Exam

1 Accounting System

Question 1.1

I have written the following test:

```
for (int i = 0; i < 20000; i++) {
    new Thread(() -> {
        accounts.deposit(n - 1, 1);
        int value = accounts.get(n - 1);
        accounts.transfer(n - 1, n - 2, value);
        assert (accounts.get(n - 1) == 0);
    });
    new Thread(() -> {
        accounts.deposit(n - 1, 1);
        int value = accounts.get(n - 1);
        accounts.transfer(n - 1, n - 2, value);
        assert (accounts.get(n - 1) == 0);
    });
}
assert (accounts.get(n - 1) == 0);
```

By transfering an amount to an account, checking that what the current value on the account is and then removing that amount 20000 across two Threads we see that the value on the account is not what we expect due to interleaved transfers.

Question 1.2

I have written the following test:

```
accounts.deposit(n - 1, 99);
new Thread(() -> {
    while (accounts.get(n - 1) != 100) {
    }
    assert (true); // Never run.
});
new Thread(() -> {
    accounts.deposit(n - 1, 1); // Makes .get(n-1) == 100
    accounts.deposit(n - 1, 1);
    accounts.deposit(n - 1, 1);
});
```

This test never succeeds, which it should.

Question 1.3

The full `LockAccounts` class is shown below:

```
import java.util.Arrays;

public class LockAccounts implements Accounts {
    private volatile Integer[] accounts;

    public LockAccounts(int n) {
        accounts = new Integer[n];
        Arrays.fill(accounts, 0, accounts.length, 0);
    }

    public void init(int n) {
        synchronized (accounts) {
            accounts = new Integer[n];
            Arrays.fill(accounts, 0, accounts.length, 0);
        }
    }

    public int get(int account) {
        synchronized (accounts[account]) {
            return accounts[account];
        }
    }

    public int sumBalances() {
        synchronized (accounts) {
            int sum = 0;
            for (int i = 0; i < accounts.length; i++) {
                sum += accounts[i];
            }
            return sum;
        }
    }

    public void deposit(int to, int amount) {
        synchronized (accounts[to]) {
            accounts[to] += amount;
        }
    }

    public void transfer(int from, int to, int amount) {
        synchronized (accounts[from]) {
            synchronized (accounts[to]) {
```

```
                accounts[from] -= amount;
                accounts[to] += amount;
            }
        }
    }

    public void transferAccount(Accounts other) {
        synchronized (accounts) {
            synchronized (other) {
                for (int i = 0; i < accounts.length; i++) {
                    accounts[i] += other.get(i);
                }
            }
        }
    }

    public String toString() {
        String res = "";
        if (accounts.length > 0) {
            synchronized (accounts) {
                res = "" + accounts[0];
                for (int i = 1; i < accounts.length; i++) {
                    res = res + " " + accounts[i];
                }
            }
        }
        return res;
    }
}
```

The accounts field is made `volatile` to ensure visibility.

The `int` array has been turned into an `Integer` array in order to enable "striping" by locking just the accounts being modified.

`init`, `sumBalances`, `transferAccount` and `toString` lock the entire `accounts` array since the operations herein involve the entire array and would give inconsistent results otherwise.

`sumBalances` can show inconsistent results since a specific account can be modified while `sumBalances` iterates over the `accounts` array.

My solution **can** deadlock in both `transfer` and `transferAccount` since one thread can obtain (for example in `transfer`) a lock on one account while another thread has a lock on the other related account (and they are both waiting for each other). The same applies to `transferAccount` with two related `Account` objects.

Running `java -ea Runner` gives:

```
class UnsafeAccounts passed sequential tests
class UnsafeAccounts passed concurrent tests
```

Question 1.4

Relevant snippets of `LockAccountsFast` (the rest of the class is identical to `LockAccounts`) below:

```
public class LockAccountsFast implements Accounts {
    private volatile Integer[] accounts;
    private volatile Integer[] sums;
    private static final int threads = 4;

    [...]

    public int sumBalances() {
        int sum = 0;
        for (int i = 0; i < sums.length; i++) {
            synchronized (sums[i]) {
                sum += suns[i];
            }
        }
        return sum;
    }

    public void deposit(int to, int amount) {
        synchronized (accounts[to]) {
            int index = Thread.currentThread().hashCode()
                % sums.length;
            synchronized (sums[index]) {
                accounts[to] += amount;
                sums[index] += amount;
            }
        }
    }

    [...]
}
```

Question 1.5

The full `STMAccounts` can seen below.

```java
import java.util.Arrays;

public class STMAccounts implements Accounts {
    private volatile Integer[] accounts;

    public STMAccounts(int n) {
        accounts = new Integer[n];
        Arrays.fill(accounts, 0, accounts.length, 0);
    }

    public void init(int n) {
        atomic(() -> {
            this.accounts = new Integer[n];
            Arrays.fill(accounts, 0, accounts.length, 0);
        });
    }

    public int get(int account) {
        return atomic(() -> this.accounts[account]);
    }

    public int sumBalances() {
        return atomic(() -> {
            int sum = 0;
            for (int i = 0; i < this.accounts.length; i++) {
                sum += this.accounts[i];
            }
            return sum;
        });
    }

    public void deposit(int to, int amount) {
        atomic(() -> this.accounts[to] += amount);
    }

    public void transfer(int from, int to, int amount) {
        atomic(() -> {
            this.accounts[from] -= amount;
            this.accounts[to] += amount;
        });
    }

    public void transferAccount(Accounts other) {
        atomic(() -> {
            for (int i = 0; i < accounts.length; i++) {
                this.accounts[i] += other.get(i);
```

```
            }
        });
    }

    public String toString() {
        return atomic(() -> {
            String res = "";
            if (this.accounts.length > 0) {
                res = "" + this.accounts[0];
                for (int i = 1; i < this.accounts.length; i++) {
                    res = res + " " + this.accounts[i];
                }
            }
            return res;
        });
    }
}
```

I have been unable to run `Runner` with the multiverse .jar file. The code looks almost identical to the code I wrote in hand-in 9, though.

I have chosen to implement the `sumBalances` that risks competing with other method calls. The other implementation option would use transactions on a more granular level, but would then risk that the array could change while `toString`'ing.

Question 1.6

My full implementation of CASAccounts can be seen below:

```
public class CASAccounts implements Accounts {
    private AtomicInteger[] accounts;
    private AtomicInteger sum = new AtomicInteger();

    public CASAccounts(int n) {
        accounts = new AtomicInteger[n];
        for (int i = 0; i < accounts.length; i++) {
            accounts[i] = new AtomicInteger(0);
        }
    }

    public void init(int n) {
        accounts = new AtomicInteger[n];
        for (int i = 0; i < accounts.length; i++) {
            accounts[i] = new AtomicInteger(0);
        }
```

```
}

public int get(int account) {
    return accounts[account].get();
}

public int sumBalances() {
    return sum.get();
}

public void deposit(int to, int amount) {
    int previous, previousSum;
    do {
        previous = accounts[to].get();
        previousSum = sum.get();
    } while (!(accounts[to].compareAndSet(previous, previous + amount)
            && sum.compareAndSet(previousSum, previousSum + amount)));
}

public void transfer(int from, int to, int amount) {
    int previousTo, previousFrom;
    do {
        previousFrom = accounts[from].get();
        previousTo = accounts[to].get();
    } while (!(accounts[from].compareAndSet(previousFrom, previousFrom - amount) && acco
}

public void transferAccount(Accounts other) {
    for (int i = 0; i < accounts.length; i++) {
        int previous, otherValue, sumPrevious;
        do {
            previous = accounts[i].get();
            otherValue = other.get(i);
            sumPrevious = sum.get();
        } while (otherValue == other.get(i) && !(accounts[i].compareAndSet(previous, oth
                && sum.compareAndSet(sumPrevious, sumPrevious + otherValue)));
    }
}

public String toString() {
    String res = "";
    if (accounts.length > 0) {
        res = "" + accounts[0].get();
        for (int i = 1; i < accounts.length; i++) {
            res = res + " " + accounts[i].get();
        }
```

```
        }
        return res;
    }
}
```

I have added the `sum` field which mainains the current total sum of the accounts. `sum` is set using CAS in every method that has to update the sum. The update of `sum` is done in the same `while` loop as the other CAS updates in order to ensure correctness of the values.

No writes are lost in the tests I have performed, and `CASAccounts` passes both sequential and concurrent tests.

The operations cannot be guaranteed to happen in constant time since any given update/set of a value might have to be retried a number of times.

I cannot guarantee that the implementation does not livelock, since i.e. a `transfer` operation could go "back and forth" between values indefinitely.

Question 1.7