Prepared by :

Said Faycal Omar
Moubarek Barre Hassan

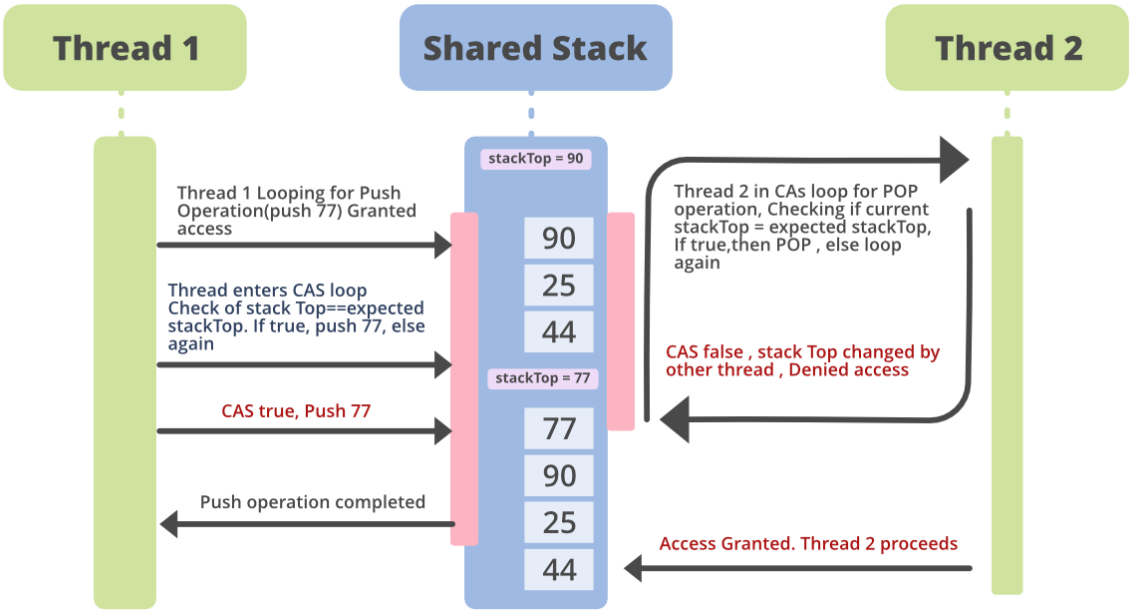
I) Lock-free data structure and algorithms

The most important operation which is the basic building block for the lock-free algorithms is the compare and swap. It compiles into a single hardware operation, which makes it faster as the synchronization appears on a granular level. Also, this operation is available in all the Atomic Classes. CAS aims at updating the value of a variable/reference by comparing it with its current value.

A) Applying CAS for a Non-Blocking Stack:

A non-blocking stack basically means that the operations of the stack are available for all the threads and no thread is blocked. To use CAS in the stack operations, a loop is written wherein the value of the top node (called stack top) of the stack is checked using CAS. If the value of stackTop is as expected, it is replaced with the new top value, else nothing is changed and the thread goes into the loop again.

Let's say we have an Integer Stack. Suppose, thread1 wants to push a value 77 onto the stack when the top of the stack value is 90. And thread2 wants to pop the top of the stack which is 90, currently. If thread1 tries to access the Stack and is granted access because no other thread is accessing it at that time, then the thread first gets the latest value of stack top. Then it enters the CAS loop and checks the stack top with the expected value (90). If the two values are the same, ie: CAS returned true, which means no other thread has modified it, the new value (77 in our case) is pushed onto the stack. And 77 becomes the new stack top. Meanwhile, thread2 keeps looping the CAS, until CAS returns true, for popping an item from the top of the stack. This is pictured below in the diagram.



B) Code Example for the Non-Blocking Stack :

The *Stack* code sample is shown below. In this example, there are two stacks defined. One which uses *traditional synchronization(*named *ClassicStack* here*)* to achieve concurrency control. The other stack uses the compare-and-set operation of the **AtomicReference** class for establishing a lock-free algorithm(named as *LockFreeStack* here). Here we are counting the number of operations performed by the Stack in a span of 1/2 a second. We compare the performance of the two stacks below :

```
// Java program to demonstrate Lock-Free
// Stack implementation
import java.io.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.locks.LockSupport;

class GFG {

    public static void main(String[] args)
```

```
        throws InterruptedException
{

        // Defining two stacks
        // Uncomment the following line to see the
        // standard stack implementation.
        // ClassicStack<Integer> operStack = new
        // ClassicStack<Integer>(); Lock-Free Stack
        // definition.

        LockFreeStack<Integer> operStack
                = new LockFreeStack<Integer>();
        Random randomIntegerGenerator = new Random();

        for (int j = 0; j < 10; j++) {
                operStack.push(Integer.valueOf(
                        randomIntegerGenerator.nextInt()));
        }

        // Defining threads for Stack Operations
        List<Thread> threads = new ArrayList<Thread>();
        int stackPushThreads = 2;
        int stackPopThreads = 2;

        for (int k = 0; k < stackPushThreads; k++) {
                Thread pushThread = new Thread(() -> {
                        System.out.println("Pushing into stack...");

                        while (true) {
                                operStack.push(Integer.valueOf(
                                        randomIntegerGenerator.nextInt()));
                        }
                });

                // making the threads low priority before
                // starting them
```

```
                pushThread.setDaemon(true);
                threads.add(pushThread);
        }

        for (int k = 0; k < stackPopThreads; k++) {
                Thread popThread = new Thread(() -> {
                        System.out.println(
                                "Popping from stack ...");
                        while (true) {
                                operStack.pop();
                        }
                });

                popThread.setDaemon(true);
                threads.add(popThread);
        }

        for (Thread thread : threads) {
                thread.start();
        }
        Thread.sleep(500);

        System.out.println(
                "The number of stack operations performed in 1/2 a second-->"
                + operStack.getNoOfOperations());
}

// Class defining the implementation of Lock Free Stack
private static class LockFreeStack<T> {

        // Defining the stack nodes as Atomic Reference
        private AtomicReference<StackNode<T> > headNode
                = new AtomicReference<StackNode<T> >();
        private AtomicInteger noOfOperations
                = new AtomicInteger(0);
```

```
public int getNoOfOperations()
{
        return noOfOperations.get();
}

// Push operation
public void push(T value)
{
        StackNode<T> newHead = new StackNode<T>(value);

        // CAS loop defined
        while (true) {
                StackNode<T> currentHeadNode
                        = headNode.get();
                newHead.next = currentHeadNode;

                // perform CAS operation before setting new
                // value
                if (headNode.compareAndSet(currentHeadNode,
                                                                newHead)) {
                        break;
                }
                else {
                        // waiting for a nanosecond
                        LockSupport.parkNanos(1);
                }
        }

        // getting the value atomically
        noOfOperations.incrementAndGet();
}

// Pop function
public T pop()
{
        StackNode<T> currentHeadNode = headNode.get();
```

```
                // CAS loop defined
                while (currentHeadNode != null) {
                        StackNode<T> newHead = currentHeadNode.next;
                        if (headNode.compareAndSet(currentHeadNode,
                                                                newHead)) {
                                break;
                        }
                        else {
                                // waiting for a nanosecond
                                LockSupport.parkNanos(1);
                                currentHeadNode = headNode.get();
                        }
                }
                noOfOperations.incrementAndGet();
                return currentHeadNode != null
                        ? currentHeadNode.value
                        : null;
        }
}

// Class defining the implementation
// of a Standard stack for concurrency
private static class ClassicStack<T> {

        private StackNode<T> headNode;

        private int noOfOperations;

        // Synchronizing the operations
        // for concurrency control
        public synchronized int getNoOfOperations()
        {
                return noOfOperations;
        }
```

```
        public synchronized void push(T number)
        {
            StackNode<T> newNode = new StackNode<T>(number);
            newNode.next = headNode;
            headNode = newNode;
            noOfOperations++;
        }

        public synchronized T pop()
        {
            if (headNode == null)
                return null;
            else {
                T val = headNode.getValue();
                StackNode<T> newHead = headNode.next;
                headNode.next = newHead;
                noOfOperations++;
                return val;
            }
        }
    }

    private static class StackNode<T> {
        T value;
        StackNode<T> next;
        StackNode(T value) { this.value = value; }

        public T getValue() { return this.value; }
    }
}
```

Output:

Pushing into stack...
Pushing into stack...
Popping from stack ...
Popping from stack ...
The number of stack operations performed in 1/2 a second-->28514750

The above output is received from implementing the **Lock Free data structure**. We see there are 4 different threads, 2 for pushing and 2 for popping from Stack. The number of operations means either Pop or Push operations on the stack.

To compare it with the standard stack version where traditional synchronization is used for concurrency, we can just uncomment the first line of the code and comment on the second line of code as follows.

```
//* Lock Based Stack programming
// This will invoke the lock-based version of the stack.
import java.io.*;

class GFG {
	public static void main(String[] args)
	{
		ClassicStack<Integer> operStack = new ClassicStack<Integer>();

		// LockFreeStack<Integer> operStack = new LockFreeStack<Integer>();
	}
}
```

The output from the **lock-based stack** is below. It clearly shows that the lock-free implementation(above) provides almost 3 times more output.

Output:

Pushing into stack...
Pushing into stack...
Popping from stack ...
Popping from stack ...
The number of stack operations performed in 1/2 a second-->8055597

II) Conclusion

Though lock-free programming offers a myriad of benefits, programming it correctly is no trivial task.

Pros:

- Truly Lock-free programming.
- Deadlock Prevention.
- Higher Throughput.

Cons

- The A-B-A problem can still happen in a lock-free algorithm(which is a change in the value of a variable from A to B then back to A while two threads are reading the same value A, without the other thread knowing it)
- Lock-free algorithms may not always be very easy to code.

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.