SAINT PETERSBURG ELECTROTECHNICAL UNIVERSITY

Computer Science and Knowledge Discovery

Master Group-6300

Report: implement a lock-free algorithm in the Java

Subject: Parallel Computing

Report by: Al-Khaykanee Mujtaba Nazar Kadhim

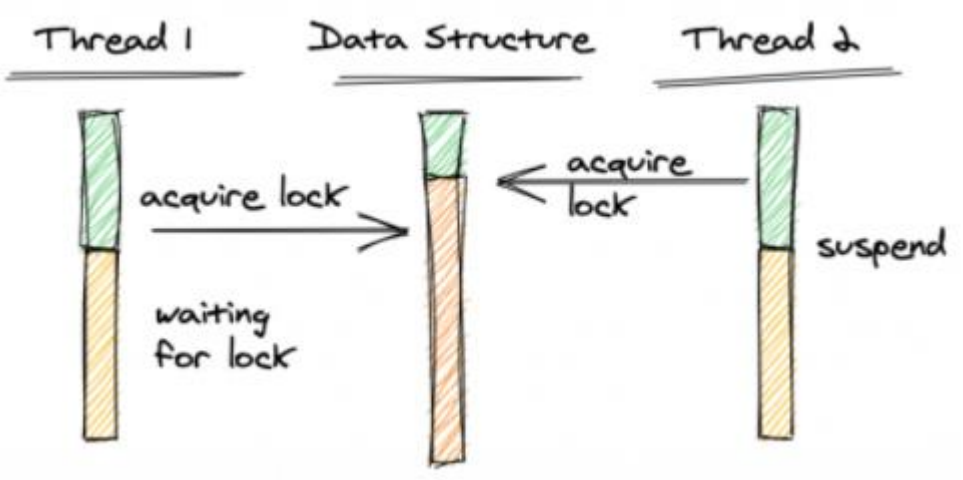
Supervisor: Natalya V. Shevskaya.

1) Introduction:

Lock-free algorithm:- an algorithm is called Lock-free if failure or suspension of any thread cannot cause failure or suspension of another thread;[1] for some operations, these algorithms provide a useful alternative to traditional blocking implementations. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress and wait-free if there is also guaranteed per-thread progress.

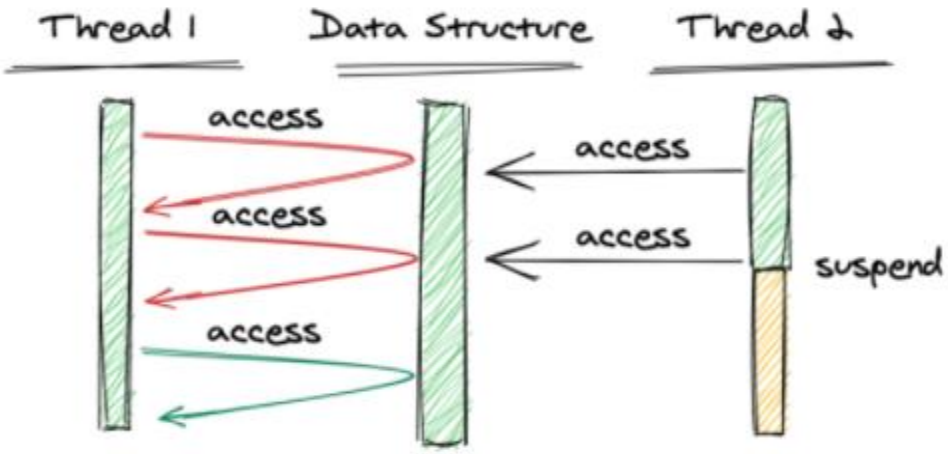
2) Lock Versus Starvation:

First, let's look at the difference between a blocked and a starving thread.



In the above picture, Thread 2 acquires a lock on the data structure. When Thread 1 attempts to acquire a lock as well, it needs to wait until Thread 2 releases the lock; it won't proceed before it can get the lock. If we suspend Thread 2 while it holds the lock, Thread 1 will have to wait forever.

The next picture illustrates thread starvation:



Here, Thread 2 accesses the data structure but does not acquire a lock. Thread 1 attempts to access the data structure at the same time, detects the concurrent access, and returns immediately, informing the thread that it could not complete (red) the operation. Thread 1 will then try again until it succeeds to complete the operation (green).

The advantage of this approach is that we don't need a lock. However, what can happen is that if Thread 2 (or other threads) access the data structure with high frequency, then Thread 1 needs a large number of attempts until it finally succeeds. We call this starvation.

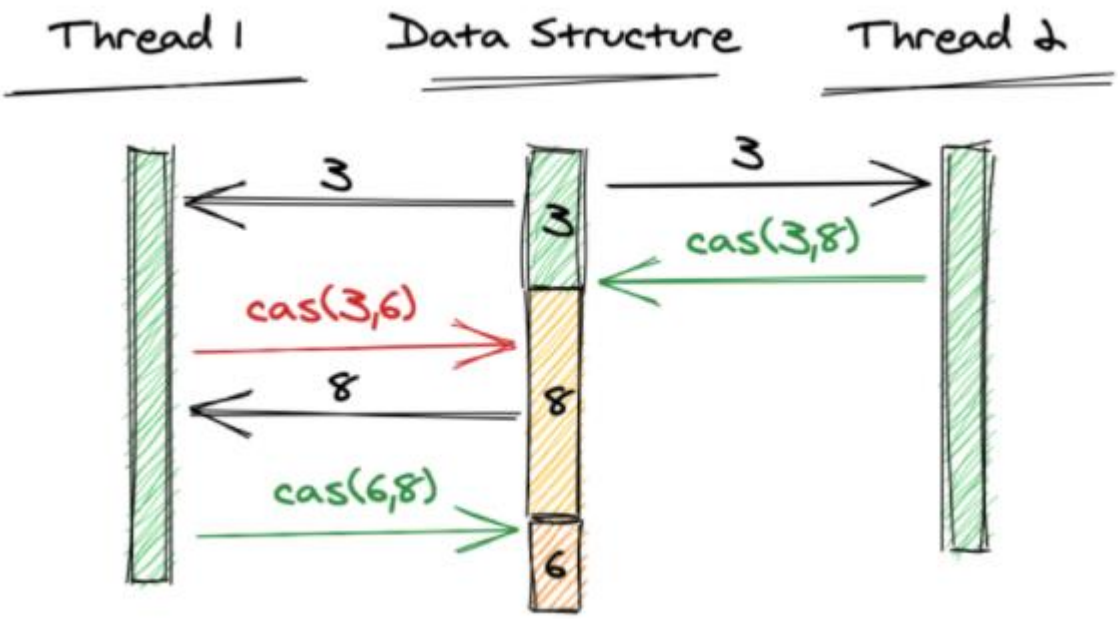
3) Non-Blocking Primitives :

In this section, we'll look at three basic operations that help us to build lock-free operations on data structures.

3.1) Compare and Swap

One of the basic operations used to avoid locking is the compare-and-swap (CAS) operation.

The idea of compare-and-swap is, that a variable is only updated if it still has the same value as at the time we had fetched the value of the variable from the main memory. CAS is an atomic operation, which means that fetch and update together are one single operation:



Here, both threads fetch the value 3 from the main memory. Thread 2 succeeds (green) and updates the variable to 8. As the first CAS by thread 1 expects the value to be still 3, the CAS fails (red). Therefore, Thread 1 fetches the value again, and the second CAS succeeds.

The important thing here is that CAS does not acquire a lock on the data structure but returns true if the update was successful, otherwise it returns false.

The following code snippet outlines how CAS works:

```
volatile int value;


boolean cas(int expectedValue, int newValue) {
    if(value == expectedValue) {
        value = newValue;
        return true;
    }
    return false;
}
```

We only update the value with the new value if it still has the expected value, otherwise, it returns false. The following code snippet shows how CAS can be called:

```
void testCas() {
    int v = value;
    int x = v + 1;


    while(!cas(v, x)) {
        v = value;
        x = v + 1;
    }
}
```

We attempt to update our value until the CAS operation succeeds, that is, returns *true*.

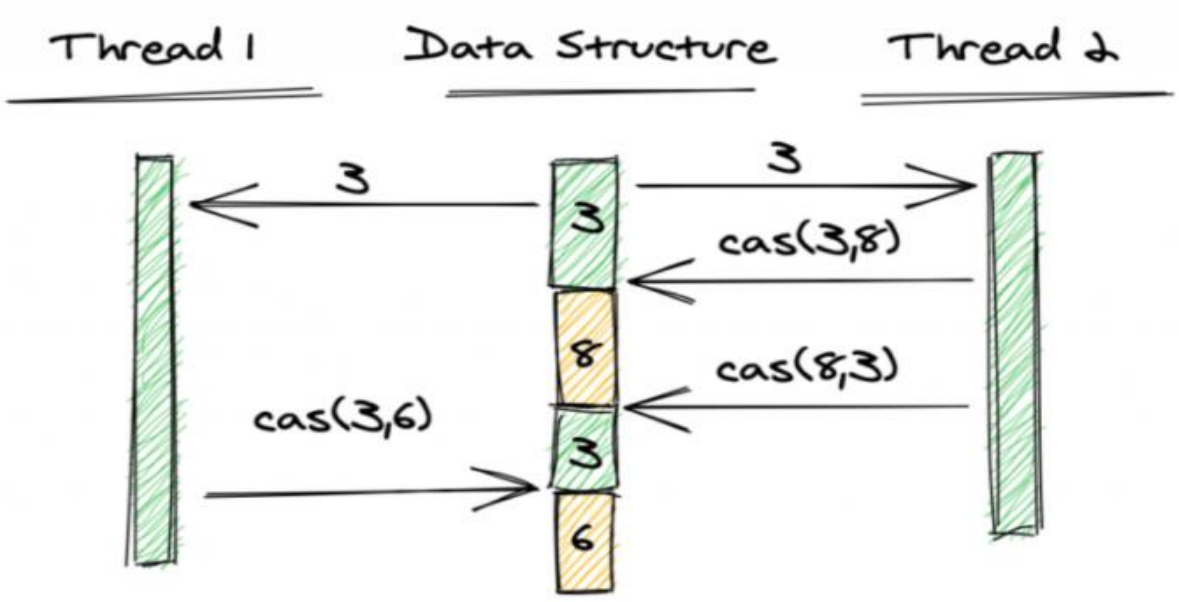
However, it's possible that a thread gets stuck in starvation. That can happen if other threads perform a CAS on the same variable at the same time, so the operation will never succeed for a particular thread (or will take an unreasonable amount of time to succeed). Still, if the compare-and-swap fails, we know that another thread has succeeded, thus we also ensure global progress, as required for lock-freedom.

3.2) Load-Link/Store-Conditional:

An alternative to compare-and-swap is load-link/store-conditional. Let's first revisit compare-and-swap. As we've seen before, CAS only updates the value if the value in the main memory is still the value we expect it to be.

However, CAS also succeeds if the value had changed, and, in the meantime, has changed back to its previous value.

The below image illustrates this situation:



Both, thread 1 and Thread 2 read the value of the variable, which is 3. Then Thread 2 performs a CAS, which succeeds in setting the variable to 8. Then again, Thread 2 performs a CAS to set the variable back to 3, which succeeds as well. Finally, Thread 1 performs a CAS, expecting the value 3, and succeeds as well, even though the value of our variable was modified twice in between.

3.3) Fetch and Add

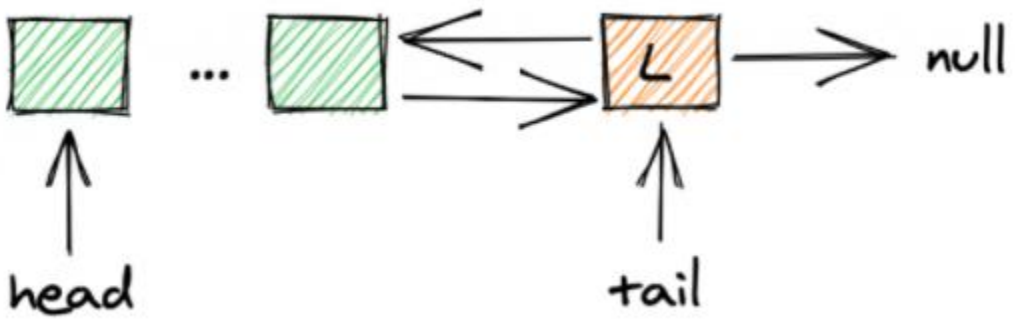
Another alternative is fetch-and-add. This operation increments the variable in the main memory by a given value. Again, the important point is that the operation happens atomically, which means no other thread can interfere.

Java provides an implementation of fetch-and-add in its atomic classes. Examples are AtomicInteger.incrementAndGet(), which increments the value and returns the new value; and AtomicInteger.getAndIncrement(), which returns the old value and then increments the value.

4) Accessing a Linked Queue from Multiple Threads

To better understand the problem of two (or more) threads accessing a queue simultaneously, let's look at a linked queue and two threads trying to add an element concurrently.

The queue we'll look at is a doubly-linked FIFO queue where we add new elements after the last element (L) and the variable tail points to that last element:

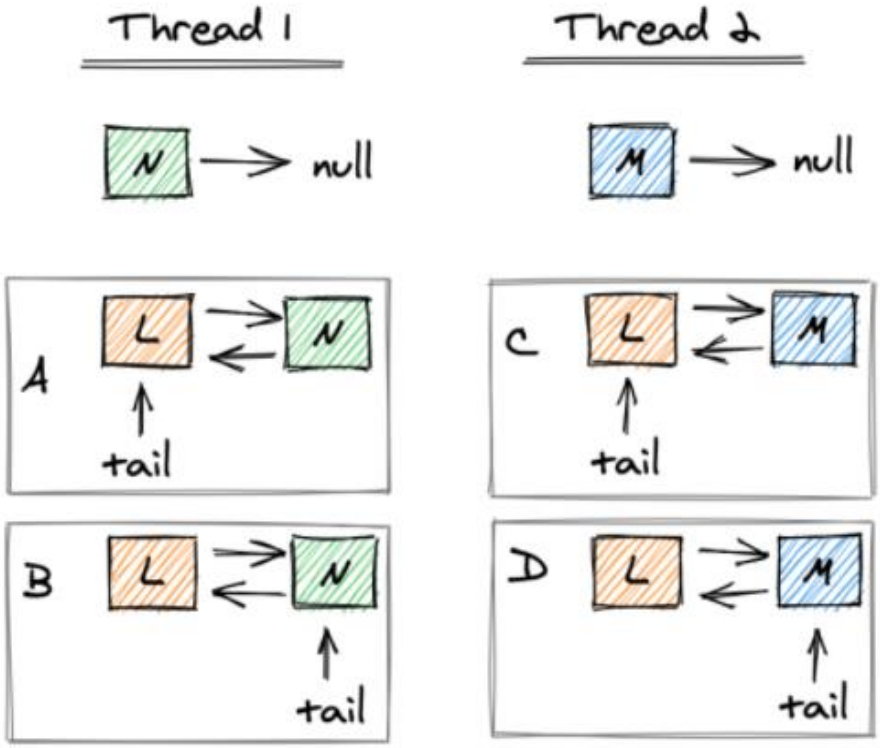


To add a new element, the threads need to perform three steps:

1) create the new elements (N and M), with the pointer to the next element set to null;

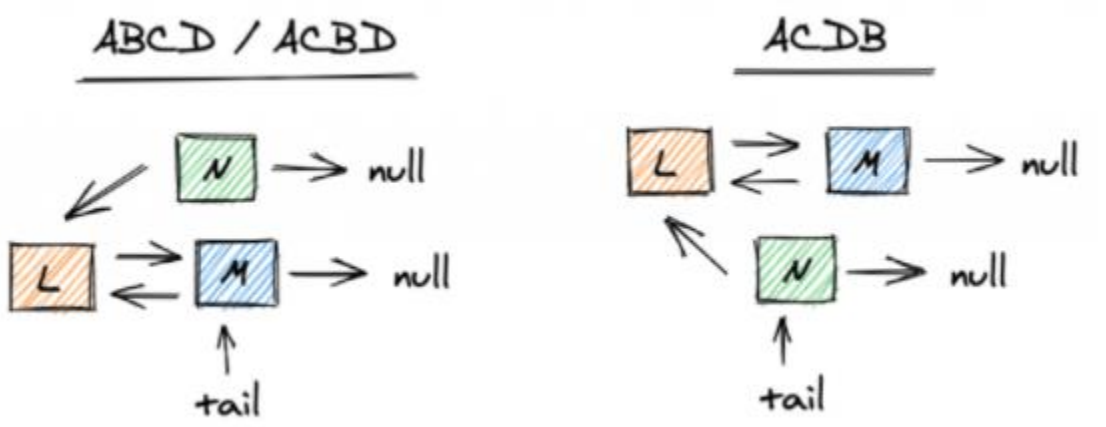
2) have the reference to the previous element point to L and the reference to the next element of L point to N (M, respectively).

3) Have tail point to N (M, respectively):



What can go wrong if the two threads perform these steps simultaneously? If the steps in the above picture execute in the order ABCD or ACBD, L, as well as tail, will point to M. N will remain disconnected from the queue.

If the steps execute in the order ACDB, tail will point to N, while L will point to M, which will cause an inconsistency in the queue:



Of course, one way to solve this problem is to have one thread acquire a lock on the queue. The solution we'll look at in the following chapter will solve the problem with the help of a lock-free operation by using the CAS operation we've seen earlier.

5) A Non-Blocking Queue in Java :

Let's look at a basic lock-free queue in Java. First, let's look at the class members and the constructor:

```
public class NonBlockingQueue<T> {
```

```
    private final AtomicReference<Node<T>> head, tail;
    private final AtomicInteger size;

    public NonBlockingQueue() {
        head = new AtomicReference<>(null);
        tail = new AtomicReference<>(null);
        size = new AtomicInteger();
        size.set(0);
    }
}
```

The important part is the declaration of the head and tail references as AtomicReferences, which ensures that any update on these references is an atomic operation. This data type in Java implements the necessary compare-and-swap operation.

Next, let's look at the implementation of the Node class:

```
private class Node<T> {
    private volatile T value;
    private volatile Node<T> next;
    private volatile Node<T> previous;

    public Node(T value) {
        this.value = value;
        this.next = null;
```

```
    }

    // getters and setters
}
```

Here, the important part is to declare the references to the previous and next node as volatile. This ensures that we update these references always in the main memory (thus are directly visible to all threads). The same for the actual node value.

6) Lock-Free add :

Our lock-free add operation will make sure that we add the new element at the tail and won't be disconnected from the queue, even if multiple threads want to add a new element concurrently:

```
public void add(T element) {
    if (element == null) {
        throw new NullPointerException();
    }

    Node<T> node = new Node<>(element);
    Node<T> currentTail;
    do {
        currentTail = tail.get();
        node.setPrevious(currentTail);
```

```
    } while(!tail.compareAndSet(currentTail, node));

    if(node.previous != null) {
        node.previous.next = node;
    }

    head.compareAndSet(null, node); // for inserting the first
element
    size.incrementAndGet();
}
```

The essential part to pay attention to is the highlighted line. We attempt to add the new node to the queue until the CAS operation succeeds to update the tail, which must still be the same tail to which we appended the new node.

7) Lock-Free get :

Similar to the add-operation, the lock-free get-operation will make sure that we return the last element and move the tail to the current position:

```
public T get() {
    if(head.get() == null) {
        throw new NoSuchElementException();
    }

    Node<T> currentHead;
```

```
    Node<T> nextNode;
    do {
        currentHead = head.get();
        nextNode = currentHead.getNext();
    } while(!head.compareAndSet(currentHead, nextNode));

    size.decrementAndGet();
    return currentHead.getValue();
}
```

Again, the essential part to pay attention to is the highlighted line. The CAS operation ensures that we move the current head only if no other node has been removed in the meantime.

Java already provides an implementation of a non-blocking queue, the ConcurrentLinkedQueue. It's an implementation of the lock-free queue from M. Michael and L. Scott described in this report. An interesting side-note here is that the Java documentation states that it's a wait-free queue, where it's actually lock-free. The Java 8 documentation correctly calls the implementation lock-free.

8) Conclusion :

In this report, we saw the fundamentals of non-blocking data structures. We explained the different levels and basic operations like compare-and-swap.

Then, we looked at a basic implementation of a lock-free queue in Java. Finally.

9) Results:

- we learned what non-blocking data structures are and why they are an important alternative to lock-based concurrent data structures.
- we have ben look at the basic building blocks of non-blocking algorithms like CAS (compare-and-swap).
- We learned how to implement a lock-free algorithm in the Java language.