CS5300: Parallel and Concurrent Programming
Final Exam

- ABBURI VENKATA SAI MAHESH
- CS18BTECH11001

1. When we consider the given implementation, we can observe that the given instruction-set is mostly similar to the CAS operation. The only difference is that the given instruction set is using pointers instead of variables as used by the CAS operation. By using pointer we can able to build locks for any type of architecture and the implementation given is not bounded to any space constraint and can be used for any number of threads(largest n for which the implementation solves the n-thread consensus doesn't exist). So, therefore we can say that the consensus number is infinite.

2. The CLH Lock uses:
```
public class CLHLock implements Lock {
        AtomicReference<Qnode> tail;
        ThreadLocal<QNode> myPred;
        ThreadLocal<QNode> myNode;
}
```
   From this, we can observe that each lock uses an atomic Qnode for the tail and thus L locks use O(L) space. We can also able to see that the algorithm is using the thread-local variable(same for all locks - space is considered for each thread) and thus making it O(n) where n is the number of threads. Therefore the total time complexity of the CLH lock algorithm is O(L+n) where L is the number of Locks and n is the number of threads.

3. In the unlock() method, a thread checks, using compareAndSet(), whether it has a successor and if so sets its pred field to AVAILABLE. So it is not safe to recycle a thread's old node at this point, since the node may be referenced by its immediate successor, or by a chain of such references. But the nodes in such a chain can be recycled as soon as a thread skips over the timed-out nodes and enters the critical section.

4. a. For CLH Lock:

```
isLocked(){
        QNode qnode = myNode.get()
        Qnode pred = tail.get()
        if(qnode.locked == true && pred.locked == true)
                return true
        return false
}
```

Explanation: Here the qnode.locked == true checks that the present thread is holding a lock and pred.locked == true check whether any other thread(not the present thread) has already acquired the lock(then the present thread hasn’t acquired the lock due to mutual exclusion)

b. For MCS Lock:

```
isLocked(){
        Qnode qnode = myNode.get()
        Qnode pred = tail.get()
        if(pred.next == qnode && qnode.locked == true)
                return true
        Return false
}
```

Explanation: Here the pred.next == qnode checks that the present thread is holding a lock and qnode.locked == true check whether any other thread(not the present thread) has already acquired the lock(then the present thread hasn’t acquired the lock due to mutual exclusion) but haven’t yet unlocked(setting qnode.next.locked = false)

5. When the contains implementation does not check the ‘marked’ bit node a is marked as removed (its marked field is set) and thread A is attempting to find the node matching a’s key. While A is traversing the list, curr A and all nodes between curr A and a including a are removed, both logically and physically. Thread A would still proceed to the point where curr A points to a, and would detect that a is marked and no longerin the abstract set. The call could be linearized at this point.

6. a. if the threads do not help each other then the synchronization problem exits. And to resolve them, there would require the use of an external lock which makes it not lock-free.
   b. Without helping, the implementation would be blocked by other threads, and therefore it isn't objection free anymore.

7. a. In the Lazy synchronization, an item is in the set if, and only if it is referred to by an unmarked reachable node. So we should always ensure that the unmarked reachable node remains reachable even if its predecessor is logically or physically deleted.
   But when we insert curr = null in line 18(as a part of garbage collection), after deleting the node logically and physically, the unmarked reachable node will no longer be reachable, and therefore the item next to the deleted one is not referred to as not there in the list. So we should not use this line as part of garbage collection.

   In particular lets say we are deletin a node *curr1* and *pred1* be its predeessor node and *succ1* be its successor node. Not consider that the following steps executed in contains() and remove() methods
   1. curr = curr1   // the curr value in contains()
   2. curr1.marked = true   // curr1 is logically deleted
   3. Pred1.next = curr1.next   // curr1 is physically deleted
   4. Curr1 = null   // clearing curr1 for garbage
   5. Curr = curr1.next   // moving to next node in contain()

   Then the line 5 will face a segmentation fault as curr1 has set to null by the remove method.

   b. For clearing null in C++ we should maintain a pred local variable for contains() method to avoid pointing to null value
   <u>The modified remove():</u>

```
        if (validate(pred, curr)) {
                if (curr.key != key) {
                        return false;
                } else {
                        curr.marked = true;
                        pred.next = curr.next;
                        Curr = null;
                        return true;
                    }
                }
```

The modified contains():

```
        int key = item.hashCode();
        Node pred = head;
        Node curr = head.next;
        while (curr.key < key){
            if(curr = null)
                Curr = pred.next;
            Pred = curr; curr = curr.next;
        }
        return curr.key == key && !curr.marked;
```

Explanation: This will free the garbage value in the remove itself and will avoid the segmentation faults in contains method too and thus a perfect implementation.

8. a. Pseudocode for Iterative Distributive Algorithm:

```
graphcolouring(){
    Barrier b
    while(true){
        thread[id].colour_local()
        b.await();
        Syncronised{
            for(v: vertex_local){
                for(v1: v.neighbours){
                    if(v.color = v1.color && priority[thread[id]] <
priority[v.thread_id])
                        v.color = colours.next
                }
            }
        }
        if(total_graph_check())
                break
}
```

Explanation: first color the vertices taken by each thread. Wait for all threads to complete their coloring(used await() barrier). Check for the neighbor nodes coloring for any conflicts. If any conflict, check the priority of the present thread with the thread that contains the neighbor node. If the priority is less then change the color of the present vertex. Loop until all the graph vertices are colored perfectly(without any conflicts).

b. If I haven't used the barrier(await() method) then each thread wouldn't wait for the other threads to complete their vertices coloring. Then we might get into an infinite while loop(as the total_graph_check() wouldn't return true and leading to no execution for break statement).

c. We can implement a non-blocking algorithm by using a continuous inner while loo for checking each time the color of the neighbor is changed. See the pseudo-code for the implementation.

```
graphcolouring(){
        thread[id].colour_local()
        while(true)
                for(v: vertex_local){
                        for(v1: v.neighbours){
                                if(v.color = v1.color && priority[thread[id]] <
priority[v.thread_id])
                                        v.color = colours.next
                        }
                }
                if(local_graph_check())
                        break
        }
```

Explanation: Here each thread will loop until the vertices assigned to the present thread has no conflicts with their neighbors and will loop until the condition is satisfied. Once the condition is satisfied the thread exits the critical state instead of waiting for all threads to complete the whole graph.