GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

CSE344 Systems Programming
Spring 2025

Midterm Project Report

A Simple Bank Simulator

Osmancan Bozali
220104004011

1. Introduction

1.1 Objective

The objective of this project is to design and implement a banking simulator using a client-server architecture. The simulator aims to manage bank accounts through a main server process, handle client transactions (deposits and withdrawals) via dedicated Teller processes, and ensure data integrity and concurrency using inter-process communication (IPC) mechanisms such as named pipes (FIFOs), shared memory, and semaphores. The system must support multiple clients (up to 20) concurrently, maintain an up-to-date log file, and handle signals for graceful shutdown, providing a practical demonstration of process management, synchronization, and IPC in operating systems.

1.2 Scope

The scope of this project includes:

- **Server Implementation**: A main server process that manages the bank database, creates Teller processes for each client, and updates a log file.
- **Client Implementation**: Client processes that simulate bank customers submitting transaction requests (deposit or withdraw) via FIFOs.
- **Teller Processes**: Intermediary processes that handle client requests and communicate with the main server using shared memory and semaphores.
- **Synchronization and Communication**: Use of FIFOs for client-server communication, shared memory for Teller-server communication, and semaphores for synchronization to ensure data consistency.

This report focuses on the system architecture and implementation details, including the testing section with test plans and results.

2. System Architecture

The bank simulator employs a client-server architecture with a multi-process design to handle concurrent client requests efficiently. The system is structured to ensure that the main server process maintains exclusive access to the bank database, while Teller processes serve as intermediaries between clients and the server. Below is a detailed explanation of the architecture:

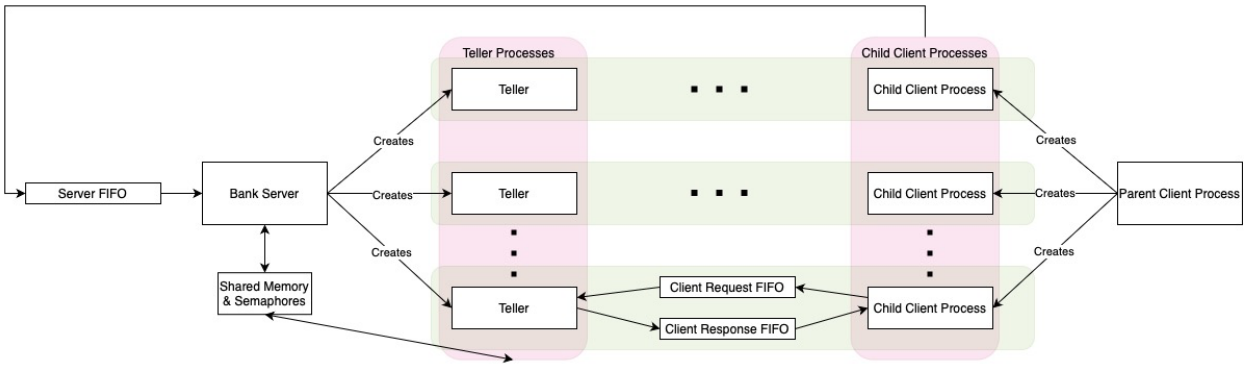


Figure: System Architecture

2.1 Components

2.1.1 Main Server Process (Bank)

- **Role**: Acts as the central authority managing the bank database and coordinating client requests.
- **Responsibilities**:
    - Initializes the bank database by parsing an existing log file or creating a new one.
    - Creates a server FIFO for receiving initial client requests.
    - Spawns Teller processes for each client request using a custom Teller function.

```
pid_t Teller(void* func, void* arg_func) {
    pid_t pid = fork();
    if (pid == 0) {
        void (*tellerFunc)(void*) = (void (*)(void*))func;
        tellerFunc(arg_func);
        exit(0);
    }
    return pid;
}
```

    - Waits teller proccesses using a custom Teller wait function.

```
void waitTeller(pid_t pid, int* status) {
    waitpid(pid, status, 0);
}
```

    - Processes transaction requests from Tellers via shared memory, updating accounts accordingly.

- Maintains a log file to persist account states.
- Handles signals (e.g., SIGINT, SIGTERM) for graceful shutdown, cleaning up resources like FIFOs, shared memory, and semaphores.

2.1.2 Teller Processes

- **Role**: Serve individual clients by processing their transaction requests and communicating with the main server.
- **Responsibilities**:
  - Receive initial client requests via client-specific FIFOs.
  - Use shared memory to request account creation, validation, deposits, or withdrawals from the main server.
  - Send responses back to clients through client-specific FIFOs.
  - Handle two types of transactions: deposits (including new account creation) and withdrawals. (see deposit() and withdraw() functions in source code.)

2.1.3 Client Processes

- **Role**: Simulate bank customers submitting transaction requests.
- **Responsibilities**:
  - Parse a client input file containing transaction details (e.g., new account deposits or withdrawals from existing accounts).
  - Create child processes for each transaction.
  - Each child process communicates with the server via the server FIFO (for initial request) and client-specific FIFOs (for transaction request/response).
  - Handle signals to ensure proper cleanup of resources upon interruption.

2.1.4 Communication Channels

- **Server FIFO**: A named pipe created by the server for clients to send initial transaction requests.
- **Client-Specific FIFOs**: Two FIFOs per client (request and response) for bidirectional communication with their assigned Teller.
- **Shared Memory**: A shared memory segment for Teller processes to communicate transaction requests and responses with the main server.
- **Semaphores**: Synchronization primitives to manage concurrent access to shared resources and coordinate Teller-server interactions.

2.2 Data Structures

- **BankAccount**: Represents a bank account with fields for ID, balance, activity status, transaction history, and client name.
- **InitialClientRequest**: Sent from clients to the server via the server FIFO, containing account ID, transaction type, client PID, and FIFO paths.
- **InitialResponse**: Sent from Tellers to clients, providing the client name and validated or assigned account ID.

- **TransactionRequest**: Sent from clients to Tellers, specifying the account ID and transaction amount.
- **TransactionResponse**: Sent from Tellers to clients, indicating the transaction outcome with an account ID and message.
- **SharedMemoryData**: Used in shared memory for Teller-server communication, including teller PID, transaction type, account ID, amount, success flag, message, and client name.

2.3 Synchronization Mechanisms

- **Semaphores**:
  - *requestSemaphore:* Signals the main server that a Teller has placed a request in shared memory.
  - *responseSemaphore:* Signals Tellers that the main server has processed a request and written a response.
  - *globalAccessSemaphore*: Protects the global account array and other shared resources from concurrent modifications.
  - *shmAccessSemaphore:* Ensures exclusive access to the shared memory structure.
  - *Account-Specific Semaphores:* One per account to synchronize access to individual account data during transactions.
- **Synchronization Flow**:
  - Tellers write requests to shared memory, signal *requestSemaphore*, and wait on *responseSemaphore*.
  - The main server processes the request upon *requestSemaphore* activation, writes the response, and signals *responseSemaphore*.
  - Account-specific semaphores prevent concurrent modifications to the same account.

3. Implementation Details

The implementation adheres to the project requirements, utilizing C with Unix system calls for process management, IPC, and synchronization. Below is a comprehensive breakdown of the key aspects:

3.1 Teller Functions

- **Deposit Teller (***deposit* function**)**:
  - Handles both new account creation and deposits to existing accounts.
  - **Steps**:
    1. Receives an *InitialClientRequest* and determines if it's a new account (N) or existing account request.
    2. Uses shared memory to request account creation or validation from the main server.
    3. Sends an *InitialResponse* to the client via the client response FIFO.
    4. Reads the *TransactionRequest* from the client request FIFO.
    5. Requests the deposit operation via shared memory.

6. Sends a *TransactionResponse* to the client with the outcome (success or failure).
    - **Key Features**:
        - Supports new account creation with unique IDs and deposits to existing accounts.
        - Prints transaction status to stdout for logging purposes.
- **Withdraw Teller (***withdraw* function**)**:
    - Handles withdrawals from existing accounts.
    - **Steps**:
        1. Receives an *InitialClientRequest* and validates the existing account.
        2. Uses shared memory to check account existence and activity status.
        3. Sends an *InitialResponse* to the client.
        4. Reads the *TransactionRequest* from the client.
        5. Requests the withdrawal via shared memory, potentially marking the account inactive if the balance reaches zero.
        6. Sends a *TransactionResponse*, indicating success, failure, or account closure.
    - **Key Features**:
        - Ensures withdrawals only occur from active accounts with sufficient balance.
        - Handles account closure when the balance becomes zero.
- **Creation Mechanism**:
    - Tellers are created using the custom *Teller* function, which forks a new process and executes the specified function (*deposit* or *withdraw*) with provided arguments.

3.2 FIFO Usage

- **Server FIFO**:
    - **Purpose**: Allows clients to send initial requests to the server.
    - **Implementation**: Created by the main server with mkfifo and opened in non-blocking read mode. The server uses select to monitor for incoming requests without blocking.
    - **Operation**: Clients write *InitialClientRequest* structures to this FIFO, which the server reads and uses to spawn Teller processes.
- **Client Request FIFO**:
    - **Purpose**: Enables clients to send transaction details to their assigned Teller.
    - **Implementation**: Each client child process creates a unique FIFO (e.g., client_<pid>_request) with mkfifo. The Teller opens it in read-only mode to receive the *TransactionRequest*.
    - **Operation**: After receiving the initial response, the client writes the transaction amount and account ID, which the Teller processes.
- **Client Response FIFO**:
    - **Purpose**: Allows Tellers to send responses back to clients.

    - **Implementation**: Created by each client child process (e.g., client_<pid>_response) and opened by the Teller in write-only mode. The client opens it in read-only mode to receive responses.
    - **Operation**: The Teller writes an *InitialResponse* followed by a *TransactionResponse*, which the client reads to confirm the transaction outcome.
- **Cleanup**: FIFOs are unlinked during client and server shutdown to prevent resource leaks.

3.3 Semaphore Usage

- **requestSemaphore**:
    - **Purpose**: Notifies the main server of a new request in shared memory.
    - **Operation**: Initialized to 0. Tellers increment it with sem_post after writing a request, and the server decrements it with sem_trywait to process the request.
- **responseSemaphore**:
    - **Purpose**: Notifies Tellers that the main server has processed a request.
    - **Operation**: Initialized to 0. The server increments it with sem_post after writing a response, and Tellers wait on it with sem_wait to read the response.
- **globalAccessSemaphore**:
    - **Purpose**: Protects the global accounts array and account creation/deletion operations.
    - **Operation**: Initialized to 1. Acquired with sem_wait and released with sem_post during account management tasks.
- **shmAccessSemaphore**:
    - **Purpose**: Prevents concurrent access to the shared memory structure.
    - **Operation**: Initialized to 1. Tellers and the server acquire it before accessing shared memory and release it afterward.
- **Account-Specific Semaphores**:
    - **Purpose**: Ensures exclusive access to individual account data during transactions.
    - **Operation**: One semaphore per account, created with a unique name (e.g., /bank_sem_account_BankID_01). Initialized to 1 and used by the main server to synchronize deposit and withdrawal operations.

3.4 Shared Memory Usage

- **Purpose**: Facilitates efficient communication between Teller processes and the main server without requiring individual pipes for each Teller.
- **Structure**: Defined as *SharedMemoryData*, containing fields for teller PID, transaction type, account ID, amount, success flag, message, and client name.
- **Implementation**:
    - Created using shm_open with the name /bank_shared_memory, sized to fit one *SharedMemoryData* structure, and mapped into memory with mmap.
    - Tellers write requests to this memory, and the main server writes responses, synchronized by *requestSemaphore* and *responseSemaphore*.
- **Usage**:

- Tellers populate the structure with request details and signal the server.
- The server processes the request, updates the structure with the response, and signals the Teller.
- Protected by *shmAccessSemaphore* to prevent race conditions.

3.5 Communication Details

- **Client to Server**:
  - Clients send an *InitialClientRequest* to the server FIFO, specifying the transaction type and FIFO paths.
  - The server reads this and creates a Teller process.
- **Server to Teller**:
  - The server forks a Teller process, passing the *InitialClientRequest* as an argument.
- **Teller to Main Server**:
  - Tellers use shared memory to send requests (e.g., create account, deposit, withdraw) and wait for responses, synchronized via semaphores.
- **Main Server to Teller**:
  - The server processes requests from shared memory, writes responses, and signals completion.
- **Teller to Client**:
  - Tellers send an *InitialResponse* via the client response FIFO, followed by a *TransactionResponse* after processing the transaction.
- **Client to Teller**:
  - Clients send a *TransactionRequest* via the client request FIFO after receiving the initial response.
- **Error Handling**:
  - Signal handlers ensure graceful shutdown, cleaning up resources like FIFOs and shared memory.

4. Testing

4.1 Banking Input Validation and Account Handling Test

This test case verifies the banking system's robustness against invalid deposits, withdrawals, operations on nonexistent accounts, and ensures proper handling of account creation, overdraw attempts, and invalid input values.

Test01.file

Results

```
❯ ./BankServer AdaBank server_fifo
AdaBank is active..
No previous logs.. Creating the bank database
Waiting for clients @server_fifo...
Received 17 clients from PIDClient82082..
-- Teller PID82087 is active serving Client01..
-- Teller PID82089 is active serving Client02..
-- Teller PID82092 is active serving Client03..
-- Teller PID82094 is active serving Client04..
-- Teller PID82096 is active serving Client05..
-- Teller PID82098 is active serving Client06..
-- Teller PID82100 is active serving Client07..
-- Teller PID82102 is active serving Client08..
-- Teller PID82104 is active serving Client09..
-- Teller PID82106 is active serving Client10..
-- Teller PID82108 is active serving Client11..
-- Teller PID82110 is active serving Client12..
-- Teller PID82112 is active serving Client01.. Welcome back Client01
-- Teller PID82113 is active serving Client01.. Welcome back Client01
-- Teller PID82114 is active serving Client01.. Welcome back Client01
-- Teller PID82115 is active serving Client01.. Welcome back Client01
-- Teller PID82116 is active serving Client01.. Welcome back Client01
Client03 withdraws 10 credits.. operation not permitted
Client04 withdraws 0 credits.. operation not permitted
Client05 withdraws -10 credits.. operation not permitted
Client06 withdraws 10 credits.. operation not permitted
Client07 withdraws 0 credits.. operation not permitted
Client08 withdraws -10 credits.. operation not permitted
Client09 deposited 10 credits.. operation not permitted
Client10 deposited 0 credits.. operation not permitted
Client11 deposited -10 credits.. operation not permitted
Client01 deposited 0 credits.. operation not permitted
Client02 deposited -10 credits.. operation not permitted
Client12 deposited 1000 credits... updating log
Client01 deposited 0 credits.. operation not permitted
Client01 deposited -100 credits.. operation not permitted
Client01 withdraws 0 credits.. operation not permitted
Client01 withdraws -100 credits.. operation not permitted
Client01 withdraws 9999 credits.. operation not permitted
Waiting for clients @server_fifo...
```

```
❯ ./BankClient Test01.file server_fifo
Reading Test01.file..
17 clients to connect.. creating clients..
Connected to the Bank...
Client01 connected.. depositing 0 credits
Client03 connected.. withdrawing 10 credits
Client04 connected.. withdrawing 0 credits
Client02 connected.. depositing -10 credits
Client05 connected.. withdrawing -10 credits
Client07 connected.. withdrawing 0 credits
Client08 connected.. withdrawing -10 credits
Client06 connected.. withdrawing 10 credits
Client09 connected.. depositing 10 credits
Client10 connected.. depositing 0 credits
Client12 connected.. depositing 1000 credits
Client11 connected.. depositing -10 credits
Client01 connected.. depositing 0 credits
Client01 connected.. depositing -100 credits
Client01 connected.. withdrawing 0 credits
Client01 connected.. withdrawing -100 credits
Client01 connected.. withdrawing 9999 credits
Client01 something went WRONG..
Client03 something went WRONG..
Client04 something went WRONG..
Client02 something went WRONG..
Client05 something went WRONG..
Client08 something went WRONG..
Client06 something went WRONG..
Client07 something went WRONG..
Client12 served.. BankID_03
Client01 something went WRONG..
Client09 something went WRONG..
Client01 something went WRONG..
Client10 something went WRONG..
Client11 something went WRONG..
Client01 something went WRONG..
Client01 something went WRONG..
Client01 something went WRONG..
exiting..
~/Desktop/Midterm ❯
```

4.2 Multi-Client Full Session Test with Account Creation and Follow-up

This test case simulates a realistic banking session with 20 clients.
First, each client independently creates a new account with an initial deposit.
Then, each client performs one additional operation — either depositing more money, partially withdrawing, or fully withdrawing to close the account.
The goal is to validate the server's ability to handle multiple clients, track account states accurately, and correctly process closures and balance updates.

Test02_1.file

```
1  N deposit 100
2  N deposit 200
3  N deposit 150
4  N deposit 300
5  N deposit 400
6  N deposit 250
7  N deposit 350
8  N deposit 500
9  N deposit 450
10 N deposit 600
11 N deposit 700
12 N deposit 550
13 N deposit 650
14 N deposit 800
15 N deposit 900
16 N deposit 750
17 N deposit 850
18 N deposit 1000
19 N deposit 950
20 N deposit 1200
```

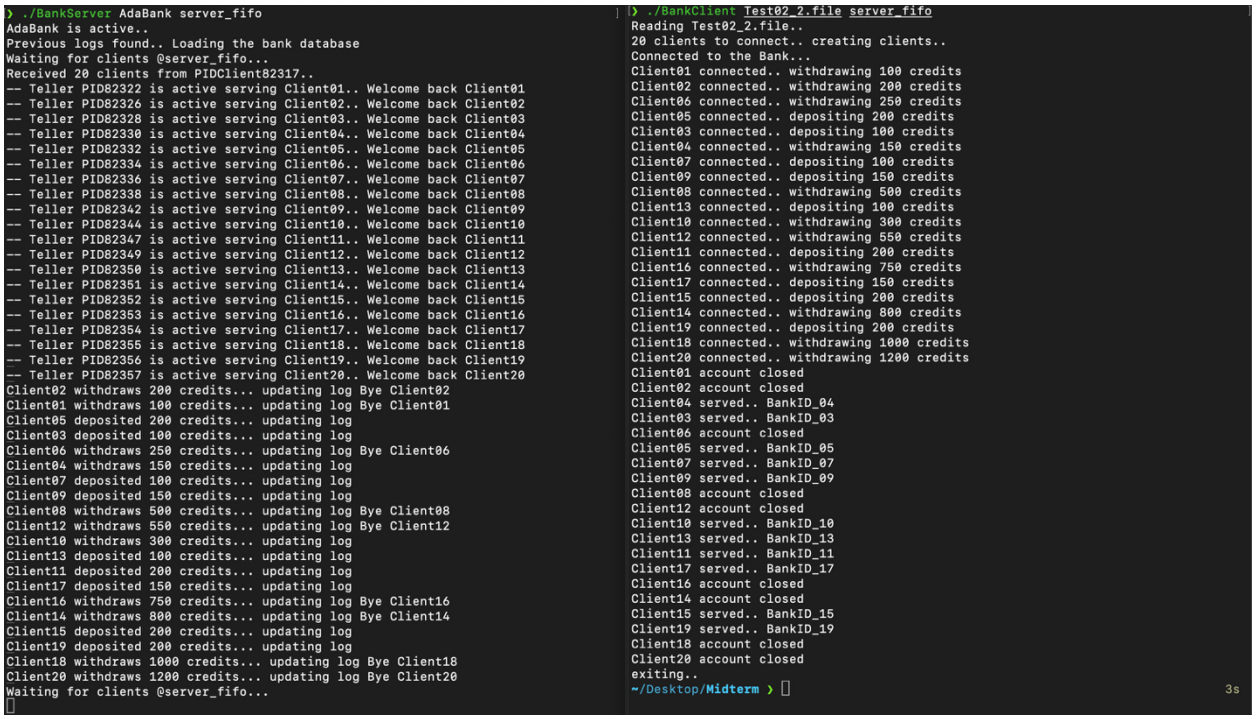
Test02_2.file

```
BankID_01 withdraw 100          # Withdraw entire balance, close BankID_1
BankID_02 withdraw 200          # Withdraw entire balance, close BankID_2
BankID_03 deposit 100           # Deposit into BankID_3
BankID_04 withdraw 150          # Partial withdraw from BankID_4 (balance remains)
BankID_05 deposit 200           # Deposit into BankID_5
BankID_06 withdraw 250          # Withdraw entire balance, close BankID_6
BankID_07 deposit 100           # Deposit into BankID_7
BankID_08 withdraw 500          # Withdraw entire balance, close BankID_8
BankID_09 deposit 150           # Deposit into BankID_9
BankID_10 withdraw 300         # Partial withdraw from BankID_10 (balance remains)
BankID_11 deposit 200          # Deposit into BankID_11
BankID_12 withdraw 550         # Withdraw entire balance, close BankID_12
BankID_13 deposit 100          # Deposit into BankID_13
BankID_14 withdraw 800         # Withdraw entire balance, close BankID_14
BankID_15 deposit 200          # Deposit into BankID_15
BankID_16 withdraw 750         # Withdraw entire balance, close BankID_16
BankID_17 deposit 150          # Deposit into BankID_17
BankID_18 withdraw 1000        # Withdraw entire balance, close BankID_18
BankID_19 deposit 200          # Deposit into BankID_19
BankID_20 withdraw 1200        # Withdraw entire balance, close BankID_20
```

Results

```
❯ ./BankServer AdaBank server_fifo
AdaBank is active..
No previous logs.. Creating the bank database
Waiting for clients @server_fifo...
Received 20 clients from PIDClient82242..
-- Teller PID82247 is active serving Client01..
-- Teller PID82250 is active serving Client02..
-- Teller PID82253 is active serving Client03..
-- Teller PID82255 is active serving Client04..
-- Teller PID82257 is active serving Client05..
-- Teller PID82260 is active serving Client06..
-- Teller PID82263 is active serving Client07..
-- Teller PID82265 is active serving Client08..
-- Teller PID82267 is active serving Client09..
-- Teller PID82269 is active serving Client10..
-- Teller PID82272 is active serving Client11..
-- Teller PID82274 is active serving Client12..
-- Teller PID82275 is active serving Client13..
-- Teller PID82276 is active serving Client14..
-- Teller PID82277 is active serving Client15..
-- Teller PID82278 is active serving Client16..
-- Teller PID82279 is active serving Client17..
-- Teller PID82280 is active serving Client18..
-- Teller PID82281 is active serving Client19..
-- Teller PID82282 is active serving Client20..
Client01 deposited 100 credits... updating log
Client02 deposited 200 credits... updating log
Client03 deposited 150 credits... updating log
Client06 deposited 250 credits... updating log
Client05 deposited 400 credits... updating log
Client04 deposited 300 credits... updating log
Client07 deposited 350 credits... updating log
Client08 deposited 500 credits... updating log
Client09 deposited 450 credits... updating log
Client10 deposited 600 credits... updating log
Client11 deposited 700 credits... updating log
Client12 deposited 550 credits... updating log
Client13 deposited 650 credits... updating log
Client15 deposited 900 credits... updating log
Client14 deposited 800 credits... updating log
Client16 deposited 750 credits... updating log
Client17 deposited 850 credits... updating log
Client18 deposited 1000 credits... updating log
Client19 deposited 950 credits... updating log
Client20 deposited 1200 credits... updating log
Waiting for clients @server_fifo...
```

```
❯ ./BankClient Test02_1.file server_fifo
Reading Test02_1.file..
20 clients to connect.. creating clients..
Connected to the Bank...
Client01 connected.. depositing 100 credits
Client02 connected.. depositing 200 credits
Client04 connected.. depositing 300 credits
Client06 connected.. depositing 250 credits
Client05 connected.. depositing 400 credits
Client03 connected.. depositing 150 credits
Client08 connected.. depositing 500 credits
Client09 connected.. depositing 450 credits
Client07 connected.. depositing 350 credits
Client10 connected.. depositing 600 credits
Client11 connected.. depositing 700 credits
Client12 connected.. depositing 550 credits
Client13 connected.. depositing 650 credits
Client15 connected.. depositing 900 credits
Client14 connected.. depositing 800 credits
Client17 connected.. depositing 850 credits
Client16 connected.. depositing 750 credits
Client18 connected.. depositing 1000 credits
Client19 connected.. depositing 950 credits
Client20 connected.. depositing 1200 credits
Client01 served.. BankID_01
Client02 served.. BankID_02
Client04 served.. BankID_04
Client03 served.. BankID_03
Client06 served.. BankID_06
Client05 served.. BankID_05
Client09 served.. BankID_09
Client08 served.. BankID_08
Client07 served.. BankID_07
Client10 served.. BankID_10
Client11 served.. BankID_11
Client12 served.. BankID_12
Client13 served.. BankID_13
Client15 served.. BankID_15
Client14 served.. BankID_14
Client16 served.. BankID_16
Client17 served.. BankID_17
Client18 served.. BankID_18
Client19 served.. BankID_19
Client20 served.. BankID_20
exiting..
~/Desktop/Midterm ❯                                                    3s
```



4.3 Example Scenario on the PDF Test

This test replicates the scenario described in the assignment document to verify that the implemented system produces expected outputs under basic operations like account creation, deposits, withdrawals, account closure, and reconnection.

Client01.file

```
1  N deposit 300
2  BankID_None withdraw 30
3  N deposit 2000
```

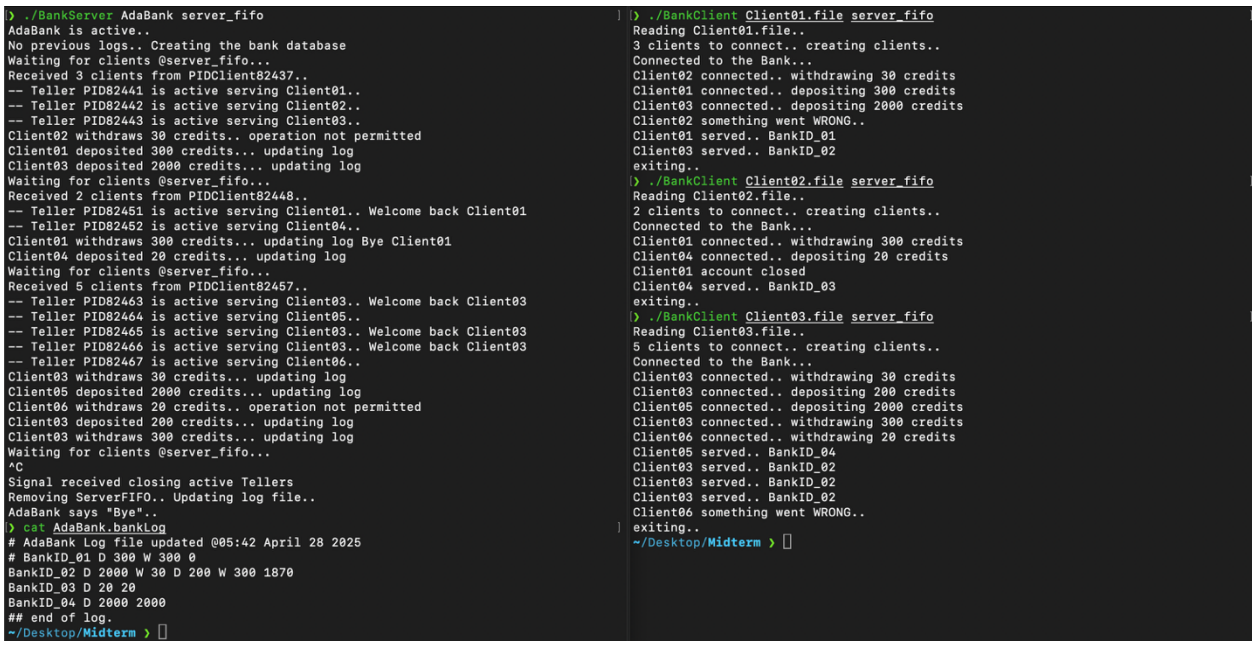
Client02.file

```
1  BankID_01 withdraw 300
2  N deposit 20
```

Client03.file

```
1  BankID_02 withdraw 30
2  N deposit 2000
3  BankID_02 deposit 200
4  BankID_02 withdraw 300
5  N withdraw 20
```

Results

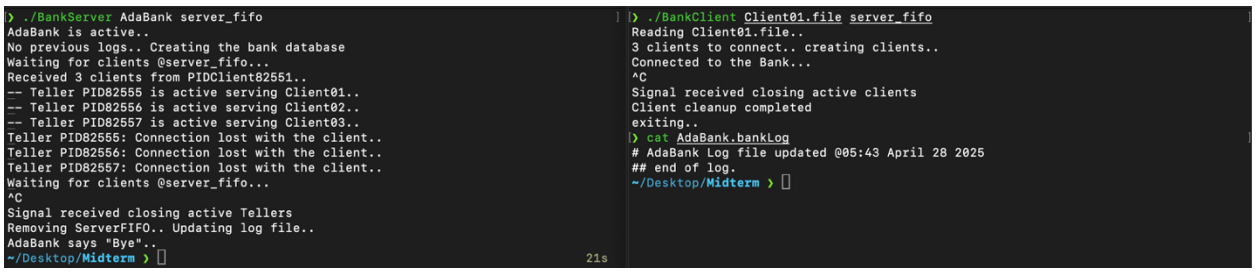


4.4 Client Interruption Test during Transaction

This test simulates the client being interrupted manually (via Ctrl+C) during an active banking operation.

The goal is to verify that the teller process handles the disconnection gracefully without crashing the server, without leaking resources, and ensuring the bank database remains consistent.

Results



4.5 Client Behavior When Server is Not Running Test

This test checks how the client behaves when the server is not available.

It simulates the client trying to connect and perform banking operations while the server FIFO is missing.

The goal is to ensure the client detects the missing server, handles the error gracefully, and exits without hanging or crashing.

Results

```
❯ ./BankClient Client01.file server_fifo
Reading Client01.file..
3 clients to connect.. creating clients..
Cannot connect server_fifo..
~/Desktop/Midterm ❯                                                    3s
```

4.6 Memory Leak Test with Example Scenario on PDF

This test checks whether memory allocated during client-server communication is properly released after operations.
It replays the example scenario from the assignment PDF while monitoring the server process using valgrind to detect any memory leaks, invalid memory access, or resource mismanagement.

Results

Client code

```
erver_fifo
==5225== Memcheck, a memory error detector
==5225== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==5225== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==5225== Command: ./BankClient Client01.file server_fifo
==5225==
Reading Client01.file..
3 clients to connect.. creating clients..
Connected to the Bank...
Client01 connected.. depositing 300 credits
Client02 connected.. withdrawing 30 credits
Client03 connected.. depositing 2000 credits
Client01 served.. BankID_01
==5226==
==5226== HEAP SUMMARY:
==5226==     in use at exit: 0 bytes in 0 blocks
==5226==   total heap usage: 3 allocs, 3 frees, 5,592 bytes allocated
==5226==
==5226== All heap blocks were freed -- no leaks are possible
==5226==
==5226== For lists of detected and suppressed errors, rerun with: -s
==5226== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Client02 something went WRONG..
==5227==
==5227== HEAP SUMMARY:
==5227==     in use at exit: 0 bytes in 0 blocks
==5227==   total heap usage: 3 allocs, 3 frees, 5,592 bytes allocated
==5227==
==5227== All heap blocks were freed -- no leaks are possible
==5227==
==5227== For lists of detected and suppressed errors, rerun with: -s
==5227== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Client03 served.. BankID_02
==5228==
==5228== HEAP SUMMARY:
==5228==     in use at exit: 0 bytes in 0 blocks
==5228==   total heap usage: 3 allocs, 3 frees, 5,592 bytes allocated
==5228==
==5228== All heap blocks were freed -- no leaks are possible
==5228==
==5228== For lists of detected and suppressed errors, rerun with: -s
==5228== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
exiting..
==5225==
==5225== HEAP SUMMARY:
==5225==     in use at exit: 0 bytes in 0 blocks
==5225==   total heap usage: 3 allocs, 3 frees, 5,592 bytes allocated
==5225==
==5225== All heap blocks were freed -- no leaks are possible
==5225==
==5225== For lists of detected and suppressed errors, rerun with: -s
==5225== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
osmancan@osmancan:~/Desktop/Midterm$
```

Server code

```
Waiting for clients @server_fifo...
^C
Signal received closing active Tellers
Removing ServerFIFO.. Updating log file..
AdaBank says "Bye"..
==5217==
==5217== HEAP SUMMARY:
==5217==     in use at exit: 0 bytes in 0 blocks
==5217==   total heap usage: 30 allocs, 30 frees, 20,729 bytes allocated
==5217==
==5217== All heap blocks were freed -- no leaks are possible
==5217==
==5217== For lists of detected and suppressed errors, rerun with: -s
==5217== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
osmancan@osmancan:~/Desktop/Midterm$
```

5. Conclusion

This bank simulator implementation demonstrates a robust client-server architecture with advanced process management and IPC techniques. The use of FIFOs enables reliable client-server communication, while shared memory and semaphores provide efficient and synchronized Teller-server interactions. The system supports concurrent client requests, maintains data integrity through proper synchronization, and ensures persistence via a log file. Signal handling enhances resilience, allowing clean termination under various conditions.

The design choices—such as a single shared memory segment for all Tellers and per-account semaphores— meets the project requirements, also making the system scalable up to the defined limits. All tests, including functional scenarios, edge cases, signal handling, server failure handling, and memory leak detection, were executed successfully. The system behaved exactly as expected in all cases, with no crashes, memory leaks, or inconsistent states observed. Given the thorough testing and correct behavior across all scenarios, I believe the project fully meets the assignment requirements. I am aiming to achieve 100 points for this assignment