Table of Contents

Specs

Increase and decrease a variable on different threads 1.000.000 times. Record its final value.

Code

The code is also available in the Race.java file.

```
public class Race {
    public static void main(String[] args) {
        Counter counter = new Counter(0);

        var iThread = new IThread(counter);
        iThread.start();

        var dThread = new DThread(counter);
        dThread.start();

        System.out.println(counter.value());
    }
}

class IThread extends Thread {

    Counter _counter;

    public IThread(Counter counter) {
        _counter = counter;
    }

    public void run() {
        for (int i = 0; i < 1_000_000; i++) {
            _counter.inc();
        }
    }
}

class DThread extends Thread {
```

```
    Counter _counter;

    public DThread(Counter counter) {
        _counter = counter;
    }

    public void run() {
        for (int i = 0; i < 1_000_000; i++) {
            _counter.dec();
        }

    }
}

class Counter {
    private int _val;

    public Counter(int n) {
        _val = n;
    }

    public void inc() {
        _val++;
    }

    public void dec() {
        _val--;
    }

    public int value() {
        return _val;
    }
}
```

Results

100 runs have been recorded.

The following graph illustrates the final value of the counter. The average value is `-12920.24`.

The final value is non-deterministic, as the code is not thread-safe. Two threads access read and modify the same variable at once without any synchronization.

For example, the following behavior may happen:

1. Thread `I` (Incrementing) reads the counter's value, which is `0`.

2. Thread `D` (Decrementing) reads the counter's value, which is `0`, as thread `I` has not written the decremented value to the shared memory yet.
3. Thread `D` writes `-1` to the counter's value.
4. Thread `I` writes `1` to the counter's value.

Locking

To solve the problem above, a simple lock based on an `AtomicBoolean` is written:

```
class Lock {
    private AtomicBoolean isLocked = new AtomicBoolean(false);

    public void lock() {
        while (!isLocked.weakCompareAndSetAcquire(false, true))
            Thread.yield();
    }

    public void unlock() {
        isLocked.set(false);
    }
}
```

And the entire program looks as follows:

```
import java.util.concurrent.atomic.AtomicBoolean;

public class Race {
    public static void main(String[] args) throws Exception {
        Counter counter = new Counter(0);

        var iThread = new IThread(counter);
        iThread.start();

        var dThread = new DThread(counter);
        dThread.start();

        iThread.join();
        dThread.join();

        System.out.println(counter.value());
    }
}

class IThread extends Thread {

    Counter _counter;

    public IThread(Counter counter) {
```

```
            _counter = counter;
        }

        public void run() {
            for (int i = 0; i < 1_000_000; i++) {
                _counter.inc();
            }
        }
    }

    class DThread extends Thread {

        Counter _counter;

        public DThread(Counter counter) {
            _counter = counter;
        }

        public void run() {
            for (int i = 0; i < 1_000_000; i++) {
                _counter.dec();
            }

        }
    }

    class Counter {
        private int _val;
        private Lock _lock = new Lock();

        public Counter(int n) {
            _val = n;
        }

        public void inc() {
            _lock.lock();
            _val++;
            _lock.unlock();
        }

        public void dec() {
            _lock.lock();
            _val--;
            _lock.unlock();
        }

        public int value() {
            return _val;
        }
    }
```

```java
class Lock {
    private AtomicBoolean isLocked = new AtomicBoolean(false);

    public void lock() {
        while (!isLocked.weakCompareAndSetAcquire(false, true))
            Thread.yield();
    }

    public void unlock() {
        isLocked.set(false);
    }
}
```

Initially, the code used `compareAndSet` instead of `weakCompareAndSetAcquire`. However, `compareAndSet` did not work entirely, even though it did bring the final value of the counter closer to zero.

It is yet to be investigated why `weakCompareAndSetAcquire` works, while `compareAndSet` doesn't.