Report: Thread Synchronization and Banking System

Ehsan R Habibagahi

Instructor: Mehrdad Ahmadzadeh Raji

1 Work done

The first step for me during this project was researching about the purpose of `threading.Condition` in Python and how to use it in production. The next step was learning about CMP (since Python built-in libs do so) and trying to make my code support that.

2 Implementation Challenges

2.1 Part 1

The most challenging and buggy section in part 1 was the `acquire()` for both Semaphore and Reentrant. I learned that I must use "Python's condition objects" to be able to handle atomicity and avoid race conditions. Also, this approach prevents busy-waiting which is highly CPU inefficient. Moreover, I encountered some problems around calculating and implementing the remaining time for the timeout functionality.

I added many additional sections in my code for debugging during testing as you may see in the code.

2.2 Part 2

The most challenging section in Part 2 was in `get_total_balance()`. It demanded to lock all accounts simultaneously to have a correct output of the real total balance. It spent a lot of time debugging this problem when it arose during system integrity test failures.

3 Synchronization Strategy

I avoided busy-wait strategy since it has 100% CPU usage and relying instead on `threading.Condition` to handle scheduling. The thread stays in a while loop until another one's done and turned back the lock. This loop prevents unexpected wakeups of the condition and checks and declares the remaining time to `wait()`.

4 Testing Results

Test results can be obtained by running `test_project.py`. As noted in the documentation, I set a 0.01s processing delay to simulate real-world conditions.

5 Final Summary

My code supports CMP and is developed modularly to make it easy for other users to import and run quickly.