Operating Systems
Lab Project

Drop-Box Clone

BSCS23143

BSCS23102

BSCS23086

Project Report

Phase 1 Design Report

Dropbox Server Implementation

1. Architecture Overview

The server follows a three-layer producer-consumer architecture:

Main Thread (Accept) → Client Queue → Client Thread Pool → Task Queue → Worker Thread Pool

1.1 Component Responsibilities

- **Main Thread**: Accepts TCP connections, pushes sockets to Client Queue
- **Client Thread Pool**: Handles authentication, command parsing, creates tasks
- **Worker Thread Pool**: Executes file operations (upload/download/delete/list)
- **Queues**: Thread-safe communication between components

2. Thread-Safe Queue Implementation

2.1 Data Structure

```
typedef struct {
    void **items;           // Circular buffer
    int capacity;           // Maximum size
    int front, rear, size;  // Queue pointers
    pthread_mutex_t lock;   // Mutual exclusion
    pthread_cond_t not_empty, not_full; // Condition variables
} Queue;
```

2.2 Synchronization Strategy

- **Mutex**: Protects all queue operations
- **Condition Variables**:
    - `not_empty`: Signals when queue has items
    - `not_full`: Signals when space available
- **Blocking Operations**:
    - `dequeue()` blocks when empty
    - `enqueue()` blocks when full

2.3 Rationale

- Prevents busy waiting with condition variables
- Ensures thread-safe access to shared queues
- Supports multiple producers/consumers

3. Thread Pool Design

3.1 Client Thread Pool

- **Size**: 5 threads (configurable)
- **Role**: Handle client communication and command parsing
- **Tasks**: Authentication, request packaging

3.2 Worker Thread Pool

- **Size**: 5 threads (configurable)

- **Role**: Execute file operations
- **Tasks**: File I/O, quota management, metadata updates

3.3 Task Structure

```
typedef struct {
    OperationType type;     // UPLOAD, DOWNLOAD, DELETE, LIST
    char username[50];      // Authenticated user
    char filename[255];     // Target file
    char local_path[255];   // Source file (upload only)
    int client_socket;      // Response channel
    UserManager *user_manager; // User database
} TaskData;
```

4. User Management System

4.1 Data Structures

```
typedef struct {
    char username[50];
    char password[50];
    UserFile files[MAX_FILES];
    int file_count;
    size_t used_quota;
    size_t quota;
    pthread_mutex_t lock;  // Per-user mutex
} User;

typedef struct {
    User users[MAX_USERS];
    int user_count;
    pthread_mutex_t lock;  // Global user manager mutex
} UserManager;
```

4.2 Synchronization Approach

- **Global Lock**: Protects user array during signup/login
- **Per-user Lock**: Protects individual user data during file operations
- **Fine-grained Locking**: Minimizes contention between different users

5. File Operations Implementation

5.1 Storage Organization

```
user_files/
```

```
├── username1/
│   ├── file1.txt
│   └── file2.pdf
└── username2/
    └── document.doc
```

5.2 Operation Flow

1. **UPLOAD**: Copy local file → user directory, update metadata
2. **DOWNLOAD**: Read file from user directory → send to client
3. **DELETE**: Remove file, update metadata and quota
4. **LIST**: Read user's file metadata

5.3 Quota Management

- Default: 10MB per user
- Real-time tracking during upload/delete
- Pre-upload quota validation

6. Synchronization Decisions

6.1 Queue Synchronization

- **Choice**: Mutex + Condition Variables
- **Reason**: Efficient blocking without busy waiting
- **Alternative Considered**: Semaphores (chosen CV for clarity)

6.2 User Data Synchronization

- **Choice**: Two-level locking (global + per-user)
- **Reason**: Balance between contention and complexity
- **Benefit**: Multiple users can operate concurrently

6.3 File Operation Synchronization

- **Choice**: Per-user mutex for metadata
- **Reason**: Prevents race conditions for same user's files
- **Limitation**: Serializes operations for same user

7. Memory Management

7.1 Allocation Patterns

- **Queue Items**: Dynamic allocation in producer, freed in consumer
- **Task Data**: Allocated in client thread, freed in worker thread
- **User Data**: Static array to simplify management

7.2 Cleanup Strategy

- Graceful shutdown signal handling
- Thread pool destruction with pending task completion
- Queue destruction after thread termination

8. Testing Methodology

8.1 Functional Tests

- Single user authentication and file operations
- Quota enforcement validation
- Error condition handling

8.2 Memory Tests

- Valgrind for leak detection
- No definite memory leaks achieved
- Minor "still reachable" memory (global structures)

8.3 Concurrency Tests

- Basic thread safety verified
- No data races detected in single-user scenario

9. Design Trade-offs

9.1 Chosen Approaches

- **In-memory user data**: Simplicity over persistence
- **Fixed thread pools**: Predictable resource usage
- **Per-user locking**: Good balance for Phase 1 requirements

9.2 Limitations Acknowledged

- No persistence across server restarts
- Single client per user in Phase 1
- Basic error recovery

10. Phase 2 Considerations

10.1 Planned Enhancements

- Multiple concurrent sessions per user
- Worker-to-client response mechanism
- Persistent metadata storage

- Enhanced conflict resolution

10.2 Open Design Questions

- Optimal worker-response communication method
- File locking strategy for concurrent access
- Metadata persistence approach

11. Conclusion

Phase 1 successfully implements the core producer-consumer architecture with proper synchronization. The design choices provide a solid foundation for Phase 2's concurrency challenges while maintaining code clarity and correctness.