CS5300 Assignment 5
Designing and Implementing Lock-Based Savings Account

Gautam Singh
CS21BTECH11018

Contents

***Abstract*—This report describes the design of a mutithreaded lock-based savings account in C++ that supports deposit and withdraw options. Additionally, there are two types of withdraws: ordinary and preferred, with the latter type of withdrawal having priority over the former. We analyze the throughput and latency of this implementation in various situations. Finally, we document some challenges faced during the implementation.**

1 Program Design

We give here an overview of the implementation details of the program.

1.1 Savings Account

The savings account object is implemented as a C++ class, following the interface given in Code 1.

We have one lock per account and two condition variables on that lock. The `preferredCondition` condition variable checks if there is a preferred withdrawal already waiting (except for the current thread). Threads check this by reading the value of `preferredWaiting`. On the other hand, the `balanceCondition` condition variable checks if the account has enough balance for withdrawal.

Code 1 C++ SavingsAccount interface.

```
class SavingsAccount {
private:
    uint32_t accNumber, preferredWaiting = 0;
    double balance = 0;
    std::mutex lock;
    std::condition_variable preferredCondition, balanceCondition;

public:
    SavingsAccount(uint32_t n);

    // Withdraw method
    void withdraw(bool preferred, double amount, int id, std::vector<std::string> &log);

    // Deposit method
    void deposit(double amount, int id, std::vector<std::string> &log);
};
```

1.2 Threads and Runner Functions

Threads are created using the C++ `std::thread` class. This makes it convenient to pass thread IDs and references to other objects using `std::ref` to the thread runner functions. In particular, we pass the following arguments to the runner function.

1) Thread ID from 0 to $N-1$, where N is the number of threads. This is because the `std::thread` class creates threads with IDs that may not be in this range, as they are only meant to be unique. However, these IDs are only used in logging and not in the snapshot algorithms.
2) Logging output vector (a reference to a C++ `std::vector<std::string>` object) to write timing information for further analysis and output. Our logs contain the (padded) time at the prefix, thus we can sort the logs directly.

1.3 Timing

The timestamps are reported using the `std::chrono` library. To prevent zero values, we report times in nanoseconds, which is the smallest available unit of time. However, results in the analysis are suitably scaled and reported in milliseconds where needed.

1.4 Random Number Generation

Random sleep times are generated using exponential distributions with a mean of α milliseconds. This is done using the `std::exponential_distribution` class. Further, the operation to be performed and the balance for the operation (whether deposit or withdrawal) is decided using uniform distributions. The randomness is generated using a Mersenne Twister, instantiated using an `std::mt19937` object. This random number generator is seeded using the current time since epoch.

2 Results and Analysis

2.1 Throughput vs. Number of Threads

The scalability of our implementation is depicted in Figure 1. In this setting, we vary the number of threads instantiated and plot the throughput curves for two sets of 10 and 50 accounts respectively. We observe the following.

1) The throughput is larger for the 50 account setting than the 10 account setting. Further, the increase is pronounced for a larger number of threads. This is because the work among a larger number of accounts is divided equally into more threads. Thus, there are fewer chances of blocking since threads invoke methods on different accounts.
2) The throughput in both cases decreases with increasing the number of threads. This might indicate more amount of blocking among threads when the number of threads increases, which is expected due to the higher competition.

2.2 Latency vs. Number of Operations per Thread

We now determine the latency of the savings account implementation by computing the average latency for all operations. Since this is inversely related to the throughput, we observe opposite trends here, as shown in Figure 2. Briefly, there are higher latencies for operating fewer number of accounts as well as for larger number of threads.

2.3 Latency vs. Number of Accounts

As observed in the previous experiments, we expect latency to be reduced as the number of accounts increase since the same number of threads can operate more accounts and are less likely. Thus, there are fewer waits on condition variables, which improves throughput. The results are shown in Figure 3.

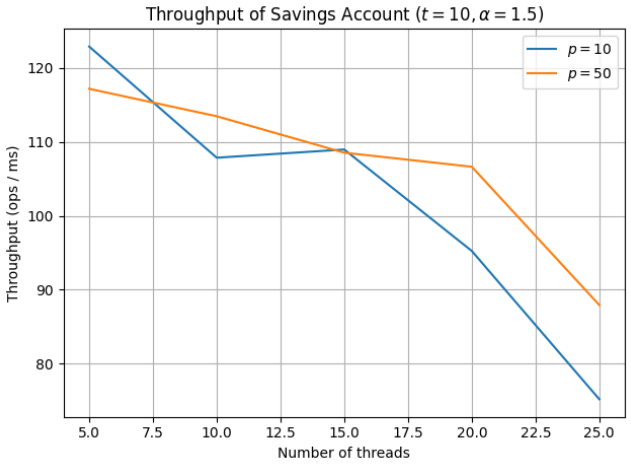

Figure 1. Throughput vs. Number of Threads

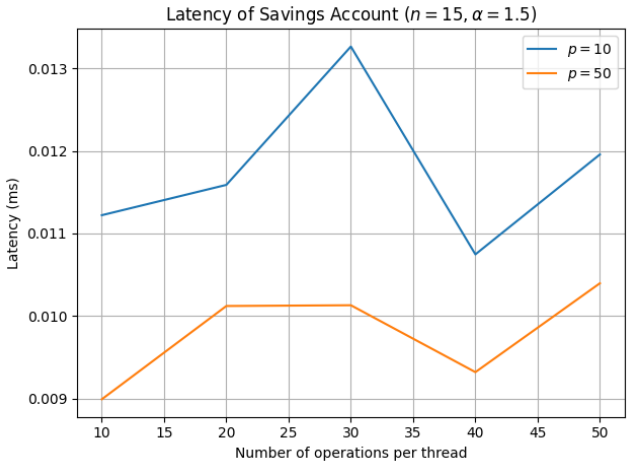

Figure 2. Latency vs. Number of operations per thread.

3 Challenges

Below, we list the various challenges faced during implementation.

1) Since locks and condition variables are used within the `SavingsAccount` class, we could not create a `SavingsAccount` object using the default constructor as that is deleted. The reason for this is that the locks and condition variables are not copyable. We fixed it by creating an array of pointers to the `SavingsAccount` objects, created using `std::make_unique`. Threads use these pointers to access the object.
2) There were instances where threads block forever due to insufficient funds. Since the amounts chosen are random, it is possible that a withdrawal blocks indefinitely and there is no deposit after. This would

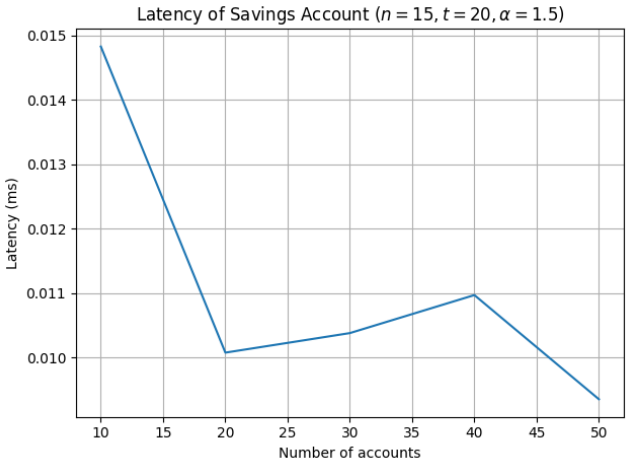

Figure 3. Latency vs. Number of operations per thread.

prevent the thread from being joined. To fix this, we simply introduced a large enough starter balance in every account.