SAINT PETERSBURG STATE
ELECTRICAL UNIVERSITY"LETI"

Department of Computer Science and Engineering

REPORT
on
LOCK-FREE DATA STRUCTURE

Submitted by ____________________ Bushra Ahmad

Supervisor ____________________ Prof. Natalya V. Shevskaya

Summary

Lock-free data structures

In a multi-threaded environment, the lock-free algorithms provide a way in which threads can access the shared resources without the complexity of Locks and without blocking the threads forever. These algorithms become a programmer’s choice as they provide higher throughput and prevent deadlocks.

This is mainly because designing lock-based algorithms, for concurrency brings its own challenges on the table. The complexity of writing efficient locks and synchronization to reduce thread contention is not everyone’s cup of tea. And besides, even after writing the complex code, many times, hard-to-find bugs occur in the production environments, where multiple threads are involved, which becomes even more difficult to resolve.

For a data structure to qualify as lock-free, more than one thread must be able to access the data structure concurrently. They don’t have to be able to do the same operations; a lock-free queue might allow one thread to push and one to pop but break if two threads try to push new items at the same time. Not only that, but if one of the threads accessing the data structure is suspended by the scheduler midway through its operation, the other threads must still be able to complete their operations without waiting for the suspended thread.

A data structure provides lock-freedom if, at any time, at least one thread can proceed. All other threads may be starving. The difference to obstruction-freedom is that there is at least one non-starving thread even if no threads are suspended.

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 ***AtomicReference***s, which ensures that any update on these references is an atomic operation**. This data type in Java implements the necessary *compare-and-swap* operation.**

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.

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.

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.

Conclusion

Lock-free algorithms and data structures is a much-debated topic in the Java World. When using lock-based or lock-free algorithms, a thorough understanding of the system must be done. One must be very mindful to use either of them. There’s no “one size fits all” solution or algorithm for different types of concurrency problems. So, deciding what algorithm suits best in a situation, is a crucial part of programming in the Multi-threaded World.