ECSE 420 - Parallel Computing

Assignment 2

Arian Omidi - 260835976
Namdar Nejad - 260893536

March 16, 2022

1 Locks & Mutual Exclusion

1.1

See Filter Lock implementation in Appendix A.

1.2

No, the Filter lock does not allow some threads to overtake others an arbitrary number of times. We can prove this by contradiction:

Assume the Filter lock does allow some threads to overtake a thread, t_0 an arbitrary number of times, n. As n approaches infinity, t_0 will starve. This is a contradiction since we know that the Filter lock is starvation free.

1.3

See Bakery Lock implementation in Appendix B.

1.4

No, the Bakery lock does not allow some threads to overtake others an arbitrary number of times. We know this as the Bakery lock respects a FIFO policy and thus, threads are not overtaken by other threads with the exception of tie-breaks, which can only happen a finite amount of times.

1.5

We can test mutual exclusion by using N threads which all have access to a shared counter. Each thread will first read the value of the counter to a register and then increment the counter based on this value M number of times. If there is mutual exclusion, then once all threads have finished executing, the counters value should equal to MN. If there is no mutual exclusion, meaning that the threads may overwrite other threads work and thus the counters value will be less than MN during termination.

1.6

See Lock Test implementation in Appendix C. Below are the results of the test run with 8 threads and a total counter value of 1000:

```
NO LOCK TEST -> Expected: 1000, Actual: 126
FILTER LOCK TEST -> Expected: 1000, Actual: 1000
BAKERY LOCK TEST -> Expected: 1000, Actual: 1000
```

This verifies that the locks are providing mutual exclusion.

2 Atomic vs. Regular Registers

2.1 LockOne

No, the mutual exclusion is not held if the atomic *flag* registers are replaced with regular registers, as memory synchronization is not guaranteed when using regular registers.

Proof. Assume that we have two threads, T_1 and T_2, and T_1 attempts to acquire the lock, locally sets its flag to *true* and is interrupted before entering the critical section. T_2 then attempts to acquire the lock and locally sets its own flag to *true*, however since the flag values are not synchronized, T_2 reads that T_1's flag is set to false and enters the critical section. Now when T_1 is awoken, it too will also enter the critical section as it is unaware that T_2 set their flag to true. Therefore, the mutual exclusion property is violated. ■

2.2 LockTwo

No, the mutual exclusion is not held if the atomic *victim* register is replaced with a regular register, as again memory synchronization is not guaranteed when using regular registers.

Proof. Assume that we have two threads, T_1 and T_2, and T_1 attempts to acquire the lock, locally sets *victim* to 1 and is interrupted before entering the critical section. T_2 then attempts to acquire the lock and locally sets *victim* to 2, and since the victim value is not synchronized, T_1's write to *victim* is only now registered and *victim* is set to 1. T_2 reads sees that its not the victim and thus enters the critical section. Now if T_2's write to *victim* is registered and T_1 is awoken, it too will also enter the critical section as it is reads that T_2 is the victim. ■

3 Flaky Computer Corporation

3.1 Mutual Exclusion

Yes, LockThree provides mutual exclusion.

Proof. Assume that threads A and B are both in the critical section. This implies that

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

and

$$write_B(turn = B) \rightarrow write_B(busy = true) \rightarrow read_B(turn = A) \rightarrow CS_B$$

Therefore, for $read_B(turn = A)$, thread A must set $write_A(turn = A)$, and similarly, for $read_A(turn = B)$, thread B must set $write_B(turn = B)$.
without loss of generality, let thread A be the first to write to the *turn* register. Thus, for both threads to be within the critical section we must have that

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

or

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

both of which are contradictions. ■

3.2 Deadlock

If we only have one thread, T, attempting to access the critical section, it will deadlock as no other thread will unlock the lock to set busy to *false*, leaving T in the while loop indefinitely. Thus, LockThree is not deadlock-free.

3.3 Starvation

Given that LockThree is not deadlock-free, it cannot be starvation-free.

4 Sequential Consistency and Linearizability

4.1

This history is sequentially consistent, and the sequence is: [A] r.write(0), [B] r.write(1), [A] r.read(1), [C] r.read(2), [A] r.write(2), [C] r.write(3), [B] r.read(2). However, the history is not lineaizable since we have overlaping read/writes which would cause interrupts, e.g. [A] r.read(1) and [B] r.write(1) overlap.

4.2

This history is not sequentially consistent since the order of execution between B and C is not sequential: [B] r.write(1), [A] r.read(1), [C] r.write(2), [C] r.read(1), [B] r.read(2). Moreover, the history is not linearizable since we have overlapping read/write which would cause interruptions, e.g. [A] r.read(1) and [B] r.write(1) overlap.

5 Reader and Writer

5.1

Yes, a division by zero would be possible. To do so we would need to call *writer()* first, in the writer in one thread the value of x would be set to 42 while in the other thread it would remain equal to 0, in the meanwhile since our boolean v is volatile it would be set to *true* in both threads. In this case, after calling the *reader()* we would get a division by zero since we have $x = 0$ and $v = true$.

5.2

No, if they are both volatile then division by zero would not be possible. This if we call *reader()* first then we have $v = false$ so we won't even do the division, and if we call *write()* first and then *reader()* then since x and v are both we would set x to 42 and $v = true$ in all threads, thus we won't get a division by zero in the *reader()* method.

If none are volatile, then the variable values are thread-local, thus v would still be false for the *reader()* method and we would not do a division at all.

6 MRSW Register

6.1

True. The new loop will set the value of r-bits array at index x to true on line 10. And the new loop will set the values of the r-bits array after the index x all to false, while the values for the array for before the index x will remain as is. When reading from the array we return the index of the first instance of true, which can be the new true value we added on line 10 or the old value from before and this is exactly the functionality of the regular M-valued MRSW register.

6.2

False. As explained above, the new loop sets the values after the index x to false and leaves the values before x untouched while setting the value ate x to true. Now we know that in a safe M-valued MRSW register we need to return the most recent value of a variable and not the old value. However, in the new loop, we might return the new or old value of the variable when returning the first true element which will not result in a "safe" M-valued MRSW register.

7 Binary Consensus I

Let's assume that binary consensus is possible for n threads where $n > 2$. Then a binary consensus would also be possible for any pair of threads in the set $n > 2$. Thus a binary consensus is also possible for 2 threads, contradicting our assumption. So we can conclude that if a binary consensus is possible for n threads where $n > 2$ then it's also possible for 2 threads. And in contradiction, we have proven that if binary consensus is not possible for 2 threads it's also impossible for n threads where $n > 2$.

8 Binary Consensus II

Let's assume that a consensus for $k > 2$ is possible, and the binary consensus is impossible. Now if consensus for $k > 2$ is possible, we could create a binary consensus by mapping $[0, k/2]$ to 0 and $[k/2, 1]$ to 1. This is a contradiction to our initial assumption. So we can conclude that if consensus for $k > 2$ is possible then binary consensus is also possible and if binary consensus is not possible neither is consensus for $k > 2$.

Appendices

A FilterLock.java

```
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 final int n;

    public FilterLock(int n) {
        this.level = new int[n];
        this.victim = new int[n];
        this.n = n;
    }

    @Override
    public void lock() {
        int i = ThreadId.get();
        for (int l = 1; l < n; l++) {
            this.level[i] = l;
            this.victim[l] = i;

            for (int k = 0; k < n; k++) {
                if (k == i) continue;

                while (level[k] >= l && victim[l] == i); // spin
            }
        }

    }

    @Override
    public void unlock() {
        int i = ThreadId.get();
        this.level[i] = 0;
    }
```

```
    @Override
    public void lockInterruptibly() throws InterruptedException {

    }

    @Override
    public boolean tryLock() {
        return false;
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        return false;
    }

    @Override
    public Condition newCondition() {
        return null;
    }
}
```

B BakeryLock.java

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

public class BakeryLock implements Lock {
    private volatile boolean[] flag;
    private volatile long[] label;
    private final int n;

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

        for (int i = 0; i < n; i++) {
            flag[i] = false;
```

```
            label[i] = 0;
        }
    }

    @Override
    public void lock() {
        int i = ThreadId.get();

        flag[i] = true;
        label[i] = Arrays.stream(label).min().getAsLong() + 1;

        for (int k = 0; k < n; k++) {
            while (flag[k] && (label[k] < label[i] || (label[k] == label[i] && k < i))); // spin
        }
    }

    @Override
    public void unlock() {
        int i = ThreadId.get();
        flag[i] = false;
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {

    }

    @Override
    public boolean tryLock() {
        return false;
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        return false;
    }

    @Override
    public Condition newCondition() {
        return null;
    }
}
```

C LockTest.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class LockTest {

    static final int THREAD_COUNT = 8;
    static final int DELAY = 1; //ms
    static final int COUNT = 1000;
    static final int PER_THREAD_COUNT = COUNT / THREAD_COUNT;
    static int count = 0;

    static final FilterLock filterLock = new FilterLock(THREAD_COUNT);
    static final BakeryLock bakeryLock = new BakeryLock(THREAD_COUNT);

    public static void main(String[] args) {
        /* NO LOCK */
        ExecutorService executor = Executors.newFixedThreadPool(THREAD_COUNT);
        for (int i = 0; i < THREAD_COUNT; i++) {
            executor.execute(new NoLockThread());
        }
        executor.shutdown();
        try {
            executor.awaitTermination(20, TimeUnit.SECONDS);
        } catch (InterruptedException ignored) {}
        System.out.format("NO LOCK TEST -> Expected: %d, Actual: %d\n", COUNT, count);

        /* FILTER LOCK */
        count = 0;
        ThreadId.reset();
        executor = Executors.newFixedThreadPool(THREAD_COUNT);
        for (int i = 0; i < THREAD_COUNT; i++) {
            executor.execute(new FilterLockThread());
        }
        executor.shutdown();
        try {
            executor.awaitTermination(20, TimeUnit.SECONDS);
        } catch (InterruptedException ignored) {}
        System.out.format("FILTER LOCK TEST -> Expected: %d, Actual: %d\n", COUNT, count);
```

```
        /* BAKERY LOCK */
        count = 0;
        ThreadId.reset();
        executor = Executors.newFixedThreadPool(THREAD_COUNT);
        for (int i = 0; i < THREAD_COUNT; i++) {
            executor.execute(new BakeryLockThread());
        }
        executor.shutdown();
        try {
            executor.awaitTermination(20, TimeUnit.SECONDS);
        } catch (InterruptedException ignored) {

        }
        System.out.format("BAKERY LOCK TEST -> Expected: %d, Actual: %d\n", COUNT, count);

        System.exit(0);
    }

    public static class NoLockThread implements Runnable {

        @Override
        public void run() {
            for (int i = 0; i < PER_THREAD_COUNT; i++) {
                int tmp = count;

                try {
                    Thread.sleep(DELAY);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                count = tmp + 1;
            }
        }
    }

    public static class FilterLockThread implements Runnable {

        @Override
        public void run() {
            for (int i = 0; i < PER_THREAD_COUNT; i++) {
                filterLock.lock();
```

```
            try {
                int tmp = count;

                try {
                    Thread.sleep(DELAY);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                count = tmp + 1;
            } finally {
                filterLock.unlock();
            }
        }
    }

    public static class BakeryLockThread implements Runnable {

        @Override
        public void run() {
            for (int i = 0; i < PER_THREAD_COUNT; i++) {
                bakeryLock.lock();
                try {
                    int tmp = count;

                    try {
                        Thread.sleep(DELAY);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    count = tmp + 1;
                } finally {
                    bakeryLock.unlock();
                }
            }
        }
    }

}
```

D ThreadId.java

```
/* FROM CLASS TUTORIALS */
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 ++;
      }

   }

}
```