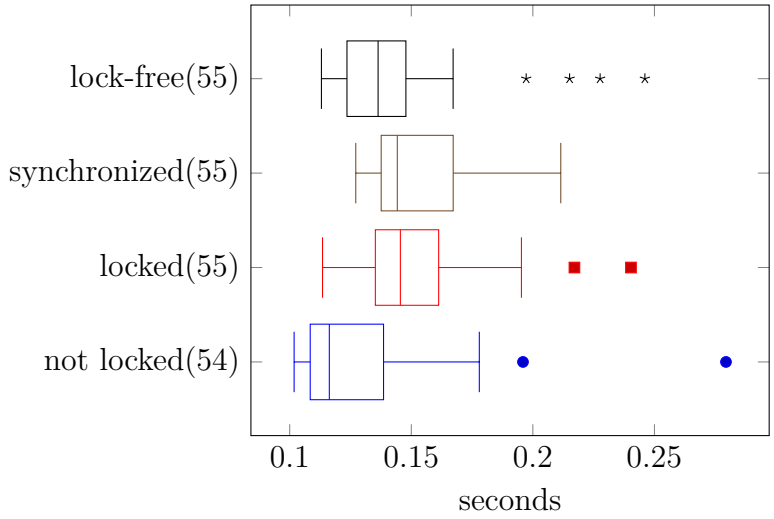

Figure 4.2.: Test results for multiple executions of Listing C.2

Each synchronized version successfully completed 55 times, the non synchronized version only successfully completed 54 times. This may be surprising at first glance, but when looking at the test code and the implementation of the stack one will see that the window for the race condition to occur is very narrow and therefore very unlikely to occur.
The Lock-Free version outperforms both conventionally blocking versions. However only by a very narrow margin and with multiple outliers. This can be attributed to the fact that the this test is focused on the read operation which requires no synchronization in the lock-free implementation.

4.2.3. Multiple Writers - Multiple Readers Comparison

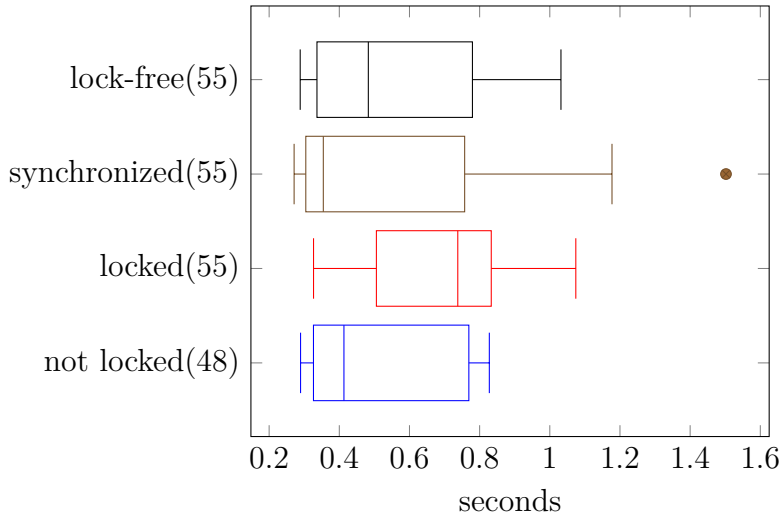

Figure 4.3.: Test results for multiple executions of Listing C.3

Each synchronized version successfully completed 55 times, while the non synchronized version is now down to 48 successful executions. This nonetheless good success rate can be attributed to the fact that the race condition is still unlikely and that the critical section of code does not take long to execute.
The 'synchronized-keyword' version outperforms the lock-free version in this test. This is likely caused by the heavy optimization done within the JVM. That theory is supported by the fact that the locked version now falls far behind the other versions, taking almost twice as long as the 'synchronized-keyword' version. Despite the fact that they both rely on the same underlying principle of blocking synchronization.

4.2.4. Thread Suspension Comparison

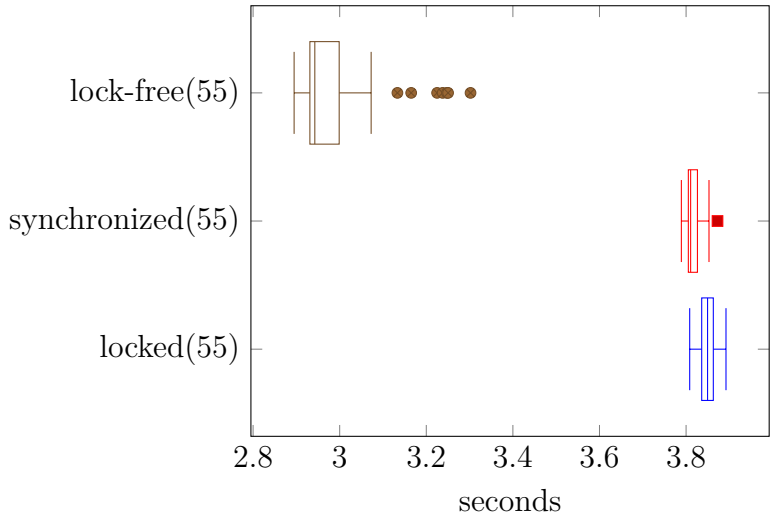

Figure 4.4.: Test results for multiple executions of Listing C.4
Note: The not locked version's results were omitted for clarity.

While the test code is very similar to the 'many writers - many readers' code on the surface(doing the same operations in the same order, compare Listing C.3 and Listing C.4), the results are vastly different.
Due to the effects described in Appendix A, the blocking version's both behave badly with thread suspension and would behave much worse if their threads were interrupted instead. In the lock-free version other threads can continue their mutations even when a thread is suspended during it's mutation attempt. This detail makes a very visible difference in the results presented in Figure 4.4.

5. Conclusion

To summarize, synchronization in a concurrent algorithm is unavoidable to obtain a correct results.
Blocking synchronization has a number of disadvantages.
Non-blocking synchronization can mitigate some of those disadvantages, but will often increase the complexity of a previously simple program.
Therefore the general advice is to use blocking synchronization in most cases. This may be shocking after a long elaboration on the disadvantages of blocking synchronization and how to overcome them. However the one upside blocking synchronization has is a major one. Simplicity. In Java there is even a keyword reserved for it. Amending a method with 'synchronized' and it immediately becomes a critical, atomically executed section of code. This simplicity is of immense value. The programmer can then focus their attention on other important aspects of the code. User-experience, features, modifiability and correctness to only name a few. Using as few fine-grained locks as possible, and those with great care will often be enough to achieve acceptable performance.

Implementing non-blocking synchronization for an algorithm can be seen as a form of optimization, and as is well documented:

> [...] premature optimization is the root of all evil [...]
>
> (Donald Knuth, 1974)

The added complexity of non-blocking synchronization is only warranted in low-level, high-performance and time-critical computing. In those circumstances non-blocking synchronization is a powerful tool to increase performance. It also has it's strengths where guarantees about performance are required. A Wait-Free algorithm can make those guarantees even in a concurrent environment.

Some frameworks/languages aim to decrease complexity whilst retaining the advantages of non-blocking synchronization techniques. Go-lang can be named as an example. It can offer a good trade-off between complexity and scalability, for the every day web programmer.

In conclusion, non-blocking synchronization techniques are a great way to optimize your code. This elaboration gave an overview of what they are, what they can do and where their limitations lie.

Bibliography

[DFL13] Damian Dechev, Steven Feldman, and Pierre LaBorde. A practical wait-free multi-word compare-and-swap operation. 9 2013.

[GAS15] Seep Goel, Pooja Aggarwal, and Smruti R. Sarangi. A wait-free stack. *CoRR*, abs/1510.00116, 2015.

[Her91] Maurice Herlihy. Wait-free synchronization. 1 1991.

[HHCP18] Attiya Hagit, Danny Hendler, Armando Castañeda, and Matthieu Perrin. Separating Lock-Freedom from Wait-Freedom. 7 2018.

[KE11] Alex Kogan and Petrank Erez. Wait-Free Queues With Multiple Enqueuers and Dequeuers. 2 2011.

[Val94] John D. Valois. Implementing lock-free queues. 10 1994.

[VCGH11] Vishwanath Venkatesan, Mohamad Chaarawi, Edgar Gabriel, and Torsten Hoefler. Design and evaluation of nonblocking collective i/o operations. 09 2011.

Appendices

A. Thread blocking side effects

In this chapter, the side-effects of blocking synchronization are discussed. These are exactly the side-effects lock-free algorithms aim to mitigate.

If a thread A is paused but is holding a lock L then that lock L is clearly blocked for as long as the thread A is paused.
That results in another problem. If then a thread B blockingly requests the lock L(i.e. pauses until lock L is available), then thread B will have to wait until thread A is unpaused and releases the lock.
Listing A.1 shows this scenario more clearly:

```
1  Lock L = new Lock();
2
3  thread-A: requests L
4  thread-A: gets L
5  thread-A: look at some array
6  thread-B: requests L
7  thread-A: sleep(10 minutes) //or is paused by scheduler
8  thread-B: waits for 10 minutes..
9  thread-A: look at some other array
10 thread-A: releases L
11 thread-B: finally acquires L
```

Listing A.1: Sleeping threads (Pseudo Code)

While the above scenario is already a major issue, the even more severe scenario is the version where thread A is not just paused, but interrupted. Then lock L can never be released and thread B will never complete.

Listing A.2 shows an example of this behavior:

```
1  Lock L = new Lock();
2
3  thread-A: requests L
4  thread-A: gets L
5  thread-A: looks at some array
6  thread-B: requests L
7
```

```
8  os|mainThread interrupts thread-A
9
10 thread-B: waits forever
```

Listing A.2: Interrupted threads (Pseudo Code)

While this may seem unlikely, at low level programming it can become a debilitating problem.

Another feasible version of this problem is if the lock has been implemented as a file on the machine. If the file exists the lock L is locked, otherwise it's unlocked. If, for example a user decides to kill the process, then the file is not deleted and other processes will not be able to access whatever resource the file was representing a lock on.

B. Implementations of a Stack

```
1  public interface Stack<E> {
2      void push(E e);
3      E pop();
4      E peek();
5
6      //functionality that requires implementation for the
           ↪ tests, but is omitted from the code definitions
           ↪ below for clarity
7      int size();
8      void println();
9      void push(E e, long sleep_at_some_point);
10     E pop(long sleep_at_some_point);
11     E peek(long sleep_at_some_point);
12 }
```

Listing B.1: Interface definition of a stack with minimal functionality

```
1  public class Node<E> {
2      public final E val;
3      public Node<E> next;
4      public Node(E val, Node<E> next) {
5          this.val = val;
6          this.next = next;
7      }
8  }
```

Listing B.2: Generic Node class used by the Stack implementations below

```
1  public class LinkedStack<E> implements Stack<E> {
2      private Node<E> head = null;
3
4      public void push(E e) {
5          head = new Node<>(e, head);
6      }
7
```

```
8       public E pop() {
9           if(head==null)
10              return null;
11          else {
12              E val = head.val;
13              head = head.next;
14              return val;
15          }
16      }
17
18      public E peek() {
19          return head==null? null: head.val;
20      }
21  }
```

Listing B.3: Simple, non concurrent Stack

```
1   import java.util.concurrent.locks.Lock;
2   import java.util.concurrent.locks.ReentrantLock;
3
4   public class LockedStack<E> extends LinkedStack<E> {
5       private Lock lock = new ReentrantLock();
6
7       @Override public void push(E e) {
8           lock.lock();
9           try {
10              super.push(e);
11          } finally {lock.unlock();}
12      }
13
14      @Override public E pop() {
15          lock.lock();
16          try {
17              return super.pop();
18          } finally {lock.unlock();}
19      }
20
21      @Override public E peek() {
22          lock.lock();
23          try {
24              return super.peek();
25          } finally {lock.unlock();}
26      }
```

```
27 }
```

Listing B.4: Locking Stack (using locks)

```
1  public class SynchronizedStack<E> extends LinkedStack<E> {
2      @Override public synchronized void push(E e) {
3          super.push(e);
4      }
5
6      @Override public synchronized E pop() {
7          return super.pop();
8      }
9
10     @Override public synchronized E peek() {
11         return super.peek();
12     }
13 }
```

Listing B.5: Locking Stack (using the synchronized keyword)

```
1  import java.util.concurrent.atomic.AtomicReference;
2  public class LFStack<E> implements Stack<E> {
3      private AtomicReference<Node<E>> head = new
          ↪ AtomicReference<>(null);
4
5      public void push(E e) {
6          Node<E> oldHead;
7          Node<E> newHead;
8          do {
9              oldHead = head.get();
10             newHead = new Node<>(e, oldHead);
11         } while(!head.compareAndSet(oldHead, newHead));
12     }
13
14     public E pop() {
15         Node<E> oldHead;
16         Node<E> newHead;
17         do {
18             oldHead = head.get();
19             if(oldHead==null)
20                 return null;
21             else
22                 newHead = oldHead.next;
```

```
23          } while(! head.compareAndSet(oldHead, newHead));
24
25          return oldHead.val;
26      }
27
28      public E peek() {
29          Node<E> headNode = head.get();
30          return headNode==null? null: headNode.val;
31      }
32  }
```

Listing B.6: Lock-Free Stack - Inspired by Lock-Free-Queue([Val94])

C. Performance Measurement Code

```
1  void singleThreadTest(Stack<String> stack) {
2      int iterations = 200000;
3      for(int i=0;i<iterations;i++) {
4          stack.push(String.valueOf(i));
5      }
6
7      assertEquals(iterations, stack.size());
8
9      for(int i=0;i<iterations;i++) {
10         assertEquals(String.valueOf(iterations-1),
               ↪ stack.peek());
11     }
12
13     assertEquals(iterations, stack.size());
14
15     for(int i=iterations-1;i>=0;i--) {
16         assertEquals(String.valueOf(i), stack.pop());
17     }
18
19     assertEquals(0, stack.size());
20 }
```

Listing C.1: single thread, correctness test

```
1  void singleWriterMultipleReaders(Stack<String> stack)
       ↪ throws Throwable {
2      stack.push("former tos");
3      stack.push("tos");
4
5      int nThreads = 500;
6      ConcurrentPoolTester pool = new
           ↪ ConcurrentPoolTester(nThreads);
7      pool.execute(() -> {
8          for(int i=0;i<nThreads*nThreads;i++) {
9              if (i%2==0)          stack.pop();
10             else                 stack.push(i + "");
```

```
11          }
12      });
13
14      for(int i=0;i<nThreads;i++) {
15          pool.execute(() -> {
16              for(int i2=0;i2<nThreads;i2++) {
17                  String val = stack.peek();
18                  assertNotNull(val); //cannot assert
                        ↪ anything else, actual state is
                        ↪ nondeterministic
19              }
20          });
21      }
22      pool.waitForShutdownOrException();
23  }
```

Listing C.2: single writer thread, multiple reader threads

```
1   void multipleWritersMultipleReaders(Stack<String> stack)
      ↪ throws Throwable {
2       int nThreads = 500;
3       ConcurrentPoolTester pool = new
          ↪ ConcurrentPoolTester(nThreads);
4       for(int i=0;i<nThreads;i++) {
5           int fi = i;
6           pool.execute(() -> {
7               stack.push(fi + "");
8           });
9       }
10      pool.waitForShutdownOrException();
11
12      assertEquals(nThreads, stack.size());
13
14      pool = new ConcurrentPoolTester(nThreads);
15
16      for(int i=0;i<nThreads;i++) {
17          pool.execute(() -> {
18              String val = stack.peek();
19              assertNotNull(val); //cannot assert anything
                    ↪ else, actual state is nondeterministic
20          });
21      }
22      pool.waitForShutdownOrException();
```

```
23
24      assertEquals(nThreads, stack.size());
25
26      pool = new ConcurrentPoolTester(nThreads);
27      for(int i=0;i<nThreads;i++) {
28          pool.execute(() -> {
29              String val = stack.pop();
30              assertNotNull(val); //cannot assert anything
                    ↪ else, actual state is nondeterministic
31          });
32      }
33      pool.waitForShutdownOrException();
34
35      assertEquals(0, stack.size());
36  }
```

Listing C.3: multiple writer threads, multiple reader threads

```
1   void writersAndReaders_withSuspension(Stack<String> stack)
      ↪ throws Throwable {
2       int nThreads = 500;
3       int suspendEveryNthThread = 100;
4       int suspendFor = 500;
5
6       ConcurrentPoolTester pool = new
          ↪ ConcurrentPoolTester(nThreads);
7       for(int i=0;i<nThreads;i++) {
8           int fi = i;
9           pool.execute(() -> {
10              if(fi %suspendEveryNthThread == 0)
11                  stack.push(String.valueOf(fi), suspendFor);
12              else
13                  stack.push(String.valueOf(fi));
14          });
15      }
16      pool.waitForShutdownOrException();
17
18      assertEquals(nThreads, stack.size());
19
20      pool = new ConcurrentPoolTester(nThreads);
21
22      for(int i=0;i<nThreads;i++) {
23          int fi = i;
```

```
24          pool.execute(() -> {
25              String val = (fi %suspendEveryNthThread == 0)?
                    ↪ stack.peek(suspendFor) : stack.peek();
26              assertNotNull(val); //cannot assert anything
                    ↪ else, actual state is nondeterministic
27          });
28      }
29      pool.waitForShutdownOrException();
30
31      assertEquals(nThreads, stack.size());
32
33      pool = new ConcurrentPoolTester(nThreads);
34      for(int i=0;i<nThreads;i++) {
35          int fi = i;
36          pool.execute(() -> {
37              String val = (fi %suspendEveryNthThread == 0)?
                    ↪ stack.pop(suspendFor) : stack.pop();
38              assertNotNull(val); //cannot assert anything
                    ↪ else, actual state is nondeterministic
39          });
40      }
41      pool.waitForShutdownOrException();
42
43      assertEquals(0, stack.size());
44  }
```

Listing C.4: multiple writer threads, multiple reader threads, plus deterministic thread suspension during execution(for each push, pop and peek operation at three different points within the synchronization mechanism)

D. Futures

One cannot write an elaboration on non-blocking synchronization without acknowledging the concept of futures[1].

It is typically used in conjunction with non-blocking IO, but can be generalized for usage with normal locks.

Listing D.1 shows a minimal Example of a client connecting to a server and awaiting a message[2]. A connection is established and a read-future obtained. Line 5 demonstrates what could also be done in a blocking environment. The result of the future is awaited and in line 6 the message is available. Before and after the thread completes some work. The downside is that no work can be completed while the thread is waiting.

Lines 9 to 14 demonstrate the alternative. Clearly the thread still has to wait for the message to arrive, but it bridges that time with 'doing some work'. The fairly expensive thread is not wasted, but optimally utilized. Additionally the time spent copying the data from NIC to the correct memory location, in this case a newly created string is also optimally bridged.

```
1  Socket connection = Socket.connect("localhost", 12345);
2  Future<String> future = connection.read_async();
3
4  //The old way, the blocking way
5  doSomeWork();
6  String message = future.get();
7  handleMessage(message);
8  doSomeWork();
9
10 //The cool way, the non-blocking way
11 while(!future.isDone()) {
12     doSomeWork();
13 }
14 String message = future.get();
15 handleMessage(message);
```

Listing D.1: List - Example of using Futures (Pseudo Code)

[1] Occasionally also referred to as 'promises'

[2] Exception-Handling abbreviated for clarity.

Using Futures and non-blocking IO to bridge the waiting time with computation can significantly increase performance [VCGH11]. However it does have an obvious limitation. When the awaited IO-message is required for further computation and there is nothing else to do at the moment[3], then Futures introduce nothing but syntactic complexity and do not need to be used.

Futures and asynchronous IO can also be used to decrease the number of threads used to handle clients on a server. Instead of one thread per client, multiple clients can be handled by the same thread. As threads are an expensive resource and the number of threads is limited on a machine, this can make a significant improvement with regards to the scalability of a server.

[3]Though one could always mine Bitcoin

List of Figures

List of Listings