Concurrent Banking System In Linux

212010020065

13 August , 2025

Table of Contents

Revision History

Version	Name	Calender	Description of Changes
1.0	Furkan can Süme	13 August , 2025	First version
1.1	Furkan Can Süme	19 August , 2025	Sending non-existent money solved

OS Assignment:Concurrent Banking System - Detailed Summary

Objective:

The goal is to implement a concurrent banking system using process creation, inter-process communication (IPC), and synchronization techniques.

General System Overview:

A main process will read transactions (withdraw, deposit, transfer) from a file and create a separate child process for each transaction. Transactions will be executed on shared memory. Synchronization and deadlock prevention will be crucial due to concurrent execution.

Tasks to be Done (Step-by-Step):

1. Account and Transaction Structure:

- Each account has a balance, stored in shared memory.
- Supported transaction types:
  - Withdraw(amount, account_id)
  - Deposit(amount, account_id)
  - Transfer(amount, from_account_id, to_account_id)

2. Process Handling:

- The main process:
  - Reads the transaction list from a file.
  - Spawns a child process for each transaction.
- The child process:
  - Updates the account balance in shared memory.
  - Logs the transaction in the shared log table.
  - Exits with code 0 on success or -1 on failure (e.g., insufficient funds).

3. Synchronization and Deadlock Prevention:

- Access to account balances must be synchronized using semaphores.
- During transfer operations, since two accounts are accessed:
  - Always lock the account with the smaller account number first, then the larger one. This prevents deadlocks.

4. IPC (Inter-Process Communication):

- The child process communicates the transaction result to the parent process via exit codes:
  - 0: success
  - -1: failure

5. Retry Mechanism:

- If a transaction fails (e.g., due to insufficient funds), the main process retries it once.

Implementation Steps:

1. Initialize accounts in shared memory with predefined balances.
2. Read the number of accounts and transactions from file.
3. Read account information as <account_id, balance>.
4. Read transactions in the format <type, from_id, to_id, amount>.
5. Create a child process for each transaction.
6. Use semaphores to ensure mutual exclusion during balance updates.
7. Retry any failed transaction once.
8. Use exit codes to communicate results from child to parent process.

Sample Output:

Final account balances and transaction logs should be displayed, for example:
Transaction 0: Deposit 100 to Account 0 (Success)
Transaction 2: Transfer 615 from Account 2 to Account 3 (Failed)
Transaction 2: Transfer 615 from Account 2 to Account 3 (Success) // retry succeeded

Used for application

Ubuntu:

Ubuntu is a Linux-based open-source operating system. It is user-friendly and is commonly preferred in areas such as software development, server management, and cybersecurity. It can be used as an alternative to Windows. It is mostly managed using terminal-based commands.(Figure 0.1)

VMware:

VMware is a virtualization software. It allows you to run multiple virtual machines on a single physical computer. For example, you can install a virtual Ubuntu system on Windows and test Ubuntu through it.(Figure 0.2)

Nano:

Nano is a simple, terminal-based text editor used in Linux systems. It is used to edit code files or configuration files. It is easy to use and operates from the command line.(Vim was not used because it is not that complex a program)

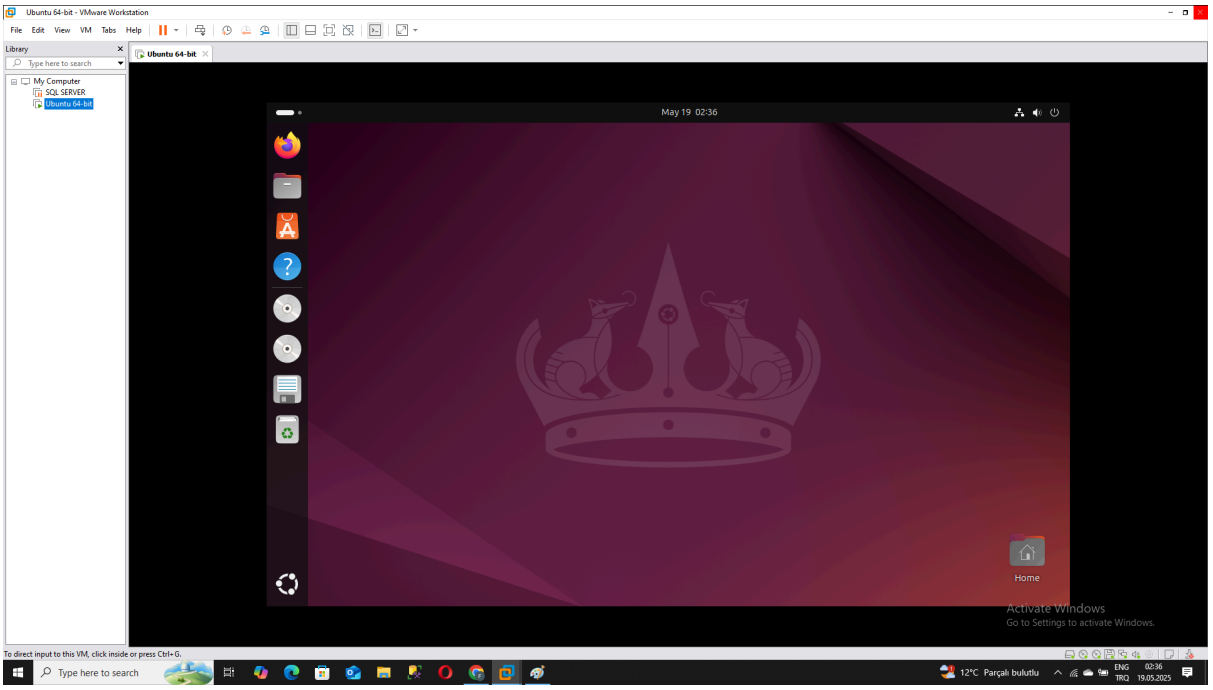


Figure 0.1 An image of Ubuntu

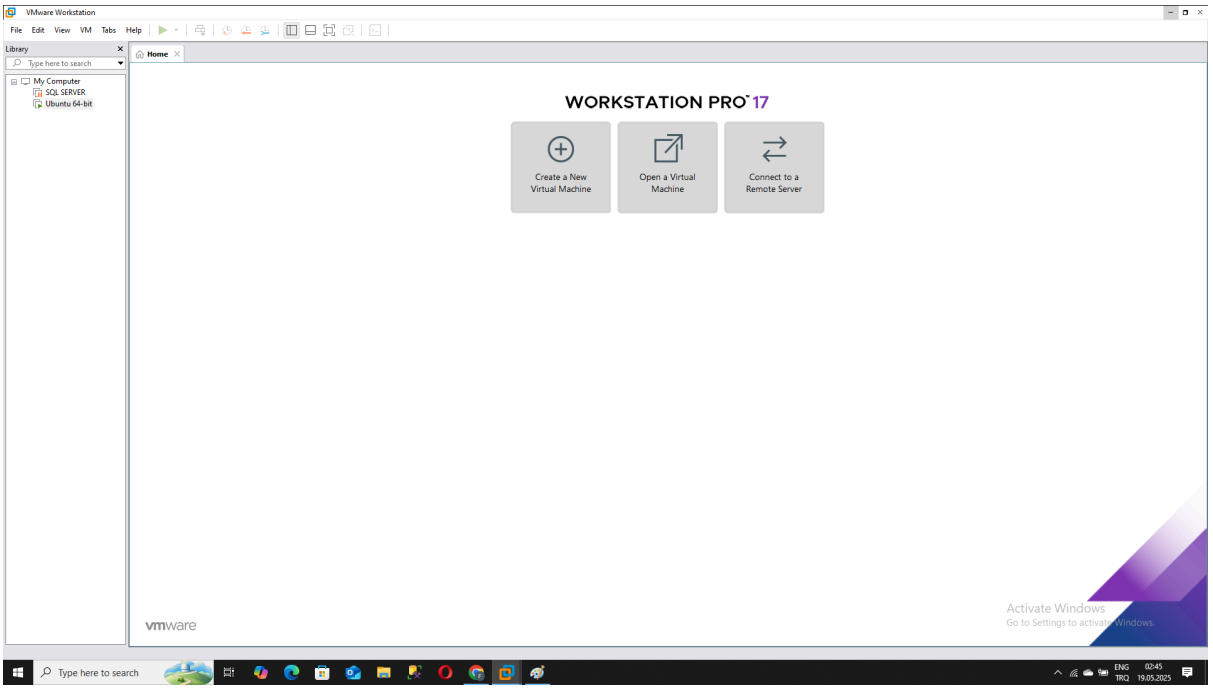


Figure 0.1 An image of VMware

Code Explanation and Implementation Details

This project is organized into three main files, each responsible for a distinct part of the system:

main.c CODE

```
#include "accounts.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <time.h>

void execute_transaction(SharedData *s, int type, int source, int target, float amount, int
current_trans);
void print_results(SharedData *s);

int main() {
    // 1. Paylaşılan bellek  Shared memory
    int shm_id = shmget(IPC_PRIVATE, sizeof(SharedData), 0666|IPC_CREAT);
    if(shm_id == -1) {
        perror("shmget failed");
        exit(1);
    }

    SharedData *shared = (SharedData*)shmat(shm_id, NULL, 0);
    if(shared == (void*)-1) {
        perror("shmat failed");
        exit(1);
    }

    // 2. Semafor init
    sem_init(&shared->global_lock, 1, 1);
    for(int i=0; i<MAX_ACCOUNTS; i++) {
        sem_init(&shared->accounts[i].lock, 1, 1);
    }

    // 3. Hesapları başlat  Start accounts
    shared->transaction_count = 0;
    for(int i=0; i<5; i++) {
```

```
        shared->accounts[i].account_id = i;
        shared->accounts[i].balance = 1000.0;
    }

    // 4. Dosyayı aç Open file
    FILE *file = fopen("transactions.txt", "r");
    if(!file) {
        perror("Failed to open file");
        exit(1);
    }

    // 5. İşlemleri oku ve işle  Read and process transactions
    int type, source, target;
    float amount;
    char line[100];

    while(fgets(line, sizeof(line), file)) {
        if(sscanf(line, "%d %d %d %f", &type, &source, &target, &amount) != 4) {
            fprintf(stderr, "Invalid line format: %s", line);
            continue;
        }

        printf("Processing: type=%d from=%d to=%d amount=%.2f\n",
               type, source, target, amount);

        sem_wait(&shared->global_lock);
if(shared->transaction_count >= MAX_TRANSACTIONS) {
    sem_post(&shared->global_lock);
    fprintf(stderr, "Transaction limit reached (%d)\n", MAX_TRANSACTIONS);
    continue;
}
int current_trans = shared->transaction_count++;
sem_post(&shared->global_lock);

        pid_t pid = fork();
        if(pid == 0) { // Child
            execute_transaction(shared, type, source, target, amount, current_trans);
            _exit(0);
        }
        else if(pid > 0) { // Parent
            int status;
            waitpid(pid, &status, 0);
        }
        else {
            perror("fork failed");
        }
```

```
    }
    fclose(file);

    // 6. Sonuçları yazdır Print results
    print_results(shared);

    // 7. Temizlik  Cleaning
    shmdt(shared);
    shmctl(shm_id, IPC_RMID, NULL);
    return 0;
}

void execute_transaction(SharedData *s, int type, int source, int target, float amount, int
current_trans) {
    // Kaydı doldur Fill out registration
    s->transactions[current_trans].transaction_id = current_trans;
    s->transactions[current_trans].type = type;
    s->transactions[current_trans].source_account = source;
    s->transactions[current_trans].target_account = target;
    s->transactions[current_trans].amount = amount;
    s->transactions[current_trans].timestamp = time(NULL);

    switch(type) {
        case 0: // Para çekme Withdrawal ( NEW 1.1 FİX)
    sem_wait(&s->accounts[source].lock);
    if(s->accounts[source].balance >= amount) {
        s->accounts[source].balance -= amount;
        s->transactions[current_trans].status = 0;
        printf("Success: Withdrew %.2f from %d\n", amount, source);
    } if(s->accounts[source].balance <= amount){
        s->transactions[current_trans].status = -1;
        printf("Failed: Insufficient balance in %d (Current: %.2f)\n",
              source, s->accounts[source].balance);
    }
    sem_post(&s->accounts[source].lock);
    break;


        case 1: // Para yatırma Deposit money
            sem_wait(&s->accounts[target].lock);
            s->accounts[target].balance += amount;
            s->transactions[current_trans].status = 0;
            printf("Success: Deposited %.2f to %d\n", amount, target);
            sem_post(&s->accounts[target].lock);
            break;
```

```
        case 2: // Transfer
            int first = (source < target) ? source : target;
            int second = (source < target) ? target : source;

            sem_wait(&s->accounts[first].lock);
            sem_wait(&s->accounts[second].lock);

            if(s->accounts[source].balance >= amount) {
                s->accounts[source].balance -= amount;
                s->accounts[target].balance += amount;
                s->transactions[current_trans].status = 0;
                printf("Success: Transferred %.2f from %d to %d\n", amount, source, target);
            } else {
                s->transactions[current_trans].status = -1;
                printf("Failed: Transfer from %d to %d\n", source, target);
            }

            sem_post(&s->accounts[second].lock);
            sem_post(&s->accounts[first].lock);
            break;
    }
}

void print_results(SharedData *s) {
    printf("\nFINAL ACCOUNT BALANCES:\n");
    for(int i=0; i<5; i++) {
        printf("Account %d: %.2f\n", i, s->accounts[i].balance);
    }

    printf("\nTRANSACTION LOG:\n");
    printf("ID | Type      | From | To  | Amount | Status   | Timestamp\n");
    printf("-------------------------------------------------------------\n");

    for(int i=0; i<s->transaction_count; i++) {
        Transaction t = s->transactions[i];
        char time_str[26];
        ctime_r(&t.timestamp, time_str);
        time_str[strlen(time_str)-1] = '\0'; // Remove newline

        const char* type_str;
        switch(t.type) {
            case 0: type_str = "WITHDRAW"; break;
            case 1: type_str = "DEPOSIT "; break;
            case 2: type_str = "TRANSFER"; break;
            default: type_str = "UNKNOWN"; break;
        }
```

```
        printf("%2d | %-8s | %4d | %3d | %7.2f | %-8s | %s\n",
       t.transaction_id,
       type_str,
       t.source_account,
       t.target_account,
       t.amount,
       t.status == 0 ? "SUCCESS" : "FAILED",
       time_str);
    }
}
```

main.c CODE DESCRIPTION

In the main.c file, I organized the code into **seven logical sections** to make the workflow clearer and easier to navigate. Although this isn't a formal modularization, it helped me keep track of the operations and avoid confusion while coding. Some inline comments may appear in Turkish — those were added for my own reference and learning. The seven sections are:

1. **Shared Memory Setup**
2. **Semaphore Initialization**
3. **Initialize Accounts**
4. **Open the Transaction File**
5. **Read and Process Transactions**
6. **Print Final Results**
7. **Cleanup and Resource Deallocation**

While there are additional operations scattered across the code, these seven segments represent the main flow of the system and provide a structured overview of how the concurrent banking system operates.

İmportant Detail

It is also worth noting that each account is **initialized with a starting balance of 1000 units**. While this approach is sufficient for testing the system and observing transaction behavior at the current stage, it can definitely be improved in the future. For instance, initial balances could be input by the user or generated randomly to simulate more realistic scenarios. However, for now, this fixed value was chosen to focus on verifying the core functionality of the system.

Transactions.txt (replaceable):

This file lists the banking transactions to be processed by the system. Each line includes a transaction type, source account, destination account (if applicable), and the transaction amount.

Transactions.txt example and actions to be taken

```
1 0 0 500.00   # Deposit 500 to account 0
0 1 0 200.00   # Withdraw 200 from account 1
2 0 1 300.00   # Transfer 300 from 0 to 1
0 2 0 1000.00  # Attempt to withdraw 1000 from account 2 (should fail)
```

Account 0: 1200.00 (1500 - 300)
Account 1: 1100.00 (1000 - 200 + 300)
Account 2: 0.00   (1000 - 1000)
Account 3: 1000.00
Account 4: 1000.00

İmportant Detail

If it is not written as in the example given above, it will not work and the operations will be written as UNKNOWN.

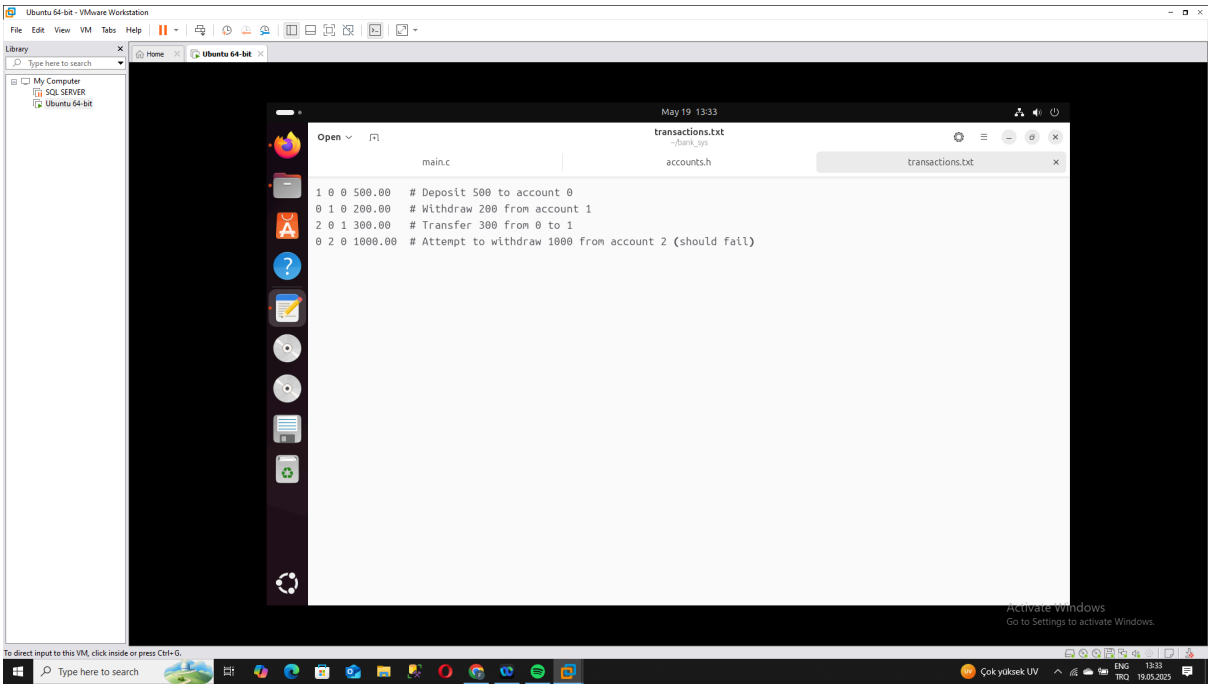


Figure 0.3 An image of Transactions.txt

account.h CODE

```
#ifndef ACCOUNTS_H
#define ACCOUNTS_H

#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>

#define MAX_ACCOUNTS 10
#define MAX_TRANSACTIONS 100

#define WITHDRAWAL 0
#define DEPOSIT 1
#define TRANSFER 2

typedef struct {
    int account_id;
    float balance;
    sem_t lock;
} Account;


typedef struct {
    int transaction_id;
    int type;
    int source_account;
    int target_account;
    float amount;
    int status;
    time_t timestamp;
} Transaction;

typedef struct {
    Account accounts[MAX_ACCOUNTS];
    Transaction transactions[MAX_TRANSACTIONS];
    sem_t global_lock;
    int transaction_count;
    time_t timestamp;
} SharedData;

#endif
```

account.h CODE DESCRIPTION

This file contains the data structures (such as account IDs and balances), shared memory declarations, semaphore identifiers, and function prototypes related to account operations.

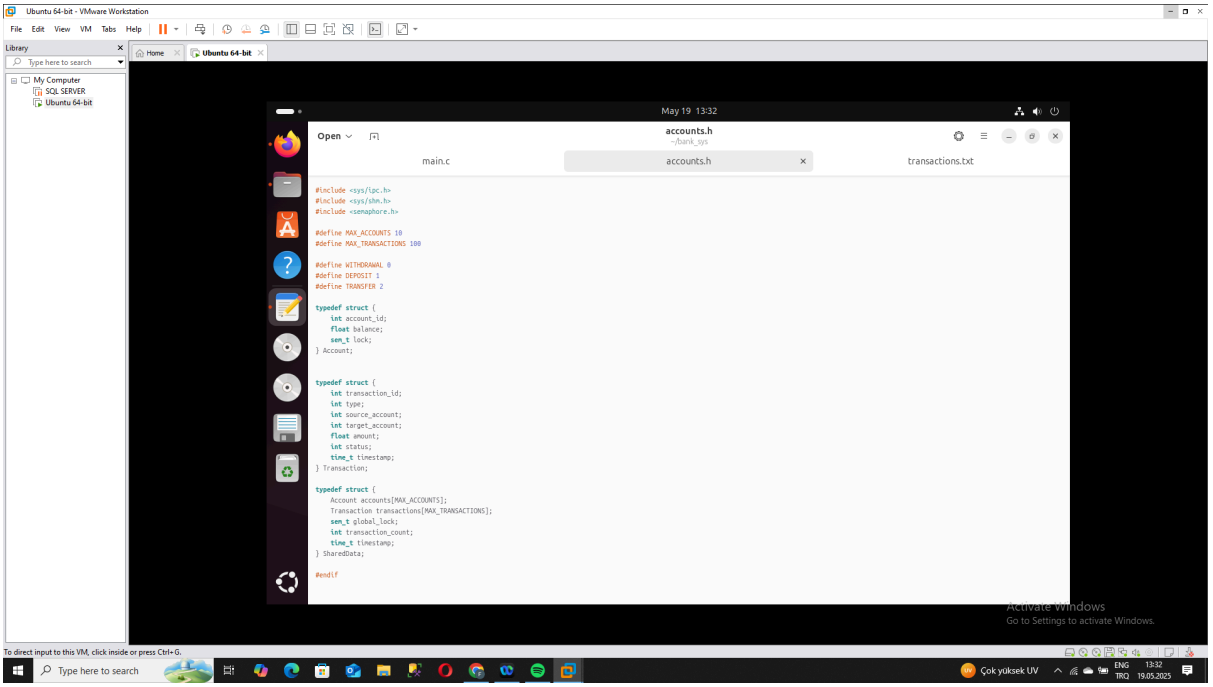
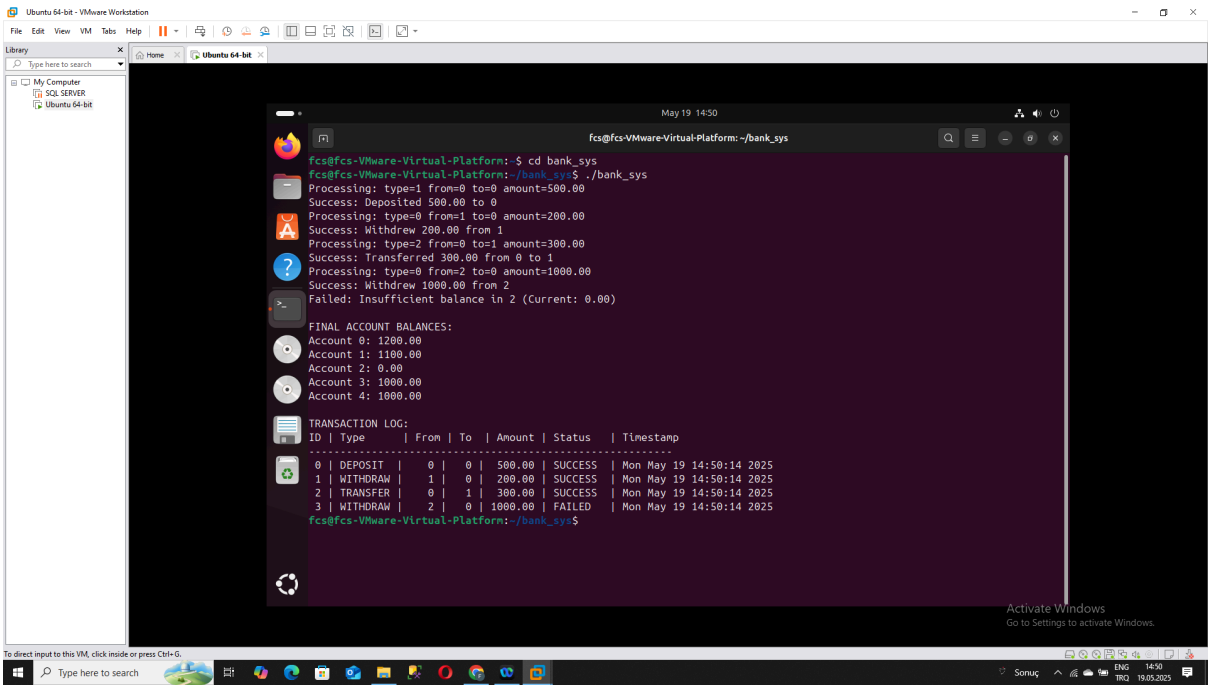


Figure 0.4 An image of account.h

Result of the study

And finally it gives an output like this



Summary

This concurrent banking system application securely manages multiple transaction types (deposits/withdrawals/transfers) using shared memory and semaphores for process synchronization. The program demonstrates robust parallel processing by creating separate child processes for each transaction while maintaining complete auditability through detailed transaction logging. Key features include:

1) Atomic incrementing of transaction IDs using semaphore-protected counters,
2) Deadlock prevention through ordered locking (lowest account number first), and
3) Automatic retry mechanisms for failed transactions. The implementation includes

comprehensive error handling for file I/O and system calls, with built-in protection against memory overflow through MAX_TRANSACTIONS limits. Each transaction is timestamped for complete audit trails, while the code maintains thread safety throughout all critical sections. The program requires compilation with gcc using -lpthread and -lrt flags for POSIX thread and real-time library support. Designed for both production use and debugging, it provides clear end-user transaction reports while offering detailed debug messages for developers. The system handles edge cases including insufficient funds, malformed input, and maximum transaction limits while ensuring all shared resources are properly cleaned up upon termination.

Personal thought:

Generally speaking, the intended goal has largely been achieved; however, there are still some areas for improvement. For example, the idea of starting each bank account with 1000 TL is one of the points I have addressed in the attached document