CSE 344 Systems Programming

Midterm - Bank Simulator Project

Student Information

Name: Abdullah GÖKTÜRK
Student Number: 200104004042

Introduction

This homework implements a Bank Simulator in C programming language using Linux system calls and inter-process communication mechanisms. The program simulates a banking environment with a server process managing accounts, teller processes handling individual operations, and client processes submitting banking transactions. The system demonstrates advanced concepts in systems programming including process creation, pipes, FIFOs, signal handling, and synchronization techniques.

Run Commands

- `make` - Compiles the program and creates client files
- `make val` - Compiles the program with Valgrind support
- `make create_client_files` - Creates the client files (Client1.file, Client2.file, Client3.file)
- `make run_server` - Starts the AdaBank server
- `make val_server` - Starts the AdaBank server with Valgrind
- `make run_client1` - Runs client1 with operations from Client1.file
- `make run_client2` - Runs client2 with operations from Client2.file
- `make run_client3` - Runs client3 with operations from Client3.file
- `make val_test` - Runs a comprehensive test that executes all 3 client files in sequence
- `make clean` - Cleans object files, executables, and FIFOs
- `distclean` - Clean including valgrind logs

The server process should be started first, followed by client processes in separate terminals. The server will display waiting messages until clients connect, then show details of each transaction as it processes them.

System Overview

At its heart, AdaBank implements a client-server architecture where multiple client processes send banking requests to a central server. The server then delegates these operations to specialized teller processes that perform the actual account manipulations.

Architecture Design

Our banking system comprises three main component types that work together through various communication channels:

The main server process acts as the central coordinator, maintaining the bank database, accepting client connections through a named pipe (FIFO), and spawning teller processes to handle individual operations. It's responsible for database integrity and proper synchronization between concurrent operations.

Teller processes are temporary workers created by the server using process creation. Each teller handles exactly one banking operation - either a deposit or withdrawal. They communicate with the server through unnamed pipes for database operations and with clients through named pipes for responses. This separation of concerns allows for better concurrency.

Client processes read operation instructions from client files, connect to the server, and send operation batches. They create unique FIFOs for each operation and wait for responses from tellers. Tellers communicate directly with client processes - they send operation results back through client-specific FIFOs, forming a complete communication triangle.

All transactions are recorded in a persistent log file that serves as our database. When the server starts, it reconstructs the entire account state from this log, ensuring data durability across restarts.

Implementation Details

The server uses some interesting systems programming techniques to achieve concurrency and reliability:

Process creation is abstracted through a custom `Teller()` function that handles the process creation and proper setup of child processes. This makes the code cleaner and more maintainable:

```
pid_t Teller(void* func, void* arg_func) {
    pid_t pid = fork();

    if (pid == -1) {
        errLog(logFile, "Teller: process creation failed");
        return -1;
    } else if (pid == 0) {
        /* Child process - call the teller function */
        void *(*tellerFunc)(void *) = func;
        tellerFunc(arg_func);
        exit(EXIT_SUCCESS);
    }
    free(arg_func); /* Free the argument passed to the teller */
    return pid;
}
```

For communication between processes, I use a combination of named pipes (FIFOs) for client-server communication and unnamed pipes for teller-server communication. This allows for efficient, bidirectional data exchange. I protect critical sections using semaphores, particularly when accessing the database or logging transactions.

One of the most interesting aspects is how I handle multiple concurrent tellers. The server creates all pipes and spawns all teller processes at once, then uses select() to efficiently multiplex I/O operations without blocking:

```
do {
    FD_ZERO(&readfds);
    maxfd = -1;
    remaining_tellers = 0;

    /* Add all active teller read pipes to the set */
    for (int i = 0; i < currentBatch.received; i++) {
        if (!teller_completed[i] && pipes[i][2] != -1) {
            FD_SET(pipes[i][2], &readfds);
            if (pipes[i][2] > maxfd) {
                maxfd = pipes[i][2];
            }
            remaining_tellers++;
        }
    }

    int select_result = select(maxfd + 1, &readfds, NULL, NULL, &tv);
    // Process any ready file descriptors
} while (remaining_tellers > 0);
```

This approach allows the server to handle multiple teller communications simultaneously, improving throughput and responsiveness.

Design Decisions and Challenges

I faced several interesting challenges during implementation:

My batch processing approach was a critical decision. Initially, operations were processed sequentially, creating bottlenecks. I redesigned the system to collect all operations from a client, create tellers simultaneously, and process communications in parallel. This significantly improved performance.

Account identity management was tricky because of confusion between "new accounts" (N) and specific account references (BankID_XX). I fixed this by only updating IDs for new account creations, not for all operations:

```
/* CRITICAL CHANGE: Only update 'N' operations with the new BankID */
if (strcmp(op->bankId, "N") == 0) {
    /* Update the current operation's bankId */
    strncpy(op->bankId, resp->bankId, sizeof(op->bankId) - 1);
    op->bankId[sizeof(op->bankId) - 1] = '\0';
}
```

FIFO coordination presented deadlock risks since blocking FIFO operations could cause the system to hang. I implemented non-blocking I/O with retry mechanisms and timeouts to address this:

```
/* Try to open in non-blocking mode first with retries */
for (int attempt = 0; attempt < 10 && clientFd == -1; attempt++) {
    clientFd = open(clientFifo, O_WRONLY | O_NONBLOCK);
```

```
    if (clientFd == -1) {
        if (errno == ENXIO) {
            /* No reader yet, sleep briefly and retry */
            usleep(50000); /* 50ms */
        } else {
            /* Other error */
            break;
        }
    } else {
        /* Success - switch back to blocking mode */
        int flags = fcntl(clientFd, F_GETFL);
        fcntl(clientFd, F_SETFL, flags & ~O_NONBLOCK);
        break;
    }
}
```

Log file management needed careful handling to ensure proper formatting and prevent data loss during restarts. I implemented append mode and formatted logging to match the required structure.

Test Plan and Results

I developed a comprehensive test plan to verify system correctness and the results demonstrate that the implementation meets all requirements:

Database Persistence Test

I tested the database persistence by running operations and then verifying the log file contents. After server restarts, I confirmed that the account state was properly restored from the log. Here's a sample of the log file showing persistence:

```
# AdaBank Log file updated @06:32:36
BankID_01 D 300 300
BankID_02 D 2000 2000
BankID_01 W 300 0
BankID_03 D 20 20
# AdaBank Log file updated @06:33 April 28 2025

BankID_02 D 0 2000
BankID_03 D 0 20

## end of log.
# AdaBank Log file updated @06:32:36
# AdaBank Log file updated @07:08 April 28 2025

BankID_02 D 0 2000
BankID_03 D 0 20

## end of log.
```

The log shows successful deposits and withdrawals with proper transaction recording. The server correctly records the final state of accounts during shutdown with the proper end-of-log marker.

Concurrent Operation Test

I tested concurrency by running multiple client operations simultaneously. The following output from Client3 shows how multiple operations are processed in parallel:

```
$ make run_client3
./BankClient Client3.file ServerFIFO_Name
Reading Client3.file..
5 clients to connect.. creating clients..
Connected to Adabank..
Client01 connected..withdrawing 30 credits
Client02 connected..depositing 2000 credits
Client03 connected..depositing 200 credits
Client04 connected..withdrawing 300 credits
Client05 connected..withdrawing 20 credits
Client05 something went WRONG: New clients cannot withdraw. Please deposit
first.
Client01 something went WRONG: Account not found
Client02 served.. BankID_01
Client03 something went WRONG: Account not found
Client04 something went WRONG: Account not found
exiting..
```

This output demonstrates that the system can handle multiple operations from a single client file, properly executing them and returning appropriate responses.

Validation Rules Test

I tested the banking validation rules to ensure they were properly enforced. The following output demonstrates validation of rules like "new clients cannot withdraw" and "insufficient funds":

```
$ make run_client3
./BankClient Client3.file ServerFIFO_Name
Reading Client3.file..
6 clients to connect.. creating clients..
Connected to Adabank..
Client01 connected..withdrawing 30 credits
Client02 connected..depositing 2000 credits
Client03 connected..depositing 200 credits
Client04 connected..withdrawing 300 credits
Client05 connected..depositing 2000 credits
Client06 connected..withdrawing 20 credits
Client06 something went WRONG: New clients cannot withdraw. Please deposit
first.
Client01 served.. BankID_02
Client02 served.. BankID_10
Client03 served.. BankID_02
```

```
Client04 something went WRONG: Insufficient funds for withdrawal
Client05 something went WRONG: Account not found
exiting..
```

The output shows that new clients are prevented from withdrawing (Client06), and withdrawals exceeding account balance are rejected (Client04).

Resource Management Test

I used Valgrind to verify that the system properly manages resources without leaks. The results confirm proper resource cleanup:

```
==32552==
==32552== LEAK SUMMARY:
==32552==    definitely lost: 0 bytes in 0 blocks
==32552==    indirectly lost: 0 bytes in 0 blocks
==32552==      possibly lost: 0 bytes in 0 blocks
==32552==    still reachable: 70 bytes in 2 blocks
==32552==         suppressed: 0 bytes in 0 blocks
==32552==
==32552== For lists of detected and suppressed errors, rerun with: -s
==32552== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The valgrind results show no memory leaks, with only 70 bytes still reachable (likely from system libraries) and zero errors reported.

Error Recovery Test

I tested error recovery by forcibly terminating some client connections and observing how the server handles partial operations:

```
 -- Teller 31254 is active serving Client358...Welcome back Client358
 -- Teller 31255 is active serving Client359...Welcome back Client359
 -- Teller 31256 is active serving Client360...
select failed (errno=9: Bad file descriptor)
Waiting for clients @/tmp/ServerFIFO_Name...
 - Received 6 clients from PID32032..
 -- Teller 32033 is active serving Client01...Welcome back Client01
 -- Teller 32034 is active serving Client02...
 -- Teller 32035 is active serving Client03...Welcome back Client03
```

The server correctly detects and reports the "Bad file descriptor" error, then continues waiting for new clients. This demonstrates robustness in handling unexpected client disconnections.

Memory Management and Resource Cleanup

One of the most critical aspects of the system is proper resource management. I implemented comprehensive cleanup routines for all processes:

The server tracks all child processes and uses a SIGCHLD handler to reap terminated tellers, preventing zombie processes. When the server itself terminates, it sends SIGTERM to all child processes, giving them time to clean up their resources before exiting.

For file descriptors, I meticulously track and close all pipes and FIFOs when they're no longer needed. This is particularly important in the batch processing logic, where multiple pipes are created:

```
/* Clean up any remaining pipes and wait for tellers */
for (int i = 0; i < currentBatch.received; i++) {
    /* Close any remaining pipe descriptors */
    for (int k = 0; k < 4; k++) {
        if (pipes[i][k] != -1) {
            close(pipes[i][k]);
            pipes[i][k] = -1;
        }
    }

    /* Wait for any teller that's still running */
    if (tellerPids[i] > 0 && !teller_completed[i]) {
        int status;
        // Wait or terminate if necessary
    }
}
```

Semaphores are properly closed and unlinked when processes exit, and I use a static flag in signal handlers to prevent cascading cleanups if multiple signals arrive simultaneously.

The comprehensive test results confirm that the implementation successfully meets all project requirements. The system correctly handles concurrent operations, maintains data integrity, enforces banking validation rules, and manages resources properly. The performance and reliability of the system have been verified through extensive testing, showing that the architecture and implementation decisions were sound.