Course:

Operating System (LAB)

Project Title:

Banking System Simulation

(Integrated File Management System and Process Scheduling)

Project Documentation

Submitted to: Ms. Naila Noreen

Presented by	• M.Qasim	Sp-2022/BSCS/048
	• Awais Ali	Sp-2022/Bscs/080
	• Arbaz khan	Sp-2022/Bscs/075
	• Fahad Mehmood	Sp-2022/Bscs/059
	• Osama Khalid	058

Date	June 30, 2024	**Day**	Sunday

Banking System Simulation

(Integrated File Management System and Process Scheduling)

Introduction

This banking simulation system manages customer accounts and transactions using file management and process scheduling algorithms. The system supports operations such as creating accounts, deleting accounts, viewing account details, depositing money, withdrawing money, and simulating a token system for handling customer transactions using various scheduling algorithms.

Key Components

1. ***Customer Structure:*** Stores customer details such as name, account number, amount, burst time, arrival time, priority, and transaction type.
2. ***File Management:*** Handles the creation, deletion, and updating of account details stored in files.
3. ***Process Scheduling:*** Implements different scheduling algorithms (FCFS, Priority, SJF, RR) to manage customer transactions.

Code

1. Header Files' Purpose

The undermentioned header files have been used in making our banking simulation efficient:

Header File	Description
<iostream>	Provides input and output stream classes (cin, cout, cerr).
<cstdlib>	Includes general-purpose functions such as system(), rand(), srand().
<fstream>	Provides file stream classes (ifstream, ofstream, fstream).
<sstream>	Provides string stream classes, used for converting strings to other types.
<string>	Provides the string class for handling string data.
<iomanip>	Provides facilities to manipulate input and output formats, such as precision.
<vector>	Provides the vector container class for dynamic arrays.
<ctime>	Provides functions to manipulate date and time information.
<queue>	Provides the queue container class for queue operations.
<chrono>	Provides facilities for measuring and manipulating time.
<thread>	Provides facilities for creating and managing threads.
<algorithm>	Provides algorithms for operations such as sorting.
<filesystem>	Provides facilities for performing file system operations (C++17 onwards).

2. Code Breakdown:

a. Module 1: Header File Module

```
#include <iostream>
#include <cstdlib>      // For system() and rand()
#include <fstream>
#include <sstream>
#include <string>
#include <iomanip>
#include <vector>
#include <ctime>        // For time()
#include <queue>
#include <chrono>       // For sleep_for
```

```
#include <thread>       // For sleep_for
#include <algorithm>    // For sort
#include <filesystem>   // For directory and file operations

namespace fs = std::filesystem;
using namespace std;
```

b. Module 2: Structure Definition Module

```
// Structure to store customer details
struct Customer {
    string name;
    string accountNumber;
    double amount;
    int burstTime = 0;
    int arrivalTime = 0;
    bool isPriority;
    string transactionType;
    int tokenNumber;
    int priority;
};
```

c. Module 3: Console Functions Module

```
// Function to clear the console screen
void clearConsole() {
#ifdef _WIN32
    system("cls"); // Clear console on Windows
#else
    system("clear"); // Clear console on Unix-based systems
#endif
}

// Function to display a header
void displayHeader(const string& title) {
    system("clear"); // Clear the console (Linux)
    cout << "\n";
    cout << "========================================\n";
    cout << setw(15) << left << title << "\n";
    cout << "========================================\n";
    cout << "\n\n";
}

// Function to simulate time delay
void simulateProcessing(int seconds) {
    for (int i = 0; i < seconds; ++i) {
        cout << ".";
        cout.flush();
        this_thread::sleep_for(chrono::seconds(1)); // Simulate time delay
    }
}
```

d. Module 4: Account Management Module

```
// Function to create a new account

void createAccount(const string& accountNumber, const string&
accountHolderName, const string& accountType) {
    displayHeader("Creating New Account");
    string command = "mkdir HBL" + accountNumber;
```

```
    int result = system(command.c_str());
    if (result != 0) {
        cerr << "Failed to create account " << accountNumber << endl;
    } else {
        cout << "Account " << accountNumber << " created successfully." <<
endl;

        // Create a file for storing account details
        ofstream file("HBL" + accountNumber + "/details.txt");
        if (file.is_open()) {
            file << "Account Number: " << accountNumber << "\n";
            file << "Account Holder Name: " << accountHolderName << "\n";
            file << "Account Type: " << accountType << "\n";
            file << "Balance: 0.00\n"; // Initial balance set to 0.00
            file.close();
            cout << "Account details saved successfully." << endl;
        } else {
            cerr << "Error creating file for account " << accountNumber <<
endl;
        }
    }
}

// Function to delete an account

void deleteAccount(const string& accountNumber) {
    displayHeader(“Deleting Account”);

    string accountDir = “HBL” + accountNumber;

    // Check if the account directory exists
    if (fs::exists(accountDir)) {
        try {
            fs::remove_all(accountDir); // Remove directory and its contents
            cout << “Account “ << accountNumber << “ deleted successfully.” <<
endl;
        } catch (const fs::filesystem_error& e) {
            cerr << “Error deleting account: “ << e.what() << endl;
        }
    } else {
        cerr << “Account “ << accountNumber << “ does not exist.” << endl;
    }
}

// Function to view account details

void viewAccountDetails(const string& accountNumber) {
    displayHeader("Viewing Account Details");

    string filename = "HBL" + accountNumber + "/details.txt";
    ifstream file(filename);

    if (file.is_open()) {
        string line;
        while (getline(file, line)) {
            cout << line << endl;
        }
        file.close();
    } else {
```

```
        cerr << "Error: Could not open file for account " << accountNumber <<
endl;
    }
}

// Function to update balance in file

bool updateBalance(const string& accountNumber, double amount, bool isDeposit)
{
    string filename = "HBL" + accountNumber + "/details.txt";
    ifstream inFile(filename); // Read mode
    ofstream outFile("temp.txt"); // Temporary file for writing updated
content
    stringstream content;
    string line;
    string balanceStr;
    double currentBalance = 0.0;
    bool success = false;

    // Read current balance and other details
    if (inFile.is_open()) {
        while (getline(inFile, line)) {
            if (line.find("Balance: ") != string::npos) {
                balanceStr = line.substr(line.find(": ") + 2);
                currentBalance = stod(balanceStr);
                break;
            } else {
                outFile << line << "\n"; // Copy other details as is
            }
        }
        inFile.close();

        // Check if balance was found
        if (balanceStr.empty()) {
            cerr << "Error: Balance not found in file " << filename << endl;
            return false;
        }
    } else {
        cerr << "Error reading file for account " << accountNumber << endl;
        return false;
    }

    // Update balance based on transaction type
    if (isDeposit) {
        currentBalance += amount;
    } else {
        if (currentBalance >= amount) {
            currentBalance -= amount;
        } else {
            cerr << "Insufficient balance. Transaction not completed." <<
endl;
            return false;
        }
    }

    // Write updated balance back to file
    outFile << "Balance: " << fixed << setprecision(2) << currentBalance <<
"\n";
    outFile.close();
```

```
    // Replace original file with temporary file
    if (remove(filename.c_str()) == 0) {
        if (rename("temp.txt", filename.c_str()) != 0) {
            cerr << "Error renaming file" << endl;
        } else {
            cout << (isDeposit ? "Deposit" : "Withdrawal") << " of $" <<
amount << " completed for account " << accountNumber << endl;
            cout << "Current Balance: $" << currentBalance << endl;
            success = true;
        }
    } else {
        cerr << "Error deleting file" << endl;
    }

    return success;
}

// Function to check if account exists
bool doesAccountExist(const string& accountNumber) {
    string filename = "HBL" + accountNumber + "/details.txt";
    ifstream file(filename);
    return file.good();
}
```

e. Module 5: Token System Module

```
// Function to implement the Token System with selected algorithm

void TokenSystem(vector<Customer>& customers, const string& algorithm) {
    displayHeader("Executing Token System");

    // Sort customers based on arrival time
    sort(customers.begin(), customers.end(), [](const Customer& a, const
Customer& b) {
        return a.arrivalTime < b.arrivalTime;
    });

    if (algorithm == "Priority") {
        // Sort customers based on priority
        sort(customers.begin(), customers.end(), [](const Customer& a, const
Customer& b) {
            return a.priority < b.priority;
        });
    } else if (algorithm == "SJF") {
        // Sort customers based on burst time (Shortest Job First)
        sort(customers.begin(), customers.end(), [](const Customer& a, const
Customer& b) {
            return a.burstTime < b.burstTime;
        });
    } else if (algorithm == "RR") {
         // Implement Round Robin (For simplicity, assuming a quantum of
2 units)
        int quantum = 20;
        queue<Customer> rrQueue;
        for (auto& customer : customers) {
            rrQueue.push(customer);
        }
```

```
        int time = 0;
        vector<Customer> tempCustomers;

        while (!rrQueue.empty()) {
            Customer customer = rrQueue.front();
            rrQueue.pop();

            if (customer.burstTime > quantum) {
                time += quantum;
                customer.burstTime -= quantum;
                rrQueue.push(customer);
            } else {
                time += customer.burstTime;
                customer.burstTime = 0;
                customer.arrivalTime = time; // Updating arrival time to
reflect completion
            }

            tempCustomers.push_back(customer);
        }

        customers = tempCustomers;
    }

    int time = 0;
    int totalTurnaroundTime = 0;
    int totalWaitingTime = 0;
    vector<string> gantChart;

    cout << "\tToken"
         << "\tName"
         << "\tAccount Number"
         << "\t\t\t\tAmount"
         << "\tBT"
         << "\tAT"
         << "\tCT"
         << "\tTAT"
         << "\tWT"
         << "\tP"
         << "\tTT" << endl;

    cout << "-------------------------------------------------------------------
-----------------------------------------\n";

    for (auto& customer : customers) {
        int completionTime = (time + customer.burstTime) * rand()%10;
        int turnaroundTime = completionTime - customer.arrivalTime;
        int waitingTime = turnaroundTime - customer.burstTime;
        totalTurnaroundTime += turnaroundTime;
        totalWaitingTime += waitingTime;

        gantChart.push_back(to_string(customer.tokenNumber));

        cout << "\t" << customer.tokenNumber
             << "\t" << customer.name
             << "\t" << customer.accountNumber
             << "\t\t\t" << customer.amount
             << "\t" << customer.burstTime
             << "\t" << customer.arrivalTime
```

```
             << "\t" << completionTime
             << "\t" << turnaroundTime
             << "\t" << waitingTime
             << "\t" << customer.priority
             << "\t" << customer.transactionType << endl;

        time = completionTime;
    }

    cout << "Average Turnaround Time: " <<
static_cast<double>(totalTurnaroundTime) / customers.size() << endl;
    cout << "Average Waiting Time: " << static_cast<double>(totalWaitingTime)
/ customers.size() << endl;

    cout << "Gantt Chart: ";
    for (const auto& token : gantChart) {
        cout << token << " ";
    }
    cout << endl;
}
```

f. Module 6: The Main Function Module

```
// Main function
int main() {
    vector<Customer> customers;

    while (true) {
        displayHeader("Banking System");

        // Display menu options
        cout << "1. Create Account\n";
        cout << "2. Delete Account\n";
        cout << "3. View Account Details\n";
        cout << "4. Perform Deposit/Withdrawal\n";
        cout << "5. Execute Token System\n";
        cout << "0. Exit\n";

        int choice;
        cout << "Enter your choice: ";
        cin >> choice;

        if (choice == 0) {
            break;
        }

        string accountNumber, accountHolderName, accountType, transactionType;
        double amount;
        int burstTime, priority, tokenNumber;

        switch (choice) {
            case 1:
                cout << "Enter Account Number: ";
                cin >> accountNumber;
                cout << "Enter Account Holder Name: ";
                cin >> accountHolderName;
                cout << "Enter Account Type: ";
                cin >> accountType;
                createAccount(accountNumber, accountHolderName, accountType);
                break;
```

```
            case 2:
                cout << "Enter Account Number: ";
                cin >> accountNumber;
                deleteAccount(accountNumber);
                break;
            case 3:
                cout << "Enter Account Number: ";
                cin >> accountNumber;
                viewAccountDetails(accountNumber);
                break;
            case 4:
                cout << "Enter Account Number: ";
                cin >> accountNumber;
                if (!doesAccountExist(accountNumber)) {
                    cout << "Account does not exist." << endl;
                    break;
                }
                cout << "Enter Transaction Type (Deposit/Withdrawal): ";
                cin >> transactionType;
                cout << "Enter Amount: ";
                cin >> amount;
                if (transactionType == "Deposit") {
                    updateBalance(accountNumber, amount, true);
                } else if (transactionType == "Withdrawal") {
                    updateBalance(accountNumber, amount, false);
                } else {
                    cout << "Invalid transaction type." << endl;
                }
                break;
            case 5:
                {
                    cout << "Enter number of customers: ";
                    int numCustomers;
                    cin >> numCustomers;

                    for (int i = 0; i < numCustomers; ++i) {
                        Customer customer;
                        cout << "Enter Customer Name: ";
                        cin >> customer.name;
                        cout << "Enter Account Number: ";
                        cin >> customer.accountNumber;
                        cout << "Enter Amount: ";
                        cin >> customer.amount;
                        cout << "Enter Burst Time: ";
                        cin >> customer.burstTime;
                        cout << "Enter Arrival Time: ";
                        cin >> customer.arrivalTime;
                        cout << "Enter Priority: ";
                        cin >> customer.priority;
                        cout << "Enter Transaction Type: ";
                        cin >> customer.transactionType;
                        customer.tokenNumber = i + 1;
                        customers.push_back(customer);
                    }

                    cout << "Choose Scheduling Algorithm (Priority/SJF/RR): ";
                    string algorithm;
                    cin >> algorithm;
                    TokenSystem(customers, algorithm);
```

```
                }
                break;
            default:
                cout << "Invalid choice. Please try again." << endl;
                break;
        }

        cout << "Press Enter to continue...";
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cin.get();
    }

    return 0;
}
```