ALLIANCE UNIVERSITY

Private University established in Karnataka State by Act No.34 of year 2010
Recognized by the University Grants Commission (UGC), New Delhi

Mini Project

Title of the Mini Project

Group-12

ATM Transaction

DA-A SECTION

SEM-5

A report submitted by--

Saya Rishi (2022BCSE07AED028)

Saya Preetham (2022BCSE07AED029)

Gadamsetty Deepthika (2022BCSE07AED110)

Sadineni Aashritha (2022BCSE07AED139)

Lagisetty Bhupala Vignesh(2022BCSE07AED283)

INDEX

1. INTRODUCTION:

In today's banking systems, Automated Teller Machines (ATMs) are essential for providing customers with convenient and secure access to their accounts, enabling various financial transactions like cash withdrawals, deposits, and balance inquiries. This project simulates an ATM transaction system, focusing on implementing key operating system concepts such as concurrency, synchronization, and file handling to ensure reliable and consistent performance during simultaneous operations.

This ATM simulation project leverages multithreading to handle concurrent transactions, which mimics the real-world scenario of multiple users accessing the ATM system simultaneously. By implementing synchronization mechanisms such as mutexes, the project ensures that shared resources, like the account balance, are accessed safely without conflicts. Additionally, file handling is employed to persist account data, allowing balance updates to be saved and retrieved across sessions, simulating a basic but crucial aspect of ATM systems—data retention.

This project demonstrates how an operating system manages resources and handles concurrent tasks efficiently. It serves as an educational tool for understanding fundamental OS concepts, especially in the context of finance-based applications.

2. OBJECTIVES

The main objective of this project is to develop a simulated ATM system that incorporates core operating system concepts, providing both functionality and reliability in a multi-user environment. Through this project, we aim to:

- **Implement Concurrency in Financial Transactions**: Design the ATM system to handle multiple types of transactions (such as withdrawals, deposits, and balance checks) concurrently. By utilizing multithreading, the project aims to simulate real-world scenarios where multiple users may access the system simultaneously, allowing us to explore the management of concurrent processes.
- **Ensure Data Consistency through Synchronization**: Utilize synchronization techniques, specifically mutex locks, to prevent race conditions and ensure data consistency during concurrent transactions. This objective demonstrates how operating systems handle shared resources safely, particularly in situations where multiple threads interact with shared data, such as an account balance.
- **Demonstrate File Handling for Persistent Data Storage**: Implement file operations to simulate a basic database that stores the account balance. This enables the ATM

system to retrieve and update account information across multiple sessions, emulating persistent storage, a critical component in real-world ATM systems for data reliability.

- **Explore Error Handling in Concurrent Environments**: Develop error-handling mechanisms to address potential issues such as insufficient funds during withdrawals. The objective is to ensure that the system remains stable and provides clear feedback to users, reinforcing the importance of managing errors in concurrent environments.
- **Enhance Understanding of System Calls and Resource Management**: Utilize system calls for thread management and file handling, giving a practical demonstration of how operating systems facilitate resource allocation and I/O operations. By working directly with these system-level operations, this project aims to provide insight into the critical role of the OS in managing processes and resources.
- **Develop a Foundation for Real-World ATM Systems**: By simulating basic ATM operations with OS principles, this project serves as a foundational step toward understanding the complexities of real-world ATM and banking systems, where efficient and reliable resource management is essential.

3. PROJECT BACKGROUND:

3.1 DESCRIPTION

The primary data in this project is the account balance, representing the user's available funds. This balance is a dynamic value, modified based on the type of transaction (deposit, withdrawal, or balance check) performed by the user. The following components describe the data used in this ATM system:

- **Account Balance**: An integer value representing the user's current funds. It is initialized with a default value if no prior data exists.
- **Transaction Amount**: An integer entered by the user to specify the amount to deposit or withdraw.
- **Transaction History (Optional)**: If included, this would record past transactions to provide insights into user activity. In this project, however, only the current balance is saved and managed.

Since the ATM system uses multithreading, the account balance data requires synchronization to ensure consistent and correct updates across concurrent transactions.

3.2 SOURCE

The account balance data is stored and retrieved from a local file (account.txt) within the project directory. This file acts as a simple, persistent storage solution for the balance, ensuring that data is preserved across sessions. On program initialization, the balance is loaded from this file or set to a default value if no file exists. After each deposit or withdrawal, the updated balance is saved back to the file. This approach simulates a basic database and highlights essential data handling within an operating system, specifically demonstrating file I/O operations in a concurrent environment.

4. IMPLEMENTION

Here we are using some User-define functions like

void load_balance()

void save_balance()

void *check_balance()

void *deposit()

void *withdraw()

etc…

<u>Code:</u>

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

#define FILENAME "account.txt"
pthread_mutex_t mutex; // Mutex for synchronizing balance access
typedef struct {
int balance;
} Account;

// Function to load the account balance from the file
 void load_balance(Account *account) {
FILE *file = fopen(FILENAME, "r");
if (file == NULL) {
   printf("File not found, initializing balance to 5000\n");
   account->balance = 5000; // Default balance if file doesn't exist
} else {
        fscanf(file, "%d", &account->balance);
        fclose(file);
    }
            }
      // Function to save the account balance to the file
     void save_balance(Account *account) {
     FILE *file = fopen(FILENAME, "w");
     if (file == NULL) {
        perror("Error opening file");
        return;
     }
     fprintf(file, "%d", account->balance);
     fclose(file);
            }

    // Function to check the balance
     void *check_balance(void *arg) {
     Account *account = (Account *)arg;
     pthread_mutex_lock(&mutex);
     printf("Your current balance is %d\n", account->balance);
     pthread_mutex_unlock(&mutex);
     return NULL;
            }

    // Function to deposit money
     void *deposit(void *arg) {
     Account *account = (Account *)arg;
     int amount;
     printf("Enter the amount to deposit: ");
     scanf("%d", &amount);
```

```
pthread_mutex_lock(&mutex);
account->balance += amount;
save_balance(account);
printf("Deposit successful! Your current balance is %d\n", account->balance);
pthread_mutex_unlock(&mutex);
return NULL;
        }

// Function to withdraw money
void *withdraw(void *arg) {
Account *account = (Account *)arg;
int amount;
printf("Enter the amount to withdraw: ");
scanf("%d", &amount);

pthread_mutex_lock(&mutex);
if (amount > account->balance) {
    printf("Insufficient funds!\n");
} else {
    account->balance -= amount;
    save_balance(account);
    printf("Withdrawal successful! Your current balance is %d\n", account->balance);
}
pthread_mutex_unlock(&mutex);
return NULL;
        }

int main() {
Account account;
```

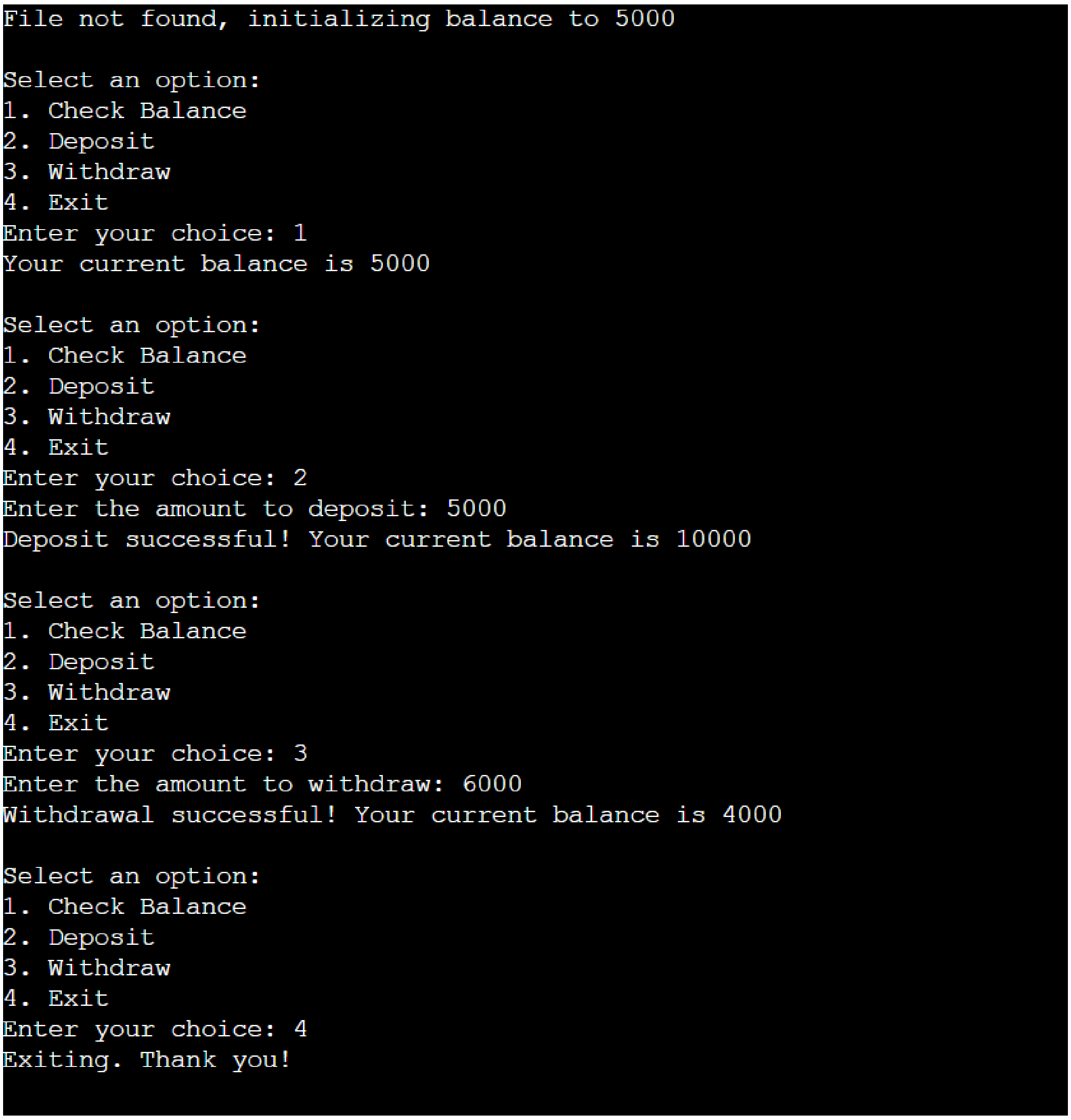
```
pthread_mutex_init(&mutex, NULL);
load_balance(&account);

    int choice;
    while (1) {
        printf("\nSelect an option:\n");
        printf("1. Check Balance\n");
        printf("2. Deposit\n");
        printf("3. Withdraw\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

    pthread_t tid;
    switch (choice) {
        case 1:
            pthread_create(&tid, NULL, check_balance, &account);
            pthread_join(tid, NULL);
            break;
        case 2:
            pthread_create(&tid, NULL, deposit, &account);
            pthread_join(tid, NULL);
            break;
        case 3:
            pthread_create(&tid, NULL, withdraw, &account);
            pthread_join(tid, NULL);
            break;
        case 4:
            printf("Exiting. Thank you!\n");
            pthread_mutex_destroy(&mutex);
    return 0;
default:
    printf("Invalid option. Try again.\n");
        }
    }

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

OUTPUT:



```
File not found, initializing balance to 5000

Select an option:
1. Check Balance
2. Deposit
3. Withdraw
4. Exit
Enter your choice: 1
Your current balance is 5000

Select an option:
1. Check Balance
2. Deposit
3. Withdraw
4. Exit
Enter your choice: 2
Enter the amount to deposit: 5000
Deposit successful! Your current balance is 10000

Select an option:
1. Check Balance
2. Deposit
3. Withdraw
4. Exit
Enter your choice: 3
Enter the amount to withdraw: 6000
Withdrawal successful! Your current balance is 4000

Select an option:
1. Check Balance
2. Deposit
3. Withdraw
4. Exit
Enter your choice: 4
Exiting. Thank you!
```

5. DISCUSSIONS AND RECOMMENDATIONS:

5.1 DISCUSSIONS:

This project successfully demonstrates the use of core operating system concepts within a simulated ATM environment. By implementing **multithreading** for concurrent transaction handling, we simulate how real-world ATMs allow multiple users to perform transactions simultaneously, ensuring efficient use of resources. **Synchronization** through mutex locks ensures that critical sections of the code, particularly those involving the shared `balance` variable, are accessed safely by each thread, preventing inconsistencies and race conditions.The use of **file handling** to store and retrieve the account balance demonstrates simple data persistence, maintaining account information across sessions and mimicking basic database functionality.

One challenge encountered was handling concurrent access to shared resources, which could lead to data inconsistency if not properly synchronized. Using mutexes resolved this issue, highlighting the importance of managing shared resources in concurrent environments. Additionally, incorporating error handling for conditions such as insufficient funds provides a better user experience and adds to the system's robustness, showing how user input validation can enhance security and reliability in financial applications

5.2 RECOMMENDATIONS:

Enhanced Data Persistence: A database system, such as SQLite, could replace the basic file-based approach for storing account data. This would provide more robust data management capabilities and allow for additional features, such as storing transaction history and supporting multiple user accounts.

Improved Security Measures: Adding security features, such as user authentication, would increase the realism and usability of the ATM system. This could involve implementing a PIN verification system or password encryption to prevent unauthorized access to accounts.

Addition of Transaction History: Tracking each transaction (deposits, withdrawals) would provide users with a more comprehensive account management experience. This feature could be implemented by recording transactions in a file or database, allowing users to view their recent activity.

Network-Based ATM Simulation: To extend this project further, the ATM system could be adapted to a client-server model where multiple ATMs (clients) connect to a central banking server. This would better represent real-world banking systems and explore OS concepts related to networking and client-server communication.

Testing and Optimization for Scalability: Testing the system's performance under heavy load (simulating multiple users accessing the ATM concurrently) would help identify areas for optimization. This could include refining thread management and improving response times, which are critical for a smooth user experience in high-traffic environments.

6. CONCLUSION:

This ATM transaction simulation project has successfully demonstrated the integration of fundamental operating system concepts, including concurrency, synchronization, and file handling, within a financial application. By employing multithreading, we simulated a real-world scenario where multiple users can perform transactions simultaneously, showcasing how operating systems handle concurrent processes and resource management.

The use of mutex locks to ensure synchronized access to the shared account balance highlights the importance of preventing race conditions and maintaining data consistency in multi-threaded applications. Additionally, the implementation of file handling for data persistence provides insight into basic file I/O operations that ensure critical data is retained across sessions, which is essential in any financial system.

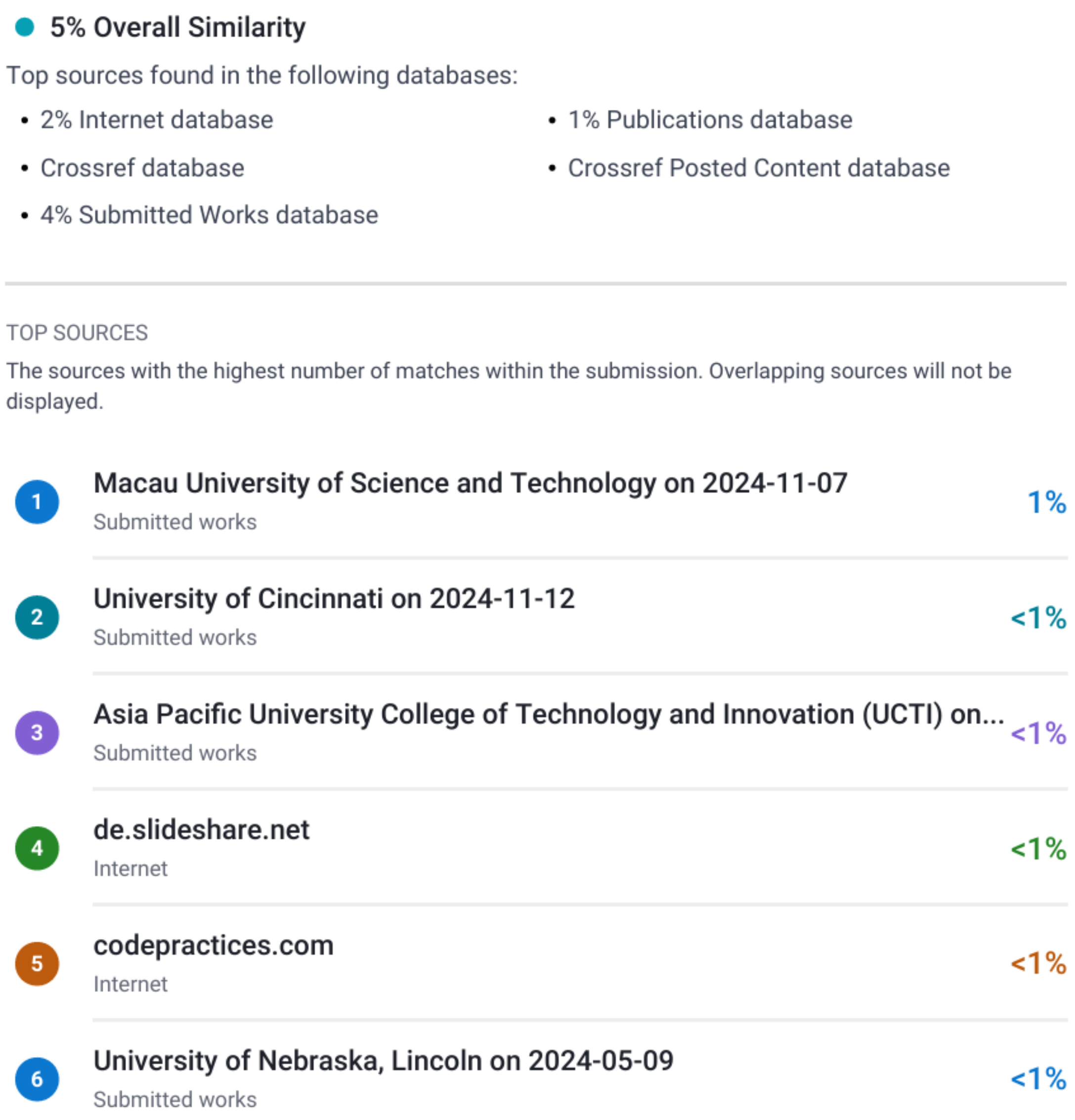
Through this project, we gained practical experience in managing shared resources and handling concurrent tasks safely, as well as an understanding of error handling to ensure system stability. While the project achieved its objectives, it also opened up possibilities for further enhancement, such as adding user authentication and transaction history, which would make the system more robust and realistic.

In summary, this project serves as a valuable learning tool in applying OS principles to real-world applications, demonstrating how effective resource management and synchronization are critical in the development of reliable and secure software systems.

7. REFERENCES:

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.
- Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.
- Stallings, W. (2018). Operating Systems: Internals and Design Principles (9th ed.). Pearson.
- IBM Knowledge Center. (n.d.). POSIX Threads Programming. https://www.ibm.com/docs/en
- GNU C Library Documentation. (n.d.). File System Interface – GNU C Library. https://www.gnu.org/software/libc/manual/
- Cprogramming.com. (n.d.). File I/O in C. https://www.cprogramming.com/tutorial/cfileio.html

8. PLAGIARISM REPORT:



● **5% Overall Similarity**

Top sources found in the following databases:

- 2% Internet database
- 1% Publications database
- Crossref database
- Crossref Posted Content database
- 4% Submitted Works database

TOP SOURCES

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

1	**Macau University of Science and Technology on 2024-11-07** Submitted works	1%
2	**University of Cincinnati on 2024-11-12** Submitted works	<1%
3	**Asia Pacific University College of Technology and Innovation (UCTI) on...** Submitted works	<1%
4	**de.slideshare.net** Internet	<1%
5	**codepractices.com** Internet	<1%
6	**University of Nebraska, Lincoln on 2024-05-09** Submitted works	<1%