**Multithreaded Banking System**

**Introduction**

The proposed project aims to develop a multithreaded banking system capable of handling multiple concurrent transactions efficiently. By leveraging the power of multithreading, the system will enhance performance, scalability, and responsiveness.

**Implementation Steps**

1. **Database Design:** Create tables for accounts, transactions, and users.
2. **Account and Transaction Classes:** Define data structures for accounts and transactions.
3. **Multithreaded Server:** Develop a server to handle multiple client connections and process requests.
4. **Transaction Processing:** Implement deposit, withdrawal, transfer, and interest calculation logic.
5. **User Management:** Implement user registration, login, and authentication.
6. **Security Implementation:** Integrate encryption, authentication, and authorization.
7. **Testing and Debugging:** Conduct thorough testing to ensure system correctness and performance.

**Source Code**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

// Define a structure to represent a bank account

typedef struct {

int balance;

pthread\_mutex\_t lock; // Mutex to ensure thread-safe access to balance

} BankAccount;

// Function to deposit money into the bank account

void\* deposit(void\* arg) {

BankAccount\* account = (BankAccount\*)arg;

int amount = 200; // Amount to deposit

pthread\_mutex\_lock(&account->lock); // Lock the mutex before modifying balance

account->balance += amount;

printf("Deposited %d. New balance: %d\n", amount, account->balance);

pthread\_mutex\_unlock(&account->lock); // Unlock the mutex after modifying balance

pthread\_exit(NULL);

}

// Function to withdraw money from the bank account

void\* withdraw(void\* arg) {

BankAccount\* account = (BankAccount\*)arg;

int amount = 150; // Amount to withdraw

pthread\_mutex\_lock(&account->lock); // Lock the mutex before modifying balance

if (account->balance >= amount) {

account->balance -= amount;

printf("Withdrew %d. New balance: %d\n", amount, account->balance);

} else {

printf("Insufficient funds for withdrawal of %d. Current balance: %d\n", amount, account->balance);

}

pthread\_mutex\_unlock(&account->lock); // Unlock the mutex after modifying balance

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[4]; // Array to hold thread identifiers

BankAccount account; // Bank account instance

// Initialize the bank account

account.balance = 1000; // Starting balance

pthread\_mutex\_init(&account.lock, NULL); // Initialize the mutex

// Create threads to perform deposit and withdrawal operations

pthread\_create(&threads[0], NULL, deposit, (void\*)&account);

pthread\_create(&threads[1], NULL, deposit, (void\*)&account);

pthread\_create(&threads[2], NULL, withdraw, (void\*)&account);

pthread\_create(&threads[3], NULL, withdraw, (void\*)&account);

// Wait for all threads to complete

for (int i = 0; i < 4; i++) {

pthread\_join(threads[i], NULL);

}

// Destroy the mutex

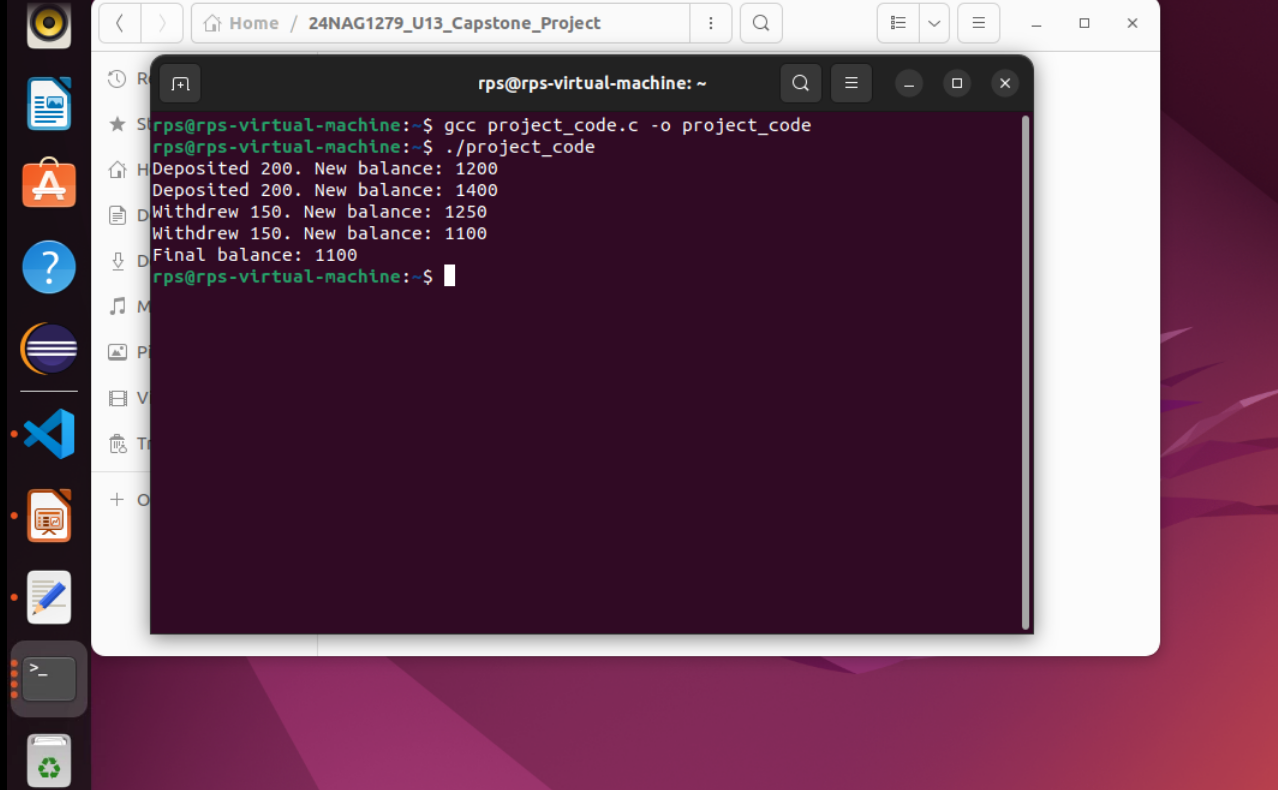
pthread\_mutex\_destroy(&account.lock);

printf("Final balance: %d\n", account.balance);

return 0;

}

**Output:**

****

**Future Enhancement**

A multithreaded banking system offers a strong foundation, but there's always room for improvement and adaptation to evolving technological landscapes. Here are some potential future enhancements:

 **Cloud Computing:** Migrate to cloud platforms for scalability, cost-efficiency, and disaster recovery.

 **Microservices Architecture:** Break down the system into smaller, independently deployable services for better maintainability and scalability.

 **Internet of Things (IoT):** Integrate IoT devices for enhanced security and user experience.

**Conclusion**

**Multithreaded banking systems** offer a substantial advantage in handling the high concurrency demands of the financial industry. By enabling simultaneous processing of multiple transactions, these systems significantly enhance performance, scalability, and responsiveness.