CS5300 Theory Assignment 3

Gautam Singh
CS21BTECH11018

1) Suppose thread A acquires and releases this lock. Here, thread A's own node is referenced by both `tail` and `myNode`. Now, when A tries to acquire the lock once again, the following happens.
   a) Thread A sets it's `locked` field to `true` in line 8.
   b) Thread A swaps `tail` with `qnode` in line 10, but obtains its thread-local node again. In other words, A becomes its own predecessor.
   c) Thread A then spins on `pred.locked` which corresponds to the `locked` field in its own memory. In other words, thread A deadlocks itself.

   On the other hand, the MCS lock sets the tail node of the queue to `null` if there are no further threads waiting in the queue to signify the queue is empty. In this way, the thread reclaims its memory and does not deadlock itself.
2) Here are the implementations of `isLocked()` for various types of locks.
   a) A test-and-set lock is acquired if the atomic boolean variable `state` is set, which is shown in Code 1.
   b) A CLH lock is acquired when the `locked` field of the tail of the queue is set. Thus, we can simply check that to infer whether the lock is acquired. This is shown in Code 2.
   c) An MCS lock is acquired when the `tail` of the queue is set and its `locked` field is also set. Thus, we obtain a reference to the tail and check these two conditions. This is depicted in Code 3.

```
1 public class TASLock implements Lock {
2     public boolean isLocked() {
3         return state.get();
4     }
5 }
```

Code 1: Implementation of `isLocked` for `TASLock`.

```
1 public class CLHLock {
2     public boolean isLocked() {
3         QNode tnode = tail.get();
4         return tnode.locked;
5     }
6 }
```

Code 2: Implementation of `isLocked` for `CLHLock`.

```
1 public class MCSLock {
2     public boolean isLocked() {
3         QNode node = tail.get();
4         return node != null && node.locked;
5     }
6 }
```

Code 3: Implementation of `isLocked` for `MCSLock`.

3) The implementation of a "nested" readers-writers lock is shown in Code 4.

```
1 import java.util.concurrent.TimeUnit;
2 import java.util.concurrent.locks.Condition;
3 import java.util.concurrent.locks.Lock;
4 import java.util.concurrent.locks.ReadWriteLock;
5 import java.util.concurrent.locks.ReentrantLock;
6
7 public class NestedReadWriteLock implements ReadWriteLock {
8     // Counters and flags to maintain the number of readers,
```

```
    // writers and writers waiting for exclusive write access.
    int readers, waitingWriters;
    boolean writer;
    // Locks to be used: one for the counters
    // and two for actual use.
    Lock lock, readLock, writeLock;
    // Conditions to be used: one each for
    // the readers and writers.
    Condition readCondition, writeCondition;

    public NestedReadWriteLock() {
        // Initialize counters and flags
        readers = waitingWriters = 0;
        writer = false;
        // Create locks
        lock = new ReentrantLock();
        readLock = new ReadLock();
        writeLock = new WriteLock();
        // Create conditions
        readCondition = lock.newCondition();
        writeCondition = lock.newCondition();
    }

    public Lock readLock() {
        return readLock;
    }

    public Lock writeLock() {
        return writeLock;
    }

    protected class ReadLock implements Lock {
        public void lock() {
            lock.lock();
            try {
                // Wait until all writers and waiting writers are serviced
                while (writer || waitingWriters > 0) {
                    readCondition.await();
                }
                // Add to number of readers
                readers++;
            } finally {
                lock.unlock();
            }
        }

        public void unlock() {
            lock.lock();
            try {
                // Decrement number of readers
                readers--;
                // Signal a single waiting writer thread to wake up
                // if it exists and there are no more readers
                if (readers == 0 && waitingWriters > 0) {
                    writeCondition.signal();
                }
            } finally {
                lock.unlock();
```

```
67                  }
68              }
69
70          }
71
72          protected class WriteLock implements Lock {
73
74              // Threads must acquire the read lock first
75              public void lock() {
76                  lock.lock();
77                  try {
78                      // Decrement the number of readers
79                      readers--;
80                      // Increment the number of waiting writers
81                      waitingWriters++;
82                      // Wait until I am the only reader
83                      while (readers > 0 || writer) {
84                          writeCondition.await();
85                      }
86                      // Decrement the number of waiting writers
87                      waitingWriters--;
88                      // Become a writer with exclusive write access
89                      writer = true;
90                  } finally {
91                      lock.unlock();
92                  }
93              }
94
95              public void unlock() {
96                  lock.lock();
97                  try {
98                      // Unset writer flag
99                      writer = false;
100                     if (waitingWriters > 0) {
101                         // First, wake up waiting writers if any
102                         writeCondition.signal();
103                     } else {
104                         // Otherwise, wake up all readers
105                         readCondition.signalAll();
106                     }
107                     // Increment the number of readers
108                     readers++;
109                 } finally {
110                     lock.unlock();
111                 }
112             }
113
114         }
115
116 }
```

Code 4: Implementation of nested readers-writers lock.

To argue correctness of the lock, observe the following.

a) Suppose a thread holds the write lock. Then, it must have acquired the read lock as well and was the only reader in the system. Another thread looking to acquire the read lock must wait on the condition in line 46 of Code 4 since there is a writer thread. Since it cannot acquire the read lock, it definitely cannot acquire the write lock. Thus, the writer thread has exclusive write access.

b) Suppose a thread holds the read lock. Then, any other thread looking to acquire the write lock will first

acquire the read lock, but will then wait on the condition in line 46 of Code 4 since there are at least two reader threads.

This implementation is fair for the writers due to the FIFO re-entrant lock used in acquiring the read and write locks. All writers will eventually acquire the write lock.