

Project 1 Report

Multithreaded Banking Simulation

Jack Vega
CSC 3502 Operating Systems

October 8, 2025

1 Phase 1

1.1 Objective

The goal of Phase 1 was to design and implement a simple multithreaded banking simulation in C using the POSIX `pthread` library. Each thread represents a bank teller performing multiple transactions on one or more shared accounts.

This phase focuses on building the basic concurrent framework, identifying race conditions, and applying synchronization mechanisms to ensure data integrity.

1.2 Thread Design

Each teller is represented by a thread executing the `teller_thread()` function. Every teller performs a fixed number of transactions (`TRANSACTIONS_PER_TELLER`) on a shared account array.

Each thread receives its unique teller ID as an argument to identify its log entries and printed output.

1.3 Data Structures

- **Account:** contains an `account_id`, `balance`, and `transaction_count`.
- **logs:** a 2D array storing each threads transaction values for later verification.

1.4 Correctness Verification

At the end of execution:

1. Each tellers logged transactions are summed to compute the expected final balance.
2. The expected balance is compared against the actual shared account balance after all threads complete.

This allows us to evaluate whether a race condition occurred or not.

1.5 Challenges and Solutions

1.5.1 Correctness Verification

At first, verifying correctness seemed quite challenging. However, after reflecting on how race conditions occur specifically when multiple threads access the same memory location I realized something important. Since arrays are inherently divided into distinct elements, as long as each thread writes to a unique index, a race condition cannot occur. This insight made it possible to safely log all thread operations and then aggregate their results in a single thread to verify the correctness of the final output.

1.6 Performance Observations

- With only one account, there's little to no overhead
- Increasing the number of accounts (planned in later phases) will allow greater parallelism, but may present deadlock challenges.
- Although increasing the number of threads increases will speed up the throughput of data, it also increases the risk of race conditions.

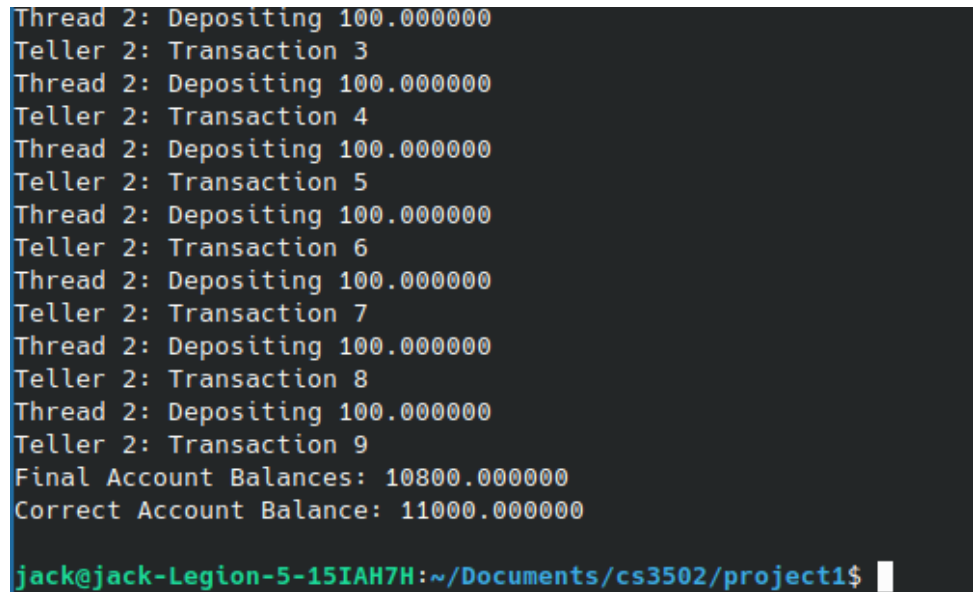
1.7 Program Output

Example excerpt from a program run (10 threads, 10 transactions each):

Listing 1: Phase 1 Output

```
1 Initial Balance: $1000.00
2 Teller 1: Transaction 2
3 Thread 1: Depositing $100.000000
4 Thread 2: Depositing $100.000000
5 Teller 2: Transaction 0
6 ...
7 Final Account Balances: $10900.00 <== Race Condition Detected!
8 Correct Account Balance: $11000.00
```

1.8 Screenshots

A terminal window with a dark background and light-colored text. The output shows a sequence of alternating messages from 'Thread 2' and 'Teller 2'. 'Thread 2' performs nine deposit transactions of 100.000000 each. 'Teller 2' performs nine transaction operations, numbered 3 through 9. At the end, the program prints the 'Final Account Balances' as 10800.000000 and the 'Correct Account Balance' as 11000.000000, indicating a discrepancy due to a race condition. The prompt at the bottom is 'jack@jack-Legion-5-15IAH7H:~/Documents/cs3502/project1\$' with a cursor.

```
Thread 2: Depositing 100.000000
Teller 2: Transaction 3
Thread 2: Depositing 100.000000
Teller 2: Transaction 4
Thread 2: Depositing 100.000000
Teller 2: Transaction 5
Thread 2: Depositing 100.000000
Teller 2: Transaction 6
Thread 2: Depositing 100.000000
Teller 2: Transaction 7
Thread 2: Depositing 100.000000
Teller 2: Transaction 8
Thread 2: Depositing 100.000000
Teller 2: Transaction 9
Final Account Balances: 10800.000000
Correct Account Balance: 11000.000000
jack@jack-Legion-5-15IAH7H:~/Documents/cs3502/project1$
```

Figure 1: Phase 1 program output showing thread activity and final balance verification with a race condition having happened.

1.9 Conclusion

Phase 1 established a multithreaded structure for a banking simulation using POSIX threads showcasing race conditions.

2 Phase 2

2.1 Objective

The objective of Phase 2 was to perfect the multithreaded banking system from Phase 1 by introducing mutexes and preventing race conditions. This phase also lays the groundwork for supporting multiple accounts.

2.2 Key Differences from Phase 1

Phase 2 builds directly on the Phase 1 foundation but introduces several structural and functional improvements:

- **Dedicated deposit() Function:** All balance modifications are now handled through a separate function that encapsulates synchronization and overdraft checks.
- **Non-Blocking Locks:** Replaced `pthread_mutex_lock()` with `pthread_mutex_trylock()` to avoid stalling threads if an account is temporarily locked by another teller. (Stalling will be demonstrated in phase 3)
- **Transaction Retry Logic:** Threads reattempt failed transactions by decrementing the loop counter, ensuring each teller still completes the target number of successful transactions.
- **Enhanced Logging Structure:** The logs are now stored in a 3D array (`logs[account][teller]`) to properly record all teller activity per account. This is to further prepare for handling multiple accounts
- **Randomized Transaction Amounts:** Each transaction amount varies randomly, both positive (deposits) and negative (withdrawals), within a defined range.

2.3 Approach

Each teller thread repeatedly attempts to perform deposits or withdrawals on a randomly selected account. If the accounts mutex cannot be acquired immediately, the thread pauses and waits until it is acquired.

- **Thread Safety:** The shared account data remains protected by mutexes removing race conditions.
- **Validation:** Before modifying a balance, the system checks that withdrawals will not overdraft an account. Invalid operations are rejected.
- **Logging:** Each successful transaction is logged, allowing a correct final balance to be computed at the end for verification.

2.4 Challenges and Solutions

1. Overdraft Rejection. In earlier designs, withdrawing from a nearly empty account could cause negative balances. By checking the condition `if (amount < 0 && balance + amount < 0)`, we safely skip such operations and keep balances non-negative.

2.5 Performance Observations

- **Overhead Trade-off:** Some CPU cycles are waiting, but this is offset by removing the possibility of race conditions.

2.6 Program Output

Example console output (abridged):

Listing 2: Phase 2 Output

```
1 Initial Balance: $1000.00
2 Thread 0: Depositing $63.22
3 Thread 0: Depositing $63.22
4 ...
5 Thread 8: Withdrawing $100.31
6 Thread 8: Withdrawing $100.31
7 ...
8
9 =====
10          Final Account Balances
11 =====
12 Account 0: $393.11
13
14 =====
15          Correct Account Balances
16 =====
17 Account 0: $393.11
```

The final reported balances match the computed correct balances, confirming that thread synchronization, validation logic worked properly, and no race conditions occurred.

2.7 Screenshots

```
Thread 5: Depositing 102.04
Thread 5: Depositing 102.04
Thread 5: Depositing 102.04
Failed to acquire lock
Thread 4: Withdrawing 62.73
Thread 4: Withdrawing 73.97
Thread 4: Withdrawing 73.97
Thread 4: Withdrawing 73.97
Thread 4: Withdrawing 73.97
Thread 4: Withdrawing 73.97
Thread 4: Withdrawing 73.97
Thread 4: Withdrawing 73.97
Thread 4: Withdrawing 73.97
Thread 6: Withdrawing 3.15

=====
                Final Account Balances
=====
Account 0: 441.17

=====
                Correct Account Balances
=====
Account 0: 441.17
```

Figure 2: Phase 2 output showing concurrent transactions and verified final balances.

2.8 Conclusion

Phase 2 introduced mutex's to the banking simulation. This lets the program avoid race conditions and improves the accuracy of the account.

3 Phase 3

3.1 Objective

The objective of Phase 3 was to change the banking simulation to conduct balance transfers between accounts and explore the concept of deadlock occurrence.

3.2 Design Overview

Building upon Phase 2, this phase introduces two major system-level features:

- **Inter-account Transfers:** Each teller now randomly selects two distinct accounts and transfers a random amount between them.
- **Deadlock Detection:** The system intentionally introduces potential circular wait conditions to observe a deadlock senario.

The introduction of multiple accounts makes synchronization significantly more complex. When two accounts are locked simultaneously for a transfer, threads risk entering a circular wait if they each hold one lock and wait for the other, this is called a deadlock.

3.3 Deadlock Mechanism

To simulate deadlocks intentionally, each transfer first locks the **from** account and then attempts to lock the **to** account. A deliberate delay (`usleep(100)`) increases the likelihood that two threads will collide in opposite directions (e.g., AB and BA).

This demonstrates the occurrence of deadlocks.

3.4 Thread Function Behavior

Each teller thread:

1. Randomly selects two distinct account IDs.
2. Chooses a random transfer amount (between \$0.00 and \$125.94).
3. Calls the `transfer()` function to perform the transaction.

The randomness ensures diverse access patterns and increases contention probability.

3.5 Program Output

Example console output excerpt:

Listing 3: Phase 3 Output

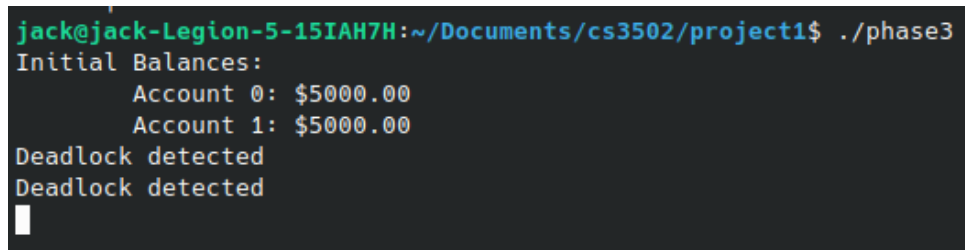
```
1 Initial Balances:
2   Account 0: $5000.00
3   Account 1: $5000.00
4 Thread 0: Transferring $12.34 from account 0 to 1
5 Thread 1: Transferring $93.48 from account 1 to 0
6 ... <== deadlocked
```

3.6 Challenges and Solutions

Deadlock Demonstration. The intentional `usleep()` delay combined with simultaneous locking of multiple accounts results in observable deadlocks. This showcases how circular wait conditions arise when two threads attempt opposite transfers.

Transfer Consistency. Even under deadlock scenarios, accounts maintain valid balances due to the locking hierarchy ensuring no partial modifications occur.

3.7 Screenshots

A terminal window with a dark background and light-colored text. The prompt is 'jack@jack-Legion-5-15IAH7H:~/Documents/cs3502/project1\$'. The command entered is './phase3'. The output shows 'Initial Balances:', 'Account 0: \$5000.00', 'Account 1: \$5000.00', and two lines of 'Deadlock detected'. A cursor is visible on the line following the second 'Deadlock detected' message.

```
jack@jack-Legion-5-15IAH7H:~/Documents/cs3502/project1$ ./phase3
Initial Balances:
    Account 0: $5000.00
    Account 1: $5000.00
Deadlock detected
Deadlock detected
█
```

Figure 3: Phase 3 output showing a deadlock.

3.8 Conclusion

Phase 3 successfully demonstrates both inter-account transfer logic and the occurrence of deadlocks in concurrent systems. This sets the foundation for **Phase 4**, where deadlock avoidance and recovery strategies will be introduced.

4 Phase 4

4.1 Objective

The objective of Phase 4 was to eliminate the deadlocks observed in Phase 3 by implementing an effective **deadlock avoidance mechanism**. This was accomplished by applying a consistent **lock ordering policy**, ensuring that circular wait conditions cannot occur.

4.2 Design Overview

In previous phases, transfers between two accounts occasionally caused deadlocks when multiple threads attempted opposite transfers simultaneously (e.g., Account 0 1 and Account 1 0). To resolve this, Phase 4 introduces the `safe_transfer()` function, which orders all lock acquisitions by account ID, guaranteeing a globally consistent locking sequence.

- **Lock Ordering:** Always lock the account with the smaller ID first.
- **Non-blocking Trylocks:** Use `pthread_mutex_trylock()` to gracefully handle contention without stalling.
- **Thread Retry:** Failed transactions are retried to ensure every teller still completes its designated number of operations.

This approach ensures that no two threads can hold locks in opposing orders, effectively preventing circular waits and deadlocks.

4.3 Implementation Details

The key innovation of Phase 4 lies in determining a global order for lock acquisition. Given two accounts `from_id` and `to_id`, the system defines:

```
first  = min(from_id, to_id)
second = max(from_id, to_id)
```

Threads then always lock the `first` account before the `second`, regardless of transfer direction.

This guarantees that no cyclic dependencies can form between threads, making deadlocks impossible by design.

4.4 Program Output

Listing 4: Phase 4 Output

```
1 Initial Balances:
2 ....Account 0: $5000.00
3 ....Account 1: $5000.00
4 Thread 2: Transferring $42.11 from account 0 to 1
5 Thread 4: Transferring $16.59 from account 1 to 0
6 Thread 7: Transferring $10.93 from account 0 to 1
7 ...
```

```
8
9 Final Account Balances
10 ....Account 0: $4988.38
11 ....Account 1: $5011.62
```

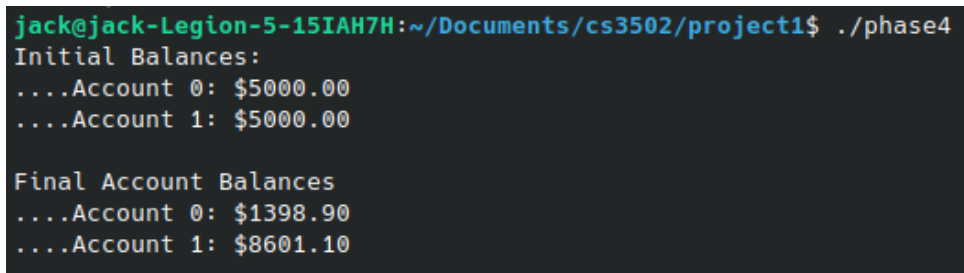
No deadlocks occur even under high contention, confirming successful implementation of lock ordering.

4.5 Challenges and Solutions

1. Balancing Correctness and Concurrency. Initially, strict lock ordering slightly reduced concurrency since threads could not lock accounts in arbitrary order. However, this trade-off was necessary to ensure total deadlock avoidance.

2. Retry Mechanism. The retry mechanism guarantees that failed transfers (due to temporarily busy accounts) are reattempted without affecting the total number of successful transactions per teller.

4.6 Screenshots



```
jack@jack-Legion-5-15IAH7H:~/Documents/cs3502/project1$ ./phase4
Initial Balances:
....Account 0: $5000.00
....Account 1: $5000.00

Final Account Balances
....Account 0: $1398.90
....Account 1: $8601.10
```

Figure 4: Phase 4 output showing no deadlocks or race conditions in the program output.

4.7 Conclusion

Phase 4 completes the core functionality of the multithreaded banking simulation by achieving both correctness and safety under concurrent inter-account transactions. By applying a simple yet effective lock ordering rule, all deadlocks were eliminated without requiring complex detection or recovery systems.

This phase demonstrates the practical importance of structured synchronization design and forms the final step toward a fully reliable multithreaded banking model.