**GEBZE TECHNICAL UNIVERSITY**

**COMPUTER ENGINEERING**

**DEPARTMENT**

**CSE344 Systems Programming**

**Spring 2025**

**Final Project Report**

**Multi-threaded Distributed Chat and File Server**

**Osmancan Bozali**

**220104004011**

## **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**

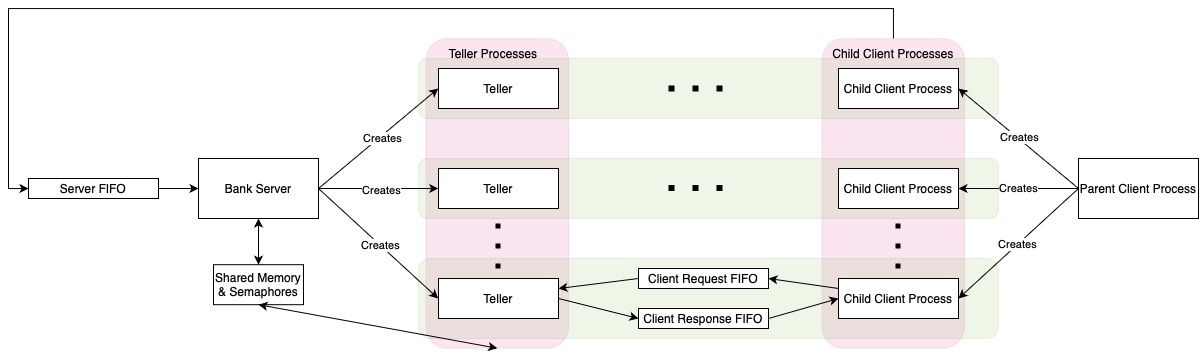
The scope of this project includes:

* **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.

## **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*

* 1. **Components**
     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.
    1. **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.)
    1. **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.
    1. **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.
  1. **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.
  1. **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.

## **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**:
    1. Supports new account creation with unique IDs and deposits to existing accounts.
    2. 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**:
    1. Ensures withdrawals only occur from active accounts with sufficient balance.
    2. 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.

### **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**

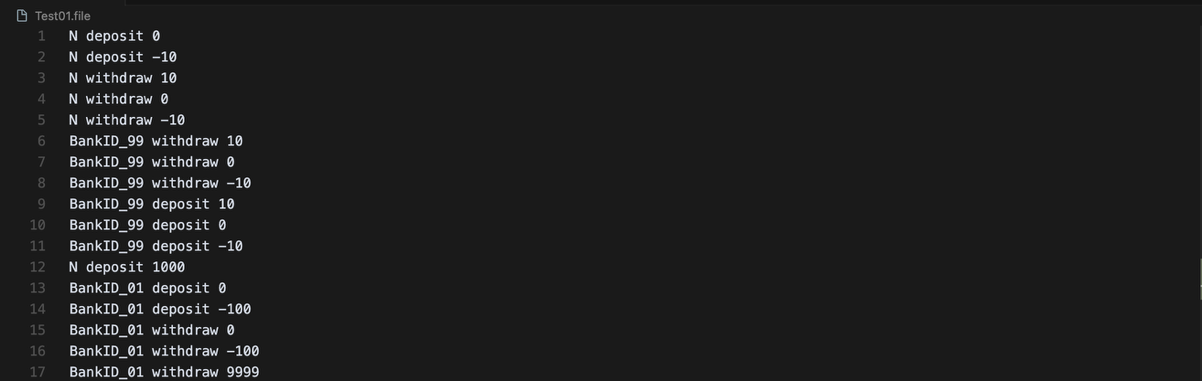
* **Client to Server**:
  + Clients send an *InitialClientRequest* to the server FIFO, specifying the transaction type and FIFO paths.
  + The server reads this and creates a Teller process.
* **Server to Teller**:
  + The server forks a Teller process, passing the *InitialClientRequest* as an argument.
* **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.

## **Testing**

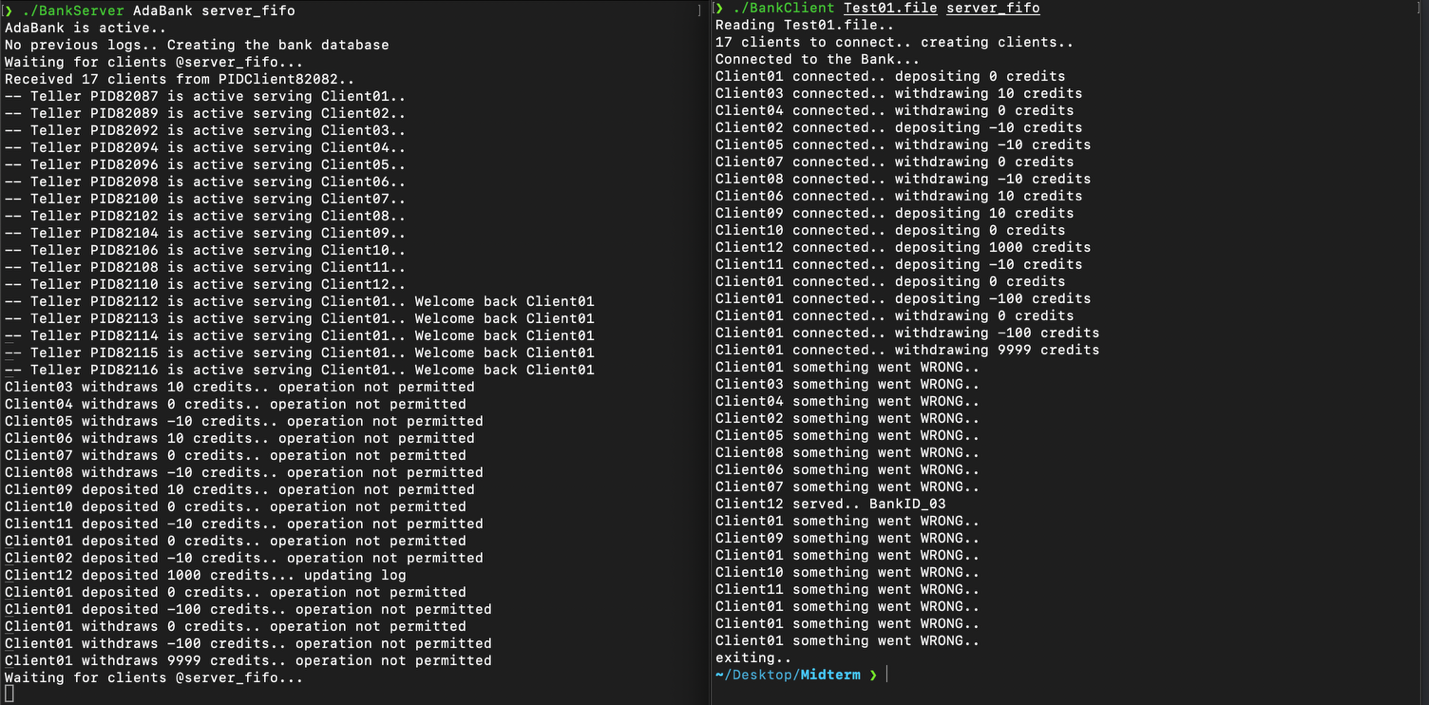
* 1. **Banking Input Validation and Account Handling Test**

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



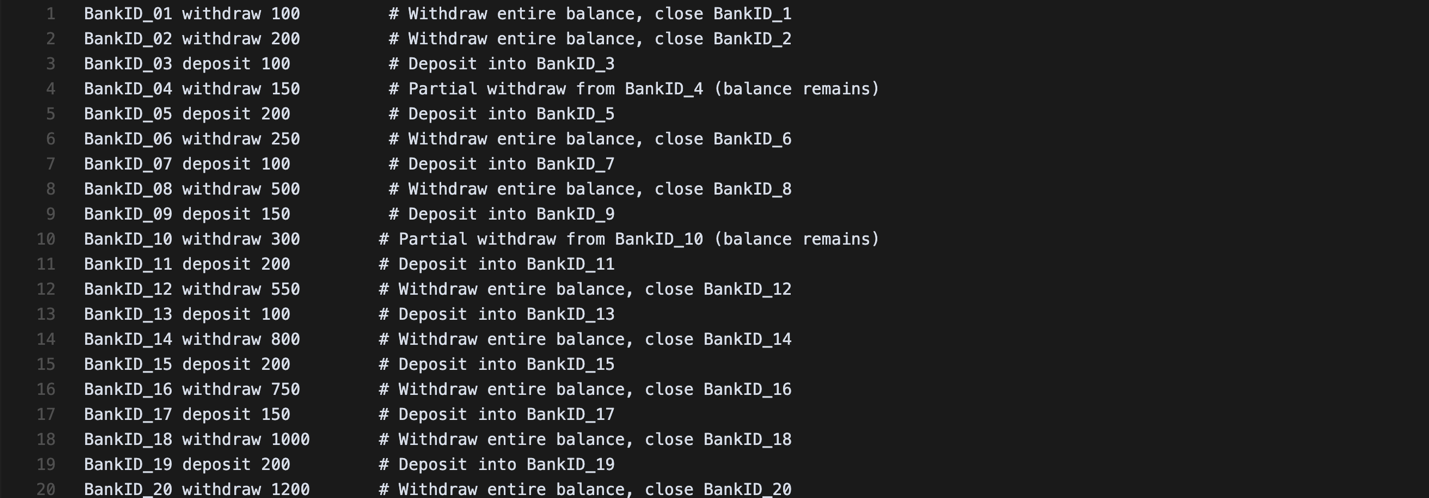
* 1. **Multi-Client Full Session Test with Account Creation and Follow-up**

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.

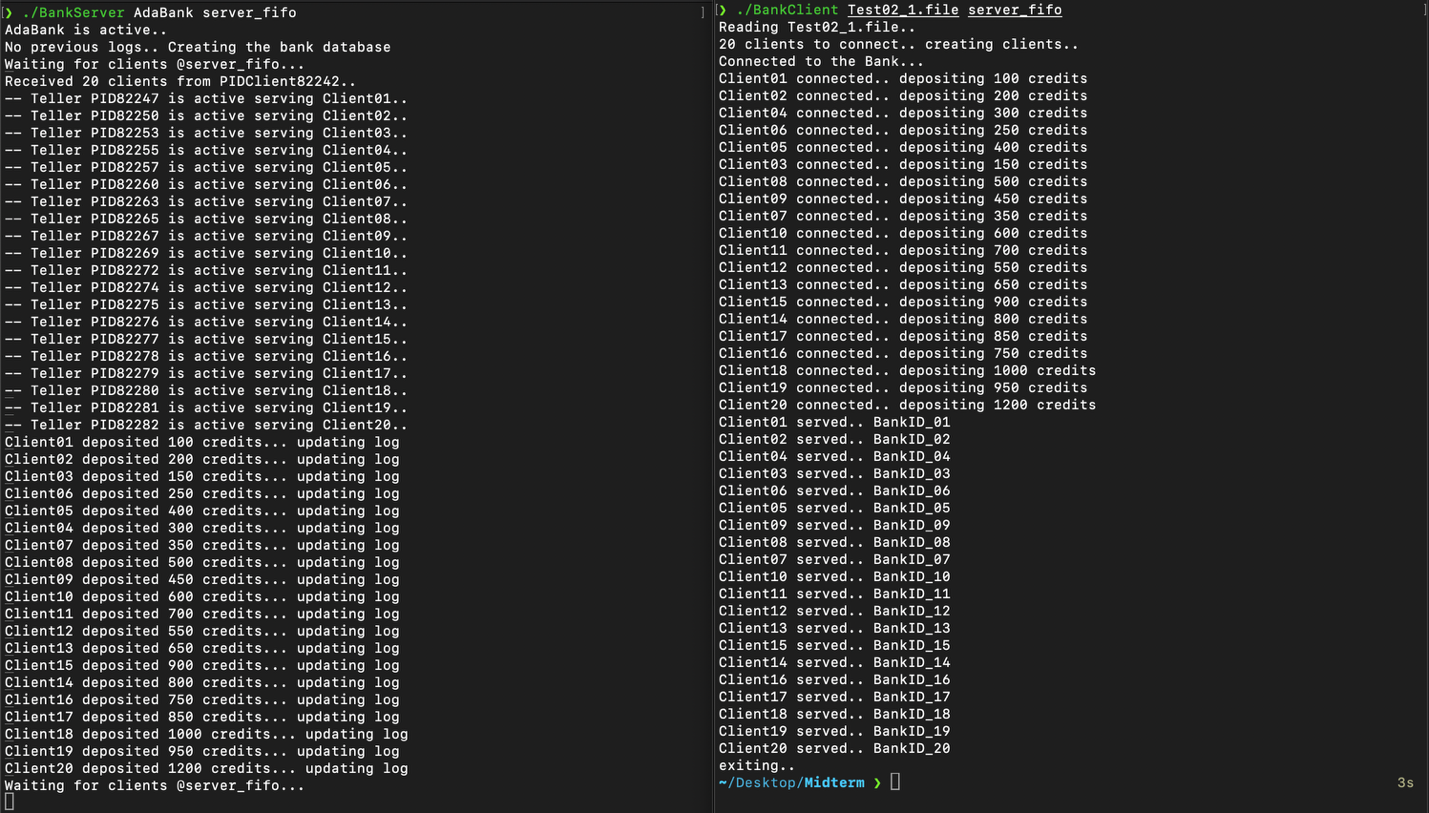
Test02\_1.file

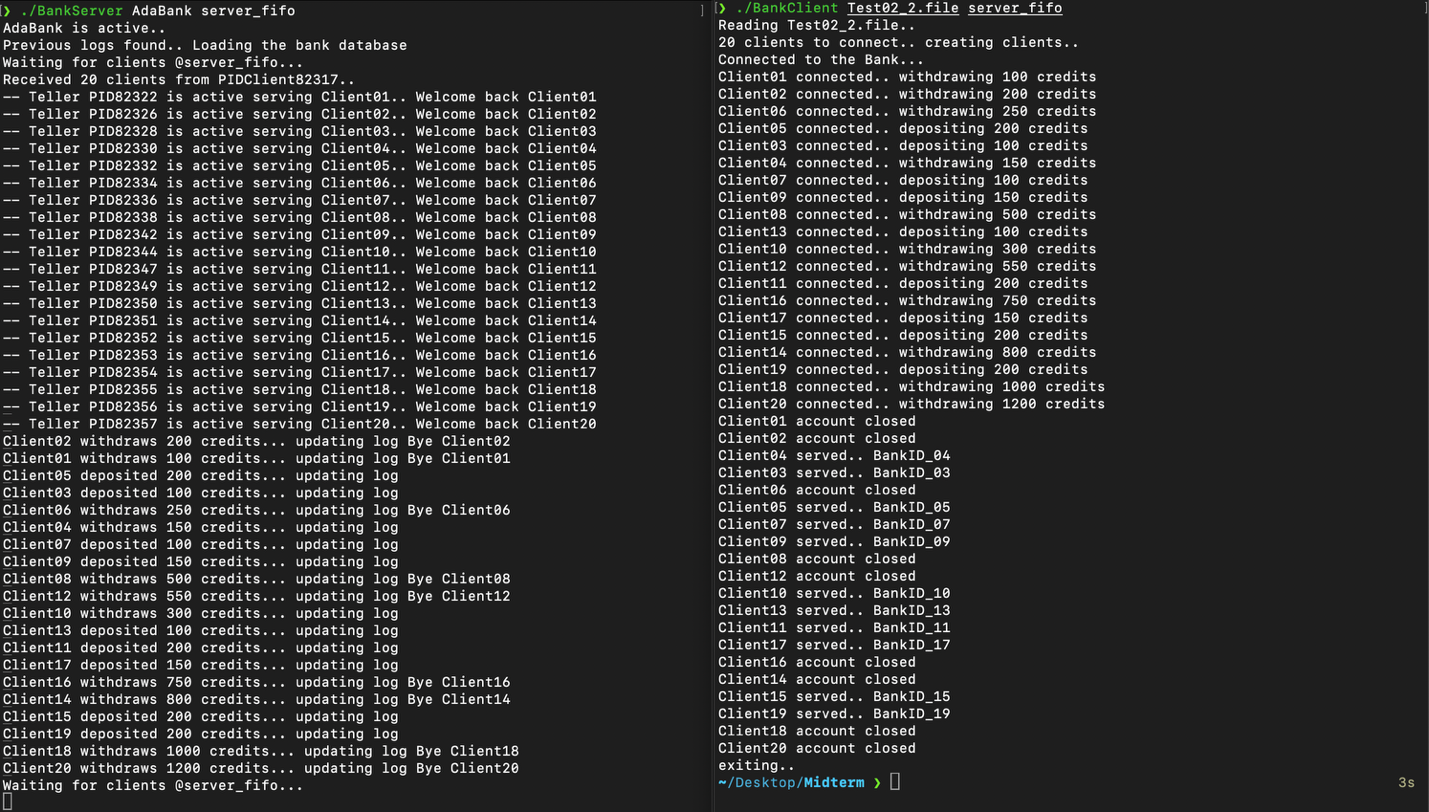


Test02\_2.file



Results





* 1. **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

****

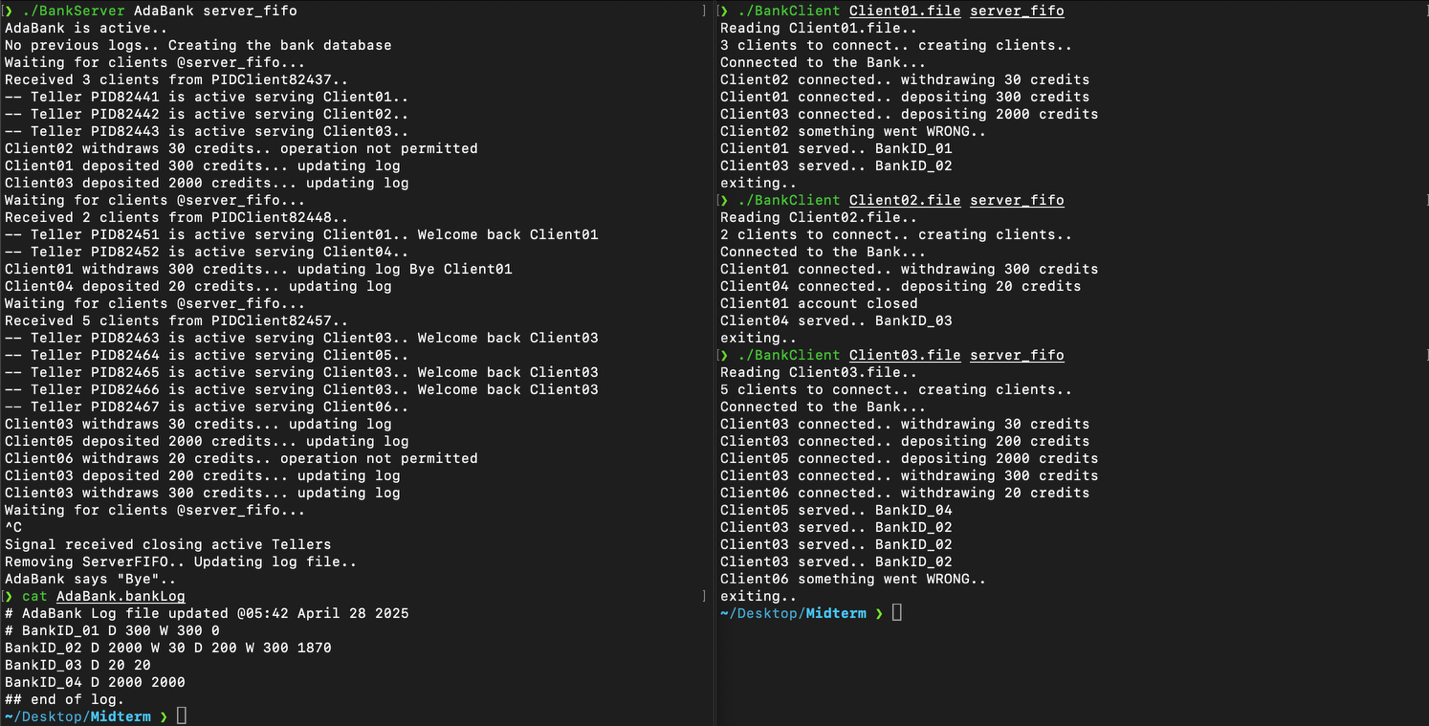
Client02.file

****

Client03.file

****

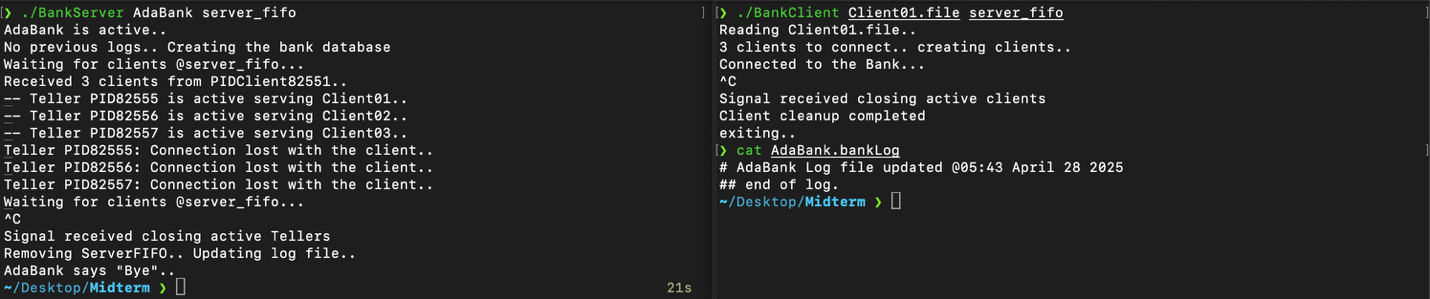
Results



* 1. **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.

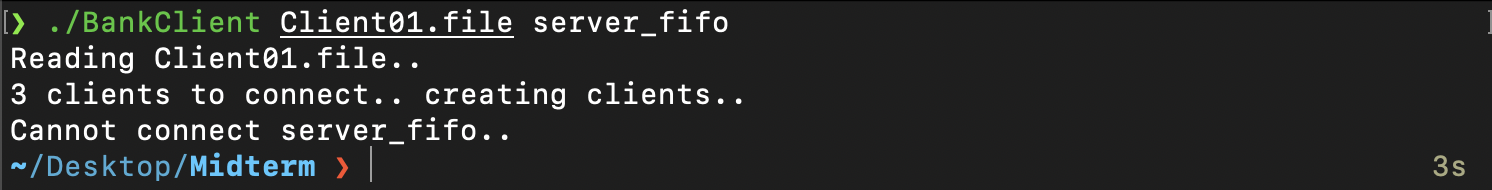
Results

****

* 1. **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

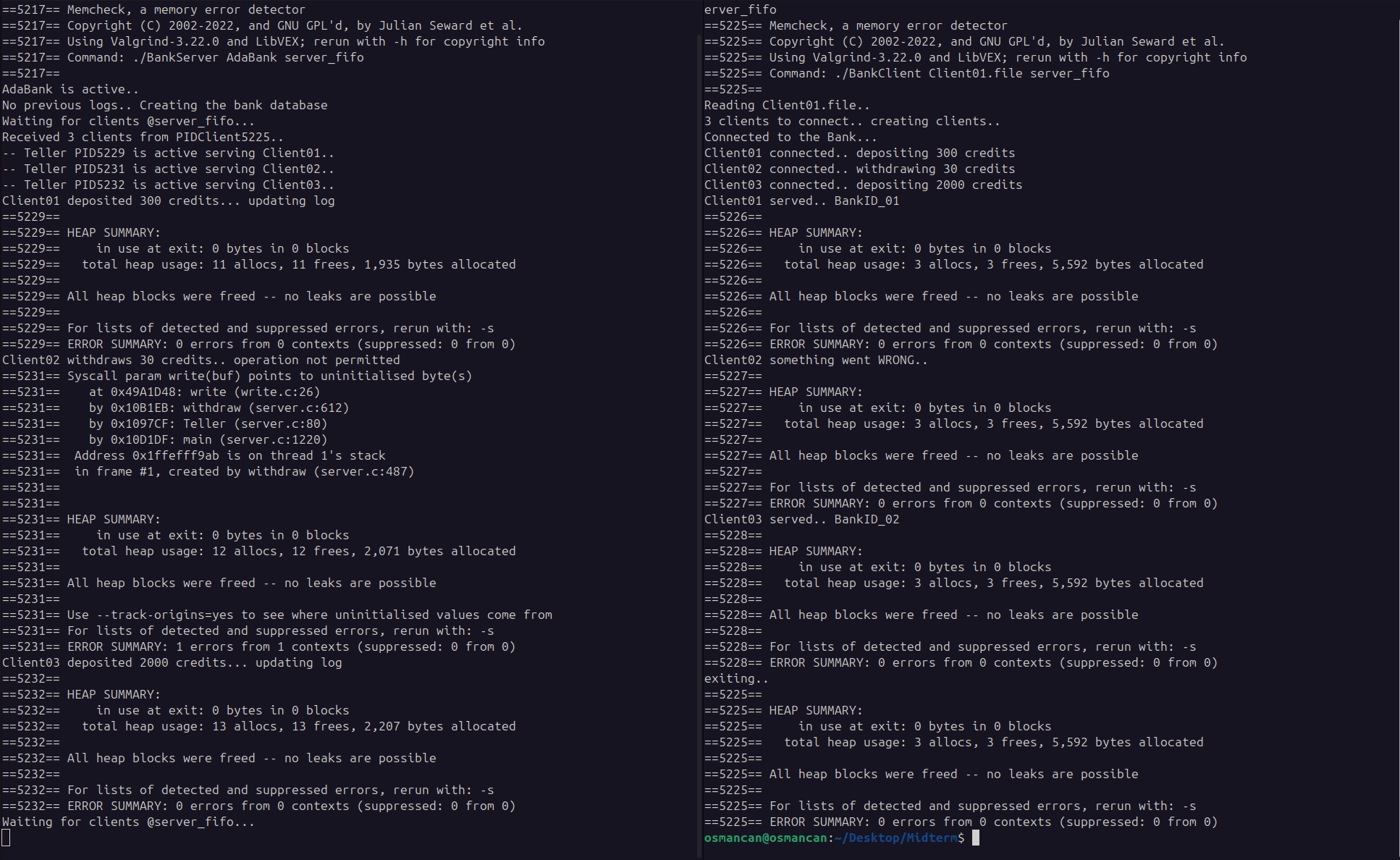


* 1. **Memory Leak Test with Example Scenario on PDF**

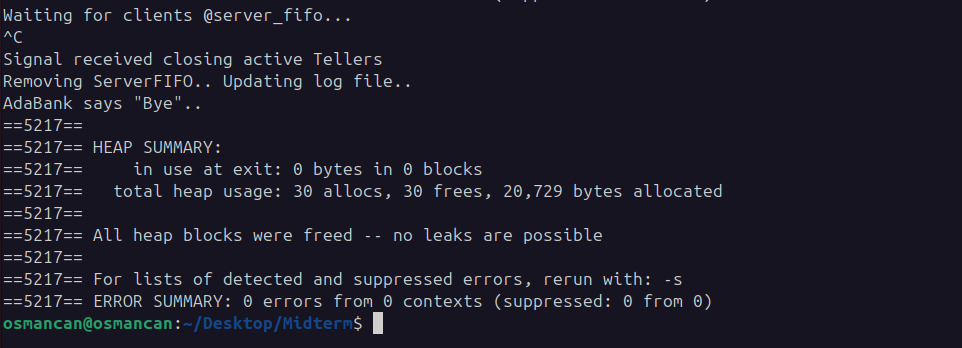
This test checks whether memory allocated during client-server communication is properly released after operations.  
It replays the example scenario from the assignment PDF while monitoring the server process using valgrind to detect any memory leaks, invalid memory access, or resource mismanagement.

Results

Client code



Server code

****

## **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