**PROJECT NAME**

Multithreaded Banking System

**INTRODUCTION**

In this project, we develop a multi-threaded simulation of banking transactions to illustrate the principles of concurrency and synchronization in C programming. The primary objective is to demonstrate how multiple threads can safely perform operations on shared resources, specifically bank accounts, using mutexes to prevent race conditions and ensure data consistency.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <string.h>

#define NUM\_ACCOUNTS 3

#define NUM\_THREADS 10

#define NAME\_LENGTH 200

typedef struct {

int id;

char name[NAME\_LENGTH];

int balance;

int deposit\_count; // Counter for deposits

int withdraw\_count; // Counter for withdrawals

int total\_deposited; // Total amount deposited

int total\_withdrawn; // Total amount withdrawn

pthread\_mutex\_t mutex;

} Account;

Account accounts[NUM\_ACCOUNTS];

int total\_deposited = 0; // Global variable to keep track of total deposited amount

int total\_withdrawn = 0; // Global variable to keep track of total withdrawn amount

pthread\_mutex\_t total\_deposited\_mutex = PTHREAD\_MUTEX\_INITIALIZER; // Mutex for accessing total\_deposited

pthread\_mutex\_t total\_withdrawn\_mutex = PTHREAD\_MUTEX\_INITIALIZER; // Mutex for accessing total\_withdrawn

// Initialize accounts with default values, names, and counters

void initialize\_accounts() {

const char\* names[NUM\_ACCOUNTS] = {"DEEPAK", "RAMESH", "SURESH"};

for (int i = 0; i < NUM\_ACCOUNTS; i++) {

accounts[i].id = i;

strncpy(accounts[i].name, names[i], NAME\_LENGTH - 1);

accounts[i].name[NAME\_LENGTH - 1] = '\0'; // Ensure null-termination

accounts[i].balance = 1000; // Initial balance

accounts[i].deposit\_count = 0; // Initialize deposit counter

accounts[i].withdraw\_count = 0; // Initialize withdrawal counter

accounts[i].total\_deposited = 0; // Initialize total deposited amount

accounts[i].total\_withdrawn = 0; // Initialize total withdrawn amount

pthread\_mutex\_init(&accounts[i].mutex, NULL);

}

}

// Deposit function with mutex locking

void deposit(Account \*account, int amount) {

pthread\_mutex\_lock(&account->mutex);

account->balance += amount;

account->deposit\_count++; // Increment deposit counter

account->total\_deposited += amount; // Update total deposited amount

printf("Deposited %d Rs to account %d (%s). New balance: %d Rs\n", amount, account->id, account->name, account->balance);

// Update the global total deposited amount

pthread\_mutex\_lock(&total\_deposited\_mutex);

total\_deposited += amount;

pthread\_mutex\_unlock(&total\_deposited\_mutex);

pthread\_mutex\_unlock(&account->mutex);

}

// Withdraw function with mutex locking

void withdraw(Account \*account, int amount) {

pthread\_mutex\_lock(&account->mutex);

if (account->balance >= amount) {

account->balance -= amount;

account->withdraw\_count++; // Increment withdrawal counter

account->total\_withdrawn += amount; // Update total withdrawn amount

printf("Withdrew %d Rs from account %d (%s). New balance: %d Rs\n", amount, account->id, account->name, account->balance);

// Update the global total withdrawn amount

pthread\_mutex\_lock(&total\_withdrawn\_mutex);

total\_withdrawn += amount;

pthread\_mutex\_unlock(&total\_withdrawn\_mutex);

} else {

printf("Insufficient funds in account %d (%s)\n", account->id, account->name);

}

pthread\_mutex\_unlock(&account->mutex);

}

// Thread function to perform random transactions

void\* transaction(void \*arg) {

int account\_id = \*(int\*)arg;

free(arg);

// Randomly deposit or withdraw

int action = rand() % 2; // 0 for deposit, 1 for withdraw

int amount = (rand() % 100) + 1; // Random amount between 1 and 100

if (action == 0) {

deposit(&accounts[account\_id], amount);

} else {

withdraw(&accounts[account\_id], amount);

}

return NULL;

}

int main() {

pthread\_t threads[NUM\_THREADS];

initialize\_accounts();

// Seed random number generator

srand(time(NULL));

for (int i = 0; i < NUM\_THREADS; i++) {

int \*account\_id = malloc(sizeof(int));

if (account\_id == NULL) {

perror("Failed to allocate memory");

return 1;

}

\*account\_id = rand() % NUM\_ACCOUNTS; // Randomly select an account

if (pthread\_create(&threads[i], NULL, transaction, account\_id) != 0) {

perror("Failed to create thread");

return 1;

}

}

for (int i = 0; i < NUM\_THREADS; i++) {

if (pthread\_join(threads[i], NULL) != 0) {

perror("Failed to join thread");

return 1;

}

}

// Print total deposited and withdrawn amounts

printf("Total amount deposited: %d Rs\n", total\_deposited);

printf("Total amount withdrawn: %d Rs\n", total\_withdrawn);

// Print deposit and withdrawal counts and totals for each account

for (int i = 0; i < NUM\_ACCOUNTS; i++) {

printf("Account %d (%s) - Deposits: %d, Total Deposited: %d Rs, Withdrawals: %d, Total Withdrawn: %d Rs\n",

accounts[i].id, accounts[i].name,

accounts[i].deposit\_count, accounts[i].total\_deposited,

accounts[i].withdraw\_count, accounts[i].total\_withdrawn);

}

// Cleanup

for (int i = 0; i < NUM\_ACCOUNTS; i++) {

pthread\_mutex\_destroy(&accounts[i].mutex);

}

pthread\_mutex\_destroy(&total\_deposited\_mutex);

pthread\_mutex\_destroy(&total\_withdrawn\_mutex);

return 0;

}

**OUTPUT**



**CONCLUSION**

The project successfully demonstrates the use of multi-threading and synchronization in C to manage concurrent transactions on shared resources. By using mutexes to control access to both per-account data and global counters, the simulation ensures accurate and consistent transaction processing. This project serves as a practical example of handling concurrency in real-world applications and highlights the importance of proper synchronization in multi-threaded environments