



MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

**Mini Project Report
of
Operating Systems Lab (CSE 3163)**

**Title: MULTITHREADED BANKING
APPLICATION**

**SUBMITTED
BY**

**PODICHETI MOUKTHIK
PILLARI SAI VENKATA AASHISH
S. HEMANTH BHARATH
GADDE SREE LAKSHMI NARASIMHA NAIDU**

**Reg.no: 210905296 Roll no: 46
Reg.no: 210905212 Roll no: 37
Reg.no: 210905178 Roll no: 31
Reg.no: 210905304 Roll no: 48**

**Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal, Karnataka
NOV 2023**



MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Manipal
12/11/2023

CERTIFICATE

This is to certify that the project titled **Multithreaded Banking Application** is a record of the bonafide work done by **PODICHETI MOUKTHIK (210905296)**, **PILLARI SAI VENKATA AASHISH (210905212)**, **S. HEMANTH BHARATH (210905178)**, and **GADDE SREE LAKSHMI NARASIMHA NAIDU (210905304)** submitted in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology (B.Tech.) in **COMPUTER SCIENCE & ENGINEERING** of Manipal Institute of Technology, Manipal, Karnataka, (A Constituent Institute of Manipal Academy of Higher Education), during the academic year 2022-2023.

Name and Signature of Examiners:

1. **Mrs. Archana Praveen Kumar, Assistant Professor - Senior Scale, CSE Dept.**

ACKNOWLEDGMENTS

We want to express our sincere gratitude to our instructor Mrs. Archana Praveen Kumar at MANIPAL INSTITUTE OF TECHNOLOGY, whose guidance and teaching have been instrumental in successfully completing this project. The foundational knowledge and concepts related to Multithreading, which form the core of this project, were imparted to us during our coursework. We greatly appreciate the expertise, support, and valuable insights provided by our educators, which have enabled us to understand and implement the Multithreaded Banking Application effectively. Their dedication to teaching and commitment to our learning journey have been invaluable, and we acknowledge their contributions with deep appreciation.

ABSTRACT

The Multithreaded Banking Application is a sophisticated software solution engineered to elevate the efficiency and responsiveness of banking operations through the strategic implementation of multithreading. This application is meticulously designed to enable concurrent user interactions, allowing seamless execution of diverse financial transactions such as deposits, withdrawals, and fund transfers. Leveraging advanced thread synchronization and locking mechanisms ensures the integrity of shared data, while robust user authentication and authorization protocols secure sensitive information. The system empowers users to create, modify, and delete accounts concurrently, enhancing overall user experience and minimizing response times. Comprehensive transaction logging and auditing mechanisms contribute to transparency and traceability, facilitating error resolution and reconciliation. Built with fault tolerance and error-handling capabilities, the application guarantees a resilient performance even in unexpected scenarios. Optimized resource utilization ensures efficient handling of concurrent users, making the application scalable to accommodate growing demands. In summary, the Multithreaded Banking Application offers a reliable and secure platform for banking operations, utilizing multithreading to concurrently process transactions, enhance user experience, and maintain the integrity of financial data.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

CHAPTER 2: PROBLEM STATEMENT

CHAPTER 3: OBJECTIVES

CHAPTER 4: METHODOLOGY

CHAPTER 5: RESULTS & SNAPSHOTS

CHAPTER 6: LIMITATIONS

CHAPTER 7: FUTURE WORK

CHAPTER 8: CONCLUSION

CHAPTER 9: REFERENCES

INTRODUCTION

Our solution is an application for handling multiple accounts in a bank. In this project, we tried to show the working of a banking account system and cover the basic functionality of banking, particularly with a focus on achieving multi-client concurrency. This project is developed using C language.

Our project makes use of multiple important operating systems concepts which include but are not limited to multithreading, process synchronization, deadlock handling, mutex locks, concurrent client-server interactions, etc.

The core of our project stems from the fundamental idea of multithreading. Most attempts at a history of multithreaded processors start with the PPU's in the CDC 6600. However, the idea of sharing a single execution path among multiple threads of execution started at least a decade earlier. After its initial appearance in the 50s, and the subsequent investigation of IBM on the implementation of Simultaneous **Multithreading** (1968), multithreading has proven to be a powerful tool to handle multiple queries simultaneously from different clients.

A significant issue faced by many real-world projects is that of different processes working parallelly without being synchronized appropriately. This can lead to major problems including but not limited to Deadlocks, inconsistency in data, the incorrect flow of data and instructions, etc. To combat the possible errors caused due to process synchronization we have utilized **semaphores, mutex locks**, etc.

Our prototype CLI application has the capacity to handle up to 20 concurrent clients in one session, the capacity can be ramped up as and when required. The code makes heavy use of threads in various aspects ranging from accepting sessions, handling client inputs, opening, and closing connections, etc.

The software has multiple functionalities, ranging from creating a user account during a session, possibly logging out and logging in, depositing and withdrawing money from an account to transferring money into another account. All these functionalities require background checks which range from checking the validity of customers, checking whether a given transaction can be processed, handling concurrent transactions, etc.

PROBLEM STATEMENT

In the realm of modern banking applications, the implementation of a multithreaded system introduces challenges related to concurrent transaction processing, data consistency, and system integrity. The efficient management of resources, synchronization of operations, and prevention of potential deadlocks are critical concerns in ensuring the seamless functioning of such applications. This project aims to develop a multithreaded banking application utilizing advanced operating system principles to address these challenges. The focus will be on creating a robust system that can handle concurrent transactions, maintain data integrity, and employ effective synchronization mechanisms to prevent deadlocks. The successful completion of this project will contribute to the advancement of secure and high-performance multithreaded banking applications, offering a practical solution for real-world financial systems operating in complex and dynamic environments.

OBJECTIVES

1. The primary objective is to introduce an application designed for managing multiple accounts in a bank. The text describes the nature of this application.
2. Demonstrate the functioning of a banking account system while focusing on the essential functionalities of banking.
3. To point out the significant issue of different processes working in parallel without proper synchronization and how this can lead to problems such as deadlocks, data inconsistency, and incorrect flow of data and instructions.
4. The use of synchronization mechanisms such as semaphores and mutex locks to prevent errors due to process synchronization and the core idea of multithreading.

METHODOLOGY

1) Client API:

[NARASIMHA]

Client-Side:

The client-side code of the Multithreaded Banking Application can be summarized as follows:

1. **Socket Connection:**
 - Establishes a socket connection to the server using the socket and connect functions. The client connects to the server with specified IP and port.
2. **User Interface Functions:**
 - Provides user interface functions for input handling (input_string), pressing enter to continue (press_enter), and clearing the screen (clear_screen).
3. **Bank Menu and Operations:**
 - Presents a bank menu (bank_menu) with options to create an account, log in, or exit. Uses functions like create_account, login, access_account, withdraw, deposit, and transfer to perform various banking operations.
4. **Communication with Server:**
 - Utilizes the send_and_recv function for sending user input to the server and receiving server responses. The client communicates with the server by writing to and reading from the socket.
5. **Socket Error Handling:**
 - Checks for errors during socket creation and connection. If errors occur, the client displays an appropriate error message and terminates.
6. **User Interaction and Session Handling:**
 - Prompts the user for account-related actions, such as creating an account, logging in, and performing transactions. Manages the user session by handling various banking operations.
7. **Credit and Disconnection Screens:**
 - Displays credit and disconnection screens using the credit_screen and disconnection functions, providing a visually appealing interface.
8. **Input Parsing and Validation:**
 - Parses user input, especially for transaction amounts, and validates it to ensure correctness. Invalid inputs trigger error messages.
9. **Graceful Exit:**
 - Implements a graceful exit mechanism by sending a "quit" command to the server when the user chooses to exit. Closes the socket before terminating the client.

In summary, the client-side code establishes a socket connection to the server, provides a user-friendly interface for interacting with the banking system, communicates with the server for various operations, handles errors, and ensures a graceful exit. It encapsulates the client-server interaction logic and offers a structured interface for users to interact with the banking application.

2) Server API:

[Hemanth Bharath]

Server-Side:

The server-side code of the Multithreaded Banking Application showcases key features:

1. Multithreading:
 - Efficiently manages multiple client connections using pthreads. The `session_acceptor_thread` function spawns threads for each client session, allowing concurrent handling.
2. Socket Communication:
 - Utilizes sockets for communication. Establishes a server socket and accepts incoming client connections using `socket`, `bind`, `listen`, and `accept` functions.
3. Account Management:
 - Maintains an array of Account structures for individual bank accounts. Thread-safe functions (`create_account`, `search_account`, `serve_account`) ensure controlled access to account information using mutex locks.
4. Transaction Handling:
 - Supports various banking operations, including deposit, withdrawal, fund transfer, query, and logout. Mutex locks ensure thread safety during transaction execution.
5. Periodic Actions and Semaphores:
 - Implements a periodic action cycle using semaphores. The `periodic_action_cycle_thread` function waits for semaphore signals, allowing periodic actions like displaying information.
6. Error Handling and Cleanup:
 - Incorporates error handling for socket and pthread functions. Ensures proper resource cleanup, such as socket closure and mutex destruction.
7. Security Measures:
 - Uses mutex locks to guarantee thread safety, preventing data inconsistencies during concurrent access.

In summary, the server-side code is designed for concurrency, handles client connections, manages bank accounts securely, supports various transactions, and incorporates error handling for robust execution.

3) Periodic Signal Raising Code and Server-Side Processing:

[Aashish]

Periodic Signal Raising Code: The code incorporates a periodic signal-raising mechanism to trigger certain actions at fixed intervals. Specifically, it uses the `SIGALRM` signal and a signal handler to implement a timer-based periodic action cycle. Here's an overview of the relevant components:

1. Signal Handling:
 - The `periodic_action_handler` function serves as the signal handler for the `SIGALRM` signal. This function is invoked at regular intervals, as defined by the timer.

- The sigaction structure is initialized to specify the behavior of the signal handler, indicating that it should be called with SIGALRM.
2. Timer Configuration:
 - The it_interval and it_value fields of the itimerval structure are set to determine the interval between successive signals and the initial time until the first signal, respectively.
 - The setitimer function is then called with ITIMER_REAL to establish a real-time timer that decrements in real time.
 3. Periodic Action Cycle Thread:
 - The periodic_action_cycle_thread function represents the thread responsible for the periodic action cycle. It repeatedly waits for the actionLock semaphore to be posted, which indicates that the timer has expired.
 - Once the semaphore is posted, the thread acquires the bankMutex and executes the periodic actions, including printing information about active connections and accounts.
 - This mechanism ensures that the periodic actions are performed at regular intervals, promoting synchronization and providing a structured approach to handling time-dependent server-side tasks.

4) Structures and Restriction to Access to Struct Through Mutex Locks and Semaphores:

[Moukthik]

Structures: The code utilizes a structure named Account to represent individual bank accounts. This structure encapsulates relevant information about an account, including its name (accountName), usage status (isInUse), and balance (balance). This structured approach allows for the organization of account-related data, facilitating readability, and maintainability. It promotes a clean separation of concerns, enhancing the overall design of the banking application.

Restriction to Access to Struct Through Mutex Locks and Semaphores: The implementation enforces thread safety and controlled access to shared resources, particularly the array of Account structures and the connectionCount variable, using mutex locks and semaphores.

1. Mutex Locks:
 - Multiple mutex locks are used to control access to critical sections of the code. For instance, bankMutex is employed to synchronize access to the connectionCount variable, ensuring that the count is updated atomically and preventing race conditions.
 - customerMutex is an array of mutex locks, with each element corresponding to a specific Account structure. These locks ensure that operations on individual accounts, such as checking and updating balances, are thread-safe.
 - The code utilizes mutex locks to protect shared resources during operations like creating accounts, serving accounts, and performing transactions. This prevents data corruption and ensures the integrity of account-related information.
2. Semaphores:

- The `actionLock` semaphore is employed to coordinate periodic actions in the banking server. It allows controlled access to the periodic action cycle, ensuring that the cycle runs at intervals and avoids conflicts between threads.
- The `sem_post(&actionLock)` call is used to signal the availability of a periodic action, allowing the periodic action cycle thread to execute. This mechanism ensures that the server periodically displays information about active connections and accounts.
- Semaphores provide a means of synchronization between threads, preventing them from accessing shared resources concurrently and ensuring orderly execution of critical sections.

By employing mutex locks and semaphores, the code establishes a robust synchronization mechanism, safeguarding shared data structures and preventing potential issues such as data inconsistency, deadlocks, and race conditions in the multithreaded environment of the banking application.

RESULTS AND SNAPSHOTS

The program successfully demonstrated the Banker's Algorithm for resource allocation and deadlock avoidance. The provided inputs were processed, and the program output indicated that the system was in a safe state, ensuring that all processes could be executed without encountering any deadlocks.

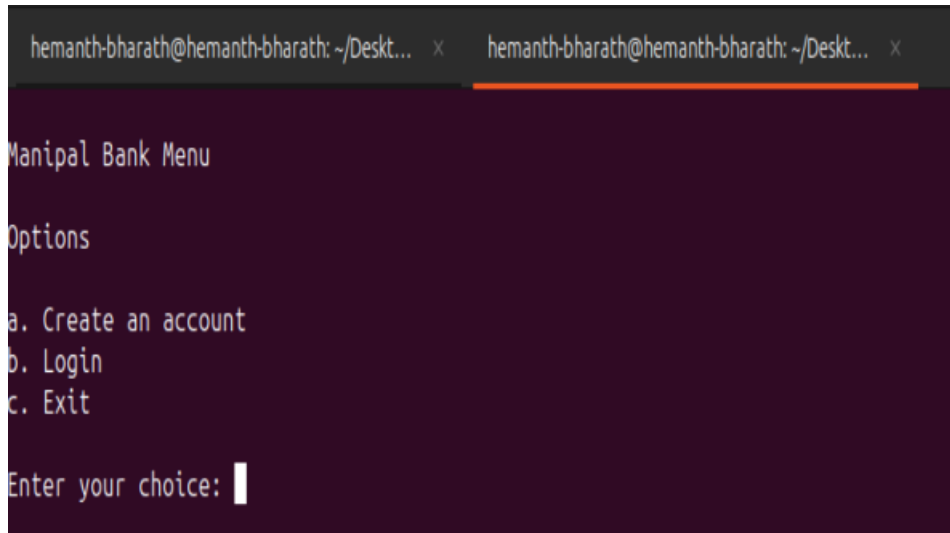
The algorithm efficiently allocated resources to processes while preventing the occurrence of deadlocks, thereby demonstrating its effectiveness in managing resources within a computer system.

1) COMPILATION AND EXECUTION

```
hemanth-bharath@hemanth-bharath:~/Desktop/os mini proj$ gcc -o server server.c
hemanth-bharath@hemanth-bharath:~/Desktop/os mini proj$ ./server
server is ready to receive client connections ...
New Client has connected!
There is 1 active connection.
Customers:
There is 1 active connection.
```

```
hemanth-bharath@hemanth-bharath:~/Desktop/os mini proj$ ./client
Attempting to connect to banking server...
Successfully connected to banking server.
```

2) **Client menu program as soon as the menu starts**



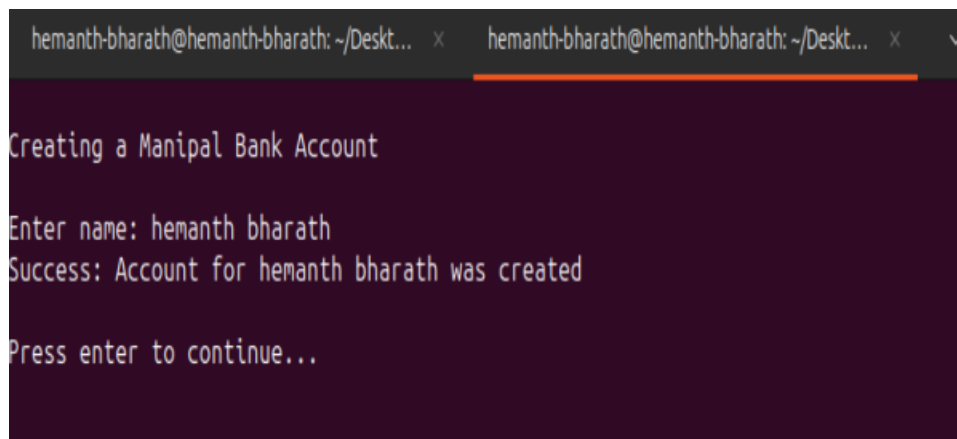
```
hemanth-bharath@hemanth-bharath: ~/Desk... x hemanth-bharath@hemanth-bharath: ~/Desk... x v
Manipal Bank Menu

Options

a. Create an account
b. Login
c. Exit

Enter your choice: |
```

3) **Creating an Account Successfully:**



```
hemanth-bharath@hemanth-bharath: ~/Desk... x hemanth-bharath@hemanth-bharath: ~/Desk... x v
Creating a Manipal Bank Account

Enter name: hemanth bharath
Success: Account for hemanth bharath was created

Press enter to continue...
```

4) **Login after successful creation of account**

```
Manipal Bank Menu

Options

a. Create an account
b. Login
c. Exit

Enter your choice: b
```

```
Manipal Bank Account Login

Enter name: hemanth bharath
```

5)Login and deposit amount functionality:

```
Accessing Manipal Bank Account

Full Name:      hemanth bharath
Balance in Rupees:  0.00

Options

a. Withdraw Money
b. Deposit Money
c. Transfer Money
d. Log Out

Enter your choice: b
Enter amount to deposit: 500

Success: Deposited Rs500.00 into hemanth bharath's account

Press enter to continue...
```

6)withdraw amount functionality:

```
Enter your choice: a
Enter amount to withdraw: 600

Error: Cannot withdraw Rs600.00 from hemanth bharath's account due to insufficient balance
```

7) Meanwhile , lets have a look on server logs on console:

```
Server received input: withdraw 600
Error: Cannot withdraw Rs600.00 from hemanth bharath's account due to insufficient balance
There is 1 active connection.
Customers:
hemanth bharath
```

8) Creating multiple users:

```
Creating a Manipal Bank Account

Enter name: moukthik
Success: Account for moukthik was created

Press enter to continue...
```

8) Transfer between accounts:

```
Creating a Manipal Bank Account

Enter name: moukthik
Success: Account for moukthik was created

Press enter to continue...
```


Server logs:

```
Creating a Manipal Bank Account  
Enter name: moukthik  
Success: Account for moukthik was created  
Press enter to continue...
```

9) Logout from account:

```
d. Log Out  
Enter your choice: d  
Success: Logged out from hemanth bharath's account
```

Server logs:

```
server received input: end  
Success: Logged out from hemanth bharath's account  
There is 1 active connection.  
Customers:  
hemanth bharath  
moukthik
```

LIMITATIONS

While the provided multithreaded banking application demonstrates a functional implementation, there are some inherent limitations and considerations:

1. Limited Scalability:

- The application might face scalability issues when the number of clients or transactions increases significantly. As the number of concurrent connections grows, the system could experience performance bottlenecks.

2. Security Concerns:

- The code lacks explicit security measures such as encryption for data transmitted over the network. Implementing secure communication (e.g., using TLS/SSL) is crucial to protect sensitive user data and prevent unauthorized access.

3. Error Handling and Robustness:

- The error handling in the code is relatively basic. In a production environment, more comprehensive error handling mechanisms should be implemented to handle various scenarios, such as network failures, unexpected user inputs, or server crashes.

4. Limited Transactional Consistency:

- The application does not implement a full-fledged transactional system. In a real banking system, ensuring transactional consistency (atomicity, consistency, isolation, and durability - ACID properties) is critical to prevent data inconsistencies and ensure the integrity of financial transactions.

5. Synchronization Overhead:

- The use of mutex locks and semaphores introduces synchronization overhead. While necessary for ensuring thread safety, excessive use of locks can lead to performance degradation, especially in scenarios with high contention for shared resources.

6. No Logging Mechanism:

- The application lacks a comprehensive logging mechanism for tracking user activities, system events, and potential errors. Logging is crucial for auditing, debugging, and monitoring the application in a real-world setting.

7. Single-Point of Failure:

- The current design has a single server, creating a potential single point of failure. In a production environment, a distributed and redundant server infrastructure should be considered to enhance system reliability.

8. No User Authentication:

- The application does not include a robust user authentication mechanism. In a real banking system, user authentication is critical for ensuring that only authorized users can access and manipulate their accounts.

9. Limited User Feedback:

- The user interface lacks detailed feedback to users about the status of their operations. Providing more informative messages and handling edge cases gracefully can improve the user experience.

It's important to note that addressing these limitations requires careful consideration of the specific requirements, security standards, and performance expectations of a real-world banking application. Additionally, compliance with industry regulations (e.g., PCI-DSS for payment systems) should be considered for a production-grade system.

FUTURE WORK

Expanding and improving the existing multithreaded banking application can involve various areas of future work. Here are some suggestions for enhancing and extending the project:

1. Database Integration:

- Implement a database system (e.g., MySQL, PostgreSQL) to store account information persistently. This allows for better data management, retrieval, and ensures data consistency across server restarts.

2. User Authentication and Authorization:

- Implement a robust user authentication system to verify the identity of clients. This can involve username/password validation or more advanced methods like multi-factor authentication. Additionally, enforce access control mechanisms to restrict unauthorized access.

3. Transaction Logging:

- Develop a comprehensive logging system to record user transactions, system events, and potential errors. Logging is crucial for auditing, debugging, and monitoring the application's behavior.

4. Error Handling and Recovery:

- Enhance error handling mechanisms to gracefully manage unexpected scenarios, such as network failures or server crashes. Implement recovery mechanisms to handle errors without compromising data integrity.

5. Scalability Improvements:

- Optimize the application for better scalability by introducing load balancing and distributed computing techniques. This ensures that the system can handle an increasing number of clients and transactions efficiently.

6. Comprehensive Documentation:

- Create thorough documentation for developers, system administrators, and end-users. This includes documentation for installation, configuration, usage, and maintenance.

CONCLUSION

We have created a robust system using **Threads** and **Mutex locks** in our Linux-based multi-threaded banking system. Since the threads are interconnected, our menu-driven application makes it simple to know some crucial facts, such as transaction and account details. In our project, thread synchronization is effectively managed using Mutex locks to ensure that two or more threads do not execute important sections concurrently. We used multithreading to decrease banking complexity through thread parallelism. Our project will walk you through financial processes including opening accounts, withdrawing money, transferring money to different accounts, etc. with the use of straightforward solutions that make it easy to resolve any pressing banking concerns. The evaluated code produced satisfactory results in a variety of real-life banking circumstances.

REFERENCES

- <https://www.geeksforgeeks.org/introduction-of-process-synchronization/>
- <https://stackoverflow.com/questions/5791860/beginners-socket-programming-in-c>
- <https://web.cs.hacettepe.edu.tr/~abc/teaching/bbs646/slides/ch04.pdf>