CO21BTECH11002
Aayush Kumar
Theory Assignment 4

1)
The given CLH Lock implementation will cause a deadlock. Consider the following situation:

There are two threads P and Q. They perform the following method calls:

P: lock();
Q: lock();
P: unlock();
P: lock();

Once Q calls lock(), it considers the previous node, which belongs to P, as its predecessor. Consequently, Q continuously checks if P's node is available. Once P releases the lock by setting its node's "locked" state to false, Q was previously waiting in a loop (line 13) and got slowed down. Before Q can check that P's "locked" state is now false, P calls lock() and switches its node to a locked state. Initially, the tail was pointing to Q's node, but P then updates the tail to its own node (line 12) and designates Q's node as its predecessor. Now, P is stuck waiting for Q's node to unlock (Q's node is still marked as locked), and Q is similarly stuck waiting for P's node to unlock. As a result, neither P nor Q can acquire the lock, leading to a deadlock situation.

In an MCS lock, this scenario is avoided because when a thread performs an unlock(), it sets its successor's "locked" state to false. This means that even if the successor is slow, it will eventually read its "qnode.locked" as false, because of the action of its predecessor.

2)
```
class UnisexBathrooms {
    mutex restroom_mutex;
    condition_variable_any restroom_cv;
    int male_count, female_count; // inside bathroom
    int male_waiting_count, female_waiting_count; // waiting to get in
    int next_gender_allowed; // 0: none, 1: male, 2: female
```

```
public:
    UnisexBathrooms() {
        male_count = female_count = 0;
        male_waiting_count = female_waiting_count = 0;
        next_gender_allowed = 0;
    }

private:
    void male_enter() {
        restroom_mutex.lock();

        male_waiting_count++;

        while (female_count != 0 || next_gender_allowed == 2)
            restroom_cv.await();
        male_waiting_count--;

        male_count++;
        restroom_mutex.unlock();
    }

    void male_exit() {
        restroom_mutex.lock();
        male_count--;

        if (female_waiting_count)
            next_gender_allowed = 2;
        else
            next_gender_allowed = 0;

        restroom_mutex.unlock();

        restroom_cv.signal_all();
    }

    void female_enter() {
        restroom_mutex.lock();

        female_waiting_count++;

        while (male_count != 0 || next_gender_allowed == 1)
            restroom_cv.await();
        female_waiting_count--;

        female_count++;
        restroom_mutex.unlock();
    }

    void female_exit() {
        restroom_mutex.lock();
        female_count--;

        if (male_waiting_count)
            next_gender_allowed = 1;
        else
            next_gender_allowed = 0;

        restroom_mutex.unlock();

        restroom_cv.signal_all();
    }

public:
    void use_restroom(bool is_male, int id) {
        if (is_male) {
            male_enter();
            use_restroom_now();
            male_exit();
        } else {
            female_enter();
            use_restroom_now();
            female_exit();
        }
    }
}
```

The implementation ensures mutual exclusion by verifying the absence of the other gender and the appropriate flag before allowing entry. This guarantees that only one gender can access the bathroom at any given time, establishing mutual exclusion.

The design also ensures weak starvation freedom as, upon exit, the departing gender sets the flag for the opposite gender to true. This means that the other gender will be permitted to enter once the current occupants have left, preventing any single gender from being indefinitely denied access and ensuring a degree of fairness.

3)

```
public
class MCSLock implements Lock
{
        AtomicReference<QNode> tail;
        ThreadLocal<QNode> myNode;
public
        MCSLock()
        {
                tail = new AtomicReference<QNode>(null);
                myNode = new ThreadLocal<QNode>()
                {
                protected
                        QNode initialValue()
                        {
                                return new QNode();
                        }
                }
        }
public
        boolean lock(long time, TimeUnit unit)
        {
                long startTime = System.currentTimeMillis();
                long patience = TimeUnit.MILLISECONDS.convert(time, unit);
```

```
            QNode qnode = myNode.get();
            QNode myPred = tail.get and set(qnode);
            if (myPred != null)
            {
                qnode.locked = true;
                myPred.next = qnode;
                qnode.pred = myPred;
                // wait until predecessor gives up the lock
                while (System.currentTimeMillis() - startTime < patience)
                {
                    if (!qnode.locked)
                        return true;
                }
                // abandoned
                qnode.pred.next = qnode.next;
                if (qnode.next != null)
                    qnode.next.pred = qnode.pred;
                // if node that was at the end of queue is abandoned, update tail
                tail.compareAndSet(qnode, myPred)
                return false;
            }
            else
            {
                return true;
            }
        }
public
        void unlock()
        {
            QNode qnode = myNode.get();
            if (qnode.next == null)
            {
                if (tail.compareAndSet(qnode, null))
                    return;
                // wait until successor fills in its next field
                while (qnode.next == null)
                {
                }
            }
            qnode.next.locked = false;
            qnode.next = null;
        }
        class QNode
        {
            volatile boolean locked = false;
            volatile QNode next = null;
            volatile QNode pred = null;
        }
}
```

This is a modified abortable MCSLock(). The lock() method is used to acquire the lock. It takes a timeout value as input. It first gets the current thread's QNode from the myNode ThreadLocal variable. It attempts to set the myNode as the new tail of the queue by using an atomic operation (compare-and-set) on the tail variable. If there was a previous tail (i.e., if myPred is not null), it means there are other threads waiting in the queue. In that case, the current thread marks itself as "locked" and updates the previous thread's next pointer and its own pred pointer to create the queue structure. The method then enters a loop where it checks if the qnode is still locked. It waits for a while, up to the specified timeout, for the predecessor thread to release the lock. If the timeout is exceeded and the qnode is still locked, it is considered abandoned, and the current thread adjusts the pointers in the queue to skip itself and returns false.