CSE 344

Midterm Project

200104004074

Bedirhan Ömer Aksoy

1) Introduction

In my implementation, the main function inside bankserver.c first cleans any left shared memory and semaphores from previous runs to do a fresh start, then it creates a named FIFO for client communication and initializes the shared memory, semaphore, and database structures, and loads existing accounts from AdaBank.bankLog if it exists. Then the server starts listening for client requests indefinitely. When a client request comes, the server forks (with pid_t pid = Teller()) a new teller process to handle the request, and then waits for that teller to complete. On SIGINT signal (Ctrl+C), the server writes the current account states into the log file, cleans up the resources and exits.

bankclient.c reads the transactions from an input file line-by-line and sends each transaction as a request to the bank server with a FIFO then waits for a response from the server with its own client FIFO and updates its BankID when receiving a new one after a successful deposit and continues processing remaining transactions.

teller.c implements the teller function that is executed with each forked child process. The teller reads the shared memory and semaphore to safely process the assigned client request with either depositing or withdrawing funds then sends the response back to client and exits after completing the operation.

utils.c provides helper functions for initializing shared memory, semaphores and the bank database and core functionalities such as deposit, withdraw, logging operations, and cleaning up resources. It also contains the Teller creation and synchronization functions.

utils.h defines the data structures such as ClientRequest, BankAccount, BankDatabase, and SharedMemory with function prototypes used in all modules for consistent access.

In this system, new accounts are created dynamically during deposits, existing accounts are updated or deleted based on transactions, and the all account database is persistently maintained with a log file between server restarts.

2) Code Explanation

2.0) bankclient.c:

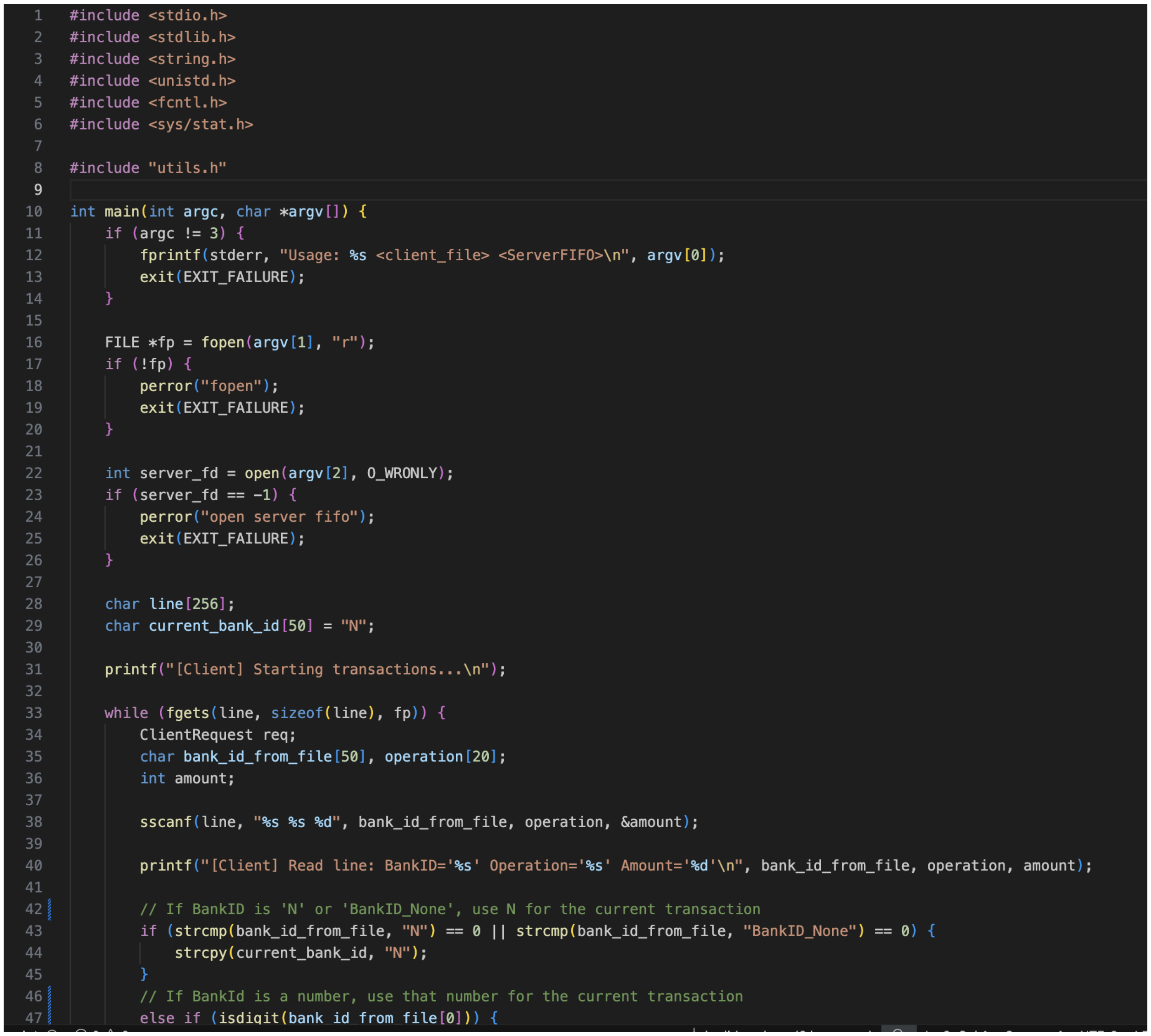
2.0.0) int main(int argc, char *argv[]): The main function starts with checking if exactly three command-line arguments are passed, which are expected to be the executable's name, the transaction file's name and the server FIFO's path. If not, it prints a usage error message and exits the program. Then it trys to open the specified transaction file to read. If opening the file fails, an error is printed and the program exits with failuer. After opening the file successfully, the function trys to open the server FIFO for writing which is necessary for sending client requests to the server.

A default current_bank_id value of "N" is used to track the latest BankID that the client should use. The program reads each line from the input transaction file, parsing it into three parts: bank_id_from_file, operation, and amount. If the parsed bank ID is not "N" or "BankID_None", the client updates it's current_bank_id with the new value; if not, it keeps using the last known BankID.

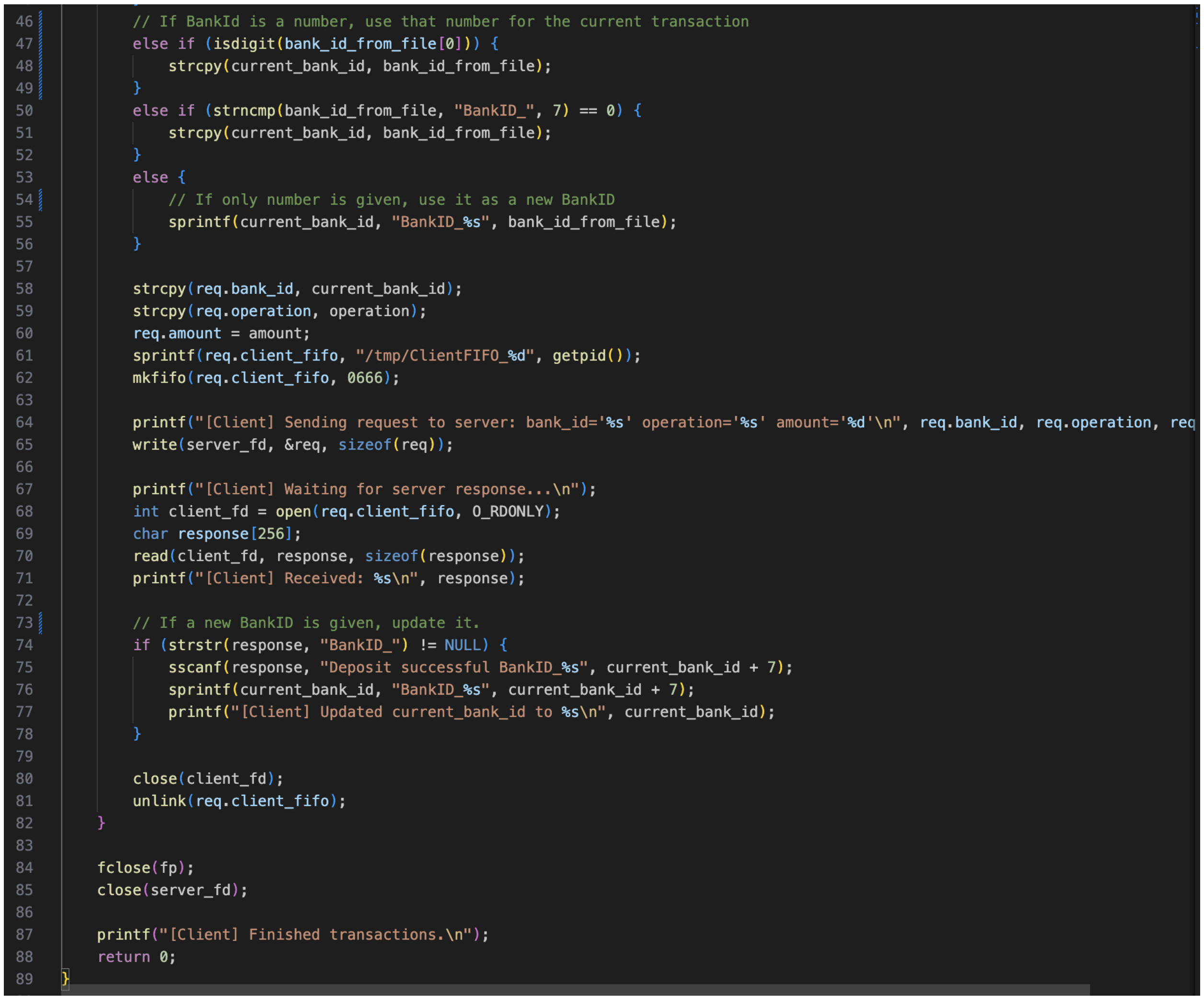
For each transaction read, the client fills the ClientRequest struct with the current BankID, operation type and amount. Then it creates a unique FIFO specific to the client by using its process ID in the FIFO name and calls mkfifo() to create it. The client sends the filled ClientRequest to the server by writing it to the server FIFO. After sending the request it opens it's own client FIFO in read-only mode and waits for the server's response. After receiving the response, the client reads the message, prints it to console and checks if the response contain a new BankID assignment. If a new BankID is provided by the server the client extracts the BankID from response and updates its current_bank_id for future transactions.

After handling the server response, the client closes the client FIFO and unlinks (deletes) to clean up. This procedure repeated for each transaction in the input file. Once all transactions are processed, the client closes the input file and the server FIFO and terminates the program successfully by returning 0.

The program ensures each transaction is independently processed with using a new FIFO per transaction, properly cleaning up FIFOs after each use to avoid resource leaks. It maintains dynamic tracking of BankID between multiple operations and handles all necessary error conditions and ensures stable communication with the bank server.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#include "utils.h"

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <client_file> <ServerFIFO>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    FILE *fp = fopen(argv[1], "r");
    if (!fp) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    int server_fd = open(argv[2], O_WRONLY);
    if (server_fd == -1) {
        perror("open server fifo");
        exit(EXIT_FAILURE);
    }

    char line[256];
    char current_bank_id[50] = "N";

    printf("[Client] Starting transactions...\n");

    while (fgets(line, sizeof(line), fp)) {
        ClientRequest req;
        char bank_id_from_file[50], operation[20];
        int amount;

        sscanf(line, "%s %s %d", bank_id_from_file, operation, &amount);

        printf("[Client] Read line: BankID='%s' Operation='%s' Amount='%d'\n", bank_id_from_file, operation, amount);

        // If BankID is 'N' or 'BankID_None', use N for the current transaction
        if (strcmp(bank_id_from_file, "N") == 0 || strcmp(bank_id_from_file, "BankID_None") == 0) {
            strcpy(current_bank_id, "N");
        }
        // If BankId is a number, use that number for the current transaction
        else if (isdigit(bank_id_from_file[0])) {
```



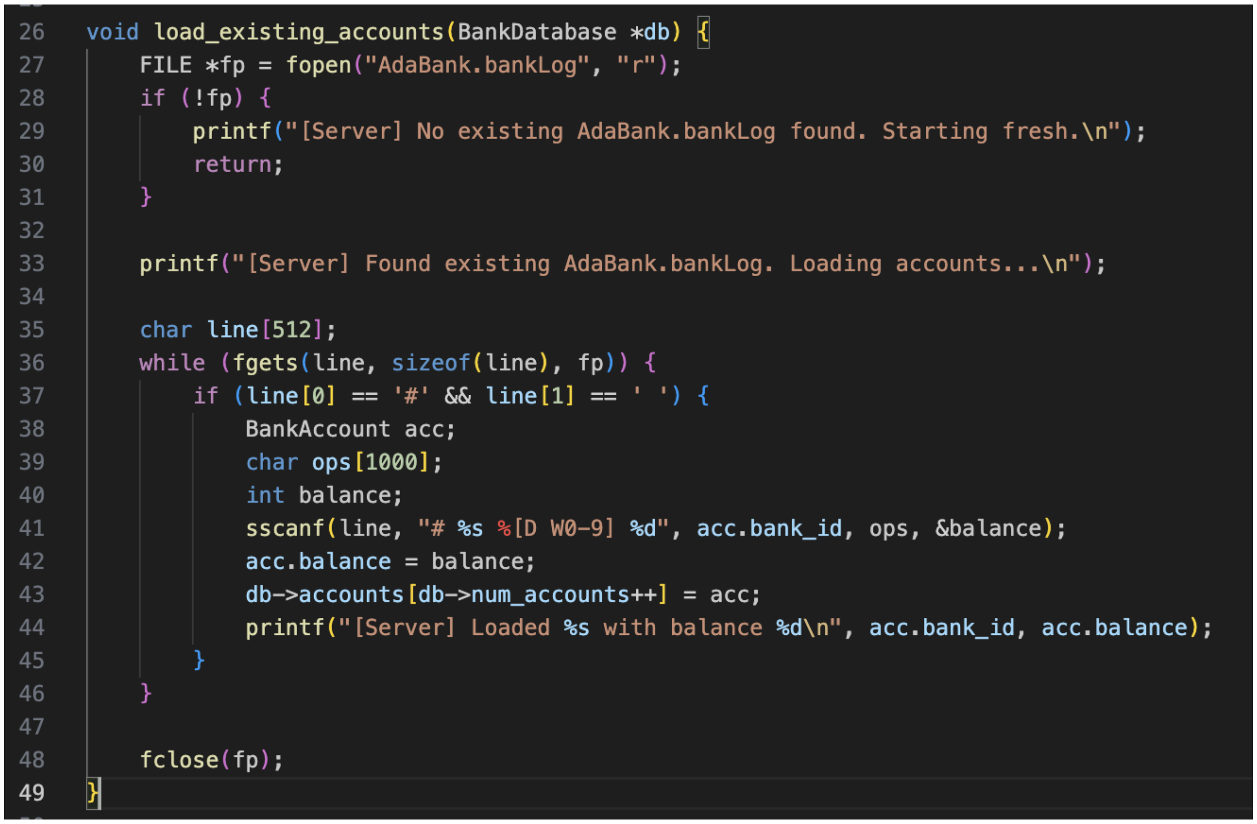
```c
        // If BankId is a number, use that number for the current transaction
        else if (isdigit(bank_id_from_file[0])) {
            strcpy(current_bank_id, bank_id_from_file);
        }
        else if (strncmp(bank_id_from_file, "BankID_", 7) == 0) {
            strcpy(current_bank_id, bank_id_from_file);
        }
        else {
            // If only number is given, use it as a new BankID
            sprintf(current_bank_id, "BankID_%s", bank_id_from_file);
        }

        strcpy(req.bank_id, current_bank_id);
        strcpy(req.operation, operation);
        req.amount = amount;
        sprintf(req.client_fifo, "/tmp/ClientFIFO_%d", getpid());
        mkfifo(req.client_fifo, 0666);

        printf("[Client] Sending request to server: bank_id='%s' operation='%s' amount='%d'\n", req.bank_id, req.operation, req
        write(server_fd, &req, sizeof(req));

        printf("[Client] Waiting for server response...\n");
        int client_fd = open(req.client_fifo, O_RDONLY);
        char response[256];
        read(client_fd, response, sizeof(response));
        printf("[Client] Received: %s\n", response);

        // If a new BankID is given, update it.
        if (strstr(response, "BankID_") != NULL) {
            sscanf(response, "Deposit successful BankID_%s", current_bank_id + 7);
            sprintf(current_bank_id, "BankID_%s", current_bank_id + 7);
            printf("[Client] Updated current_bank_id to %s\n", current_bank_id);
        }

        close(client_fd);
        unlink(req.client_fifo);
    }

    fclose(fp);
    close(server_fd);

    printf("[Client] Finished transactions.\n");
    return 0;
}
```

2.1) bankserver.c:

2.1.0) variables: In bankserver.c, the program defines local variables inside the main function for handling server operations. These variables are server_fd for the server FIFO file descriptor, SharedMemory *shm for managing the shared memory region, and sem_t *sem and sem_t *sem_request for synchronization of semaphores. A BankDatabase *db pointer is used for accessing the local in-memory database of bank accounts. volatile sig_atomic_t keep_running flag is declared globally to manage the server's running status that allows clean shutdown handling when receiving a SIGINT signal from the user.

2.1.1) void sigint_handler(int sig): The sigint_handler function is a signal handler for handling SIGINT signals during server execution. When a SIGINT is received, the function simply sets the keep_running flag to 0, signaling the main server loop to terminate gracefully. This enables the server to clean up resources, write the final state into the log file and exit cleanly without immediate termination or data loss.

2.1.2) void load_existing_accounts(BankDatabase *db): The load_existing_accounts function attempts to open the AdaBank.bankLog file if it exists in the current directory. If the file is successfully opened, it reads the file line by line, parsing each line to extract the BankID and the latest account balance. For each parsed entry, the function populates a new BankAccount object with the extracted BankID and balance and inserts it into the db->accounts array, incrementing the num_accounts counter after each insertion. If the file does not exist, the server prints a message indicating a fresh start with no existing accounts.



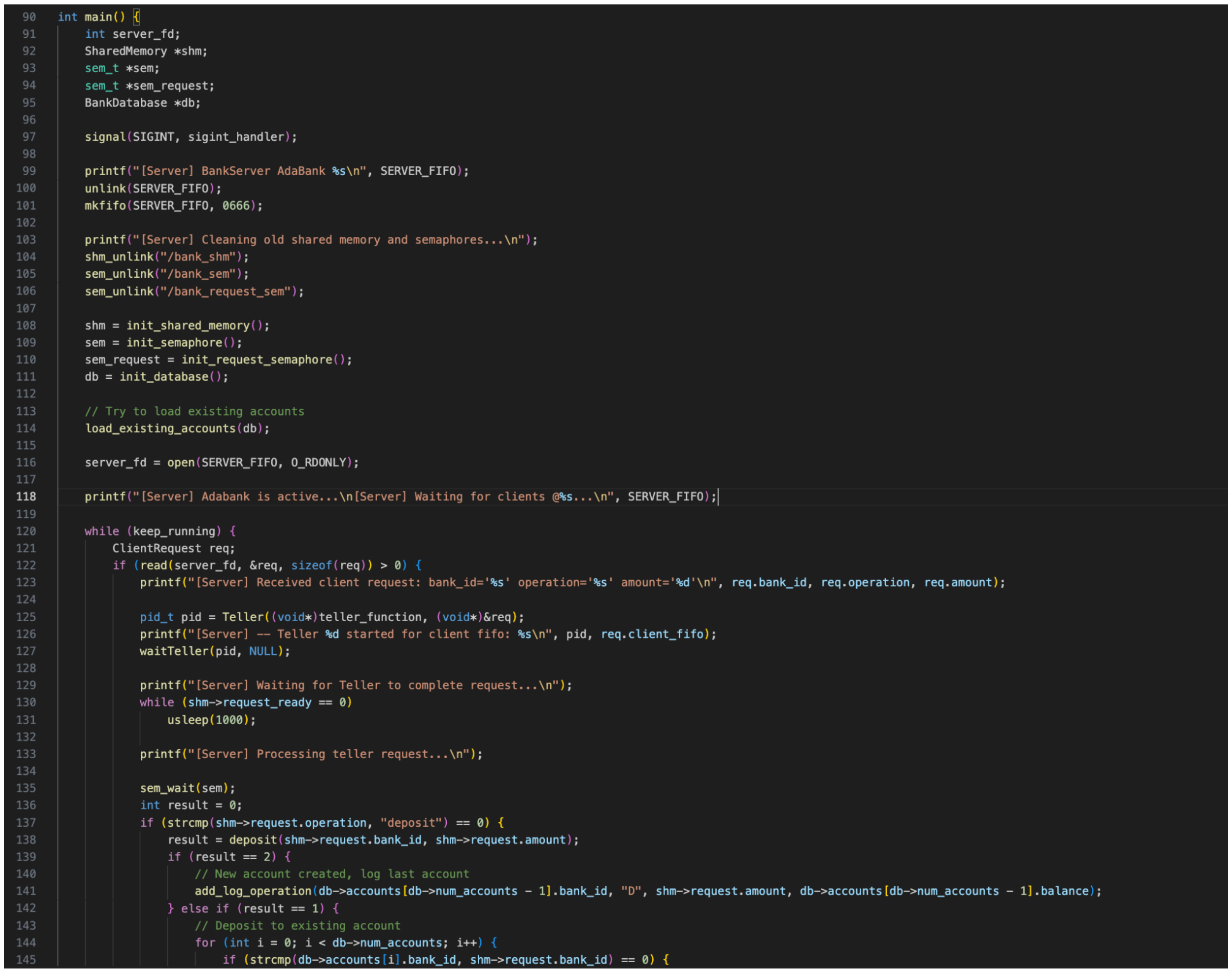
```c
void load_existing_accounts(BankDatabase *db) {
    FILE *fp = fopen("AdaBank.bankLog", "r");
    if (!fp) {
        printf("[Server] No existing AdaBank.bankLog found. Starting fresh.\n");
        return;
    }

    printf("[Server] Found existing AdaBank.bankLog. Loading accounts...\n");

    char line[512];
    while (fgets(line, sizeof(line), fp)) {
        if (line[0] == '#' && line[1] == ' ') {
            BankAccount acc;
            char ops[1000];
            int balance;
            sscanf(line, "# %s %[D W0-9] %d", acc.bank_id, ops, &balance);
            acc.balance = balance;
            db->accounts[db->num_accounts++] = acc;
            printf("[Server] Loaded %s with balance %d\n", acc.bank_id, acc.balance);
        }
    }

    fclose(fp);
}
```

2.1.3) int main(): The main function sets up the BankServer's entire runtime environment and coordinates the handling of client requests. It first sets the SIGINT handler by calling signal(SIGINT, sigint_handler) to allow a clean server shutdown on interruption. The function prints an initial startup message, removes any existing ServerFIFO file using unlink() and creates a new FIFO with mkfifo() to establish communication with clients.

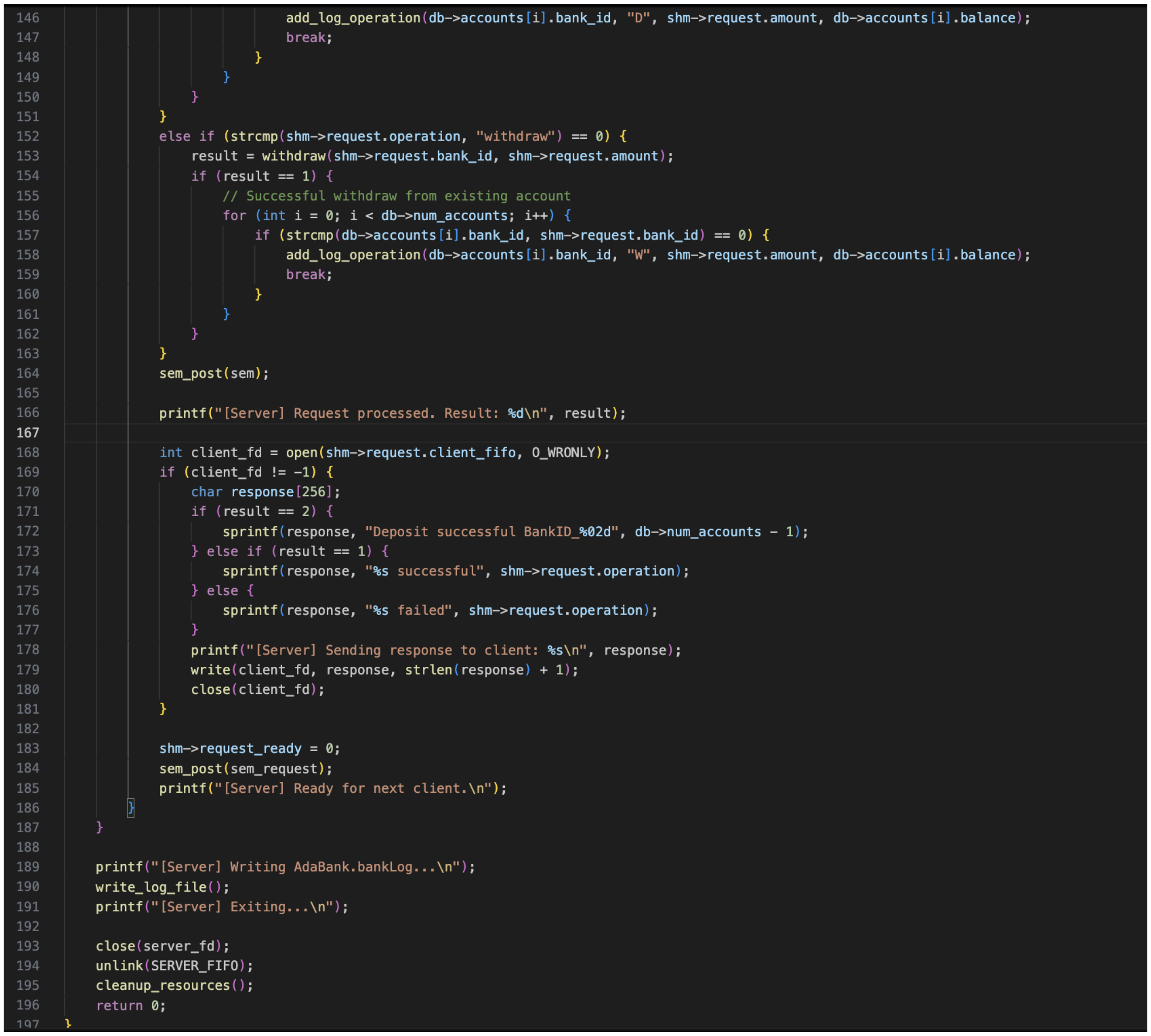
Before initializing the main resources, the program calls shm_unlink() and sem_unlink() functions to clean up any leftover shared memory or semaphore objects from previous executions to avoid conflicts. It then initializes shared memory, semaphores, and a fresh or loaded banking database by calling init_shared_memory(), init_semaphore(), init_request_semaphore(), and init_database(), respectively. If the AdaBank.bankLog file exists, load_existing_accounts(db) is called to reconstruct the in-memory bank database from the file.

After completing resource initialization, the server opens the SERVER_FIFO for reading and enters a main processing loop, which continues as long as keep_running is true. Inside this loop, the server waits for client requests by reading from the FIFO. Upon receiving a valid ClientRequest, it forks a child process through the Teller() function to handle the request concurrently using the teller_function. It prints debug messages showing which teller process is assigned to which client FIFO and waits for the teller process to complete by calling waitTeller(pid, NULL).

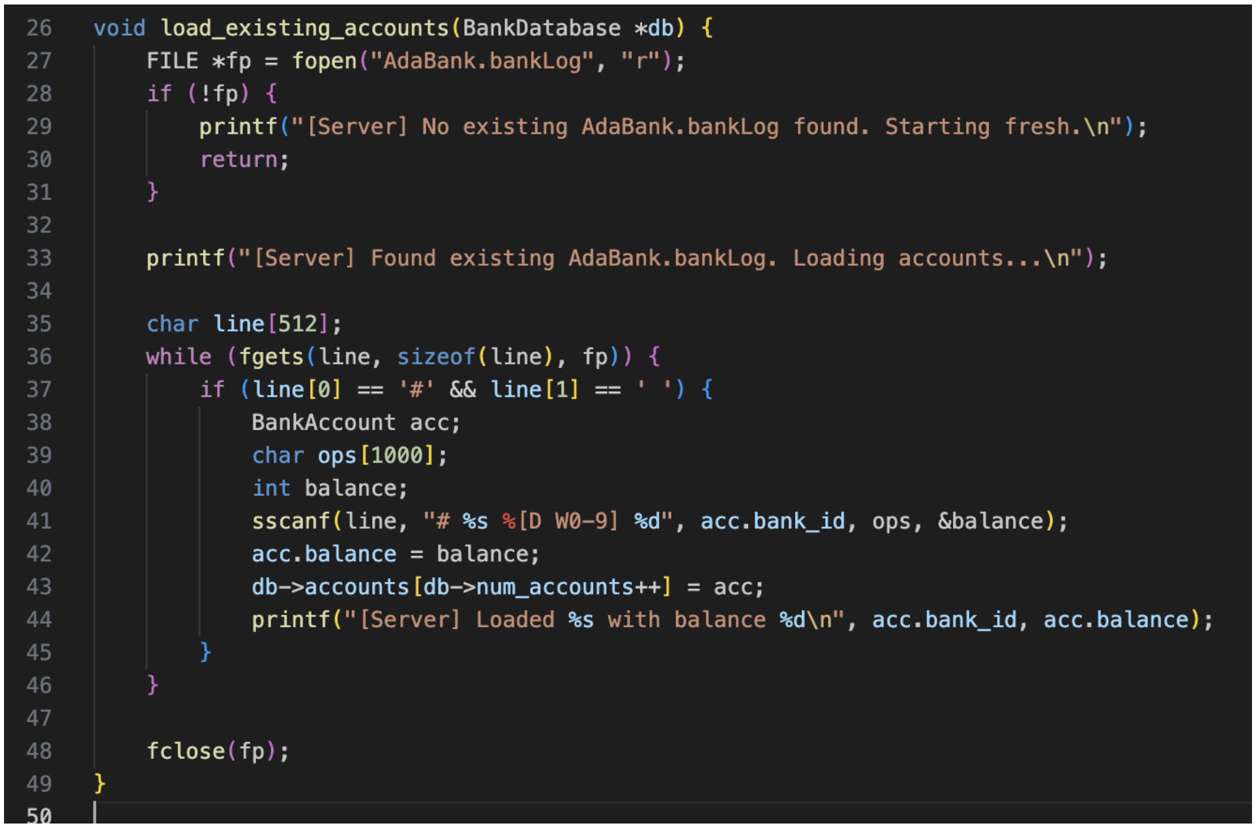
When a SIGINT signal interrupts the server, the main loop terminates, and the server begins its shutdown sequence. It prints shutdown messages, closes the server FIFO, unlinks the FIFO file, writes the current database state into the AdaBank.bankLog file and calls cleanup_resources() to release the shared memory and semaphore resources before exiting cleanly.



```c
int main() {
    int server_fd;
    SharedMemory *shm;
    sem_t *sem;
    sem_t *sem_request;
    BankDatabase *db;

    signal(SIGINT, sigint_handler);

    printf("[Server] BankServer AdaBank %s\n", SERVER_FIFO);
    unlink(SERVER_FIFO);
    mkfifo(SERVER_FIFO, 0666);

    printf("[Server] Cleaning old shared memory and semaphores...\n");
    shm_unlink("/bank_shm");
    sem_unlink("/bank_sem");
    sem_unlink("/bank_request_sem");

    shm = init_shared_memory();
    sem = init_semaphore();
    sem_request = init_request_semaphore();
    db = init_database();

    // Try to load existing accounts
    load_existing_accounts(db);

    server_fd = open(SERVER_FIFO, O_RDONLY);

    printf("[Server] Adabank is active...\n[Server] Waiting for clients @%s...\n", SERVER_FIFO);

    while (keep_running) {
        ClientRequest req;
        if (read(server_fd, &req, sizeof(req)) > 0) {
            printf("[Server] Received client request: bank_id='%s' operation='%s' amount='%d'\n", req.bank_id, req.operation, req.amount);

            pid_t pid = Teller((void*)teller_function, (void*)&req);
            printf("[Server] -- Teller %d started for client fifo: %s\n", pid, req.client_fifo);
            waitTeller(pid, NULL);

            printf("[Server] Waiting for Teller to complete request...\n");
            while (shm->request_ready == 0)
                usleep(1000);

            printf("[Server] Processing teller request...\n");

            sem_wait(sem);
            int result = 0;
            if (strcmp(shm->request.operation, "deposit") == 0) {
                result = deposit(shm->request.bank_id, shm->request.amount);
                if (result == 2) {
                    // New account created, log last account
                    add_log_operation(db->accounts[db->num_accounts - 1].bank_id, "D", shm->request.amount, db->accounts[db->num_accounts - 1].balance);
                } else if (result == 1) {
                    // Deposit to existing account
                    for (int i = 0; i < db->num_accounts; i++) {
                        if (strcmp(db->accounts[i].bank_id, shm->request.bank_id) == 0) {
```

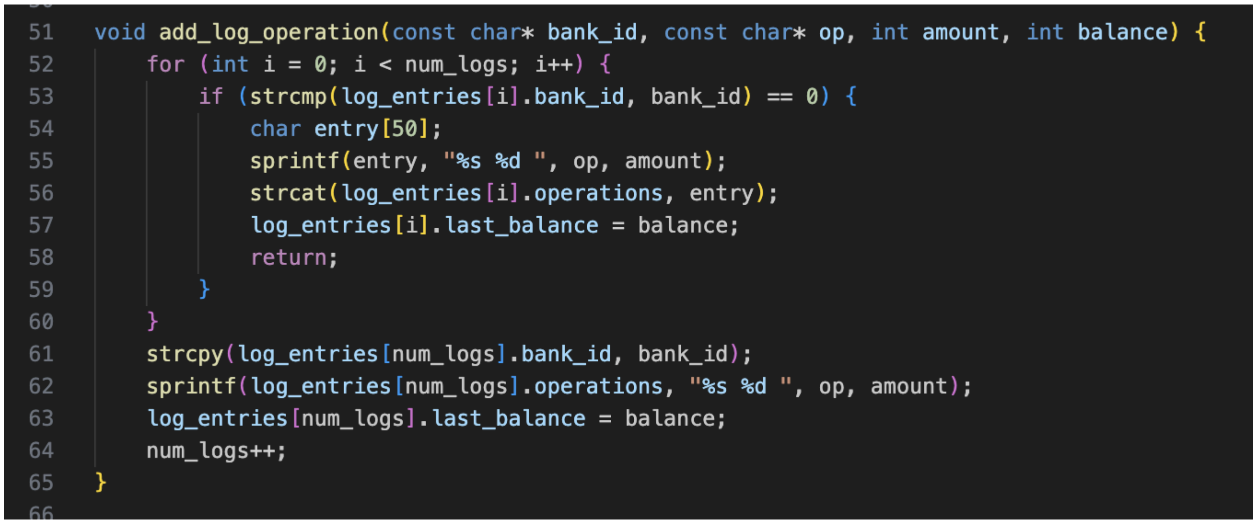

```
                            add_log_operation(db->accounts[i].bank_id, "D", shm->request.amount, db->accounts[i].balance);
                            break;
                        }
                    }
                }
            }
            else if (strcmp(shm->request.operation, "withdraw") == 0) {
                result = withdraw(shm->request.bank_id, shm->request.amount);
                if (result == 1) {
                    // Successful withdraw from existing account
                    for (int i = 0; i < db->num_accounts; i++) {
                        if (strcmp(db->accounts[i].bank_id, shm->request.bank_id) == 0) {
                            add_log_operation(db->accounts[i].bank_id, "W", shm->request.amount, db->accounts[i].balance);
                            break;
                        }
                    }
                }
            }
            sem_post(sem);

            printf("[Server] Request processed. Result: %d\n", result);

            int client_fd = open(shm->request.client_fifo, O_WRONLY);
            if (client_fd != -1) {
                char response[256];
                if (result == 2) {
                    sprintf(response, "Deposit successful BankID_%02d", db->num_accounts - 1);
                } else if (result == 1) {
                    sprintf(response, "%s successful", shm->request.operation);
                } else {
                    sprintf(response, "%s failed", shm->request.operation);
                }
                printf("[Server] Sending response to client: %s\n", response);
                write(client_fd, response, strlen(response) + 1);
                close(client_fd);
            }

            shm->request_ready = 0;
            sem_post(sem_request);
            printf("[Server] Ready for next client.\n");
        }
    }

    printf("[Server] Writing AdaBank.bankLog...\n");
    write_log_file();
    printf("[Server] Exiting...\n");

    close(server_fd);
    unlink(SERVER_FIFO);
    cleanup_resources();
    return 0;
}
```

2.1.4) void load_existing_accounts(BankDatabase *db): The load_existing_accounts function is responsible for reconstructing the in-memory bank database from a previous session's log file to ensure persistence between server restarts. It attempts to open the file named AdaBank.bankLog in read mode. If the file does not exist, the function prints a message indicating that no previous log was found and the server will start fresh with an empty database. If the file exists, the function reads it line by line, identifying lines that represent account records. For each valid account line, it extracts the BankID, deposit and withdrawal operations and the final balance using sscanf(). A new BankAccount structure is filled with the extracted BankID and balance and added to the bank database's accounts array. The function increments the number of accounts each time when new account is added. After processing all entries the file is closed.



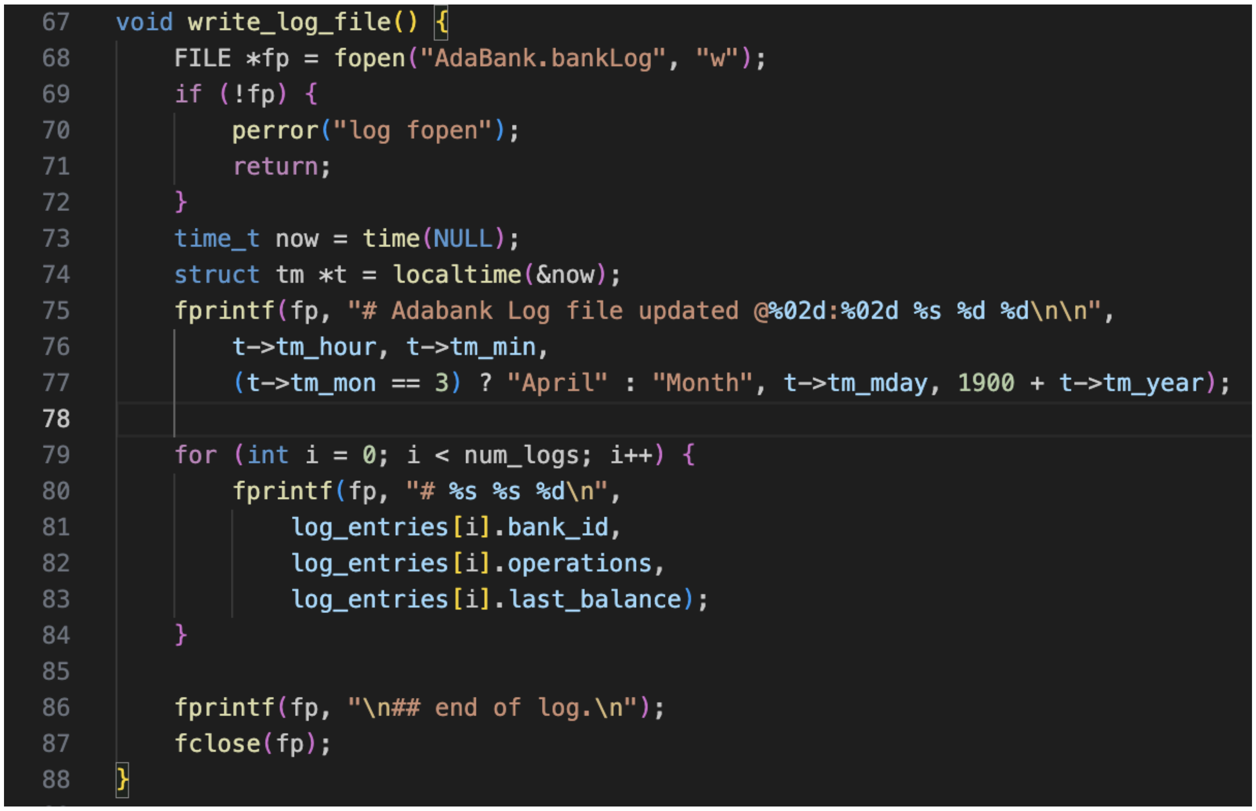
```c
void load_existing_accounts(BankDatabase *db) {
    FILE *fp = fopen("AdaBank.bankLog", "r");
    if (!fp) {
        printf("[Server] No existing AdaBank.bankLog found. Starting fresh.\n");
        return;
    }

    printf("[Server] Found existing AdaBank.bankLog. Loading accounts...\n");

    char line[512];
    while (fgets(line, sizeof(line), fp)) {
        if (line[0] == '#' && line[1] == ' ') {
            BankAccount acc;
            char ops[1000];
            int balance;
            sscanf(line, "# %s %[D W0-9] %d", acc.bank_id, ops, &balance);
            acc.balance = balance;
            db->accounts[db->num_accounts++] = acc;
            printf("[Server] Loaded %s with balance %d\n", acc.bank_id, acc.balance);
        }
    }

    fclose(fp);
}

```

2.1.5) void add_log_operation(const char* bank_id, const char* op, int amount, int balance): The add_log_operation function manages the runtime logging system with updating or creating new log entries in the memory array log_entries. When a deposit or withdrawal occurs, the function searches the current log entries for the matching BankID. If the BankID is found, it formats the new operation and appends it to the existing operations string while updating the last known balance. If the BankID is not already present in the log array, the function initializes a new log entry by copying the BankID, formatting the first operation string, setting the latest balance, and incrementing the count of log entries. It ensures multiple operations on the same BankID during a session are connected into a single log entry, updating the final log output before the server writes the complete AdaBank.bankLog at shutdown.



```c
void add_log_operation(const char* bank_id, const char* op, int amount, int balance) {
    for (int i = 0; i < num_logs; i++) {
        if (strcmp(log_entries[i].bank_id, bank_id) == 0) {
            char entry[50];
            sprintf(entry, "%s %d ", op, amount);
            strcat(log_entries[i].operations, entry);
            log_entries[i].last_balance = balance;
            return;
        }
    }
    strcpy(log_entries[num_logs].bank_id, bank_id);
    sprintf(log_entries[num_logs].operations, "%s %d ", op, amount);
    log_entries[num_logs].last_balance = balance;
    num_logs++;
}
```

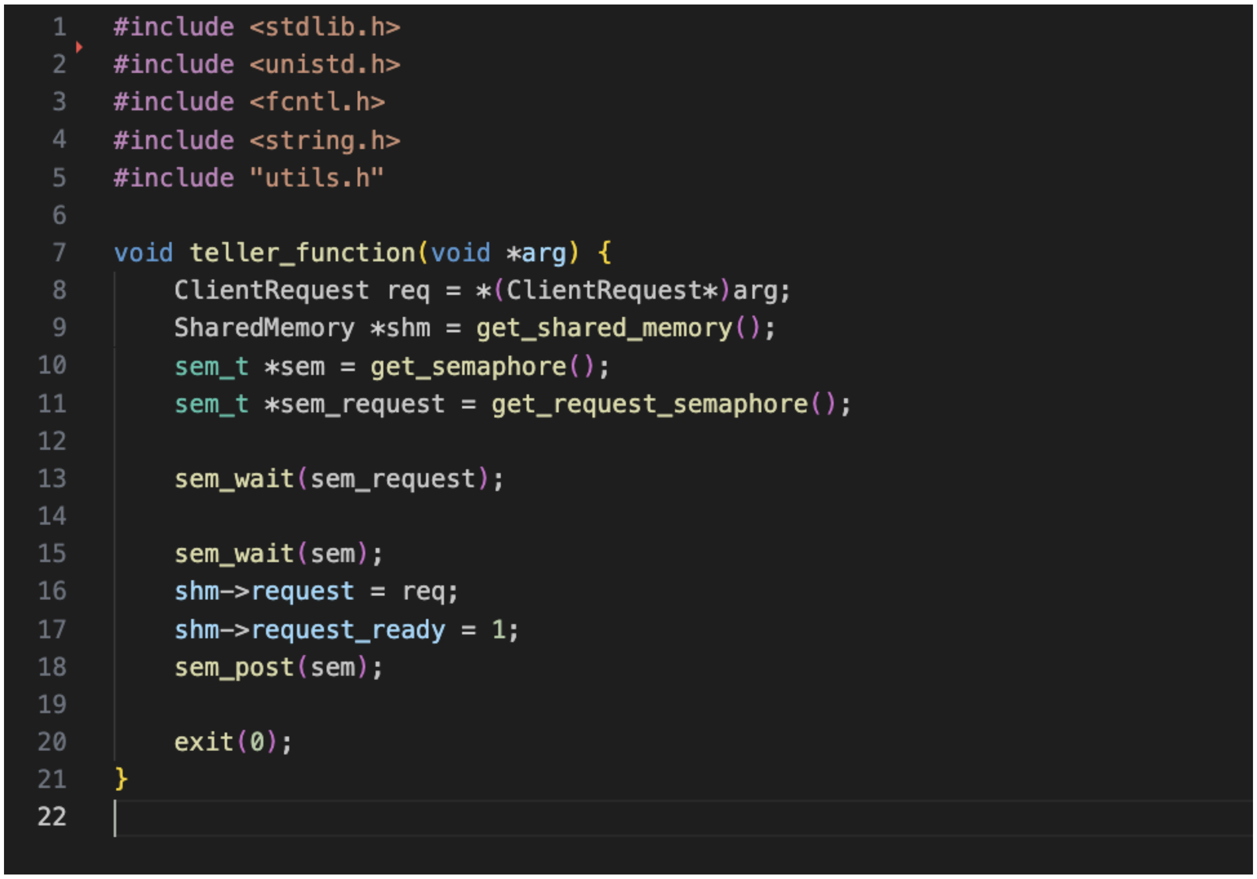
2.1.6) void write_log_file(): The write_log_file function is responsible for writing the in-memory session log entries into the persistent AdaBank.bankLog file when the server shuts down. It attempts to open the log file in write mode. If the file cannot be opened, it prints an error message using perror() and returns immediately without proceeding. Upon successful file opening, the function obtains the current time using time(NULL) and formats it using localtime() to retrieve a human-readable timestamp. It writes a header line at the top of the log file, indicating the update time including the hour, minute, month, day, and year. After the header, the function iterates through the log_entries array, writing each log entry line by line. Each line contains the BankID, the list of deposit and withdrawal operations that occurred during the session, and the final balance after all operations. After completing the log entries, it writes a termination line ## end of log. to mark the end of the file. Finally, the file is closed. This function ensures that all account activities during a server session are persistently recorded, allowing accurate recovery on the next server startup.



```c
void write_log_file() {
    FILE *fp = fopen("AdaBank.bankLog", "w");
    if (!fp) {
        perror("log fopen");
        return;
    }
    time_t now = time(NULL);
    struct tm *t = localtime(&now);
    fprintf(fp, "# Adabank Log file updated @%02d:%02d %s %d %d\n\n",
        t->tm_hour, t->tm_min,
        (t->tm_mon == 3) ? "April" : "Month", t->tm_mday, 1900 + t->tm_year);

    for (int i = 0; i < num_logs; i++) {
        fprintf(fp, "# %s %s %d\n",
            log_entries[i].bank_id,
            log_entries[i].operations,
            log_entries[i].last_balance);
    }

    fprintf(fp, "\n## end of log.\n");
    fclose(fp);
}
```

2.2) teller.c:

2.2.0) void teller_function(void *arg): The teller_function is executed by each forked teller process and is responsible for preparing client requests for the bank server to process. It receives a generic pointer argument, which is cast into a ClientRequest structure representing the client's transaction request. The teller first obtains access to the shared memory region by calling get_shared_memory(), and retrieves both the shared memory access semaphore and the request semaphore through get_semaphore() and get_request_semaphore() respectively. The teller process begins by calling sem_wait(sem_request), ensuring that it waits until the server signals readiness to accept a new request. Once the server is ready, the teller acquires exclusive access to the shared memory by calling sem_wait(sem). Inside the critical section, the teller writes the client's request into the shared memory, sets the request_ready flag to 1 to indicate that a new request is available, and releases the shared memory lock by calling sem_post(sem). After completing these steps, the teller process calls exit(0) to terminate.



```c
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include "utils.h"

void teller_function(void *arg) {
    ClientRequest req = *(ClientRequest*)arg;
    SharedMemory *shm = get_shared_memory();
    sem_t *sem = get_semaphore();
    sem_t *sem_request = get_request_semaphore();

    sem_wait(sem_request);

    sem_wait(sem);
    shm->request = req;
    shm->request_ready = 1;
    sem_post(sem);

    exit(0);
}

```

2.3) utils.c:

2.3.0) global variables: In utils.c, three static pointers are defined globally: SharedMemory *shm, sem_t *sem, and BankDatabase *db. These pointers are initialized and managed inside the utility functions and represent the shared memory region, the semaphore for controlling access to the database, and the in-memory copy of the bank's database, respectively. Declaring them as static ensures that they are visible only within this source file and allows all functions inside utils.c to access and modify these shared structures consistently without exposing globally across the program.

2.3.1) BankDatabase* init_database(): The init_database function is responsible for initializing the in-memory bank database. It allocates memory dynamically for a BankDatabase structure using malloc() and clears all fields using memset() to zero-initialize the newly allocated memory block. This ensures that the accounts array is initially empty and that the number of accounts is set to zero. The initialized database pointer is assigned to the static db pointer and returned for use by the caller. It holds all active bank accounts and their balances during the server operations.

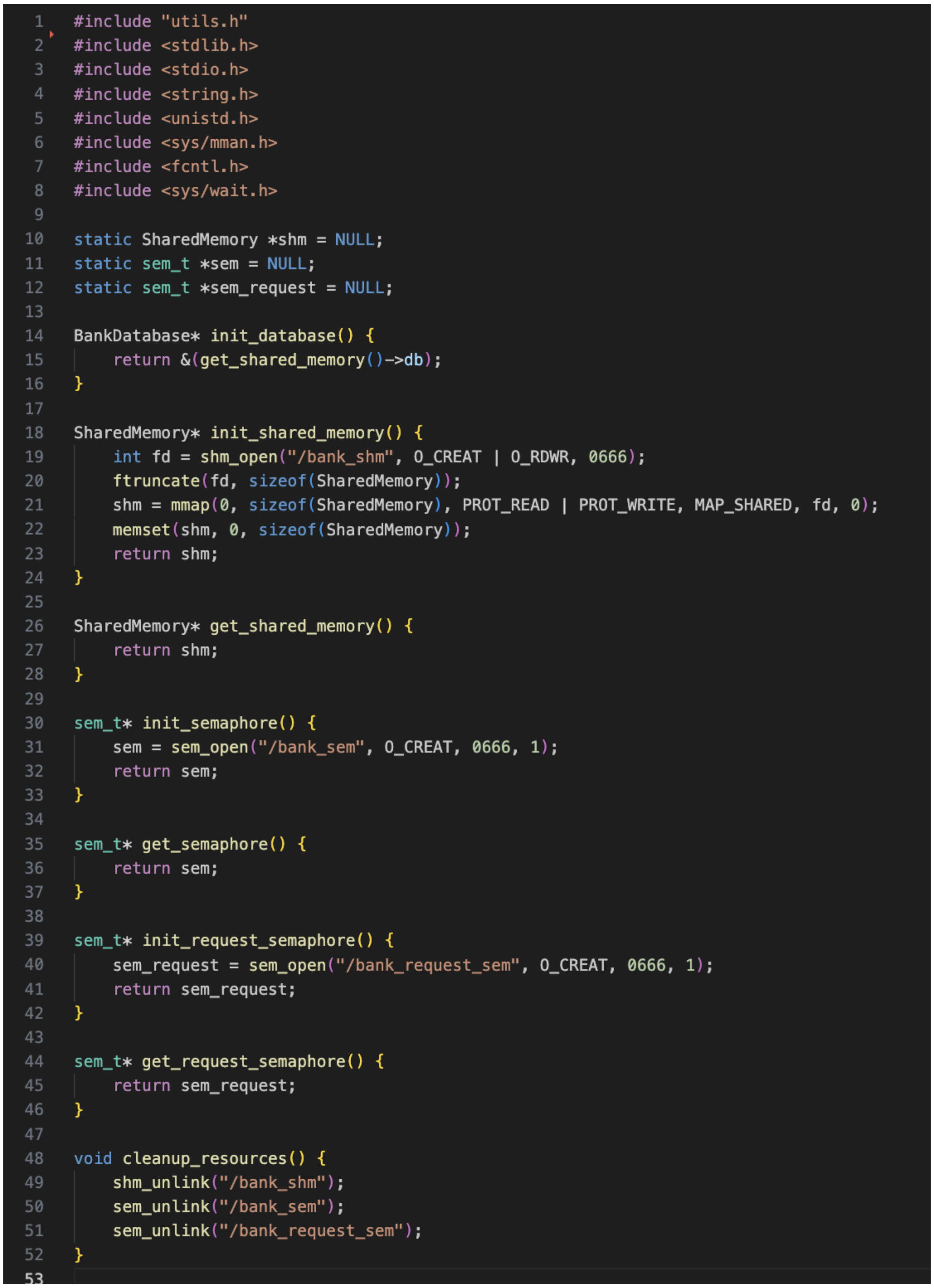
2.3.2) SharedMemory* init_shared_memory(): The init_shared_memory function sets up the shared memory object used for server-client communication and synchronization. It opens or creates a POSIX shared memory object with the name /bank_shm using shm_open() with read and write permissions. It then resizes the shared memory segment to the size of the SharedMemory structure with ftruncate() and maps it into the process's address space using mmap(). The mapped memory pointer is stored in the static shm pointer and returned. This shared memory object is used to share common information safely.

2.3.3) SharedMemory* get_shared_memory(): The get_shared_memory function simply returns the static shm pointer that holds the address of the previously mapped shared memory region. It provides other parts of the program access to the shared memory without needing to remap or reopen it.

2.3.4) sem_t* init_semaphore(): The init_semaphore function initializes the main semaphore used for controlling access to the shared memory bank database. It opens or creates a named POSIX semaphore with the name /bank_sem, with read and write permissions and an initial value of 1. This value ensures that only one process can access the shared memory at a time. The created semaphore pointer is assigned to the static sem variable and returned to the caller.

2.3.5) sem_t* get_semaphore(): The get_semaphore function simply returns the static sem pointer that holds the reference to the initialized semaphore object. It allows other parts of the program to synchronize access to the shared memory safely.
2.3.6) void cleanup_resources(): The cleanup_resources function is responsible for cleaning up all persistent IPC resources at the end of the server's execution. It calls shm_unlink("/bank_shm") to unlink the shared memory object and calls sem_unlink("/bank_sem") to unlink the main semaphore. It also frees the

dynamically allocated BankDatabase structure by calling free(db). This function ensures no system resources are leaked after server process terminates.

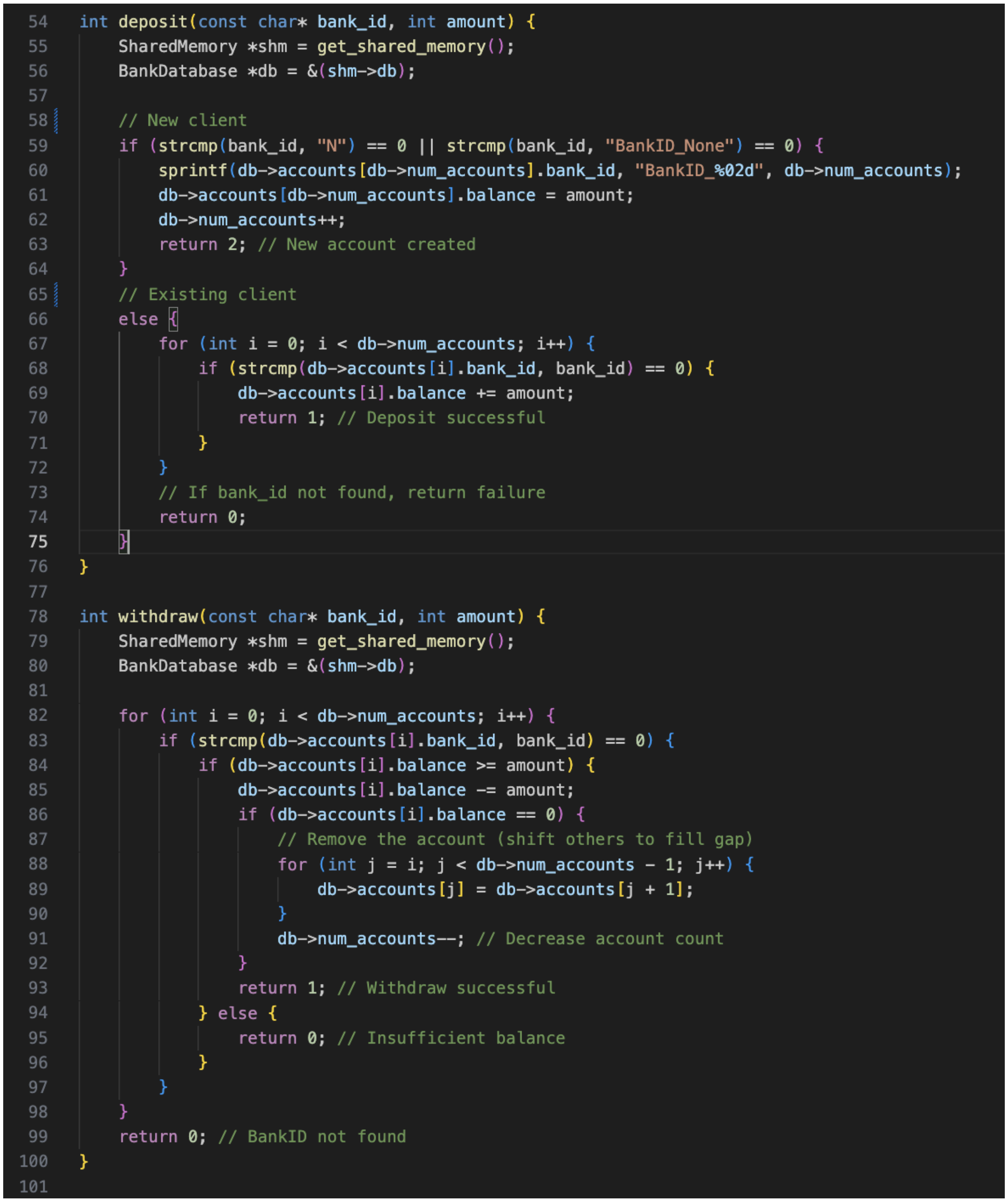


```c
#include "utils.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/wait.h>

static SharedMemory *shm = NULL;
static sem_t *sem = NULL;
static sem_t *sem_request = NULL;

BankDatabase* init_database() {
    return &(get_shared_memory()->db);
}

SharedMemory* init_shared_memory() {
    int fd = shm_open("/bank_shm", O_CREAT | O_RDWR, 0666);
    ftruncate(fd, sizeof(SharedMemory));
    shm = mmap(0, sizeof(SharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    memset(shm, 0, sizeof(SharedMemory));
    return shm;
}

SharedMemory* get_shared_memory() {
    return shm;
}

sem_t* init_semaphore() {
    sem = sem_open("/bank_sem", O_CREAT, 0666, 1);
    return sem;
}

sem_t* get_semaphore() {
    return sem;
}

sem_t* init_request_semaphore() {
    sem_request = sem_open("/bank_request_sem", O_CREAT, 0666, 1);
    return sem_request;
}

sem_t* get_request_semaphore() {
    return sem_request;
}

void cleanup_resources() {
    shm_unlink("/bank_shm");
    sem_unlink("/bank_sem");
    sem_unlink("/bank_request_sem");
}
```

2.3.7) int deposit(const char* bank_id, int amount): The deposit function handles deposit operations requested by clients. If the bank ID provided is "N" or

"BankID_None", indicating a new account creation, the function generates a new BankID based on the current number of accounts, initializes the account with the given deposit amount, and increments the number of accounts. If the bank ID already exists, the function locates the account by comparing BankIDs and adds the deposited amount to the account's existing balance. It returns 2 if a new account is created, 1 if an existing account is successfully updated, or 0 if the BankID could not be found.

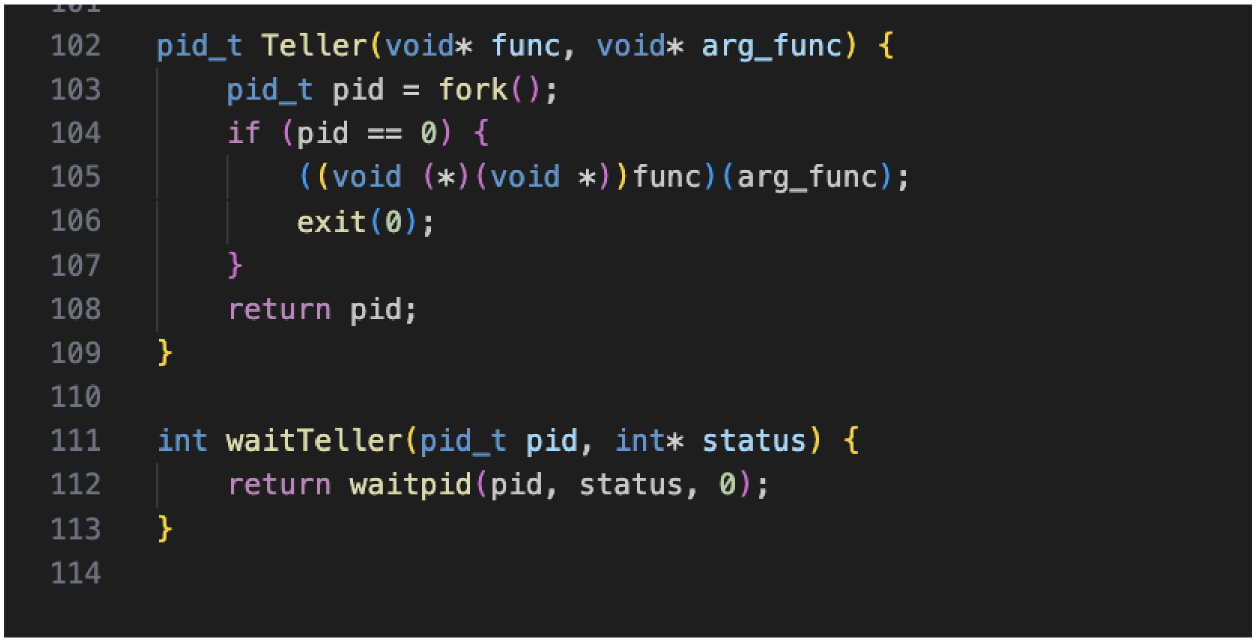
2.3.8) int withdraw(const char* bank_id, int amount): The withdraw function handles withdrawal operations requested by clients. It searches the bank database for the provided BankID and checks if the account's balance is sufficient for the requested withdrawal amount. If sufficient, it deducts the amount from the balance. If the resulting balance is zero after the withdrawal, the function removes the account from the accounts array by shifting the remaining accounts left by one index and decrements the total account count. It returns 1 for successful withdrawal and 0 if the withdrawal fails due to insufficient funds or a non-existing BankID.



```c
int deposit(const char* bank_id, int amount) {
    SharedMemory *shm = get_shared_memory();
    BankDatabase *db = &(shm->db);

    // New client
    if (strcmp(bank_id, "N") == 0 || strcmp(bank_id, "BankID_None") == 0) {
        sprintf(db->accounts[db->num_accounts].bank_id, "BankID_%02d", db->num_accounts);
        db->accounts[db->num_accounts].balance = amount;
        db->num_accounts++;
        return 2; // New account created
    }
    // Existing client
    else {
        for (int i = 0; i < db->num_accounts; i++) {
            if (strcmp(db->accounts[i].bank_id, bank_id) == 0) {
                db->accounts[i].balance += amount;
                return 1; // Deposit successful
            }
        }
        // If bank_id not found, return failure
        return 0;
    }
}

int withdraw(const char* bank_id, int amount) {
    SharedMemory *shm = get_shared_memory();
    BankDatabase *db = &(shm->db);

    for (int i = 0; i < db->num_accounts; i++) {
        if (strcmp(db->accounts[i].bank_id, bank_id) == 0) {
            if (db->accounts[i].balance >= amount) {
                db->accounts[i].balance -= amount;
                if (db->accounts[i].balance == 0) {
                    // Remove the account (shift others to fill gap)
                    for (int j = i; j < db->num_accounts - 1; j++) {
                        db->accounts[j] = db->accounts[j + 1];
                    }
                    db->num_accounts--; // Decrease account count
                }
                return 1; // Withdraw successful
            } else {
                return 0; // Insufficient balance
            }
        }
    }
    return 0; // BankID not found
}

```

2.3.9) pid_t Teller(void* func, void* arg_func): The Teller function forks a new process to serve a client request by executing the provided teller function. It calls fork(), and in the child process, casts and calls the given function pointer with the provided argument. After executing the teller function, the child process exits. In the parent process, it simply returns the child's PID.

2.3.10) int waitTeller(pid_t pid, int* status): The waitTeller function waits for a teller child process to terminate by calling waitpid() with the child process's PID. It ensures that the server does not proceed until the teller handling a specific request has finished, maintaining orderly processing and preventing resource leaks from orphaned processes.



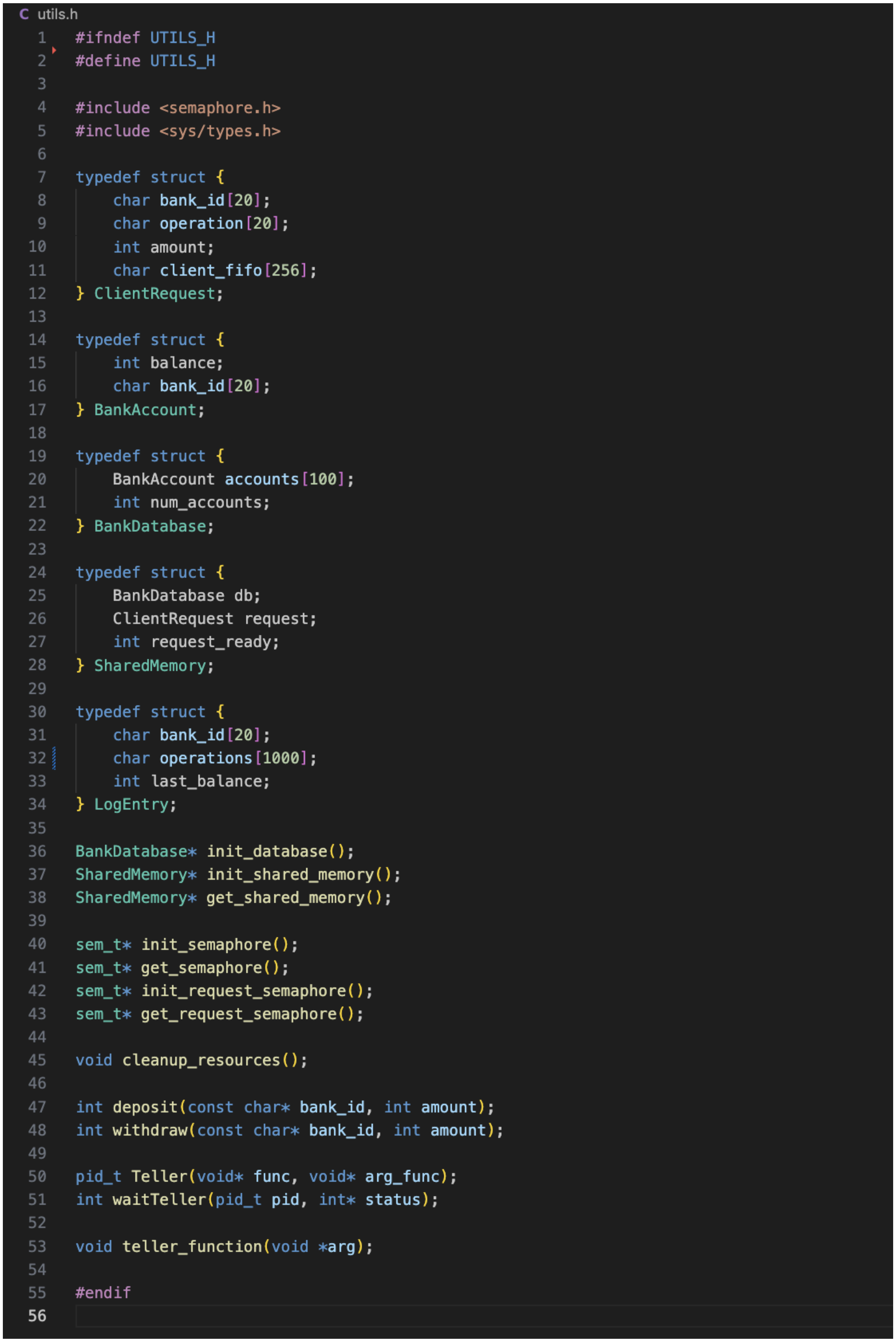
```c
pid_t Teller(void* func, void* arg_func) {
    pid_t pid = fork();
    if (pid == 0) {
        ((void (*)(void *))func)(arg_func);
        exit(0);
    }
    return pid;
}

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

2.4) utils.h:

2.0) Definitions and Typedefs: In utils.h, several structure definitions are provided to standardize the communication and data storage formats across different modules of the project. The ClientRequest structure defines the format of a request that a client sends to the server, containing fields for the bank ID, the requested operation (deposit or withdraw), the amount involved, and the path to the client FIFO. The BankAccount structure represents an individual account in the banking database, storing the account’s balance and its associated BankID. The BankDatabase structure represents the overall banking database, containing an array of BankAccount entries and an integer num_accounts to track how many accounts are currently active. The SharedMemory structure holds a single ClientRequest and used to structure the shared memory segment between the server and teller processes.

2.1) Function Declarations: The utils.h provides function prototypes for initializing and managing key system components. Functions like init_database(), init_shared_memory(), init_semaphore(), and init_request_semaphore() are declared to initialize the bank database, shared memory, and semaphores, respectively. Corresponding getter functions such as get_shared_memory() and get_semaphore() are declared to allow access to already initialized shared resources. The cleanup_resources() function is declared for releasing shared memory and semaphores when the server exits.

It declares two core banking operations, deposit(const char* bank_id, int amount) and withdraw(const char* bank_id, int amount), which handle the actual modification of account balances based on client transactions. Process control functions Teller(void* func, void* arg_func) and waitTeller(pid_t pid, int* status) are also declared, which handle the creation and synchronization of teller processes serving client requests. Lastly, the teller worker function teller_function(void *arg) is declared to define the operation that each teller child process will perform.



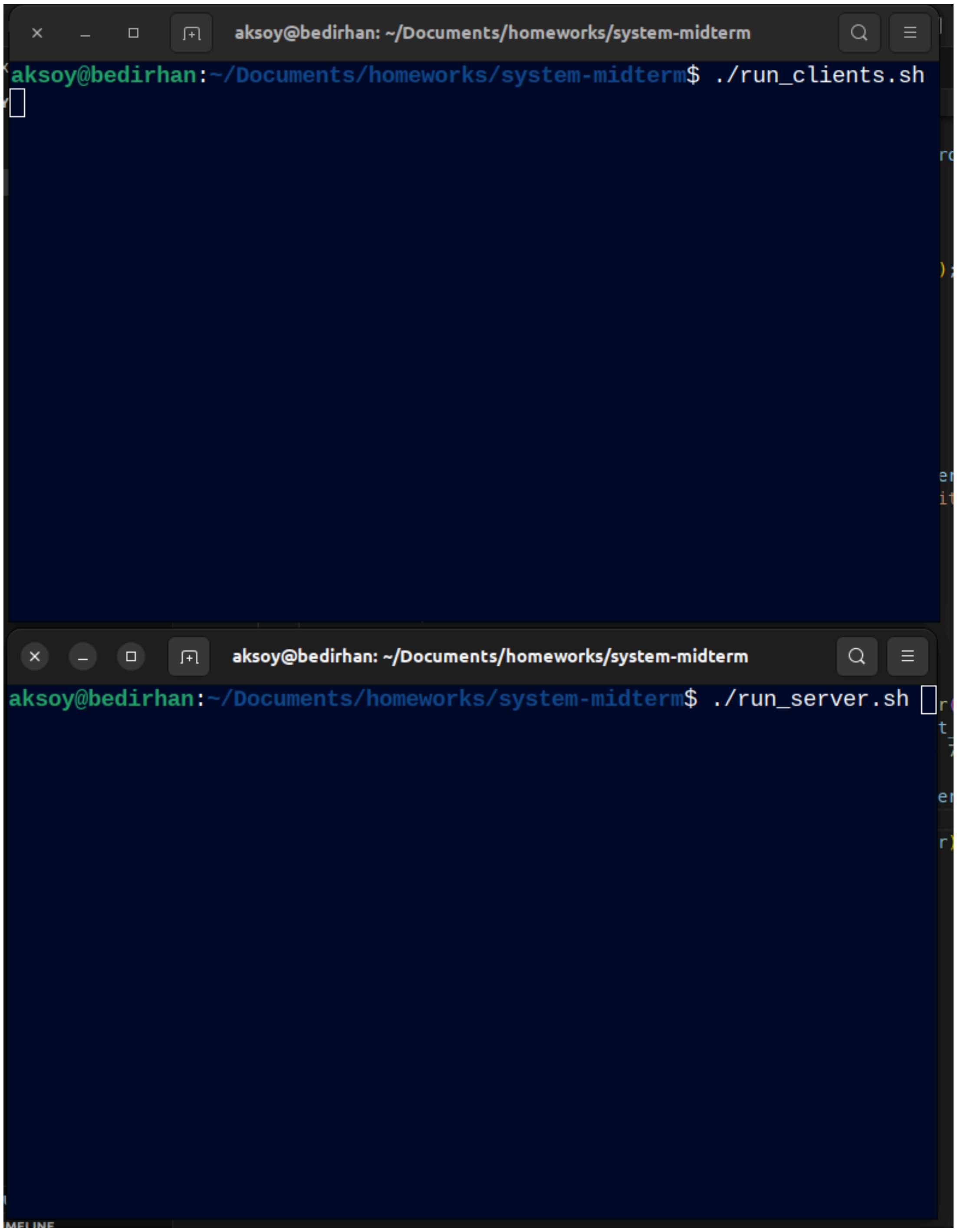
utils.h

```c
#ifndef UTILS_H
#define UTILS_H

#include <semaphore.h>
#include <sys/types.h>

typedef struct {
    char bank_id[20];
    char operation[20];
    int amount;
    char client_fifo[256];
} ClientRequest;

typedef struct {
    int balance;
    char bank_id[20];
} BankAccount;

typedef struct {
    BankAccount accounts[100];
    int num_accounts;
} BankDatabase;

typedef struct {
    BankDatabase db;
    ClientRequest request;
    int request_ready;
} SharedMemory;

typedef struct {
    char bank_id[20];
    char operations[1000];
    int last_balance;
} LogEntry;

BankDatabase* init_database();
SharedMemory* init_shared_memory();
SharedMemory* get_shared_memory();

sem_t* init_semaphore();
sem_t* get_semaphore();
sem_t* init_request_semaphore();
sem_t* get_request_semaphore();

void cleanup_resources();

int deposit(const char* bank_id, int amount);
int withdraw(const char* bank_id, int amount);

pid_t Teller(void* func, void* arg_func);
int waitTeller(pid_t pid, int* status);

void teller_function(void *arg);

#endif

```

3) Screenshots

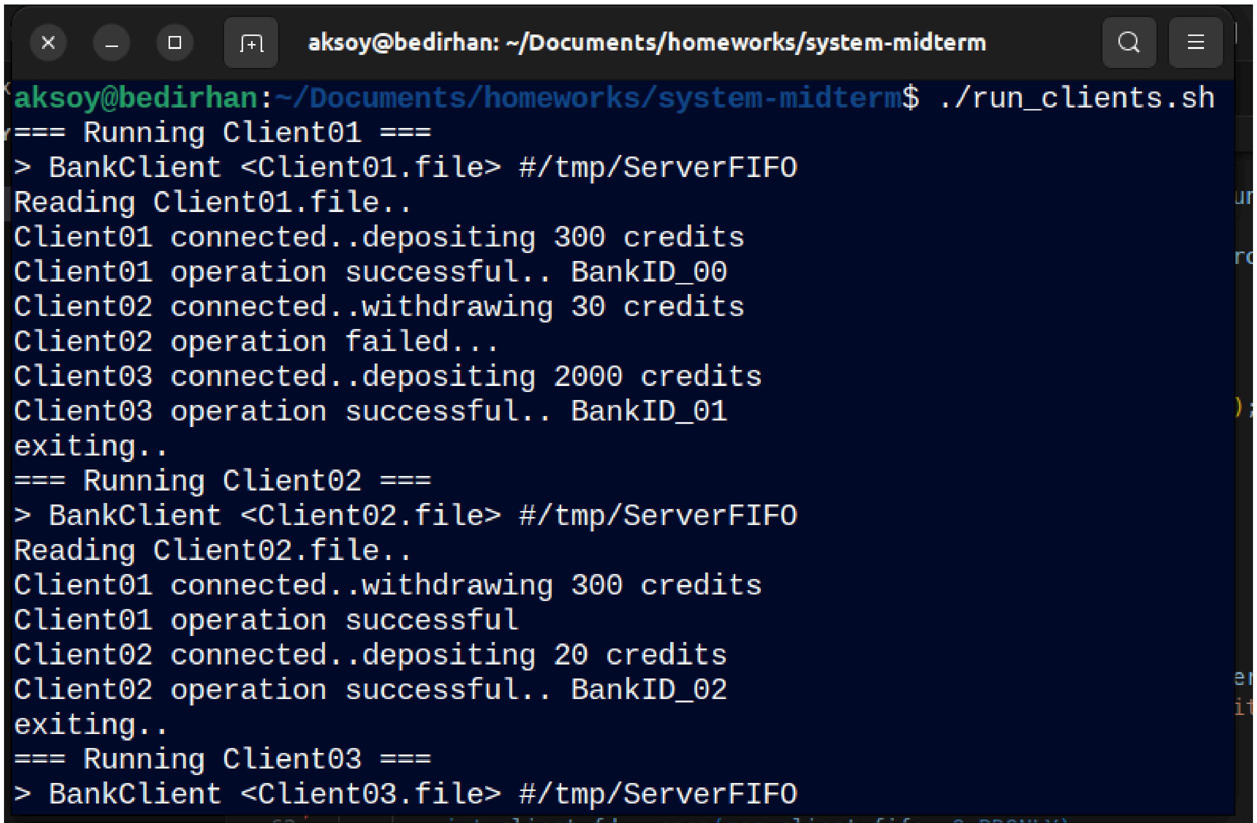
Testing program in my Ubuntu and Debian 12 machines:

First run the ./run_server.sh after giving execute permissions (it will perform the make), then run client in a different console:

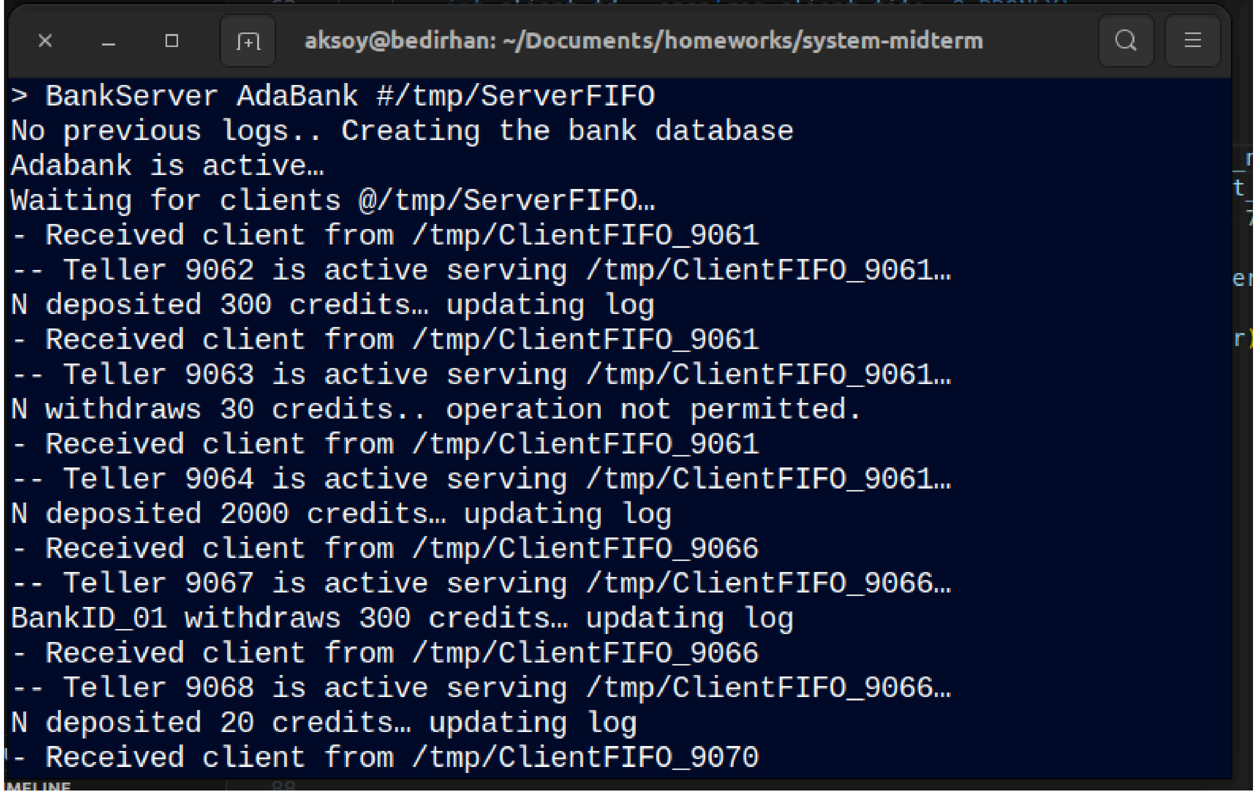


```
aksoy@bedirhan:~/Documents/homeworks/system-midterm$ ./run_clients.sh
```

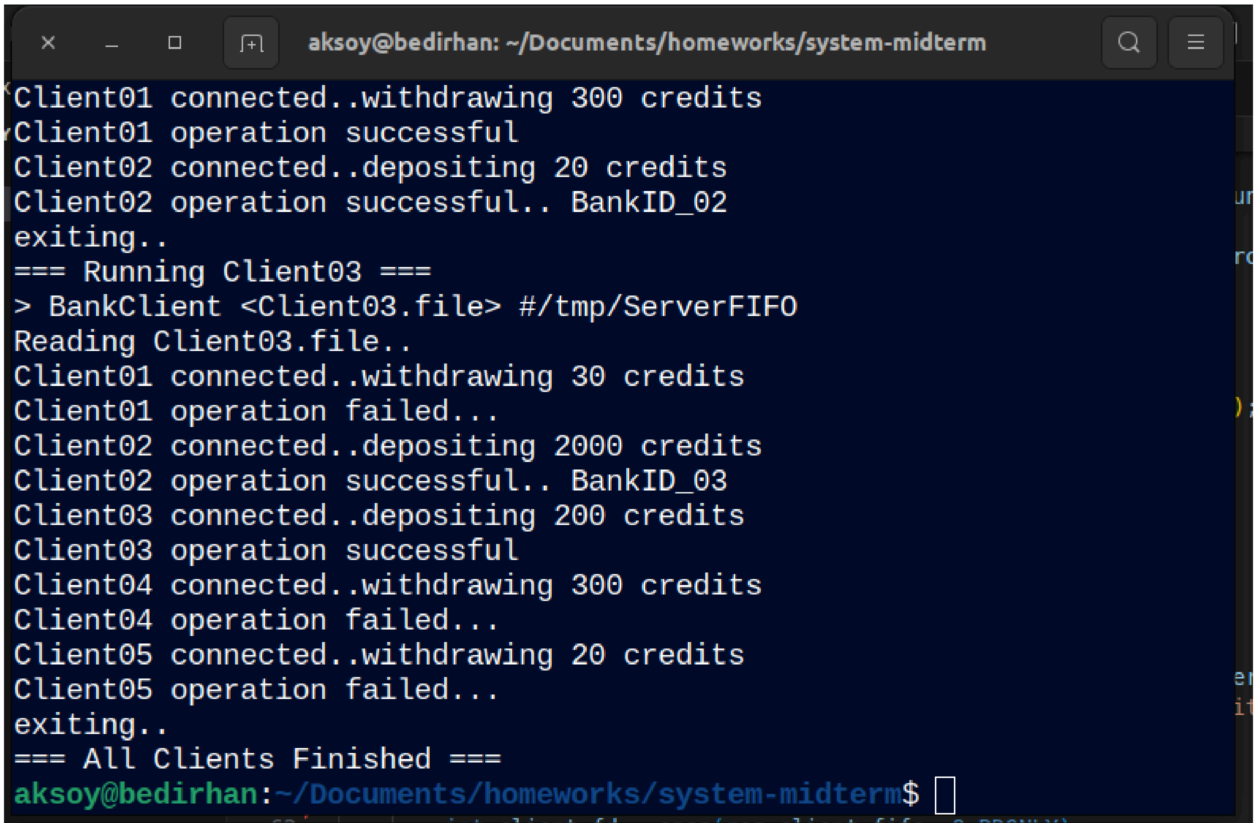
```
aksoy@bedirhan:~/Documents/homeworks/system-midterm$ ./run_server.sh
```

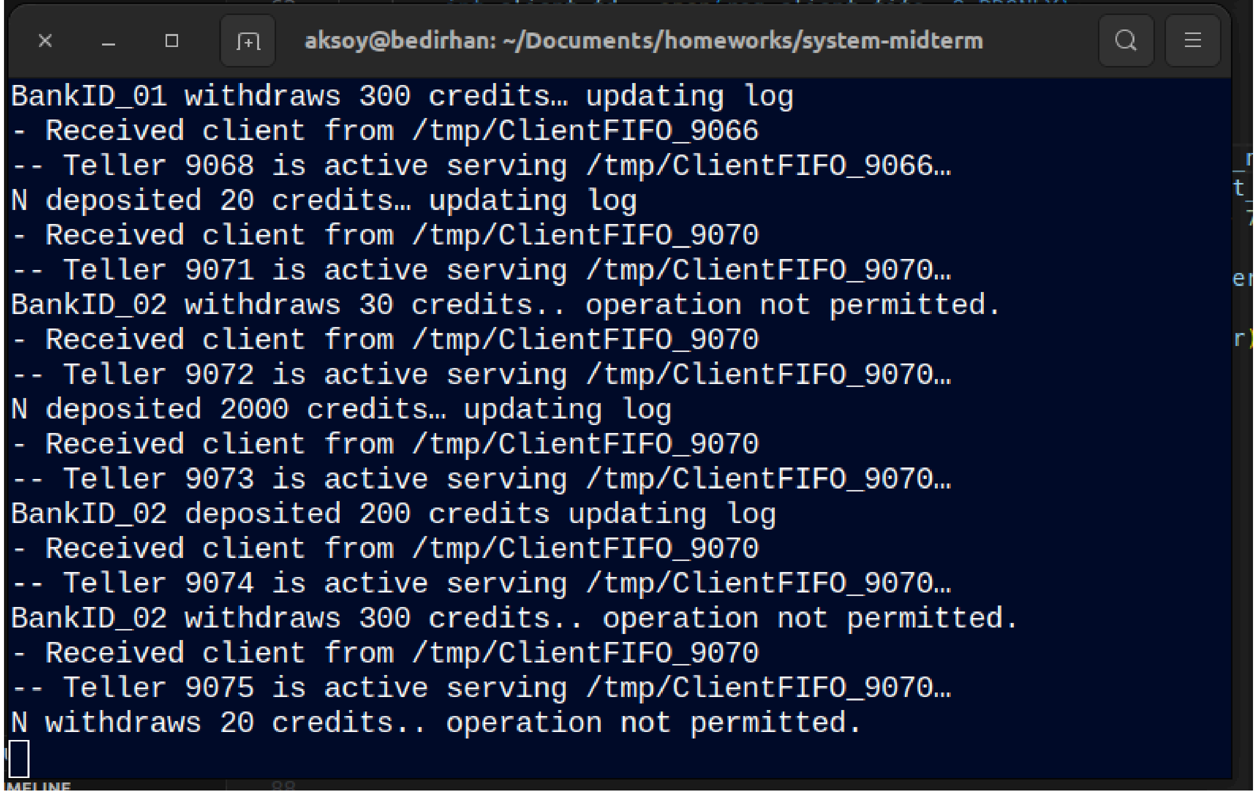

aksoy@bedirhan: ~/Documents/homeworks/system-midterm

```
aksoy@bedirhan:~/Documents/homeworks/system-midterm$ ./run_clients.sh
=== Running Client01 ===
> BankClient <Client01.file> #/tmp/ServerFIFO
Reading Client01.file..
Client01 connected..depositing 300 credits
Client01 operation successful.. BankID_00
Client02 connected..withdrawing 30 credits
Client02 operation failed...
Client03 connected..depositing 2000 credits
Client03 operation successful.. BankID_01
exiting..
=== Running Client02 ===
> BankClient <Client02.file> #/tmp/ServerFIFO
Reading Client02.file..
Client01 connected..withdrawing 300 credits
Client01 operation successful
Client02 connected..depositing 20 credits
Client02 operation successful.. BankID_02
exiting..
=== Running Client03 ===
> BankClient <Client03.file> #/tmp/ServerFIFO
```


aksoy@bedirhan: ~/Documents/homeworks/system-midterm

```
> BankServer AdaBank #/tmp/ServerFIFO
No previous logs.. Creating the bank database
Adabank is active…
Waiting for clients @/tmp/ServerFIFO…
- Received client from /tmp/ClientFIFO_9061
-- Teller 9062 is active serving /tmp/ClientFIFO_9061…
N deposited 300 credits… updating log
- Received client from /tmp/ClientFIFO_9061
-- Teller 9063 is active serving /tmp/ClientFIFO_9061…
N withdraws 30 credits.. operation not permitted.
- Received client from /tmp/ClientFIFO_9061
-- Teller 9064 is active serving /tmp/ClientFIFO_9061…
N deposited 2000 credits… updating log
- Received client from /tmp/ClientFIFO_9066
-- Teller 9067 is active serving /tmp/ClientFIFO_9066…
BankID_01 withdraws 300 credits… updating log
- Received client from /tmp/ClientFIFO_9066
-- Teller 9068 is active serving /tmp/ClientFIFO_9066…
N deposited 20 credits… updating log
- Received client from /tmp/ClientFIFO_9070
```


```
Client01 connected..withdrawing 300 credits
Client01 operation successful
Client02 connected..depositing 20 credits
Client02 operation successful.. BankID_02
exiting..
=== Running Client03 ===
> BankClient <Client03.file> #/tmp/ServerFIFO
Reading Client03.file..
Client01 connected..withdrawing 30 credits
Client01 operation failed...
Client02 connected..depositing 2000 credits
Client02 operation successful.. BankID_03
Client03 connected..depositing 200 credits
Client03 operation successful
Client04 connected..withdrawing 300 credits
Client04 operation failed...
Client05 connected..withdrawing 20 credits
Client05 operation failed...
exiting..
=== All Clients Finished ===
aksoy@bedirhan:~/Documents/homeworks/system-midterm$
```


```
BankID_01 withdraws 300 credits… updating log
- Received client from /tmp/ClientFIFO_9066
-- Teller 9068 is active serving /tmp/ClientFIFO_9066…
N deposited 20 credits… updating log
- Received client from /tmp/ClientFIFO_9070
-- Teller 9071 is active serving /tmp/ClientFIFO_9070…
BankID_02 withdraws 30 credits.. operation not permitted.
- Received client from /tmp/ClientFIFO_9070
-- Teller 9072 is active serving /tmp/ClientFIFO_9070…
N deposited 2000 credits… updating log
- Received client from /tmp/ClientFIFO_9070
-- Teller 9073 is active serving /tmp/ClientFIFO_9070…
BankID_02 deposited 200 credits updating log
- Received client from /tmp/ClientFIFO_9070
-- Teller 9074 is active serving /tmp/ClientFIFO_9070…
BankID_02 withdraws 300 credits.. operation not permitted.
- Received client from /tmp/ClientFIFO_9070
-- Teller 9075 is active serving /tmp/ClientFIFO_9070…
N withdraws 20 credits.. operation not permitted.
```

Client01 (Client01.file):

1. **Deposit 300** (bank_id = N):
   No account exists, a new account created. BankID_00 assigned, balance = 300.

2. **Withdraw 30** (bank_id = N):
   No valid account because "N" is not linked to any BankID. Operation failed.

3. **Deposit 2000** (bank_id = N):
   Again treated as a new account request. A new account created. BankID_01 assigned, balance = 2000.

Client02 (Client02.file):

1. **Withdraw 300** (bank_id = BankID_01):
   BankID_01 exists with balance 2000. Withdraw 300 successful. New balance = 1700.

2. **Deposit 20** (bank_id = N):
   Treated as new account creation again (because BankID not specified correctly).
   New account created. BankID_02 assigned, balance = 20.

Client03 (Client03.file):

1. **Withdraw 30** (bank_id = BankID_02):
   BankID_02 exists with balance 20, but not enough for 30. Operation failed.

2. **Deposit 2000** (bank_id = N):
   New account created. BankID_03 assigned, balance = 2000.

3. **Deposit 200** (bank_id = BankID_02):
   BankID_02 exists with balance 20. After deposit, new balance = 220.

4. **Withdraw 300** (bank_id = BankID_02):
   BankID_02 has 220 balance, not enough for 300. Operation failed.

5. **Withdraw 20** (bank_id = N):
   "N" means no associated account. Operation failed.

Final account states:

- BankID_00 → balance = 300
- BankID_01 → balance = 1700
- BankID_02 → balance = 220
- BankID_03 → balance = 2000

4) Conclusion

I encountered several challenges related to synchronization and maintaining consistent communication between the bankserver, teller processes, and clients. One of the issues I faced was ensuring that all shared memory and semaphore objects were properly cleaned and they recreated on every server startup. At first, the server was not resetting the database correctly when the previous shared memory and semaphores were left in the system which causing unexpected behavior when new clients connected. I solved this by unlinking old shared memory and semaphores before initializing them again during server startup.

Another challenge was implementing a persistent logging system. Initially, I was only logging operations at runtime but not reconstructing the bank database correctly from the previous session's log file. I realized that I needed a parser that could read the structured log and rebuild accounts accurately. When handling multiple operations on the same account during a session, I had to develop a way to dynamically update existing log entries instead of duplicating them that required an intermediate in-memory log representation.

Process synchronization between the tellers and the server also difficult. In the early stages, the teller processes were directly performing operations on the shared memory that lead to potential race conditions. I modified the design so that tellers only post requests into shared memory and the main server process is responsible for safely processing each request. This change simplified synchronization and made the system more robust.

Ensuring that BankIDs remained consistent and new accounts were correctly assigned after deposits from unknown clients was another point where errors occurred early. In some cases, clients could continue using outdated BankIDs after a server restart and it lead to transaction failures. I solved this by dynamically updating the client's current_bank_id after receiving a deposit confirmation from the server.