ECSE 420 – Assignment 3

Group 11

Yan Ren Fei Feng

Q1

After a thread has held and released the lock, the tail and myNode pointed to the same node. If this thread wants to hold the lock again, it will set qnode.locked equals to true, and its predecessor is also this node. Therefore it will always wait for this node to become unlocked, which is a deadlock. Hence this implementation is wrong.

Q2

2.1 *The complete source code is in FineList.java*

```
public boolean contains(T item) {
    Node pred = null;
    Node curr = null;
    int key = item.hashCode();
    head.lock();
    try {
        pred = head;
        curr = pred.next;
        curr.lock();
        try {
            while (curr.key < key) {
                pred.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock();
            }
            return (curr.key == key);
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }
}
```

2.2 The test verifies that contains() method works well. In the contains() method, it uses hand-over-hand locking, which locks both pred and curr to ensure the safety. Additionally, it acquires locks in the same order as in the add() method, so there will be no deadlock. The test is done in JUnit.

The complete source code is in FineListTest.java

```
@Test
public void testContains() {
    FineList l = new FineList();
    l.add(1);
    l.add("a");

    assertTrue(l.contains(1));
    assertTrue(l.contains("a"));
    assertFalse(l.contains(5));
    assertFalse(l.contains("b"));
}
```

Q3

3.1 To design a bounded lock-based queue using array. We define array “items” for storing data. Two locks “headLock” for dequeue method, and “tailLock” for enqueue method. “headLock” has condition “notEmpty” for the case when the queue is empty. “tailLock” has the condition “notFull” for the case when the queue is full. We also define two integers “head” and “tail” for tracking the index where to deq and enq in array. We define AtomicInteger “size” for tracking the empty space of the queue.

In enq method, we acquire “tailLock” first, and use “size” to check if the queue is full, if the queue is full the thread waits on “notFull” condition. If the queue is not full we save the data to array by using “tail” index and increment index. Then we check if the queue size was zero before adding. If it was zero meaning there are probably some threads that are waiting on “notEmpty” condition to dequeue the data. In that case we need to “notEmpty.signalAll()” to wake up waiting threads. Note that for Java implementation, since “notEmpty” is associated with “headLock”, we need to acquire lock before using signal otherwise will get illegalMonitorStateException.

In deq method, we acquire “headLock” first, and use “size” to check if the queue is empty, if the queue is empty the thread waits on “notEmpty” condition. If the queue is not empty we get the data from array by using “head” index and increment index. Then we check if the queue size was full before deq. If it was full meaning there are probably some threads that are waiting on “notFull” condition to enqueue the data. In that case we need to “notFull.signalAll()” to wake up waiting threads. Note that for Java implementation, since “notFull” is associated with “tailLock”, we need to acquire lock before using signal otherwise will get illegalMonitorStateException.

The complete Java code implementation is as following and also available in BoundedLockBasedQueue.java

```
package ca.mcgill.ecse420.a3;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
```

```
import java.util.concurrent.locks.ReentrantLock;

public class BoundedLockBasedQueue<T> {
   private T[] items;

   private int head = 0;
   private int tail = 0;

   AtomicInteger size;

   private Lock headLock = new ReentrantLock();
   private Lock tailLock = new ReentrantLock();

   private Condition notEmpty = headLock.newCondition();
   private Condition notFull = tailLock.newCondition();

   @SuppressWarnings("unchecked")
   public BoundedLockBasedQueue(int capacity) {
         items = (T[]) new Object[capacity];
         size = new AtomicInteger(0);
   }

   public void enq(T item) {
         boolean mustWakeDequeuers = false;
         tailLock.lock();
         try {
               // check if queue is full
               while (size.get() == items.length) {
                     notFull.await();
               }
               // save into queue
               items[tail % items.length] = item;
               tail++;
               // if queue was empty, signal notEmpty condition just in case
               // some threads are waiting on the condition
               if (size.getAndIncrement() == 0)
                     mustWakeDequeuers = true;

         } catch (Exception e) {
               e.printStackTrace();
         } finally {
               tailLock.unlock();
         }
```

```
        if (mustWakeDequeuers) {
                headLock.lock();
                try {
                        notEmpty.signalAll();
                } finally {
                        headLock.unlock();
                }
        }
}

public T deq() {
        boolean mustWakeEnqueuers = false;
        T result = null;
        headLock.lock();
        try {
                // check if queue is empty
                while (size.get() == 0) {
                        notEmpty.await();
                }
                // remove from queue
                result = items[head % items.length];
                head++;
                // if queue was full, signal notFull condition just in case
                // some enq threads are waiting on the condition
                if (size.getAndDecrement() == items.length) {
                        mustWakeEnqueuers = true;
                }
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                headLock.unlock();
        }
        if (mustWakeEnqueuers) {
                tailLock.lock();
                try {
                        notFull.signalAll();
                } finally {
                        tailLock.unlock();
                }
        }
        return result;
```

```
    }

}
```

3.2 The queue has one array “items” for storing data. It also has three indexes. “tail” is where a new element will be inserted to. “head” is where the next element will be extracted. “tailCommit” points to the place where the latest “committed” data has been inserted. This index is for indicating where the actual data is located. If “tailCommit” is not the same as “tail” meaning there is a place reserved for data inserting but the actual data is not in the queue yet. The thread trying to read will have to wait for saving the data completes. All three indexes use AtomicInteger which have CAS operation.

In enq method, when a new element is going to be added, we reserve the place by incrementing “tail” index. Then the data is added to place where is reserved. It is also the index of “tailCommit”. Then increments the “tailCommit”. If multiple threads insert data at the same time, they will only insert to the place they reserved and in a strict order. This line of code “while (!tailCommit.compareAndSet(currentWriteIndex, currentWriteIndex + 1)) { };” will guarantee the order.

In deq method, “head“ index points to the next data will be read from the queue. Deq thread first copies the data pointed by “head” index and perform CAS on “head” index. If the CAS succeeds, the thread gets the data from the queue. Only one thread can increment “head” index by CAS. If CAS failed, deq thread will enter loop again to repeat.

The complete Java code implementation is as following and also available in BoundedLockFreeQueue.java

```
package ca.mcgill.ecse420.a3;

import java.util.concurrent.atomic.AtomicInteger;

public class BoundedLockFreeQueue<T> {
    private T[] items;

    private AtomicInteger head = new AtomicInteger(0);
    private AtomicInteger tail = new AtomicInteger(0);
    private AtomicInteger tailCommit = new AtomicInteger(0);

    @SuppressWarnings("unchecked")
    public BoundedLockFreeQueue(int capacity) {
        items = (T[]) new Object[capacity];
    }
```

```
public void enq(T item) throws InterruptedException {
	int currentWriteIndex;
	int currentReadIndex;

	do {
		currentWriteIndex = tail.get();
		currentReadIndex = head.get();

		// check queue is full
		if (currentWriteIndex - currentReadIndex == items.length) {
			return;
		}
	} while (!tail.compareAndSet(currentWriteIndex, currentWriteIndex + 1));

	// index is reserved for us. Use it to save the data
	items[currentWriteIndex % items.length] = item;

	//
	while (!tailCommit.compareAndSet(currentWriteIndex, currentWriteIndex + 1)) { };
}

public T deq() throws InterruptedException {
	int currentReadIndex;
	int currentReadMaxIndex;
	T item = null;

	do {
		currentReadIndex = head.get();
		currentReadMaxIndex = tailCommit.get();
		// the queue is empty
		if(currentReadMaxIndex - currentReadIndex == 0) {
			break;
		}

		//retrieve the data from the queue
		item = items[currentReadIndex % items.length];

		// try to perform now the CAS operation on the head index.
		if(head.compareAndSet(currentReadIndex, currentReadIndex+1)) {
			break;
		}

	}while(true);
```

```
            return item;

    }
}
```

Q4

4.1 MatrixVectorSeqMul.java implements the sequential matrix vector multiplication. Run the code in main to test the runtime for 2000 by 2000 matrix.

4.2 Complete code implementation is in Matrix.java, Vector.java and MatrixTask.java. Matrix.java class provides put() and get() methods to access matrix elements, along with a constant-time split() that splits an n-by-n matrix into four (n/2)-by-(n/2) submatrices. Vector.java class provides put() and get() methods to access vector elements, along with a constant-time split() that splits an n-length vector into two (n/2)-length subvectors. Matrix vector multiplication can be decomposed as four product terms and two sum terms. Product terms can be computed in parallel, and when those computations are done, sum terms can be computed in parallel. The sum terms are actually vector sum. Therefore, in MatrixTask.java we implement static AddTask class for parallel vector multiplication and static MulTask class for the parallel matrix vector multiplication task. The MulTask class creates two scratch arrays to hold the matrix product terms, which are two vectors. It splits all matrices and vectors, submits tasks to compute the four product terms in parallel, and waits for them to complete. Once they are complete, the thread submits tasks to compute the two vectors' sums in parallel, and waits for them to complete. Node that in AddTask, recursion stops when vector size is one. And in MulTask the recursion stops when matrix size is one.

4.3 Test program for execution time for sequential algorithm is in MatrixVectorSeqMul.java main. In MatrixTask.java, we have two methods test_matrix_vector_parallel_multiplication and test_vector_parallel_addition that test matrix vector multiplication and vector addition is giving correct result by using numerical values. Main method in MatrixTask.java capture the runtime for 2000 by 2000 matrix. To make full use of resource, we use CachedThreadPool and use ThreadPoolExecutor to get information about thread pool like largest number of threads that have ever been in the pool, approximate number of tasks that have ever been scheduled for execution and current thread numbers, etc.
Sequential runtime is 113 ms. Parallel runtime time is 5715 ms with largest number of simultaneous executions: 35806 and total number of threads ever scheduled: 3143680. On the 4 processors machine with 64G RAM, parallel algorithm is slower than sequential algorithm.

4.4 Let Ap(n) be the number of steps needed to add two length n vectors on P processors. Vector addition is split to two half-size vector additions, plus a constant amount of work to split the vector. The work $A_1(n)$ is given by the recurrence $A_1 = 2\ A_1(n/2) + \Theta(1) = \Theta(n)$. Let $M_p(n)$ be the number of steps needed to multiply one nxn matrices with n vector on P processors. Matrix vector multiplication requires four half-size matrix vector multiplications and two matrix additions.

The work $M_1(n)$ is given by the recurrence: $M_1(n) = 4M_1(n/2) + 2A_1(n) = 4M_1(n/2) + \Theta(n) = \Theta(n^2)$.
We divide the matrix to size one in multiplication, all multiplication can be done in parallel, therefore the work is $M_\infty(n) = M_\infty(n/2) + \Theta(1) = \Theta(\log n)$
The parallelism for matrix multiplication is given by:
$M1(n)/M_\infty(n) = \Theta(n^2/\log n)$