Laboratory 1

### Problem: 2 Bank accounts

100 – accounts

50000 - operations  
  
Threads:  
 10 -> Time elapsed: 0.576 seconds   
 100 -> Time elapsed: 0.896 seconds  
 1000 -> Time elapsed: 1.023 seconds

Just for fun:

100 - threads

1000 – accounts

500000- operations

With logging into console:

Time elapsed: 3.968 seconds

In the banking simulation program, multiple threads concurrently perform transfer operations between accounts. To ensure the correctness and consistency of account states, synchronization mechanisms using mutexes (locks) are employed. This documentation outlines the rules for synchronization, including the mutexes used and the invariants they protect.

Memory updated

Here’s the updated documentation based on your code:

## Mutexes and Invariants

### 1. Account Mutex (Mtx in BankAccount class):

* **Mutex**: Each account has its own mutex (Mtx), which is a lock object that protects access to the account's state (balance and operation log).
* **Invariants Protected**:
  + **Balance Consistency**: The balance of the account must remain accurate during concurrent access. The operations that modify the balance (transfers) are protected by locking the account's mutex to prevent race conditions.
  + **Operation Log Integrity**: The log of operations (transfers) must accurately reflect all transactions for that account. The mutex ensures that only one thread can modify the log at a time, preventing incomplete or conflicting log entries.

### 2. Bank Mutex (Mtx in Bank class):

* **Mutex**: A shared mutex (Mtx) is used in the Bank class to control access to the correctness check mechanism.
* **Invariants Protected**:
  + **Correctness Check Execution**: The correctness checker runs in a separate thread and requires access to the bank's accounts. By locking the bank's mutex, it ensures that the checker does not interfere with ongoing transfer operations.
  + **Consistency of the Check Flag**: The shared boolean flag (check\_flag) is used to indicate when to stop the checker thread. The mutex protects this flag to ensure that it is safely read and modified, preventing inconsistent states.

## Rules for Synchronization

### 1. Locking Order:

* To avoid deadlocks, a consistent locking order is enforced:
  + When transferring funds between two accounts, the mutex for the accounts is acquired in a consistent order based on their unique identifiers (account IDs). The account with the lower ID is locked first.

java

Copy code

if (this.account\_id < other.account\_id) {

this.mutex.lock();

other.mutex.lock();

} else {

other.mutex.lock();

this.mutex.lock();

}

### 2. Avoiding Long-Lived Locks:

* Locks should be held for the shortest time possible. Transfer operations lock the accounts only for the duration necessary to update their balances and logs.

### 3. Correctness Checks:

* The correctness check should run periodically without blocking transfer operations. It releases its lock during checks to allow transfers to proceed while maintaining the integrity of the correctness evaluation.

### 4. Single Thread Access:

* Only one thread can access the critical sections (modifying balance and logs) at any given time due to the use of locks, ensuring that the program operates correctly in a concurrent environment.
* A correctness check could take place with a probability of approximately 11.1% for each transfer.

