Lecture: Replication and Consistency
Exercise Sheet 5
https://pl.cs.uni-kl.de/homepage/de/teaching/ws19/rac/

Bad L(u/o)ck

Consider the following (incorrect) variant of a CLH Lock implementation:

```
public class BadCLHLock implements Lock {
    // most recent lock holder
    AtomicReference<Qnode> tail;
    // thread-local variable
    ThreadLocal<Qnode> myNode;

    public void lock() {
        Qnode qnode = myNode.get();
        qnode.locked = true; // I'm not done
        // Make me the new tail, and find my predecessor
        Qnode pred = tail.getAndSet(qnode);
        // spin while predecessor holds lock
        while (pred.locked) {}
    }

    public void unlock() {
        // reuse my node next time
        myNode.get().locked = false;
    }

    static class Qnode { // Queue node inner class
        public boolean locked = false;
    }
}
```

- How does this lock implementation differ from the CLH Lock that we discussed in class?
- Show how this implementation can go wrong!

Synchronization primitives

A common synchronization primitive implemented in hardware like MIPS, PowerPC and ARM is *load-link/store-conditional (ll/sc)*. Register objects with this primitive implement two operations (slightly simplified):

- *load-link* reads the value.
- *store-conditional* tries to write a value into the register. The write succeeds for a process p only if no other process has modified the register since the last *load-link* operation on it by p (returns `true`). Otherwise, it returns `false`.

Show how to implement an atomic increment operation and a lock using ll/sc.
Can you implement a constant-time CAS from ll/sc? What about implementing ll/sc from CAS?