Parallel Computing.
Assignment Report 2

Estelle Saab - 261027752
Rafi Latif - 261120830
Group 37
November 12th, 2025

ECSE 420
Parallel Computing

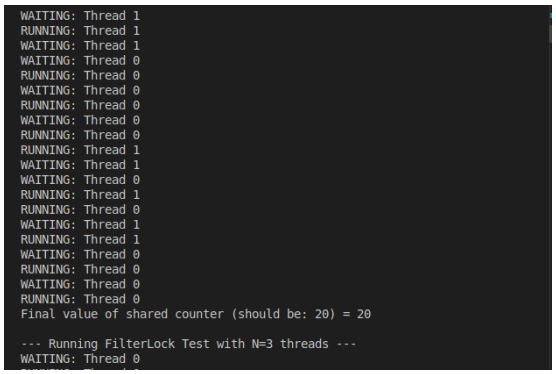
Question Explanations

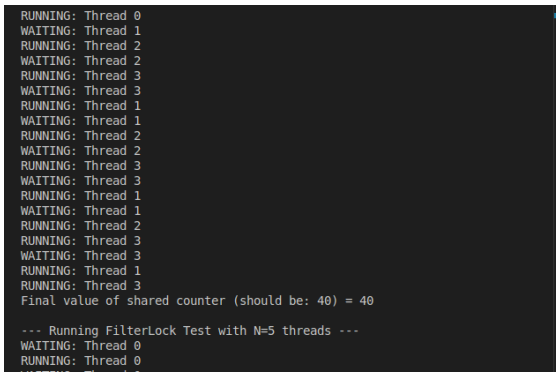
1) N-Thread Mutual Exclusion Algorithms

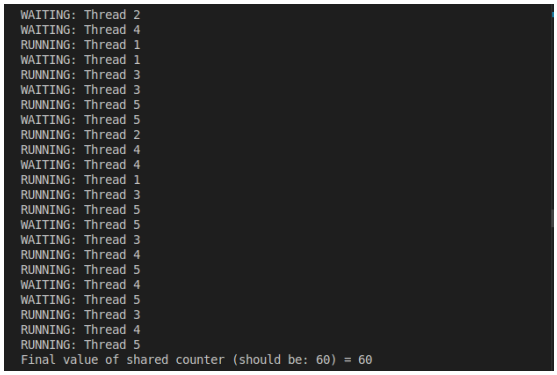
1.1) Filter Lock Implementation

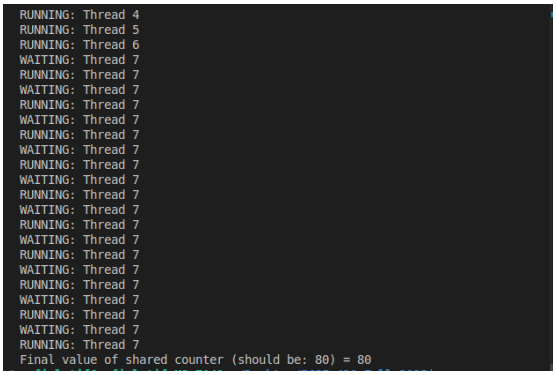
The Filter Lock implementation can be found in A.1.0 Filter Lock Implementation.

1.2) Filter Lock Test with n = 2 to n = 8 Threads







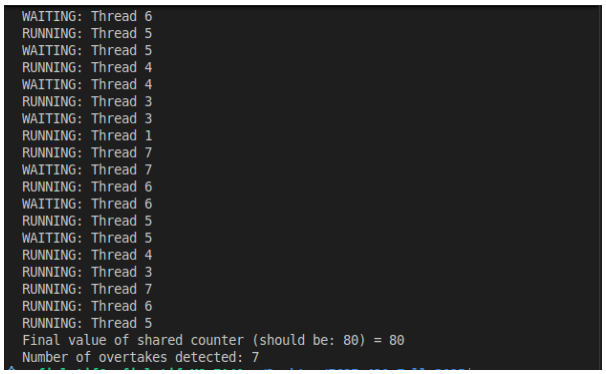


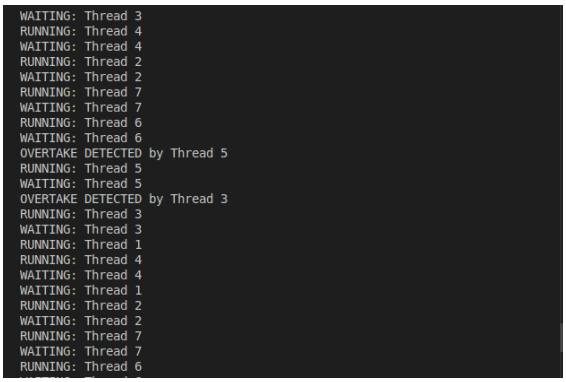
The results from the Filter Lock test (A.1.1 Filter Lock Test) shown above confirm that the lock works.

1.3) Does Filter Lock allow “Overtaking” an Arbitrary Number of Times?

No it does not, because at each level, a thread indicates its intention to enter that level and gives priority to other threads by setting itself as the victim. This mechanism ensures that if multiple threads are trying to enter the same level, the one that is not the victim will proceed, while the victim will wait. As a result, a thread cannot indefinitely overtake others since it must wait for its turn at each level of the filter lock.

1.4) Filter Lock “Overtaking” Test with n = 8 Threads



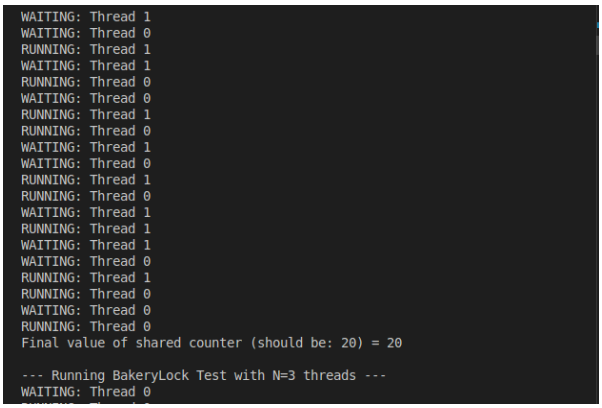


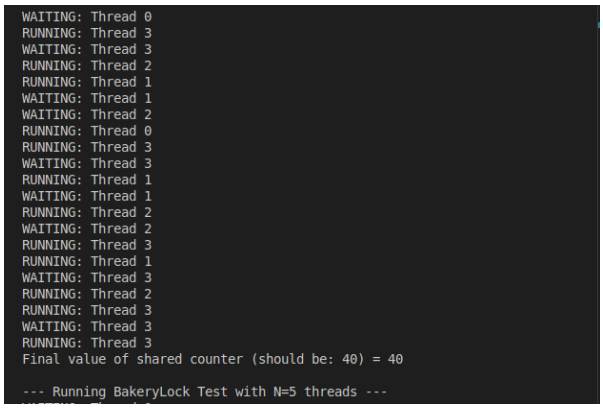
The counter task using n = 8 threads did detect overtaking when using the FilterLock, as shown above. An ordered array list of thread ids was used as the overtaking detection mechanism within this test (A.1.1 Filter Lock Test). It can be observed that overtaking occurred 7 times during execution, an example of which is demonstrated in the image to the right. As shown in the execution log, Thread 5 and Thread 3 overtake a thread that was waiting to receive the lock, longer than them.

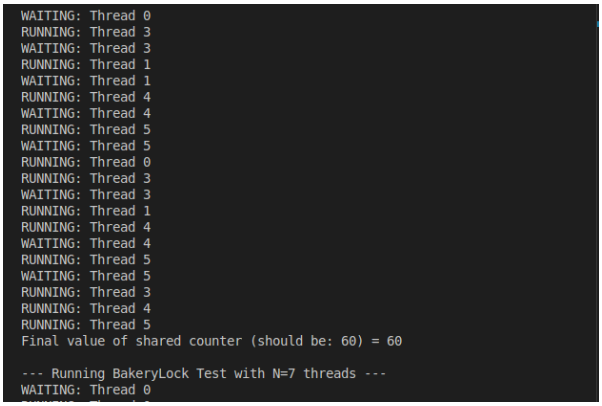
1.5) Lamport’s Bakery Lock Implementation

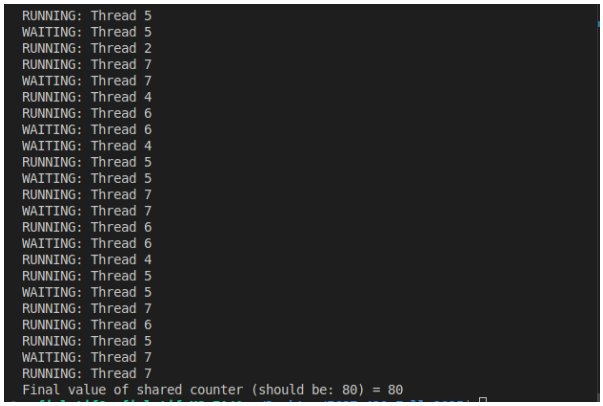
Lamport’s Baker Lock implementation can be found in B.1.0 Bakery Lock Implementation.

1.6) Bakery Lock Test with n = 2 to n = 8 Threads







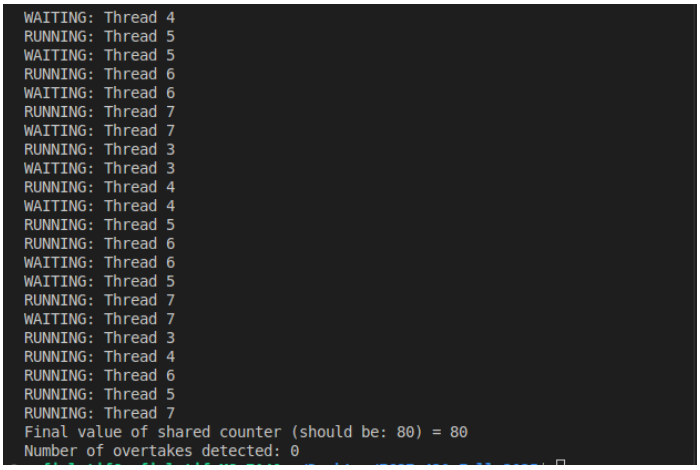


The results from the Baker Lock test (B.1.1 Bakery Lock Test) shown above confirm that the lock works.

1.7) Does Bakery Lock allow “Overtaking” an Arbitrary Number of Times?

No, the Bakery lock does not allow threads to overtake others an arbitrary number of times. The Bakery lock uses a ticketing system where each thread takes a "ticket" before entering the critical section. Threads with lower numbered tickets get priority to enter the critical section. If two threads have the same ticket number, the thread with the lower thread ID gets priority. This mechanism ensures that once a thread has taken a ticket, it cannot be indefinitely overtaken by other threads, as they will have to wait for their turn based on their assigned numbers.

1.8) Bakery Lock “Overtaking” Test with n = 8 Threads



There was no overtaking detected when executing a counter task with n = 8 threads with the BakeryLock, as shown above. Similar to the FilterLock, an order array of thread ids was used to order threads based on their intent to acquire the lock (B.1.1 Bakery Lock Test). Considering the lock takes lexicographical ordering into consideration if ever there is tie between two threads attempting to progress towards the critical section, overtaking is not likely to occur, as reflected by the results of the test.

2) The Shaky Protocol

2.1) Does the Protocol Satisfy Mutual Exclusion?

To enter the critical section, a thread must observe busy = false and turn = me. Assuming Thread A and B are in the critical section at the same time:

1. Suppose Thread A entered the inner loop and set turn = A.
2. Then Thread B immediately followed, overwriting turn = B
3. Both threads then exit the inner loop and set busy = true
4. Thread B fails to satisfy the outer loop condition (turn != me), and enters the critical section. On the other hand,Thread A does satisfy the outer loop condition and is stuck busy waiting.

Contradiction met, Thread A and B are not in the critical section at the same time. This protocol satisfies mutual exclusion.

2.2) Is the Protocol Deadlock-free?

Example of execution that generates deadlock using Thread A and B.

1. Lock initializes busy = false
2. Thread A calls lock().
3. Thread A enters the inner loop and sets turn = A, then exits the inner loop.
4. Thread A sets busy = true, exits the outer loop, and is now in the critical section
5. Thread B calls lock()
6. Thread B enters the inner loop and sets turn = B, but since busy = true, it spins in the inner loop.
7. Thread A exits the critical section, sets busy = false.
8. Thread B sees busy = false, exits the inner loop and sets busy = true.
9. Thread A calls lock() again.
10. Thread A enters the inner loop and sets turn = A, but since busy = true, it spins in the inner loop.
11. Thread B sees turn = A, so it loops back into the inner loop and sets turn = B.
12. Both Thread A and B are now stuck in the inner loop since busy = true, and it cannot be changed.

This example shows that the protocol is not deadlock-free.

2.3) Is the Procol Starvation-free?

Considering the protocol is not deadlock-free, then it also not starvation-free. Moreover, due to a race condition, only the last thread to execute line 8 can exit the outer loop and enter the critical section. Meaning, multiple threads can continuously overtake a waiting thread in the outer loop, causing starvation.

3) Linearizability & Sequential Consistency

3.1) Fig. 2: History A

Linearizability:

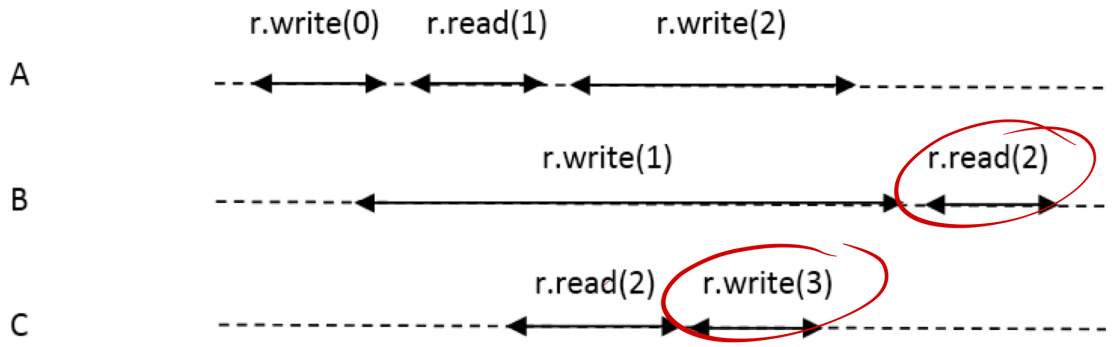
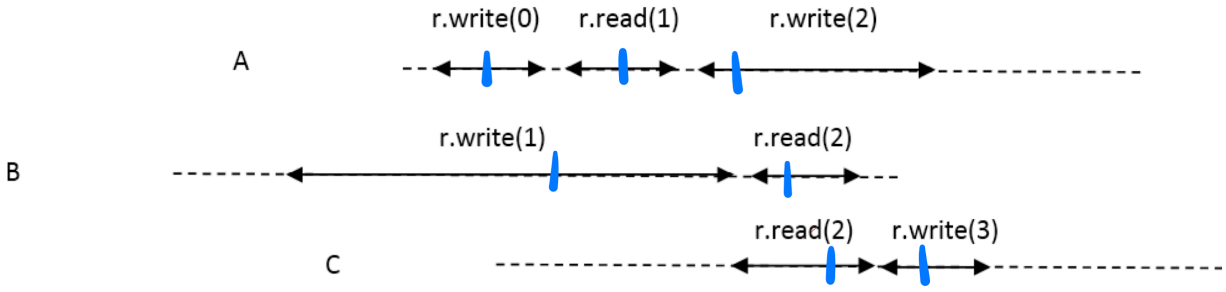


Fig. 2: History A

History A is not linearizable, due to the order of events. Thread C executes the r.write(3) event before Thread B executes r.read(2). Therefore, Thread B attempts to read a value that does not exist within the register.

Sequential Consistency:



History A is sequentially consistent as indicated by the order of execution above, where the blue lines represent the instantaneous effect of the event.

3.2) Fig. 3: History B

Linearizability:

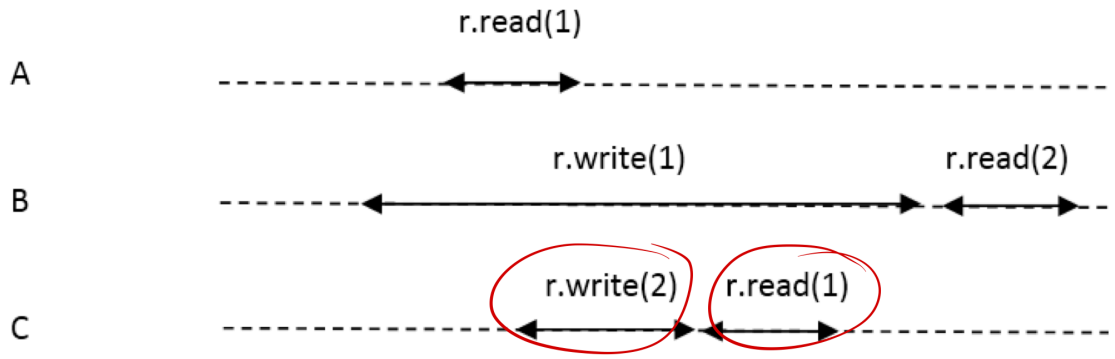
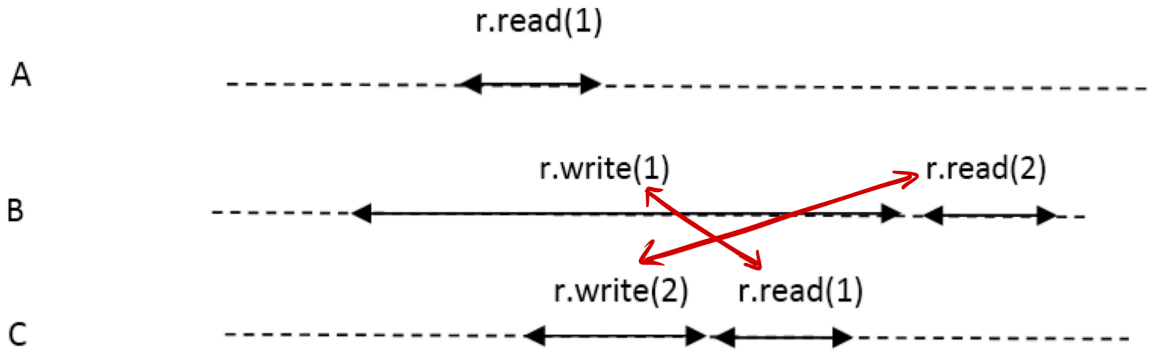


Fig. 3: History B

History B is not linearizable, due to the order of events. Assuming register r is not initialized with 1, Thread B must execute r.write(1) first, in order for Thread A to successfully execute r.read(1). However, the issue occurs when Thread C executes r.write(2) and is immediately followed by r.read(1). Therefore, Thread C attempts to read a value that is no longer in the register.

Sequential Consistency:



History B is not sequentially consistent, due to the order of events. The primary issue lies in the conflict of events between Thread B and C. In order for r.read(1) and r.read(2) to execute successfully, the register must be overwritten by either r.write(2) or r.write(2). In either situation, if one read operation succeeds, the other will inevitably fail.

4) Volatile Example

4.1)

Under the Java Memory Model, a division by zero cannot occur in this program.
The variable v is declared as volatile, which establishes a happens-before relationship between any write to v and any subsequent read of v that observes the written value. In the writer() method, the assignment x = 42 occurs before v = true. Therefore, once another thread observes v == true, the Java Memory Model guarantees that it will also see all writes that occurred before this volatile write, including the update to x. As a result, when the reader() method executes int y = 100 / x, it will always read x = 42, not 0. A division by zero could only occur if v were not declared as volatile, allowing v = true to become visible before the update to x = 42. However, with v declared volatile, such a reordering is disallowed.

4.2)

Case A - Both x and v are volatile
A division by zero is not possible.
In thread A, the write to x occurs before the volatile write to v in program order. The volatile write v = true establishes a happens-before relationship with any subsequent volatile read of v that returns true in thread B. Because volatile reads and writes also guarantee visibility of all previous actions, the assignment x = 42 in thread A becomes visible to thread B before it reads v == true. Thus, when thread B executes reader(), it must observe x = 42, and the statement 100 / x cannot cause a division by zero.

Case B - Neither x nor v are volatile
A division by zero is possible.
Without the volatile keyword, there is no happens-before relationship between the two threads. The compiler or CPU may reorder operations or cache values locally. Consequently, thread A might make the update v = true visible before x = 42 is propagated to main memory. Thread B could therefore observe v == true while still seeing the old value x == 0, leading to a division by zero in reader().

Therefore, declaring both variables as volatile enforces visibility and ordering guarantees, eliminating the risk of division by zero. Without volatile, no such guarantees exist, allowing stale reads and instruction reordering that can cause the error.

5) LockOne & LockTwo Mutual Exclusion

If the shared variables (flag in LockOne and victim in LockTwo) are replaced by regular (non-volatile) registers, mutual exclusion is no longer guaranteed in either algorithm.

In LockOne, each thread sets its own flag[i] = true and then checks the other thread's flag in a loop. With regular registers, values read from the registers could return the old or new value. As a result, both threads could read the other's flag[j] as false (i.e the old value) and enter the critical section simultaneously, violating mutual exclusion.

In LockTwo, the shared variable victim is used to indicate which thread should wait. If victim is A regular register instead of an atomic register, an overlapping write may cause a reader to see old or new value. This can cause both threads to see outdated values of the victim register, making them believe it is safe to enter the critical section at the same time.

Therefore, both LockOne and LockTwo rely on atomic registers to guarantee they function correctly. If regular registers are used, mutual exclusion cannot be ensured.

6) Regular M-valued MRSW class

6.1)

False

In the original implementation, the write(x) method first sets r_bit[x] = true and then clears all bits with indices smaller than x (i.e., for (int i = x - 1; i >= 0; i--)). This ensures that, at any given moment, exactly one bit, corresponding to the most recently written value, remains set to true. As a result, when the read() method scans the array from index 0 upward and returns the first true bit it encounters, it always returns the value of the most recent write operation, thereby satisfying the regularity condition of the register.

However, if we modify the loop to for (int i = x + 1; i < RANGE; i++), the method would instead clear only the bits with indices greater than x, leaving all lower bits set to true. This breaks the intended invariant, since multiple bits could remain true after a write operation. For instance, if the register initially has r_bit[0] = true and we call write(5), both r_bit[0] and r_bit[5] will be true. In that case, the read() method, which always returns the lowest index that is true, would incorrectly return 0 instead of 5. With the modified loop, the construction is no longer a regular M-valued MRSW register.

6.2)

False.

A safe register must return the last completed written value for any read that does not overlap a write; only reads that overlap a write may return an arbitrary value from the domain. With the proposed change (for (int i = x+1; i < RANGE; i++)), write(x) clears only bits above x and leaves all lower bits unchanged. Starting from the initialized state r_bit[0]=true, after a completed write(5) both r_bit[0] and r_bit[5] are true. A subsequent read (not overlapping any write) scans from index 0 and returns 0 rather than 5. Since a non-overlapping read fails to return the last written value with the modified loop, the construction is no longer a regular M-valued MRSW register.

7) Atomic Registers for Two Threads

Proof by contrapositive:

Assume, for contradiction, that there exists a wait-free algorithm An that solves binary consensus for n threads using only atomic registers. We construct a 2-thread algorithm A2 as follows:

- Run An but let only threads T1 and T2 take steps.
- The remaining n-2 threads are treated as crashed at time 0 (they take no steps and make no writes). This is a legal execution for a wait-free algorithm.

Because An is wait-free, T1 and T2 must each decide in a finite number of their own steps, regardless of the behavior (or absence) of the other n-2 threads. Because An satisfies validity and agreement, the decisions of T1 and T2 are identical and equal to one of their inputs. Hence A2 solves 2-thread binary consensus using only atomic registers-contradicting the assumed impossibility.

Therefore, no such An can exist. If 2-thread consensus is impossible with atomic registers, then n-thread consensus is also impossible.

8) Atomic Registers for N-Threads

Proof by reduction (contradiction).

Assume there exists a wait-free algorithm Ak that solves consensus over k>2 values for n threads using only atomic registers. Pick any two distinct values v0, v1 from the k values. To solve binary consensus:

- Map input bit 0 to proposal v0 and bit 1 to proposal v1
- Run Ak unchanged.

Because Ak guarantees agreement and validity, all deciding threads choose either v0 or v1, and the choice equals one of the proposed values-hence we have a correct binary consensus algorithm. This contradicts the assumed impossibility of binary consensus for n threads using atomic registers.

Therefore, if binary consensus is impossible, k-valued consensus is also impossible.

Appendix A: Filter Lock Source Code

A.1.0 Filter Lock Implementation

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

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

public class FilterLock implements Lock{
   private volatile int[] level;
   private volatile int[] victim;
   private int n;

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

   public void lock() {
       int currentThread = ThreadID.get();
       for (int L = 1; L < n; L++) { // one level at a time
           level[currentThread] = L; // intention to enter level L
           victim[L] = currentThread; // give priority to others
           boolean conflicted = true;
           while (conflicted) {
               conflicted = false;
               for (int k = 0; k < n; k++) {
                   if (k != currentThread && level[k] >= L && victim[L] == currentThread) {
                       conflicted = true;
                       break;
                   }
               }
           }
       }
   }

   public void unlock() {
       int current = ThreadID.get();
       level[current] = 0;
   }

   @Override
   public void lockInterruptibly() throws InterruptedException {
       // TODO Auto-generated method stub
       throw new UnsupportedOperationException("Unimplemented method 'lockInterruptibly'");
```

```
    }

    @Override
    public Condition newCondition() {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'newCondition'");
    }

    @Override
    public boolean tryLock() {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'tryLock'");
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'tryLock'");
    }
}
```

A.1.1 Filter Lock Test

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

import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;

public class TestFilterLock {
   private final int nThreads;
   private final int iters;
   private final int PER_THREAD;

   private final AtomicInteger counter;
   private int[] counterArray;
   private int[] threadIdArray;

   private ArrayList<Integer> waiting = new ArrayList<>();
   private int overtakeCount = 0;

   private FilterLock lock;

   public TestFilterLock(int nThreads, int iters) {
       this.nThreads = nThreads;
       this.iters = iters; // dynamically assign to ensure results are not truncated
       this.PER_THREAD = iters / nThreads;

       this.lock = new FilterLock(nThreads);
       ThreadID.reset();

       this.counter = new AtomicInteger(0);
       this.counterArray = new int[iters];
       this.threadIdArray = new int[iters];
   }

   private class FilterThread implements Runnable {
       @Override
       public void run() {
           for (int i = 0; i < PER_THREAD; i++) {

               System.out.println("WAITING: Thread " + ThreadID.get());
               waiting.add(ThreadID.get());

               lock.lock();
               try {
                   // if (waiting.get(0) != ThreadID.get()) {
                   //     overtakeCount++;
                   //     System.out.println("OVERTAKE DETECTED by Thread " + ThreadID.get());
                   // }
                   System.out.println("RUNNING: Thread " + ThreadID.get());
```

```
                    int counterValue = counter.getAndIncrement();
                    counterArray[counterValue] += 1;
                    threadIdArray[counterValue] = ThreadID.get();

                    waiting.remove(Integer.valueOf(ThreadID.get()));
                } finally {
                    lock.unlock();
                }
            }
        }
    }

    public void runTest() {
        ExecutorService executor = Executors.newFixedThreadPool(nThreads);
        for (int i = 0; i < nThreads; i++) {
            executor.execute(new FilterThread());
        }
        executor.shutdown();
        while (!executor.isTerminated()) {
            // wait for all threads to finish
        }

        // Verify the results
        System.out.flush();
        System.out.println("Final value of shared counter (should be: " + iters + ") = " +
counter.get());
        System.out.println("Number of overtakes detected: " + overtakeCount);
    }

    public static void main(String[] args) {
        int[] nThreadsOptions = {2, 3, 4, 5, 6, 7, 8};
        for (int n : nThreadsOptions) {
            System.out.println("\n--- Running FilterLock Test with N=" + n + " threads ---");
            TestFilterLock test = new TestFilterLock(n, (n*10));
            test.runTest();
        }
    }
}
```

Appendix B: Bakery Lock Source Code

B.1.0 Bakery Lock Implementation

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

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

public class BakeryLock implements Lock {
   volatile boolean[] flag;
   volatile int[] label;

   public BakeryLock(int n) {
       flag = new boolean[n];
       label = new int[n];
       for (int i = 0; i < n; i++) {
           flag[i] = false;
           label[i] = 0;
       }
   }

   public void lock() {
       int i = ThreadID.get();
       flag[i] = true;
       // Find max label by reading all labels
       int max = 0;
       for (int j = 0; j < label.length; j++) {
           if (label[j] > max) {
               max = label[j];
           }
       }
       label[i] = max + 1;

       // Wait until no one has priority
       for (int k = 0; k < flag.length; k++) {
           if (k == i) continue;
           while (flag[k]) {
               if (label[k] != 0 && (label[k] < label[i] || (label[k] == label[i] && k < i)))
{
                   // Busy wait
               } else {
                   break;
               }
           }
       }
   }
   public void unlock() {
       int i = ThreadID.get();
       label[i] = 0;
```

```
        flag[i] = false;
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'lockInterruptibly'");
    }

    @Override
    public Condition newCondition() {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'newCondition'");
    }

    @Override
    public boolean tryLock() {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'tryLock'");
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'tryLock'");
    }
}
```

B.1.1 Bakery Lock Test

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

import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;

public class TestBakeryLock {
   private final int nThreads;
   private final int iters;
   private final int PER_THREAD;

   private final AtomicInteger counter;
   private int[] counterArray;
   private int[] threadIdArray;

   private ArrayList<Integer> waiting = new ArrayList<>();
   private int overtakeCount = 0;

   private BakeryLock lock;

   public TestBakeryLock(int nThreads, int iters) {
       this.nThreads = nThreads;
       this.iters = iters; // dynamically assign to ensure results are not truncated
       this.PER_THREAD = iters / nThreads;

       this.lock = new BakeryLock(nThreads);
       ThreadID.reset();

       this.counter = new AtomicInteger(0);
       this.counterArray = new int[iters];
       this.threadIdArray = new int[iters];
   }

   private class BakeryThread implements Runnable {
       @Override
       public void run() {
           for (int i = 0; i < PER_THREAD; i++) {

               System.out.println("WAITING: Thread " + ThreadID.get());
               waiting.add(ThreadID.get());

               lock.lock();
               try {
                   if (waiting.get(0) != ThreadID.get()) {
                       overtakeCount++;
                       System.out.println("OVERTAKE DETECTED by Thread " + ThreadID.get());
                   }
                   System.out.println("RUNNING: Thread " + ThreadID.get());
```

```
                    int counterValue = counter.getAndIncrement();
                    counterArray[counterValue] += 1;
                    threadIdArray[counterValue] = ThreadID.get();

                    waiting.remove(Integer.valueOf(ThreadID.get()));
                } finally {
                    lock.unlock();
                }
            }
        }
    }

    public void runTest() {
        ExecutorService executor = Executors.newFixedThreadPool(nThreads);
        for (int i = 0; i < nThreads; i++) {
            executor.execute(new BakeryThread());
        }
        executor.shutdown();
        while (!executor.isTerminated()) {
            // wait for all threads to finish
        }

        // Verify the results
        System.out.flush();
        System.out.println("Final value of shared counter (should be: " + iters + ") = " +
counter.get());
        System.out.println("Number of overtakes detected: " + overtakeCount);
    }

    public static void main(String[] args) {
        int[] nThreadsOptions = {2, 3, 4, 5, 6, 7, 8};
        for (int n : nThreadsOptions) {
            System.out.println("\n--- Running BakeryLock Test with N=" + n + " threads ---");
            TestBakeryLock test = new TestBakeryLock(n, (n*10));
            test.runTest();
        }
    }
}
```

Appendix C: ThreadID Implementation

C.1.0 Thread ID Implementation

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

public class ThreadID {

   private static volatile int currentID = 0;
   private static ThreadLocalID threadLocalID = new ThreadLocalID();

   public static int get() {
       return threadLocalID.get();
   }

   public static void reset() {
       currentID = 0;
   }

   private static class ThreadLocalID extends ThreadLocal<Integer>{
       protected synchronized Integer initialValue() {
           return currentID ++;
       }

   }

}
```