ASSIGNMENT 3

CS 454: Principles of Concurrent Programming / Spring 2024

Description

In this assignment, you will update your Assignment 1 blockchain.

Changes between Assignment 3 and Assignment 1 are highlighted in this document.

Your submission should extend the following abstract class:

```
abstract class Blockchain {
  abstract Wallet createWallet(int capacity);

  abstract Coin createCoin(int id);

  abstract boolean addCoins(Wallet w, Set coins);

  abstract boolean transferCoins(Wallet from, wallet to, Set coins);

  abstract boolean payRent(Wallet w, Set coins);

  abstract boolean redeemCoins(Wallet w, Set coins);

  abstract Set getCoins();

  abstract Set getCoins(Wallet w);

  abstract Set getCoins(List<Wallet> ws);

  abstract List<Action<Coin>> audit(Wallet w);

  abstract List<Action<Wallet>> audit(Coin c);
}
```

Each operation (method) behaves as follows:

- **createWallet**: Creates a wallet that can store a number of coins – `capacity`
- **createCoin**: Registers a new mined coin with a given `id`. The `id` is unique across the same `Blockchain`.
- **addCoins**: Adds all coins `coins` to the wallet `w`.
    - This operation either adds all the coins, if the wallet has enough capacity, or none.

    - For instance, attempting to add two coins to a wallet that only has room for one should not change the contents of the wallet.
    - If all the coins are added to the wallet, this operation returns `true`. If the wallet remains unchanged, this operation returns `false`.
- **transferCoins**: Moves `coins` currently present in the `from` wallet to the `to` wallet.
    - If there is not enough room in the `to` wallet, this operation should fail and return `false`.
    - If any coin is not present in the `from` wallet, this operation should fail and return `false`.
    - This operation returns `true` if it succeeds in moving all the coins between wallets.
- **redeemCoin**: Marks all the given `coins` provided as redeemed, which should be present on the given `wallet`.
    - Similarly to **addCoins**, this operation either marks all the coins or none.
    - Trying to redeem a coin that is not in the current wallet results in failure of the whole operation (i.e., no coins are redeemed).
    - If all the coins are redeemed, this operation returns `true`. If any coin was not redeemed, then no coins are modified and this operation returns `false`.
- **payRent**: Similar to redeem described above, but marks all coins as used in rent payments
- **getCoins**: Gets the coins in a given wallet (i.e., added to the wallet, and not redeemed or used for rent).
    - Without arguments, this operation lists all the coins that the blockchain produced that are still in circulation.
    - With a `wallet` argument, this operation lists all the coins currently in that wallet that are still in circulation.
- ~~**audit**: Returns an audit log that tracks coins and wallets.~~
    - ~~With a `coin` argument, returns a list of all the wallets in which the coin ever was~~
        - ~~The order of the list matters~~
        - ~~When transferring, coins should be removed from one wallet before being added to another wallet~~
    - ~~With a `wallet` argument, returns a list of all the coins that passed by that wallet~~
        - ~~The order of the list matters~~
            - ~~If an operation changes many coins at once, the order between those coins does not matter.~~
            - ~~However, all those coins should be on the list after preceding operations and before later operations~~

Besides the **Blockchain** interface, your solution should also implement the **Coin** interface for each coin, which defines the **getStatus** operation:

```
interface Coin {
    enum Status { MINED, IN_CIRCULATION, RENT , REDEEMED }
    Status getStatus();
}
```

Each coins should behave as follows:

- All coins are created as **MINED**
- A **MINED** coin can be added to a wallet, in which case it becomes **IN_CIRCULATION**
- Once a coin is in circulation, it cannot become **MINED** again
- A **IN_CIRCULATION** coin can be used to pay rent, and become **RENT**
- A **IN_CIRCULATION** coin can be redeemed, and become **REDEEMED**
- Once a coin is not in circulation, it cannot go back in circulation

Correctness Requirements

Your implementation should keep the following properties at all times:

Correctness 1. **getCoins** operations never list more coins than a wallet's capacity

Correctness 2. **getCoins** operations never list more items for the whole blockchain than the sum of the capacity of all the wallets.

Correctness 3. Adding mined coins to a wallet successfully results in those coins being listed in later **getCoins** operations.

Correctness 4. Once a coin is used to pay rent or redeemed, that coin is not listed in later **getCoins** operations.

Correctness 5. Each coin is listed in one wallet at most by **getCoins** operations.

Correctness 6. Coins are never "in-transit" due to transfer operations (i.e., **getCoins** operations not listing coins removed from the **from** wallet and still not added to the **to** wallet).

Correctness 7. Once the status of a coin is observed to be **RENT** or **REDEEMED**, it cannot be observed to be anything else from that point on.

Correctness 8. It is not possible to observe partial results of any operation. Each operation either happens completely, or not at all.

~~**Correctness 9.** The current contents of any wallet can be explained by following the entries in the audit log, by the order in which they appear in the log.~~

~~**Correctness 10.** The current state and location of any coin can be explained by following the entries in the audit log, by the order in which they appear in the log.~~

Concurrency Requirements – Progress

Your implementation should allow multiple threads to make progress at the same time, according to the following properties:

Progress 1. Concurrent **getCoins** operations should all make progress in parallel. Concurrent **getStatus** operations on the same coin should all make progress in parallel.

Progress 2. The following operations should all make progress in parallel when they are applied to different wallets and coins: **transferCoins, addCoins**, **redeemCoins**, **payRent**.

For instance, consider 3 threads executing at the same time as follows:

1. transfer coin 1 from wallet 1 to wallet 2
2. redeem coin 2 on wallet 3
3. addCoins coin 3 on wallet 4

In this example, all 3 threads should make progress at the same time as they operate on different wallets and different coins.

Progress 3. **transferCoins, addCoins**, **redeemCoins**, **payRent** and **getCoins** operations that do not target the same wallet should all make progress in parallel.

For instance, consider 3 threads executing at the same time as follows:

1. **addCoins** to wallet 1
2. **getCoins** on wallet 2
3. **getCoins** on wallets 3, and 4
4. **getCoins** without arguments (on the whole blockchain, wallets 1, 2, 3, and 4)

In this example, thread 1, 2, and 3 should all make progress in parallel because threads 2 and 3 read the contents of wallets not being changed by thread 1. Thread 4 cannot make progress in parallel with Thread 1, and it is acceptable for only one of these threads (i.e., either 1 or 4) to make progress until the other finishes its operation.

Concurrency Requirements – Correctness

Your implementation should be **linearizable**. All operations in any execution should appear to take place in a single point. You can use locks to ensure linearizability.

Linearizability 1. When adding multiple coins C1 and C2 to a wallet W with **addCoins**, it is not possible for other threads to see C1 added and C2 missing. Other threads calling **getCoins** either see all coins C1 and C2 in W, or none. It is also not possible for other threads to see C1.**getStatus** as MINED and C2.**getStatus** as IN_CIRCULATION (either both are MINED or both are IN_CIRCULATION).

Linearizability 2. If two threads attempt to add the same coin C at the same time to wallets W1 and W2, only one thread will succeed. This holds if the wallets are different (W1 ≠ W2) or the same (W1 = W2).

Linearizability 3. When moving multiple coins C1 and C2 between wallets W1 and W2 with **transferCoins**, it is not possible for other threads calling **getCoins** to see C1 in W1 and C2 in W2, C1 in W2 and C2 in W1. Either all coins moved are in W1 or W2. This does not affect other coins in W1 or W2 not targeted by the move.

Linearizability 4. When paying rent or redeeming coins C1 and C2 from a wallet with **payRent,** or **redeemCoins**, it is not possible for other threads calling **Coin.getStatus** to see C1 as IN_CIRCULATION and C2 as RENT/REDEEMED. Either all the coins are IN_CIRCULATION or all the coins are RENT/REDEEMED. This does not affect other coins in the same site not being used in the rent/redeem operations.

Linearizability 5. If two threads attempt to pay rent/redeem with the same coin C at the same time, only one thread will succeed.

Bonus Extra Functionality

Besides implementing all of the above, you can claim a 10% bonus by implementing **Coin.weakGetStatus**:

- **weakGetStatus** performs less work than **getStatus**, and should execute faster. You don't have to show that it executes faster, but you have to explain why you expect it to.
- Property **Correctness 7** holds for **weakGetStatus**
- Properties **Linearizability 1 and 4** do not hold for **weakGetStatus**, it is possible for other threads to see C1 as IN_CIRCULATION and C2 as RENT/REDEEMED (or the other way around).
- **weakGetStatus** does not have data-races.

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method Blockchain.createBlockchain so that it creates an instance of the class you added. You cannot change any other part of the code that is provided to you.

```
abstract class Blockchain {
    static Blockchain createBlockchain() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **March 9 2024** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%** for undergraduate students, and **50%** for graduate students. You can resubmit your assignment until **March 16 2024** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment-3` label, and non-anonymous to the instructors to count towards the bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 70%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 7 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

You need to pass Test 3 to get credit for Tests 4, 5, 6, and 7. This requirement prevents submitting an unchanged Assignment 1 and still get points without any effort towards the progress/concurrency requirements explained above.

Together with the code, you should submit three video screen-cast (**through Gradescope**) that answers the three questions below by explaining how your code works (one video per question). The questions focus on concurrency/multi-threading and are worth 10% each. You can record such a video using Zoom (create a meeting and record your screen)

1. Which operations can result in deadlocks, and what steps did you take to avoid them?
2. How do you ensure concurrent progress on operations that get the contents of the same testing site, or the status of a coin (**Progress 1**)?
3. How do you ensure linearizability when adding coins to a wallet and when redeeming/paying rent (**Linearizability 1 and 4**)?
4. (Bonus) How did you ensure that **weakGetStatus** performs as described in this document?

The maximum length for each video is 1 minute, instructors will stop watching at the 1 minute mark (nothing past that point in the video will be graded). This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `System.out.println("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.