**Task 7**

**Description**

This task’s goal was to implement a deadlock detection algorithm. This system represents a banking where 2 accounts try to transfer at the same time causing a deadlock because there isn’t enough system resources to account for these processes.

**Code**

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <unistd.h>  
  
#define MAX\_ACCOUNTS 10  // Max accounts  
#define MAX\_TRANSACTIONS 10  // Max transactions  
  
pthread\_mutex\_t accounts[MAX\_ACCOUNTS];  // Mutex locks for accounts  
  
// Structure for a transaction  
typedef struct {  
    int from;  
    int to;  
} Transaction;  
  
// Transaction data  
Transaction transactions[MAX\_TRANSACTIONS];  
int num\_accounts = 0, num\_transactions = 0;  
  
// Track allocation and requests for deadlock detection  
int allocation[MAX\_TRANSACTIONS][MAX\_ACCOUNTS] = {0};  
int request[MAX\_TRANSACTIONS][MAX\_ACCOUNTS] = {0};  
int available[MAX\_ACCOUNTS] = {0};  // Initially, all resources (accounts) are free  
  
void read\_input\_file(const char\* filename) {  
    FILE\* file = fopen(filename, "r");  
    if (file == NULL) {  
        perror("Failed to open file");  
        exit(1);  
    }  
  
    fscanf(file, "%d", &num\_accounts);  // Read number of accounts  
    fscanf(file, "%d", &num\_transactions);  // Read number of transactions  
  
    printf("Number of accounts: %d\n", num\_accounts);  
    printf("Number of transactions: %d\n", num\_transactions);  
  
    // Initialize mutexes for accounts  
    for (int i = 0; i < num\_accounts; i++) {  
        pthread\_mutex\_init(&accounts[i], NULL);  
    }  
  
    // Read transaction details  
    for (int i = 0; i < num\_transactions; i++) {  
        fscanf(file, "%d %d", &transactions[i].from, &transactions[i].to);  
        printf("Transaction %d: Account %d to Account %d\n", i, transactions[i].from, transactions[i].to);  
    }  
  
    fclose(file);  
}  
  
// Function to simulate transaction processing  
void\* transfer(void\* arg) {  
    Transaction\* t = (Transaction\*) arg;  
    int tid = t->from;  // Use account index as thread ID  
  
    // Request resources (account from and account to)  
    printf("Transaction %d: Requesting Account %d\n", tid, t->from);  
    request[tid][t->from] = 1;  // Mark the resource request  
    pthread\_mutex\_lock(&accounts[t->from]);  
    allocation[tid][t->from] = 1;  // Mark as allocated  
    printf("Transaction %d: Locked Account %d\n", tid, t->from);  
  
    sleep(1);  // Simulate some processing time  
  
    printf("Transaction %d: Requesting Account %d\n", tid, t->to);  
    request[tid][t->to] = 1;  // Mark the resource request  
    pthread\_mutex\_lock(&accounts[t->to]);  
    allocation[tid][t->to] = 1;  // Mark as allocated  
    printf("Transaction %d: Locked Account %d\n", tid, t->to);  
  
    // Release resources after transfer (no deadlock here)  
    pthread\_mutex\_unlock(&accounts[t->to]);  
    pthread\_mutex\_unlock(&accounts[t->from]);  
  
    allocation[tid][t->from] = 0;  
    allocation[tid][t->to] = 0;  
    printf("Transaction %d: Completed\n", tid);  
  
    return NULL;  
}  
  
// Deadlock detection function  
void detect\_deadlock() {  
    int finish[MAX\_TRANSACTIONS] = {0};  // 0 means unfinished  
    int work[MAX\_ACCOUNTS];  
  
    // Copy available resources  
    for (int i = 0; i < num\_accounts; i++) work[i] = available[i];  
  
    // Try to find a process that can complete  
    int deadlock\_detected = 0;  
    while (1) {  
        int found = 0;  
        for (int i = 0; i < num\_transactions; i++) {  
            if (!finish[i]) { // If the process isn't finished  
                int canProceed = 1;  
                for (int j = 0; j < num\_accounts; j++) {  
                    if (request[i][j] > work[j]) { // If it still needs more than available  
                        canProceed = 0;  
                        break;  
                    }  
                }  
                if (canProceed) {  
                    // Process can finish, release its resources  
                    for (int j = 0; j < num\_accounts; j++)  
                        work[j] += allocation[i][j];  
                    finish[i] = 1;  
                    found = 1;  
                }  
            }  
        }  
  
        if (!found) {  // If no process can proceed, stop  
            break;  
        }  
    }  
  
    // Check for deadlock  
    printf("\nDeadlock Detection:\n");  
    for (int i = 0; i < num\_transactions; i++) {  
        if (!finish[i]) {  
            printf("❌ Deadlock detected: Transaction %d is stuck.\n", i);  
            deadlock\_detected = 1;  
        }  
    }  
  
    if (!deadlock\_detected) {  
        printf("✅ No deadlock detected.\n");  
    }  
}  
  
int main() {  
    // Use the full path to the input file  
    read\_input\_file("/home/ahmed/task7/transaction.txt");  
  
    pthread\_t threads[MAX\_TRANSACTIONS];  
  
    // Create transaction threads  
    for (int i = 0; i < num\_transactions; i++) {  
        pthread\_create(&threads[i], NULL, transfer, &transactions[i]);  
    }  
  
    // Wait for a while before running deadlock detection  
    sleep(3);  
    detect\_deadlock();  
  
    // Join threads (this won't happen if deadlock occurs)  
    for (int i = 0; i < num\_transactions; i++) {  
        pthread\_join(threads[i], NULL);  
    }  
  
    // Destroy mutexes  
    for (int i = 0; i < num\_accounts; i++) {  
        pthread\_mutex\_destroy(&accounts[i]);  
    }  
  
    return 0;

}

**Gitlink**

[**https://github.com/FirasAhmed2/Operating-systems-coursework.git**](https://github.com/FirasAhmed2/Operating-systems-coursework.git)