ELEC-C7310 Assignment #2: ThreadBank - Learning Diary

Introduction

This learning documents is my experience implementing the ThreadBank application, a multi-threaded bank server with client-server communication. The assignment required building a concurrent system with proper synchronization, IPC communication, and persistent data storage.

Initial Planning and Design Phase

The first challenge was understanding the system architecture. In summary I needed to create the following things: a multi-threaded server, client programs that communicate using IPC, queue management for distributing clients to desks, thread-safe account operations, persistent storage and logging.

I decided to use Unix domain sockets for IPC communication instead of named pipes or shared memory, as they provide reliable bidirectional communication and are well-suited for local processes.

Threading and Synchronization Challenges

The most complex part was implementing proper thread synchronization. I encountered several challenges when implementing the task, I will discuss about them next.

Account Access Synchronization

Initially, I considered using simple mutexes for account protection, but realized that read operations, for example balance inquiries, could be concurrent while write operations needed exclusive access. This led me to implement pthread_rwlock for each account, allowing multiple readers or single writer access patterns.

Transfer Operation Deadlocks

The transfer operation between two accounts posed a classic deadlock scenario. If two transfers happen simultaneously between the same accounts in opposite directions, they could deadlock waiting for each other's locks. I solved this by implementing ordered locking - always acquiring locks in ascending account ID order, regardless of transfer direction.

Queue Management

Implementing thread-safe queues for each service desk required careful coordination. I used condition variables to make worker threads wait efficiently when their queues were empty, and to signal when new clients arrived. The queue selection algorithm (also needed synchronization to prevent race conditions.

IPC Communication

I chose Unix domain sockets over other IPC mechanisms for few reasons: they are reliable connection-oriented communication, there is an automatic cleanup when processes terminate, there is a simpler error handling compared to named pipes and there is no network overhead concerns.

The client-server protocol was pretty straightforward: client connects, waits for "ready" message, sends commands, receives responses.

Data Persistence and Logging

Implementing persistent storage taught me about file I/O in concurrent environments. I used a simple binary format for account data and implemented atomic save operations. The logging system required its own mutex to prevent interleaved log entries from multiple threads.

Signal Handling and Graceful Shutdown

Implementing graceful shutdown was more complex than expected. The signal handler needed to:

- Set a global shutdown flag
- Wake up all waiting worker threads
- Close the server socket to stop accepting new connections
- Allow current transactions to complete

The challenge was ensuring that threads check the shutdown flag at appropriate points without affecting performance. I used pthread_cond_broadcast to wake sleeping threads and made sure all loops checked the shutdown condition.

Testing and Debugging

Testing revealed several issues that weren't apparent during initial implementation:

Race Conditions in Queue Management

The testbench exposed race conditions in my queue counting logic. Multiple threads updating queue counts simultaneously caused incorrect shortest-queue selection. I fixed this by ensuring all queue operations were atomic within their mutex sections.

Buffer Management

Initial testing showed buffer overflow issues when clients sent malformed commands. I added proper bounds checking and input validation to prevent crashes.

Resource Cleanup

Memory leaks and file descriptor leaks appeared during stress testing. I implemented proper cleanup paths and ensured all allocated resources were freed even in error conditions.

Testing

I created a test.sh script (also ran chmod +x test.sh), to allow the make command to access that.

The testbench was particularly valuable as it revealed concurrency issues that were difficult to reproduce manually. When running the testbench, the final code worked without errors

Current Limitations and What Works

What Works Well:

- Concurrent client handling with proper synchronization
- All required banking operations (balance, withdraw, deposit, transfer)
- Graceful shutdown without data corruption
- Persistent account storage
- Transaction logging
- Deadlock prevention in transfers

Current Limitations:

- Fixed maximum number of accounts (100)
- Simple binary storage format (not human-readable)
- No authentication or security features
- Limited error recovery from disk I/O failures
- Queue balancing could be more sophisticated

Further Development

Performance Improvements:

- **Better load balancing**: Instead of shortest queue, consider desk utilization or estimated service time
- **Lock-free data structures**: For high-frequency operations like balance inquiries

- **Asynchronous I/O**: For logging and persistence to reduce blocking

Robustness Enhancements:

- **Write-ahead logging**: Ensure no transactions are lost even on crashes
- **Backup and recovery**: Implement periodic backups and consistency checking
- **Connection pooling**: Reuse connections to reduce overhead

Feature Extensions:

- **Account types**: Different account types with different rules
- **Transaction history**: Per-account transaction logs
- **Interest calculation**: Background thread for calculating interest
- **Network support**: TCP sockets for remote clients

Security Improvements:

- **Authentication**: Client authentication before allowing operations
- **Encryption**: Encrypt sensitive data in storage
- **Audit logging**: Detailed security logs for compliance

Key Learning Outcomes

This assignment provided good experience with many things, but few important things are:

- **Concurrent programming**: Understanding thread synchronization, deadlock prevention, and race condition debugging
- **System programming**: Unix IPC mechanisms, signal handling, and process management
- **Software architecture**: Designing modular, maintainable concurrent systems

The most important lesson was that concurrent programming requires careful design and extensive testing. Issues that never appear in single-threaded code become critical in multi-threaded environments.

I know also that the code is not optimal, especially the server.c is far from optimal and I would consider the solution somewhat dirty as it is quite lengthy, but it seemed to do the trick. So I thought that it is probably “good enough” in the scope of this assignment.

Conclusion

This assignment demonstrated nicely the complexity and rewards of concurrent systems programming. While challenging, the experience of building a working multi-threaded server with proper synchronization.