Parsa Yadollahi - 260869949
Nicholas Nikas - 260870980

ECSE 420 - Assignment 2 Report

Question 1:

1.1)
The implementation of the Filter Lock is shown in figure 1 in the appendix.

1.2)
Yes, it is possible for one thread to overtake another. To prove this we can use r-bounded waiting which implies there exists a fixed value of *r* such that after a process had made a request to enter its critical section and before it's been granted that permission to enter, no more than *r* other processes may enter. In the Filter Algorithm seen in class, there is no value for *r* which means that a thread can in fact be overtaken.

1.3)
The implementation of the Bakery Lock is shown in figure 2 in the appendix.

1.4)
No, the Bakery lock does not allow a thread to overtake another since the algorithm works on a first come first serve basis meaning the *r* in our r-bounded waiting is equal to 0. When the *r* is equal to 0, this means that no other process can overtake the current process requesting resources.

1.5)
We could use a counter and have say 20 threads increment this counter in the *run()* method. We have a set value that this counter can increase to, once it reaches that value, each of the threads should no longer increase its value. Using both Filter and Bakery, once we shut down all threads, the value should be at exactly 10, whereas if we were to not use any lock, this value would be at an arbitrary number.

1.6)
By implementing the test described in part 1.5, we can see from figure 1 that when using the FilterLock and BakeryLock classes, we achieve mutual exclusion because the counter never increments past 10, but when not using any lock, we get a random number greater than 10. We ran this code multiple times and noticed that without any lock, we would not always get the same number but it would always be over 10.

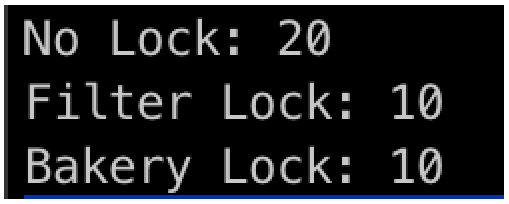

```
No Lock: 20
Filter Lock: 10
Bakery Lock: 10
```

Figure 1: Output when testing different locks.

Question 2:

LockOne will still satisfy two-thread mutual exclusion if we change its atomic register to regular register. This is because the flag variables do not require to be read a finite amount of times hence they are not affected by the value they receive whether it be before or after the overlapping write.

LockTwo will still satisfy two-thread mutual exclusion if we change its atomic register to regular register. This is because the victim variable is shared by both threads which is set to their own id to allow other threads to enter the CS. Going from atomic to regular register will just cause a minor delay in the case where the victim value is being read in the while loop of one thread and another thread is setting the victim to itself, the overlap will either cause the first thread to enter CS normally or loop once more before entering CS.

Question 3:

3.1)
Assume two threads CS_A^k and CS_b^j

From the code:

- $write_A(turn=A) \rightarrow write_A(busy=true)$
- $write_A(busy=true) \rightarrow read_A(turn==B) \rightarrow read_A(busy==true)$

From the assumption:

- $read_A(turn==B) \rightarrow write_B(turn=B)$
- $read_B(turn==A) \rightarrow write_A(turn=A)$

Combining all:

- $write_A(turn=A) \rightarrow write_A(busy=true) \rightarrow read_A(turn==B) \rightarrow write_B(turn=B) \rightarrow write_B(busy=true) \rightarrow read_B(turn==A) \rightarrow write_A(turn=A)$

Notice that the first term is the same as the last, meaning we get a cycle. In other words, CS_A^j overlaps with CS_B^k causing mutual exclusion.

3.2)

The protocol is not deadlock-free (i.e. it will deadlock). Once the process enters the *do-while* loop it will never exit that loop, especially since the *busy* variable will never get set to False.

When running the code and calling the *thirdlock.lock()*, the function stays stuck in the *lock()* method, and there's no chance for that or another thread to call *unlock()* on the thread no matter how many threads we use meaning it will never break out of the loop since the *busy* variable will always be set to true.

3.3)

The protocol can not be starvation-free since we showed in part 3.2 that the *ThirdLock* class causes deadlock. For example, we can look at the scenario proposed in part 3.2 where one thread waits forever for another one to arrive and release it from its loop causing starvation.

Question 4)

4.1

This history is sequentially consistent. We show this with the following execution order:

[A]r.write(0) → [B]r.write(1) → [A]r.read(1) → [A]r.write(2) → [B]r.read(2) → [C]r.read(2) → [C]r.write(3).

It is not linearizable, this is because [A] r.read(1) and r.write(1) overlap and [A] r.write(2) and r.read(2) also overlap. This overlap would cause some kind of interrupt.

4.2)

This history is not sequentially consistent. We show the following execution order:

[B]r.write(1) → [A]r.read(1) → [C]r.write(2) → [C]r.read(1) → [B]r.read(2).

We can allow B to execute r.read(2) right agter C executes r.write(2) but then it would be impossible for C to execute r.read(1) since B's r.write(1) will be overwritten by C's r.write(2).

It is also not linearizable because C has to execute r.read(1) before B does r.read(2) but there is no thread in the meantime that does r.write(2). Also, A and C's r.read(1) overlap with B's r.write(1).

Question 5)

5.1)

Yes. Since the boolean *v* is a volatile variable, this means that it guarantees visibility of changes across threads since reading and writing of that variable is done through the main memory and not from the CPU cache. So for a division by 0 to occur, the program would need to call *writer()* in one of the threads first then call *reader()* in another thread. By doing so we set the volatile variable *v* to true for all threads since it was written to the main memory, but the variable *x* is unchanged (equal to 0) for the second thread. Meaning, our second thread would go into the *reader()* function with *v* set to true and *x* set to 0 hence causing a division by 0.

5.2)

If both *x* and *v* are volatile, then **division by 0 is impossible** since the variables would be the same across all threads. Thus division by 0 could only occur if *v* is set to true, but if *v* is set to true then *x* is also first set to 42.

If both *x* and *v* are non-volatile then the divide by zero is possible because the *reader()* and *write()* function could be running at the same interval method call.

Question 6)

6.1) Condition for regular m-valued MRSW:

1. If a *write* (W*a*) is before a *read (*R*b)* s.t. a ≤ b and there is no other *write* (W*c*) between the initial write and read (i.e. W$a < Wc < Rb$), then the *read* should return the value written by the initial *write* W*a*.

False, adding the change to line 11 will cause the values before *x* to stay as they were and after *x* to be set to False. Doing so will interrupt the *read()* methods functionality since it reads values starting from the first index, incrementing until it reaches the first value set to True and returns that index. If the values before x do not get set to False, then the *read()* method could return an old index that was not the most recent change. Now assume we have two *write*'s, W*a* and W*b* s.t. $a < b$ then our *read* function would return the value set by W*a* which breaks the first condition.

6.2) Condition for safe m-valued MRSW:

1. If a *write* (W*a*) is before a *read (*R*b)* s.t. a < b and there is no other *write* (W*c*) between the initial write and read (i.e. W$a < Wc < Rb$), then the *read* should return the value written by the initial *write* W*a*.

False, similarly to 6.1, the states of all the indices before *x* remain the same but after *x* are set to False. We do not want this behavior in a safe Boolean MRSW. During overlaps, while reading and writing, we want to return the most recent value of a variable and not the old. But replacing line 11 could potentially return an old value.

Question 7)

We use proof by contradiction to show that binary consensus using atomic registers is impossible for two threads.

Assume that binary consensus is possible for *n > 2* threads, then for any 2 sets of threads in the set *n > 2*, the consensus is also possible. This creates a contradiction to our initial statement, implying it must also be impossible for *n* threads.

Question 8)

Similarly to question 7 we solve this by contradiction.

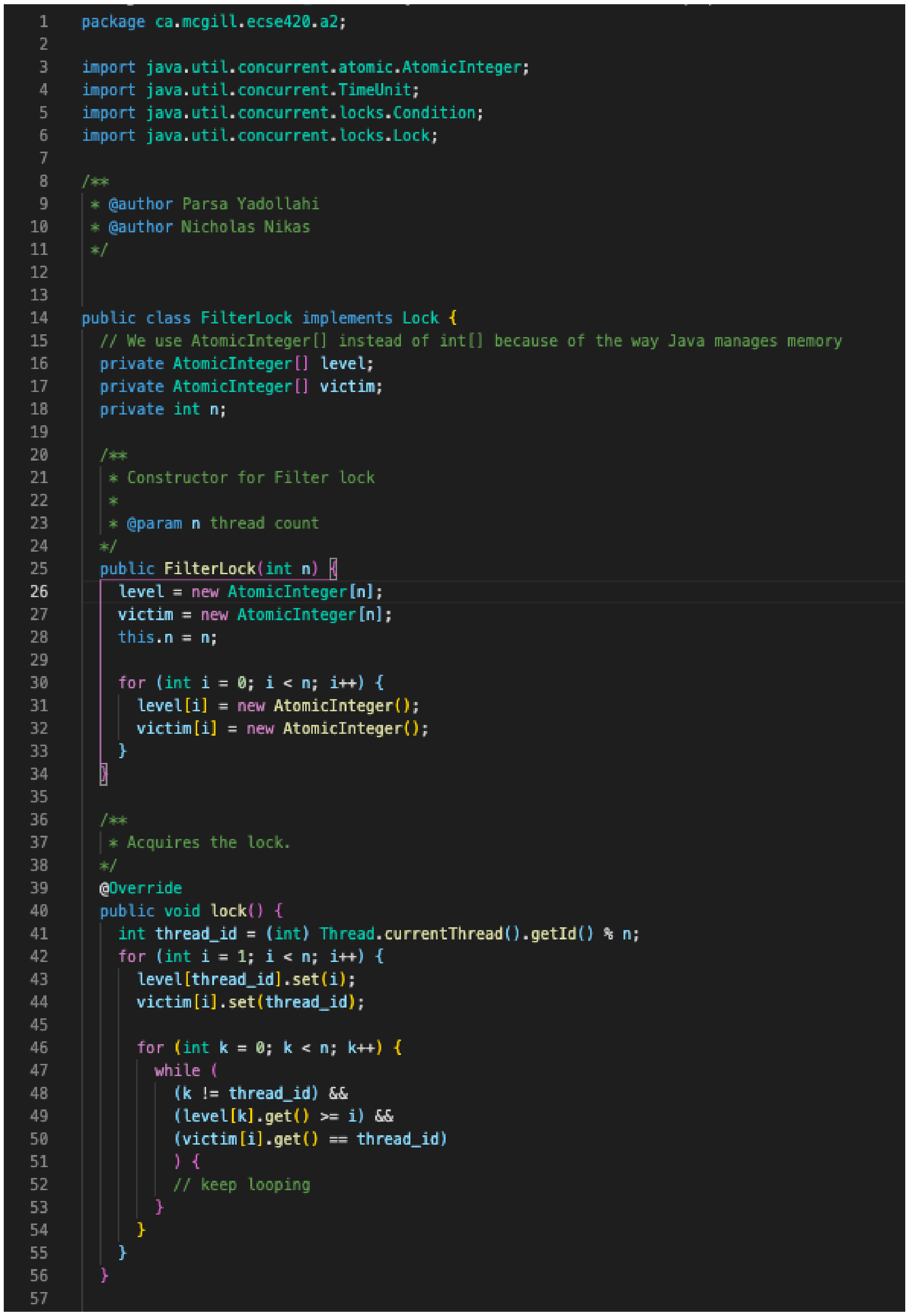
Assume that consensus is possible for *k* value greater than 2, and that binary consensus is impossible. Then we could reduce the consensus protocol to a binary consensus protocol by mapping [0, k/x] to 0 and [k/x, k] to 1 where $x \in N\, s.t.\, k/x \in \mathbb{N}$. This creates a contradiction to our initial statement since this protocol proves binary consensus is possible. Thus, consensus with k > 2 is impossible if binary consensus is also impossible.

Appendix

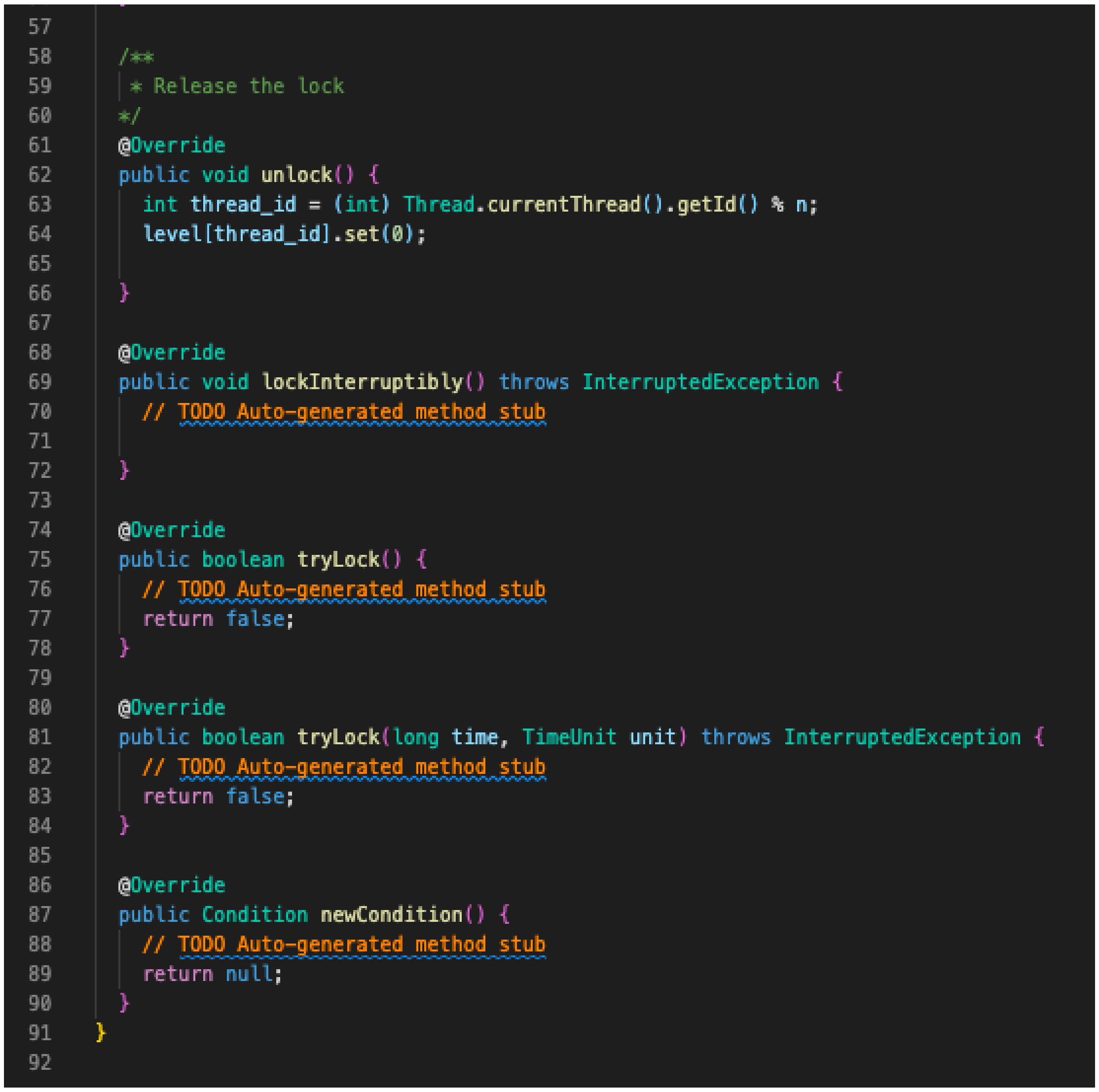
Figure 1 - Filter Lock:

To run this file, simply run the following commands from the src folder

- javac ca/mcgill/ecse420/a2/FilterLock.java
- java ca/mcgill/ecse420/a2/FilterLock



```java
package ca.mcgill.ecse420.a2;

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

/**
 * @author Parsa Yadollahi
 * @author Nicholas Nikas
 */


public class FilterLock implements Lock {
  // We use AtomicInteger[] instead of int[] because of the way Java manages memory
  private AtomicInteger[] level;
  private AtomicInteger[] victim;
  private int n;

  /**
   * Constructor for Filter lock
   *
   * @param n thread count
  */
  public FilterLock(int n) {
    level = new AtomicInteger[n];
    victim = new AtomicInteger[n];
    this.n = n;

    for (int i = 0; i < n; i++) {
      level[i] = new AtomicInteger();
      victim[i] = new AtomicInteger();
    }
  }

  /**
   * Acquires the lock.
  */
  @Override
  public void lock() {
    int thread_id = (int) Thread.currentThread().getId() % n;
    for (int i = 1; i < n; i++) {
      level[thread_id].set(i);
      victim[i].set(thread_id);

      for (int k = 0; k < n; k++) {
        while (
          (k != thread_id) &&
          (level[k].get() >= i) &&
          (victim[i].get() == thread_id)
        ) {
          // keep looping
        }
      }
    }
  }

```

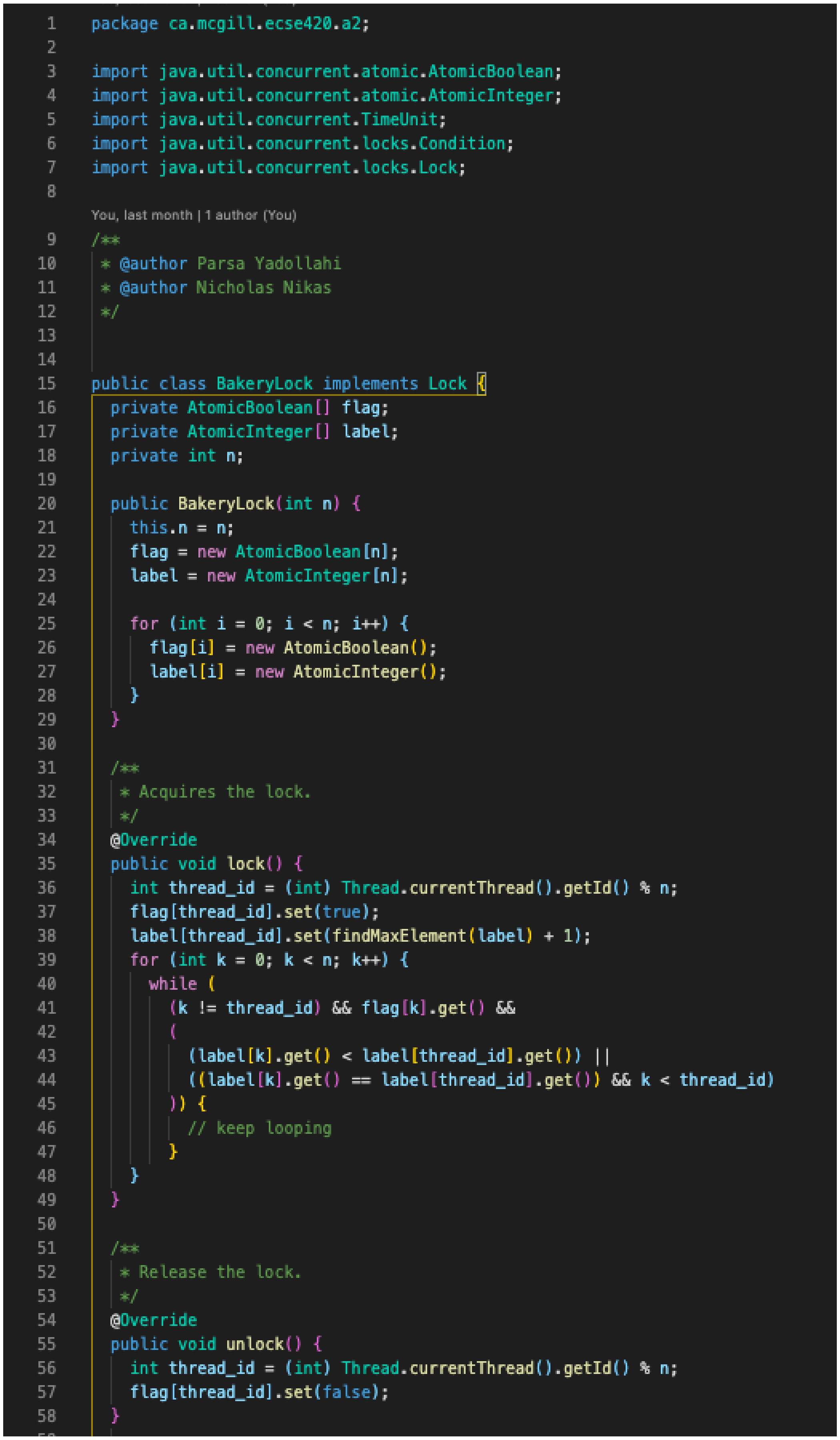


```java

    /**
     * Release the lock
    */
    @Override
    public void unlock() {
        int thread_id = (int) Thread.currentThread().getId() % n;
        level[thread_id].set(0);

    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        // TODO Auto-generated method stub

    }

    @Override
    public boolean tryLock() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public Condition newCondition() {
        // TODO Auto-generated method stub
        return null;
    }
}

```

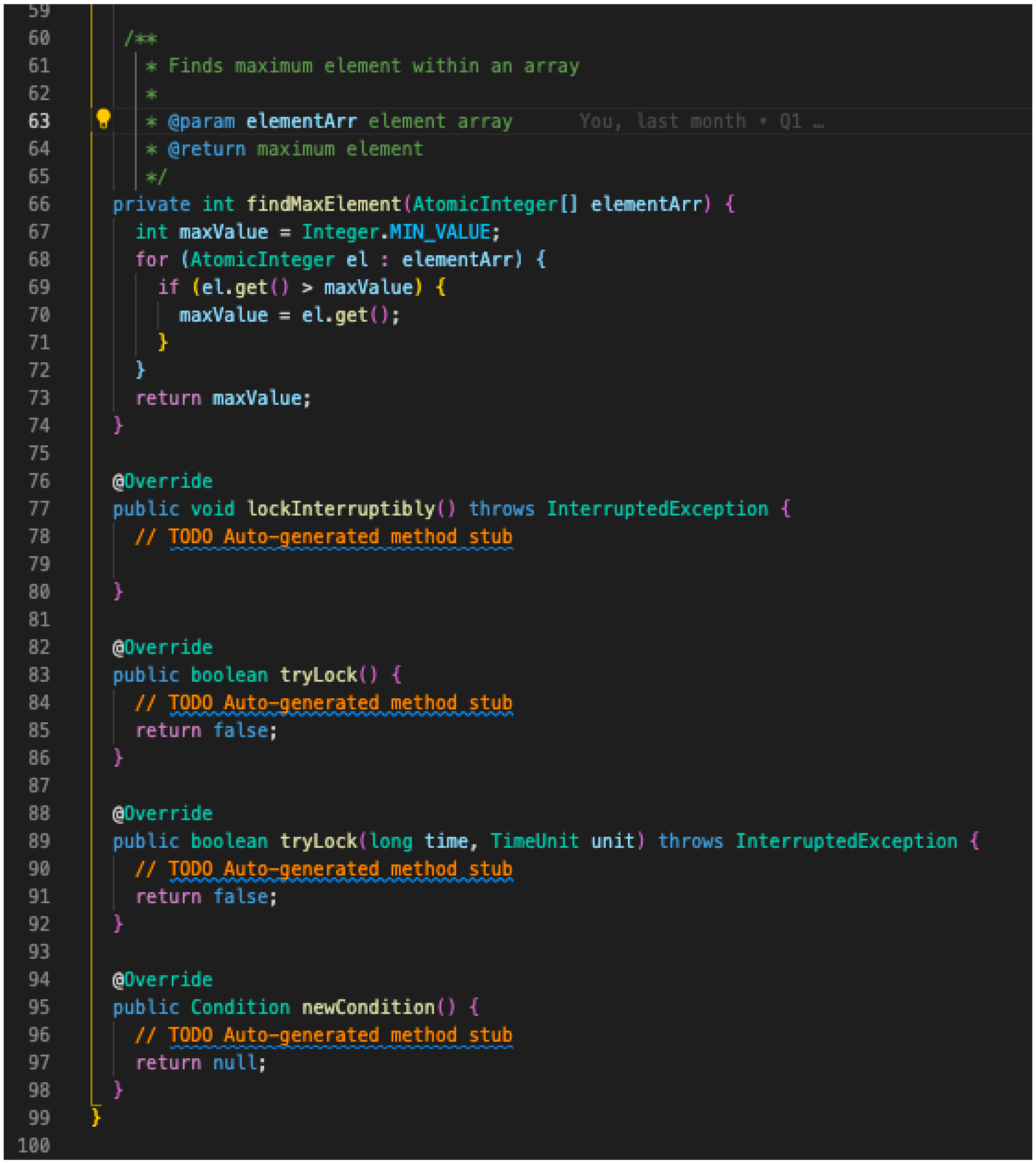
Figure 2 - Bakery Lock:

To run this file, simply run the following commands from the src folder

- javac ca/mcgill/ecse420/a2/BakeryLock.java
- java ca/mcgill/ecse420/a2/BakeryLock



```java
package ca.mcgill.ecse420.a2;

import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

You, last month | 1 author (You)
/**
 * @author Parsa Yadollahi
 * @author Nicholas Nikas
 */


public class BakeryLock implements Lock {
  private AtomicBoolean[] flag;
  private AtomicInteger[] label;
  private int n;

  public BakeryLock(int n) {
    this.n = n;
    flag = new AtomicBoolean[n];
    label = new AtomicInteger[n];

    for (int i = 0; i < n; i++) {
      flag[i] = new AtomicBoolean();
      label[i] = new AtomicInteger();
    }
  }

  /**
   * Acquires the lock.
   */
  @Override
  public void lock() {
    int thread_id = (int) Thread.currentThread().getId() % n;
    flag[thread_id].set(true);
    label[thread_id].set(findMaxElement(label) + 1);
    for (int k = 0; k < n; k++) {
      while (
        (k != thread_id) && flag[k].get() &&
        (
          (label[k].get() < label[thread_id].get()) ||
          ((label[k].get() == label[thread_id].get()) && k < thread_id)
        )) {
          // keep looping
        }
    }
  }

  /**
   * Release the lock.
   */
  @Override
  public void unlock() {
    int thread_id = (int) Thread.currentThread().getId() % n;
    flag[thread_id].set(false);
  }
```

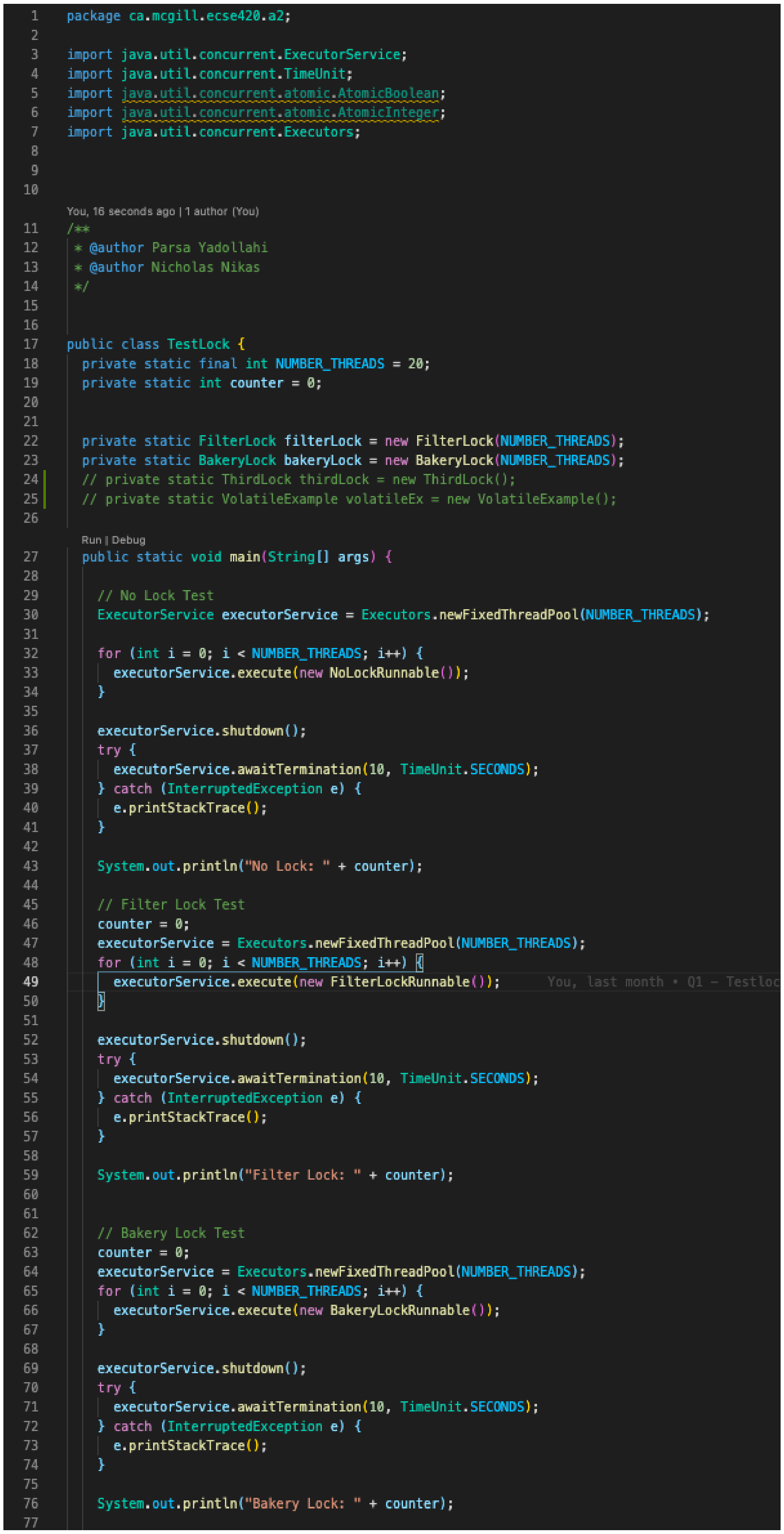


```java

    /**
     * Finds maximum element within an array
     *
     * @param elementArr element array      You, last month • Q1 …
     * @return maximum element
     */
    private int findMaxElement(AtomicInteger[] elementArr) {
      int maxValue = Integer.MIN_VALUE;
      for (AtomicInteger el : elementArr) {
        if (el.get() > maxValue) {
          maxValue = el.get();
        }
      }
      return maxValue;
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
      // TODO Auto-generated method stub

    }

    @Override
    public boolean tryLock() {
      // TODO Auto-generated method stub
      return false;
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
      // TODO Auto-generated method stub
      return false;
    }

    @Override
    public Condition newCondition() {
      // TODO Auto-generated method stub
      return null;
    }
  }

```

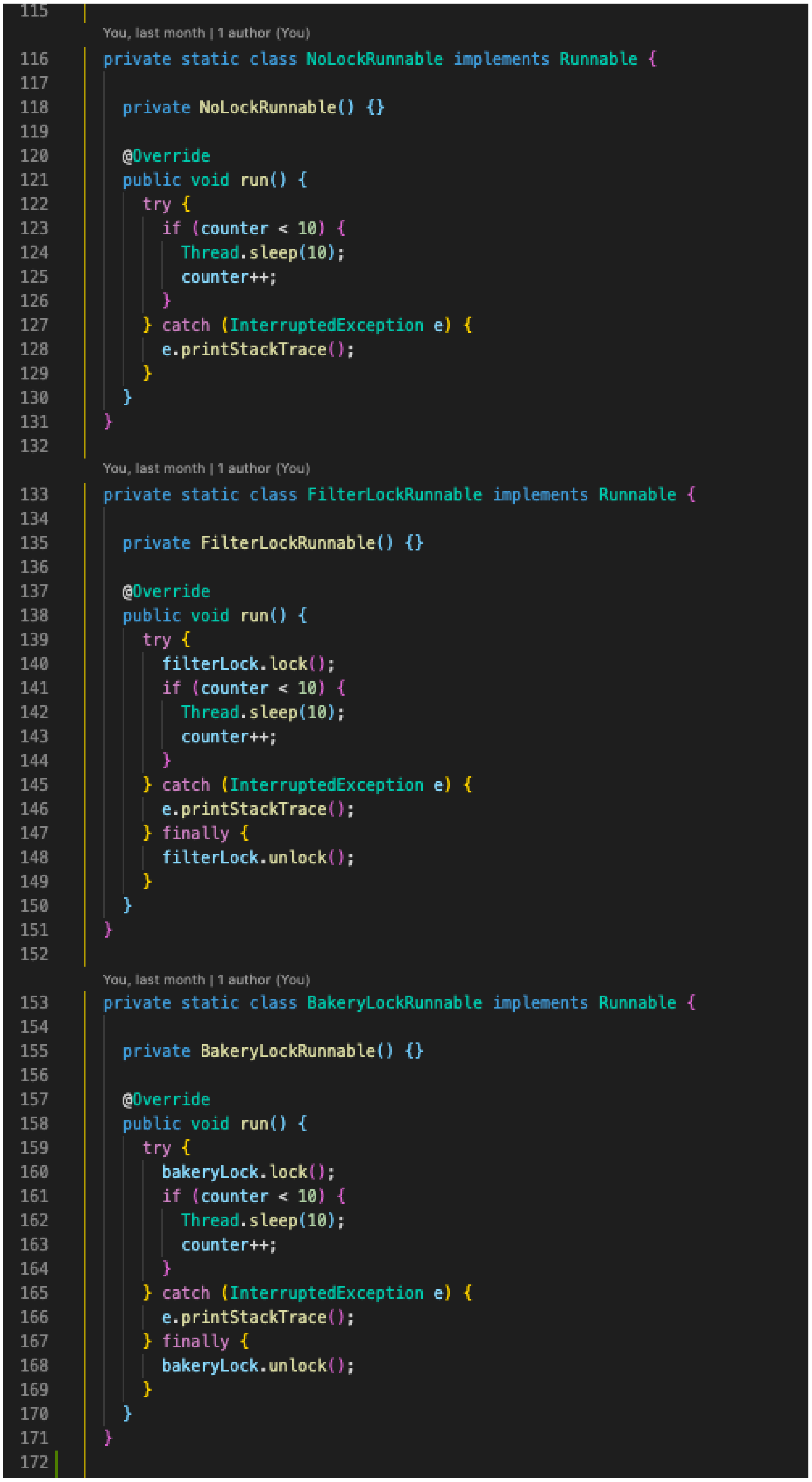
Figure 3 - Test Lock:

To run this file, simply run the following commands from the src folder

- javac ca/mcgill/ecse420/a2/TestLock.java
- java ca/mcgill/ecse420/a2/TestLock



```java
package ca.mcgill.ecse420.a2;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.Executors;



You, 16 seconds ago | 1 author (You)
/**
 * @author Parsa Yadollahi
 * @author Nicholas Nikas
 */


public class TestLock {
  private static final int NUMBER_THREADS = 20;
  private static int counter = 0;


  private static FilterLock filterLock = new FilterLock(NUMBER_THREADS);
  private static BakeryLock bakeryLock = new BakeryLock(NUMBER_THREADS);
  // private static ThirdLock thirdLock = new ThirdLock();
  // private static VolatileExample volatileEx = new VolatileExample();

  Run | Debug
  public static void main(String[] args) {

    // No Lock Test
    ExecutorService executorService = Executors.newFixedThreadPool(NUMBER_THREADS);

    for (int i = 0; i < NUMBER_THREADS; i++) {
      executorService.execute(new NoLockRunnable());
    }

    executorService.shutdown();
    try {
      executorService.awaitTermination(10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("No Lock: " + counter);

    // Filter Lock Test
    counter = 0;
    executorService = Executors.newFixedThreadPool(NUMBER_THREADS);
    for (int i = 0; i < NUMBER_THREADS; i++) {
      executorService.execute(new FilterLockRunnable());      You, last month • Q1 - Testloc
    }

    executorService.shutdown();
    try {
      executorService.awaitTermination(10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Filter Lock: " + counter);


    // Bakery Lock Test
    counter = 0;
    executorService = Executors.newFixedThreadPool(NUMBER_THREADS);
    for (int i = 0; i < NUMBER_THREADS; i++) {
      executorService.execute(new BakeryLockRunnable());
    }

    executorService.shutdown();
    try {
      executorService.awaitTermination(10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Bakery Lock: " + counter);

```



```
You, last month | 1 author (You)
private static class NoLockRunnable implements Runnable {

  private NoLockRunnable() {}

  @Override
  public void run() {
    try {
      if (counter < 10) {
        Thread.sleep(10);
        counter++;
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}

You, last month | 1 author (You)
private static class FilterLockRunnable implements Runnable {

  private FilterLockRunnable() {}

  @Override
  public void run() {
    try {
      filterLock.lock();
      if (counter < 10) {
        Thread.sleep(10);
        counter++;
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      filterLock.unlock();
    }
  }
}

You, last month | 1 author (You)
private static class BakeryLockRunnable implements Runnable {

  private BakeryLockRunnable() {}

  @Override
  public void run() {
    try {
      bakeryLock.lock();
      if (counter < 10) {
        Thread.sleep(10);
        counter++;
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      bakeryLock.unlock();
    }
  }
}
```