**Hour 1: Introduction to Concurrency and Problems in Concurrency**

1. **Introduction to Concurrency (20 minutes)**
   * Explain why concurrency is important in modern computing systems.
   * Discuss real-world examples like operating systems and internet services that require concurrency.
   * Mention complexity introduced by concurrency.
2. **Role Play: Updating a Student's Bank Account (15 minutes)**
   * Explain the scenario: multiple entities (Rik, Loan Company, Gran, Bank Manager) updating a bank account concurrently.
   * Identify potential issues like race conditions, mutual exclusion, and critical sections.
3. **Java Implementation: Simple Bank Account Simulation (25 minutes)**
   * Demonstrate a Java program simulating updates to Rik's bank account.
   * Implement without synchronization first to show race conditions.
   * Code sample demonstrating unsynchronized access.

**Hour 2: Synchronization and Solving Concurrency Problems**

1. **Problems in Concurrency (15 minutes)**
   * Outline issues like race conditions, deadlock, and livelock.
   * Describe key concepts such as mutual exclusion, critical sections, and synchronization mechanisms.
2. **Java Implementation: Using synchronized to Solve Race Conditions (25 minutes)**
   * Explain how to use Java’s synchronized keyword to handle mutual exclusion.
   * Modify the bank account example to include synchronized methods for balance updates.
3. **Java Practical Exercise (20 minutes)**
   * Task: Students modify the provided Java code to include proper synchronization.
   * Include bonus challenge: Implement a scenario with multiple accounts being updated by different threads.

**Hour 3: Advanced Synchronization Mechanisms**

1. **Avoiding Deadlock: Resource Locking Strategies (20 minutes)**
   * Explain how deadlock occurs and discuss strategies to avoid it.
   * Discuss resource ordering, the use of timeouts, and lock ordering.
2. **Java Implementation: ReentrantLock and tryLock() (25 minutes)**
   * Introduce ReentrantLock and show how to use it in Java.
   * Provide an example demonstrating the use of tryLock() to avoid deadlock in the bank account scenario.
3. **Class Discussion: Best Practices for Concurrency (15 minutes)**
   * Discuss common patterns and practices to manage concurrency in Java, including using locks effectively and minimizing critical sections.

**Detailed Notes:**

**Hour 1: Introduction to Concurrency and Problems in Concurrency**

**Introduction to Concurrency (20 minutes)**

**Why Concurrency is Important in Modern Computing Systems**

**Definition of Concurrency:**

Concurrency refers to the ability of a system to manage multiple tasks simultaneously. It allows multiple sequences of operations to be executed in overlapping periods, improving resource utilization and system performance.

**Importance in Modern Systems:**

1. **Multi-Core Processors:**
   * Modern CPUs come with multiple cores.
   * Concurrency allows programs to utilize these cores effectively.
   * Enables parallel execution, improving performance.
2. **Responsive User Interfaces:**
   * Concurrency allows UI applications to remain responsive.
   * Background tasks can run without freezing the UI.
3. **Resource Utilization:**
   * Efficient use of system resources like CPU, memory, and I/O devices.
   * Allows for better throughput and performance in applications.
4. **Scalability:**
   * Concurrency enables systems to handle increased loads.
   * Essential for applications like web servers and databases.

**Real-World Examples Requiring Concurrency**

1. **Operating Systems:**
   * Manages multiple processes and threads.
   * Schedules CPU time among processes.
   * Handles I/O operations concurrently.
2. **Web Servers:**
   * Serve multiple client requests simultaneously.
   * Each request can be handled by a separate thread or process.
3. **Database Systems:**
   * Handle concurrent transactions.
   * Ensure data consistency and isolation.
4. **Network Services:**
   * Email servers, chat applications, and streaming services.
   * Manage concurrent connections and data streams.
5. **Multimedia Applications:**
   * Play audio and video while processing user inputs.
   * Handle streaming and decoding concurrently.

**Complexity Introduced by Concurrency**

1. **Race Conditions:**
   * Occur when multiple threads access shared data simultaneously.
   * The outcome depends on the sequence or timing of threads.
2. **Deadlocks:**
   * Occur when two or more threads are waiting indefinitely for resources locked by each other.
3. **Resource Contention:**
   * Multiple threads competing for limited resources.
   * Can lead to performance degradation.
4. **Synchronization Overhead:**
   * Managing access to shared resources requires additional code.
   * Can introduce complexity and reduce performance if not handled properly.
5. **Debugging Challenges:**
   * Concurrency bugs are often non-deterministic.
   * Difficult to reproduce and fix.

**Role Play: Updating a Student's Bank Account (15 minutes)**

**Scenario Explanation**

**Characters Involved:**

1. **Rik (Student):**
   * Has a bank account with a certain balance.
2. **Loan Company:**
   * Deposits loan funds into Rik's account.
3. **Gran (Grandparent):**
   * Sends money to Rik's account as a gift.
4. **Bank Manager:**
   * Applies monthly interest to the account.

**Concurrent Updates:**

* All entities attempt to update Rik's bank account at the same time.
* Each operation affects the account balance.

**Potential Issues Identified**

1. **Race Conditions:**
   * Simultaneous access to the bank account balance without proper synchronization.
   * Final balance may not reflect all transactions correctly.
2. **Mutual Exclusion:**
   * Need to ensure that only one entity updates the balance at a time.
   * Prevents data corruption.
3. **Critical Sections:**
   * Sections of code where shared resources are accessed.
   * Require protection to avoid inconsistent states.

**Example of a Race Condition:**

* **Initial Balance:** $1000
* **Transactions:**
  + Loan Company deposits $5000.
  + Gran deposits $200.
* **Expected Final Balance:** $6200 after all transactions.
* **Race Condition Outcome:**
  + Due to overlapping operations, the final balance might be incorrect (e.g., $5200 or $1200).

**Java Implementation: Simple Bank Account Simulation (25 minutes)**

**Objective**

* Create a Java program that simulates concurrent updates to Rik's bank account.
* Implement the simulation without synchronization to demonstrate race conditions.

**Program Structure**

1. **BankAccount Class:**
   * Represents Rik's bank account.
   * Contains methods to deposit and withdraw funds.
   * Maintains the account balance.
2. **Transaction Threads:**
   * Each entity (Loan Company, Gran, Bank Manager) is represented by a thread.
   * Threads perform operations on the BankAccount object.
3. **Main Class:**
   * Initializes the BankAccount.
   * Starts multiple threads to simulate concurrent transactions.
   * Displays the final account balance.

**Code Implementation**

// BankAccount.java

public class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

// Method to deposit money

public void deposit(double amount) {

double newBalance = balance + amount;

// Simulate processing time

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

balance = newBalance;

}

// Method to withdraw money

public void withdraw(double amount) {

double newBalance = balance - amount;

// Simulate processing time

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

balance = newBalance;

}

public double getBalance() {

return balance;

}

}

// BankSimulation.java

public class BankSimulation {

public static void main(String[] args) {

BankAccount account = new BankAccount(1000);

// Thread for Loan Company deposit

Thread loanCompany = new Thread(() -> {

account.deposit(5000);

System.out.println("Loan Company deposited $5000");

});

// Thread for Gran's deposit

Thread gran = new Thread(() -> {

account.deposit(200);

System.out.println("Gran deposited $200");

});

// Thread for Bank Manager applying interest

Thread bankManager = new Thread(() -> {

double interest = account.getBalance() \* 0.05;

account.deposit(interest);

System.out.println("Bank Manager applied interest: $" + interest);

});

// Start all threads

loanCompany.start();

gran.start();

bankManager.start();

// Wait for all threads to finish

try {

loanCompany.join();

gran.join();

bankManager.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Display final balance

System.out.println("Final account balance: $" + account.getBalance());

}

}

**Explanation of the Code**

1. **BankAccount Class:**
   * **balance:** A private variable holding the account balance.
   * **deposit(double amount):** Adds the specified amount to the balance.
     + Reads the current balance.
     + Calculates the new balance.
     + Simulates processing time with Thread.sleep(100).
     + Updates the balance.
   * **withdraw(double amount):** Subtracts the specified amount from the balance.
     + Similar steps as the deposit method.
   * **getBalance():** Returns the current balance.
2. **BankSimulation Class:**
   * Creates an instance of BankAccount with an initial balance of $1000.
   * **Threads:**
     + **loanCompany:** Deposits $5000 into the account.
     + **gran:** Deposits $200 into the account.
     + **bankManager:** Applies 5% interest to the current balance and deposits it.
   * **Starting Threads:**
     + Calls the start() method on each thread to begin execution.
   * **Joining Threads:**
     + Uses join() to ensure the main thread waits for all transactions to complete before proceeding.
   * **Final Output:**
     + Prints the final account balance after all transactions.

**Demonstrating the Race Condition**

When you run the program multiple times, you may notice inconsistent final balances. This is due to the race condition caused by unsynchronized access to the balance variable.

**Sample Outputs:**

1. **Run 1:**

Gran deposited $200

Loan Company deposited $5000

Bank Manager applied interest: $50.0

Final account balance: $6250.0

1. **Run 2:**

Loan Company deposited $5000

Gran deposited $200

Bank Manager applied interest: $310.0

Final account balance: $6510.0

1. **Run 3:**

Bank Manager applied interest: $50.0

Gran deposited $200

Loan Company deposited $5000

Final account balance: $6250.0

**Analysis:**

* The final balance varies because the threads may read and write the balance at different times.
* Without synchronization, the updates to balance can overlap, leading to incorrect calculations.

**Conclusion of Hour 1**

* **Key Takeaways:**
  + Concurrency is essential for efficient modern computing.
  + It introduces complexity that must be managed carefully.
  + Without proper synchronization, concurrent programs can produce incorrect results due to race conditions.

**Hour 2: Synchronization and Solving Concurrency Problems**

**Problems in Concurrency (15 minutes)**

**Overview of Concurrency Issues**

Concurrency introduces several problems that can compromise the correctness and performance of a program. Understanding these issues is crucial for writing reliable concurrent applications.

**1. Race Conditions**

* **Definition**: A race condition occurs when the behavior of a software system depends on the relative timing of events, such as the order in which threads execute.
* **Cause**: Multiple threads accessing and modifying shared data simultaneously without proper synchronization.
* **Effect**: Leads to inconsistent or unexpected results.

**Example**: In the unsynchronized bank account simulation, multiple threads updating the balance simultaneously can cause incorrect final balances.

**2. Deadlocks**

* **Definition**: A deadlock occurs when two or more threads are blocked forever, each waiting for the other to release a resource.
* **Cause**: Circular wait conditions where threads hold locks and wait for locks held by others.
* **Effect**: The program hangs and cannot proceed further.

**Real-world Analogy**: Two people trying to pass through a narrow hallway from opposite ends and refusing to step back.

**3. Livelocks**

* **Definition**: A livelock occurs when threads continuously change their state in response to other threads without making any progress.
* **Cause**: Threads reacting to each other's actions, leading to an infinite loop of state changes.
* **Effect**: The program is active but not progressing towards completion.

**4. Starvation**

* **Definition**: A thread is perpetually denied access to resources it needs to make progress.
* **Cause**: Improper scheduling or priority settings.
* **Effect**: The thread cannot complete its task, affecting program correctness.

**Key Concepts in Concurrency**

**1. Mutual Exclusion**

* **Definition**: Ensuring that only one thread accesses a shared resource or critical section at a time.
* **Mechanisms**: Locks, synchronized blocks, and methods.

**2. Critical Sections**

* **Definition**: Parts of the code that access shared resources and must not be executed by more than one thread at a time.
* **Purpose**: Protect shared data from concurrent modifications.

**3. Synchronization Mechanisms**

* **Locks/Monitors**: Used to control access to critical sections.
* **Semaphores**: Counters that control access to shared resources.
* **Barriers**: Synchronize threads at a certain point in the program.

**Java Implementation: Using synchronized to Solve Race Conditions (25 minutes)**

**Objective**

* Modify the previous bank account simulation to include synchronization.
* Use Java's synchronized keyword to handle mutual exclusion.
* Ensure that race conditions are eliminated, and the final account balance is consistent.

**Understanding the synchronized Keyword**

**How synchronized Works**

* When a method or block is declared as synchronized, a thread must acquire the intrinsic lock (monitor) for the object before executing the code.
* Other threads attempting to enter the synchronized code will block until the lock is released.

**Types of Synchronization**

1. **Synchronized Methods**: The entire method is synchronized.
2. **Synchronized Blocks**: A block of code within a method is synchronized on a specific object.

**Modifying the BankAccount Class**

**Synchronizing Methods**

We will make the deposit and withdraw methods synchronized to ensure that only one thread can execute them at a time.

// BankAccount.java

public class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

// Synchronized method to deposit money

public synchronized void deposit(double amount) {

double newBalance = balance + amount;

// Simulate processing time

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

balance = newBalance;

}

// Synchronized method to withdraw money

public synchronized void withdraw(double amount) {

double newBalance = balance - amount;

// Simulate processing time

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

balance = newBalance;

}

// Synchronized method to get the balance

public synchronized double getBalance() {

return balance;

}

}

**Explanation**

* **public synchronized void deposit(double amount)**:
  + The method is now synchronized.
  + Threads must acquire the lock on the BankAccount instance before executing.
* **public synchronized double getBalance()**:
  + Synchronizing the getter ensures that the balance is read consistently.

**Running the Synchronized Simulation**

// BankSimulation.java

public class BankSimulation {

public static void main(String[] args) {

BankAccount account = new BankAccount(1000);

// Thread for Loan Company deposit

Thread loanCompany = new Thread(() -> {

account.deposit(5000);

System.out.println("Loan Company deposited $5000");

});

// Thread for Gran's deposit

Thread gran = new Thread(() -> {

account.deposit(200);

System.out.println("Gran deposited $200");

});

// Thread for Bank Manager applying interest

Thread bankManager = new Thread(() -> {

double interest;

synchronized (account) {

interest = account.getBalance() \* 0.05;

}

account.deposit(interest);

System.out.println("Bank Manager applied interest: $" + interest);

});

// Start all threads

loanCompany.start();

gran.start();

bankManager.start();

// Wait for all threads to finish

try {

loanCompany.join();

gran.join();

bankManager.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Display final balance

System.out.println("Final account balance: $" + account.getBalance());

}

}

**Explanation**

* **Synchronizing on the account object in the bank manager thread**:
  + Ensures that the balance read is consistent with the latest updates.
* **Consistency of Results**:
  + With synchronization, the final account balance should be consistent across runs.

**Sample Output**

Loan Company deposited $5000

Gran deposited $200

Bank Manager applied interest: $310.0

Final account balance: $6510.0

**Analysis**

* The final balance is now consistently correct.
* **Calculation**:
  + Initial Balance: $1000
  + After Loan Company Deposit: $6000
  + After Gran's Deposit: $6200
  + Interest Applied: 5% of $6200 = $310
  + Final Balance: $6200 + $310 = $6510

**Using Synchronized Blocks**

Alternatively, we can use synchronized blocks to limit the scope of synchronization.

public void deposit(double amount) {

synchronized (this) {

double newBalance = balance + amount;

// Simulate processing time

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

balance = newBalance;

}

}

**Explanation**

* Synchronizing only the critical section (code that modifies balance).
* This can improve performance by reducing the time the lock is held.

**Java Practical Exercise (20 minutes)**

**Task**

* **Objective**: Modify the provided Java code to include proper synchronization.
* **Bonus Challenge**: Implement a scenario with multiple accounts being updated by different threads.

**Steps**

1. **Add Synchronization**: Ensure all methods that access shared resources are properly synchronized.
2. **Create Multiple Accounts**:
   * Instantiate multiple BankAccount objects.
   * Simulate transfers between accounts.
3. **Handle Concurrency**:
   * Use threads to perform transfers concurrently.
   * Ensure that transfers are atomic and consistent.

**Sample Solution**

**BankAccount Class with Transfer Method**

public class BankAccount {

private double balance;

private final int accountNumber;

public BankAccount(int accountNumber, double initialBalance) {

this.balance = initialBalance;

this.accountNumber = accountNumber;

}

public synchronized void deposit(double amount) {

balance += amount;

// Simulate processing time

try {

Thread.sleep(50);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public synchronized void withdraw(double amount) {

balance -= amount;

// Simulate processing time

try {

Thread.sleep(50);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public double getBalance() {

return balance;

}

public int getAccountNumber() {

return accountNumber;

}

// Transfer method

public void transfer(BankAccount targetAccount, double amount) {

synchronized (this) {

synchronized (targetAccount) {

this.withdraw(amount);

targetAccount.deposit(amount);

System.out.println("Transferred $" + amount + " from Account "

+ this.accountNumber + " to Account " + targetAccount.getAccountNumber());

}

}

}

}

**BankSimulation Class with Multiple Accounts**

public class BankSimulation {

public static void main(String[] args) {

BankAccount account1 = new BankAccount(1, 5000);

BankAccount account2 = new BankAccount(2, 3000);

BankAccount account3 = new BankAccount(3, 2000);

// Thread for transferring from account1 to account2

Thread t1 = new Thread(() -> {

account1.transfer(account2, 1000);

});

// Thread for transferring from account2 to account3

Thread t2 = new Thread(() -> {

account2.transfer(account3, 500);

});

// Thread for transferring from account3 to account1

Thread t3 = new Thread(() -> {

account3.transfer(account1, 700);

});

// Start all threads

t1.start();

t2.start();

t3.start();

// Wait for all threads to finish

try {

t1.join();

t2.join();

t3.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Display final balances

System.out.println("Final balance of Account 1: $" + account1.getBalance());

System.out.println("Final balance of Account 2: $" + account2.getBalance());

System.out.println("Final balance of Account 3: $" + account3.getBalance());

}

}

**Expected Output**

Transferred $1000.0 from Account 1 to Account 2

Transferred $500.0 from Account 2 to Account 3

Transferred $700.0 from Account 3 to Account 1

Final balance of Account 1: $4700.0

Final balance of Account 2: $3500.0

Final balance of Account 3: $1800.0

**Analysis**

* **Consistency**: The balances reflect all transfers accurately.
* **Synchronization**: The transfer method acquires locks on both accounts involved to prevent race conditions.
* **Deadlock Potential**: Note that synchronizing on multiple objects can introduce deadlocks, which will be addressed in the next hour.

**Hour 3: Advanced Synchronization Mechanisms**

**Avoiding Deadlock: Resource Locking Strategies (20 minutes)**

**Understanding Deadlocks**

**Deadlock Conditions (Coffman Conditions)**

A deadlock can occur if all the following conditions are true:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode.
2. **Hold and Wait**: A thread holds a resource and waits for another.
3. **No Preemption**: Resources cannot be forcibly removed from threads.
4. **Circular Wait**: A circular chain of threads exists, where each thread holds at least one resource needed by the next thread.

**Strategies to Avoid Deadlock**

**1. Resource Ordering**

* **Assign a Global Order**: Assign a numeric or logical order to all locks/resources.
* **Acquire Locks in Order**: Ensure that all threads acquire locks in the predefined order.
* **Result**: Prevents circular wait conditions.

**Example**:

public void transfer(BankAccount targetAccount, double amount) {

BankAccount firstLock, secondLock;

if (this.accountNumber < targetAccount.accountNumber) {

firstLock = this;

secondLock = targetAccount;

} else {

firstLock = targetAccount;

secondLock = this;

}

synchronized (firstLock) {

synchronized (secondLock) {

// Perform transfer

}

}

}

**2. Use of Timeouts**

* **Try to Acquire Lock with Timeout**: Use methods that attempt to acquire a lock within a certain time frame.
* **Handle Failure Gracefully**: If the lock cannot be acquired, the thread can back off or try again later.
* **Result**: Reduces the chance of threads waiting indefinitely.

**3. Avoid Holding Multiple Locks**

* **Single Lock Design**: Design your system to require only one lock at a time.
* **Reduce Complexity**: Simplifies the synchronization and reduces deadlock risk.

**4. Lock Ordering**

* **Consistent Lock Acquisition**: Always acquire locks in the same order across all threads.
* **Avoid Circular Waits**: Eliminates the possibility of deadlocks due to circular dependencies.

**Java Implementation: ReentrantLock and tryLock() (25 minutes)**

**Introduction to ReentrantLock**

**What is ReentrantLock?**

* A lock with the same basic behavior as the implicit monitor lock accessed using synchronized methods and statements.
* Provides extended capabilities, such as:
  + **Fairness Policy**: Option to grant locks in the order they were requested.
  + **Interruptible Lock Acquisition**: Ability to interrupt threads waiting for a lock.
  + **Lock Polling**: Methods to check if the lock is available.

**Using ReentrantLock**

**Importing the Class**

import java.util.concurrent.locks.ReentrantLock;

**Creating a ReentrantLock**

private final ReentrantLock lock = new ReentrantLock();

**Implementing tryLock() to Avoid Deadlocks**

**Modifying the BankAccount Class**

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.ReentrantLock;

public class BankAccount {

private double balance;

private final int accountNumber;

private final ReentrantLock lock = new ReentrantLock();

// Constructors and other methods...

public boolean transfer(BankAccount targetAccount, double amount) throws InterruptedException {

boolean fromLockAcquired = false;

boolean toLockAcquired = false;

try {

// Try to acquire both locks within a timeout

fromLockAcquired = this.lock.tryLock(1000, TimeUnit.MILLISECONDS);

toLockAcquired = targetAccount.lock.tryLock(1000, TimeUnit.MILLISECONDS);

if (fromLockAcquired && toLockAcquired) {

this.withdraw(amount);

targetAccount.deposit(amount);

System.out.println("Transferred $" + amount + " from Account "

+ this.accountNumber + " to Account " + targetAccount.getAccountNumber());

return true;

} else {

System.out.println("Failed to acquire locks for transfer from Account "

+ this.accountNumber + " to Account " + targetAccount.getAccountNumber());

return false;

}

} finally {

if (fromLockAcquired) {

this.lock.unlock();

}

if (toLockAcquired) {

targetAccount.lock.unlock();

}

}

}

}

**Explanation**

* **tryLock(long timeout, TimeUnit unit)**:
  + Attempts to acquire the lock within the specified timeout.
  + Returns true if the lock was acquired, false otherwise.
* **Handling Lock Acquisition**:
  + If both locks are acquired, proceed with the transfer.
  + If unable to acquire both locks, handle the failure gracefully.
* **Finally Block**:
  + Ensures that any locks acquired are released, preventing deadlocks.

**Modifying the BankSimulation Class**

public class BankSimulation {

public static void main(String[] args) {

BankAccount account1 = new BankAccount(1, 5000);

BankAccount account2 = new BankAccount(2, 3000);

BankAccount account3 = new BankAccount(3, 2000);

// Thread for transferring from account1 to account2

Thread t1 = new Thread(() -> {

try {

account1.transfer(account2, 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

// Thread for transferring from account2 to account3

Thread t2 = new Thread(() -> {

try {

account2.transfer(account3, 500);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

// Thread for transferring from account3 to account1

Thread t3 = new Thread(() -> {

try {

account3.transfer(account1, 700);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

// Start all threads

t1.start();

t2.start();

t3.start();

// Wait for all threads to finish

try {

t1.join();

t2.join();

t3.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Display final balances

System.out.println("Final balance of Account 1: $" + account1.getBalance());

System.out.println("Final balance of Account 2: $" + account2.getBalance());

System.out.println("Final balance of Account 3: $" + account3.getBalance());

}

}

**Sample Output**

Transferred $1000.0 from Account 1 to Account 2

Transferred $500.0 from Account 2 to Account 3

Transferred $700.0 from Account 3 to Account 1

Final balance of Account 1: $4700.0

Final balance of Account 2: $3500.0

Final balance of Account 3: $1800.0

**Possible Failed Lock Acquisition**

If a lock cannot be acquired within the timeout, the transfer will not occur, and a message will be displayed.

Failed to acquire locks for transfer from Account 3 to Account 1

**Advantages of ReentrantLock**

* **Flexibility**: More control over lock acquisition and release.
* **Non-blocking Attempts**: Ability to attempt to acquire a lock without blocking indefinitely.
* **Fairness Policies**: Can create fair locks that grant access in the order requested.

**Class Discussion: Best Practices for Concurrency (15 minutes)**

**Effective Use of Locks**

* **Minimize the Scope of Locks**:
  + Synchronize only the necessary parts of the code.
  + Reduces contention and improves performance.
* **Avoid Nested Locks**:
  + Reduces the risk of deadlocks.
  + Simplifies the locking strategy.
* **Consistent Lock Ordering**:
  + Always acquire multiple locks in a consistent order.

**Designing for Concurrency**

* **Immutable Objects**:
  + Objects that cannot be modified after creation.
  + Thread-safe by default.
* **Use High-Level Concurrency Utilities**:
  + Java provides classes like ConcurrentHashMap, BlockingQueue, etc.
  + These classes handle synchronization internally and are optimized for concurrency.
* **Prefer Synchronization Utilities Over synchronized Keyword**:
  + Classes from java.util.concurrent package often provide better performance and scalability.

**Minimizing Critical Sections**

* **Keep Locks Held for Short Durations**:
  + Perform minimal work inside synchronized blocks.
  + Release locks as soon as possible.
* **Avoid I/O Operations Inside Critical Sections**:
  + I/O can be slow and unpredictable.
  + Holding locks during I/O can lead to thread contention.

**Testing and Debugging Concurrent Programs**

* **Use Thread-Safe Testing Tools**:
  + Tools like ThreadSafe or FindBugs can detect concurrency issues.
* **Simulate High Concurrency Scenarios**:
  + Test with many threads to uncover potential issues.
* **Log Thread Activities**:
  + Logging can help trace the sequence of events leading to an issue.