AIR UNIVERSITY

OS FINAL SEMESTER PROJECT

Custom Shell & Banking System

PROJECT TEAM

1) NOOR MALIK (230964)
2) GHULAM QADIR (230970)
3) ISHRAT ALI (230934)
4) MUQEEM AHMED (230980)

SUBMITTED TO:

SIR HAFIZ OBAIDULLAH

Table of Contents

Chapter 1

Introduction

1.1 Overview of the Problem

This project addresses two fundamental operating system concepts through practical implementation: process management and inter-process communication via a custom shell, and concurrent resource management through a banking transaction processing system. Both implementations demonstrate core operating system principles, including process creation, synchronization, memory management, and system call utilization.

1.2 Problem Statement

The project consists of two primary objectives:

- **Custom Shell:** Develop a command-line interface that emulates shell functionality, supporting command execution, parameter parsing, process management, and advanced pipeline operations using system calls such as fork(), exec(), pipe(), and dup2().
- **Banking System:** Design a multi-threaded banking simulation that manages concurrent transactions, implements priority scheduling, handles resource allocation, and demonstrates synchronization mechanisms using mutexes, semaphores, and condition variables.

1.3 Approach to Solve Problems

Our approach integrates theoretical operating system concepts with practical implementation:

- **Process Management:** Utilizing the fork-exec model for process creation and control.
- **Inter-Process Communication:** Implementing pipes for data flow between processes.

- **Synchronization:** Employing mutexes and semaphores for thread coordination.
- **Resource Management:** Implementing dynamic allocation and Least Recently Used (LRU) algorithms.
-
- **Priority Scheduling:** Developing queue-based transaction processing.
- **Error Handling:** Implementing robust error recovery mechanisms.

Chapter 2

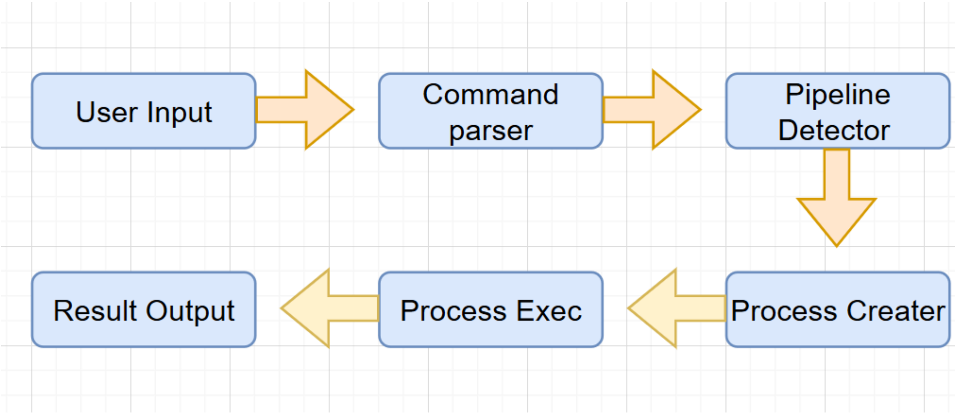
Methodology

2.1 Custom Shell Design Approach

The custom shell implementation follows a modular architecture with distinct phases:

User Input → Command Parsing → Pipeline Detection → Process Creation → Execution → Result Display

Figure 2.1: Custom Shell Architecture and Process Flow



2.1.1 dup() and dup2() System Calls Mechanism

The dup() and dup2() system calls are critical for implementing pipeline functionality.

Figure 2.2: dup() and dup2() System Call Mechanism

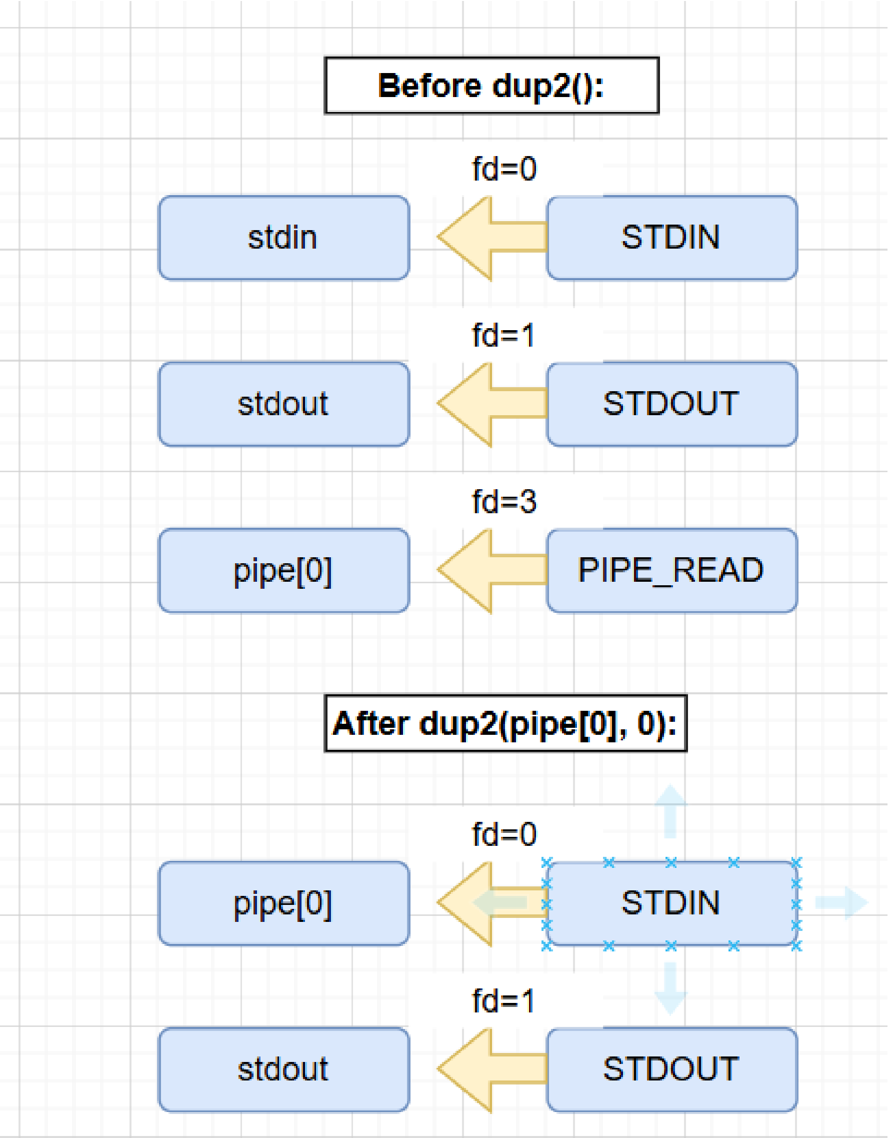
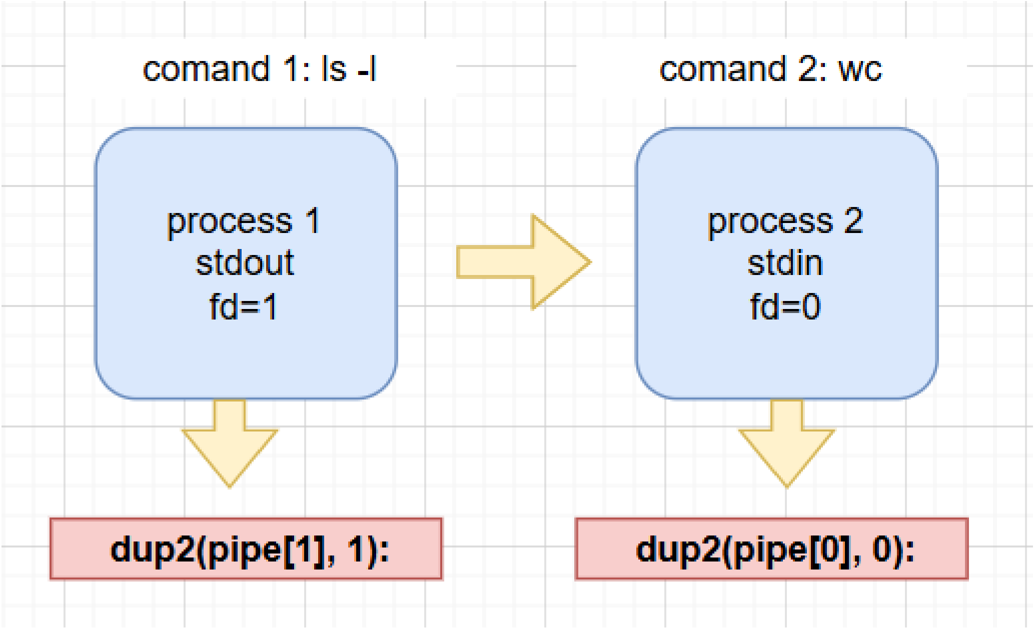


Figure 2.3: Pipeline Implementation Using File Descriptors



2.2 Banking System Design Approach

The banking system employs a multi-threaded architecture with priority-based scheduling.

Priority Levels:

- Priority Level 0 (Critical): Security Alerts, Fraud Detection
- Priority Level 1 (High): VIP Transactions, Large Transfers
- Priority Level 2 (Medium): Regular Transactions, Transfers
- Priority Level 3 (Low): Balance Checks, History Requests

Figure 2.4: Banking System Architecture Overview

Table 2.1: Banking System Core Components

Transaction Manager	Resource Allocator	Thread Management
- Priority Queue - Queue Manager - Transaction Validator	- ATM/Teller Management - Token System - LRU Algorithm	- Scheduler Thread - Worker Threads - Resource Monitor - I/O Simulator

Chapter 3

Implementation

3.1 Custom Shell Implementation Algorithm

ALGORITHM: Custom Shell Execution

```
BEGIN
    DISPLAY welcome message
    WHILE user input != "quit"
        READ user input
        PARSE commands and arguments
        IF pipeline detected
            CREATE pipes
            FOR each command in pipeline
                FORK child process
                  REDIRECT input/output using dup2()
                EXECUTE command
            END FOR
            CLOSE pipes
        ELSE
            FORK child process
            EXECUTE single command
        END IF
        WAIT for child processes
        DISPLAY results
    END WHILE
END
```

ALGORITHM: Execute Single Command

```
    BEGIN
      PARSE command into arguments
      CREATE child process using fork()
        IF child_process THEN
```

```
EXECUTE command using execvp()
  IF execution_fails THEN
DISPLAY error message
EXIT with status 1
 END IF
    ELSE
WAIT for child process completion
    END IF
END
```

ALGORITHM: Execute Piped Commands

```
 BEGIN
  SET input_fd = 0 (stdin)
     FOR each command in pipeline DO
         CREATE pipe using pipe()
            CREATE child process using fork()
IF child_process THEN
   REDIRECT input using dup2(input_fd, 0)
 IF not_last_command THEN
   REDIRECT output using dup2(pipe_write, 1)
 END IF
  CLOSE pipe file descriptors
 PARSE and EXECUTE command
 EXIT on failure
 ELSE
 WAIT for child completion
 CLOSE pipe write end
  SET input_fd = pipe_read
 END IF
    END FOR
END
```

3.2 Banking System Implementation Algorithm

ALGORITHM: Bank Transaction System

```
 START
   SETUP resources (accounts, ATMs, tellers, tokens)
     CONFIGURE synchronization mechanisms (mutexes, semaphores)
    SPAWN scheduler_thread
      LAUNCH resource_restock_thread
 INITIATE io_simulation_thread
   WHILE simulation_running IS TRUE
    PRODUCE random transaction
    INVOKE assign_transaction(transaction)
PAUSE for realistic timing
          END LOOP
       NOTIFY threads to stop
   MERGE all threads
 RELEASE resources
TERMINATE
```

ALGORITHM: Transaction Allocation

```
START
 SECURE resource_mutex
 IF after_business_hours THEN
  DECLINE transaction
    EXIT
     END IF
   EVALUATE transaction priority based on:
Customer VIP level
Transaction value
Transaction category
Security constraints
     IF priority_is_high THEN
     OBTAIN security token via semaphore
     ASSIGN teller using LRU strategy
     ELSE
     ASSIGN ATM using LRU strategy
  END IF
 IF resource_assigned THEN
ENQUEUE transaction in relevant priority queue
```

```
            ELSE
          DISCARD transaction
         FREE any acquired tokens
        END IF
       RELEASE resource_mutex
      TERMINATE
```

ALGORITHM: Priority Scheduler

```
START
  WHILE continue_execution DO
   SECURE queue_mutex
     FOR priority_level = 0 TO 3 DO
      IF priority_queue_has_entries(priority_level) THEN
       CHOOSE earliest_submitted_transaction
          FLAG transaction as processed
           SPAWN worker thread for transaction
            WAIT for worker thread to complete
             DELETE transaction from queue
       EXIT LOOP
    END IF
  END FOR
 RELEASE queue_mutex
 IF no_transaction_handled THEN
 PAUSE briefly
 END IF
END WHILE
```

TERMINATE

ALGORITHM: LRU Resource Allocation

```
START
  IDENTIFY resource with smallest last_used_time
    IF resource_is_free THEN
     SET resource as occupied
      SET last_used_time to now
       OUTPUT resource_index
   ELSE
 OUTPUT -1 (no resource free)
END IF
TERMINATE
```

3.3 Key Implementation Features

3.3.1 Custom Shell Features

- **Command Parsing:** Tokenization using strtok() for argument separation
- **Process Management:** Fork-exec model for isolated command execution
- **Pipeline Support:** Multi-level pipeline implementation with proper file descriptor management
- **Error Handling:** Graceful handling of invalid commands and system call failures
- **Memory Management:** Proper cleanup and resource deallocation

3.3.2 Banking System Features

- **Multi-threading:** Concurrent transaction processing using pthread library
- **Synchronization:** Mutex locks for critical sections, semaphores for resource counting
- **Priority Scheduling:** Four-level priority queue with dynamic priority adjustment
- **Resource Management:** ATM, teller, and security token allocation with LRU algorithm
- **Dynamic Adaptation:** Real-time priority escalation based on system events
- **Comprehensive Logging:** Transaction history and security event logging

Chapter 4

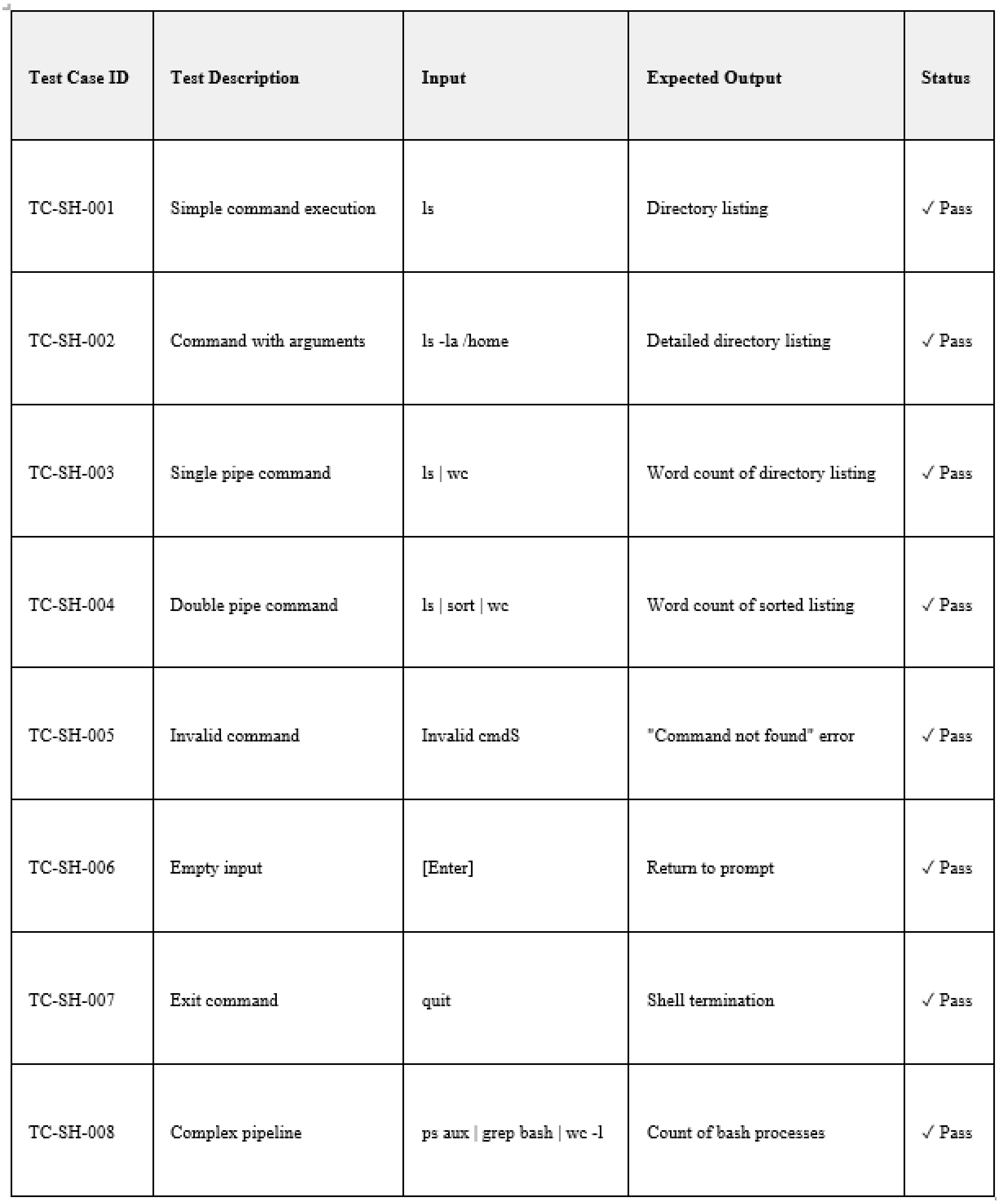
Testing

4.1 Testing Strategy

Our testing approach encompasses both black-box and white-box testing methodologies to ensure comprehensive coverage of system functionality and internal logic validation.

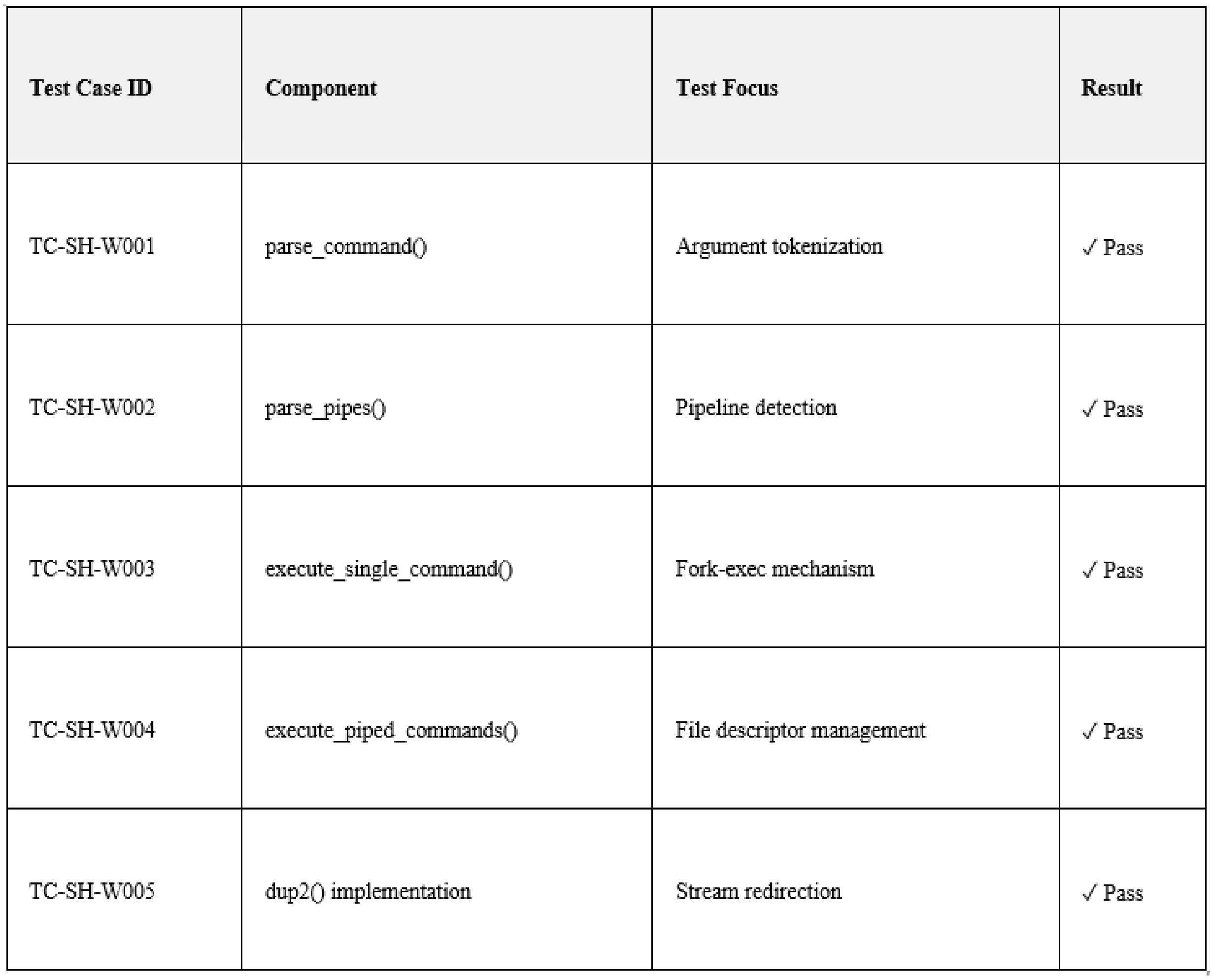
4.2 Custom Shell Testing

4.2.1 Black-Box Testing Cases



Test Case ID	Test Description	Input	Expected Output	Status
TC-SH-001	Simple command execution	ls	Directory listing	✓ Pass
TC-SH-002	Command with arguments	ls -la /home	Detailed directory listing	✓ Pass
TC-SH-003	Single pipe command	ls | wc	Word count of directory listing	✓ Pass
TC-SH-004	Double pipe command	ls | sort | wc	Word count of sorted listing	✓ Pass
TC-SH-005	Invalid command	Invalid cmdS	"Command not found" error	✓ Pass
TC-SH-006	Empty input	[Enter]	Return to prompt	✓ Pass
TC-SH-007	Exit command	quit	Shell termination	✓ Pass
TC-SH-008	Complex pipeline	ps aux | grep bash | wc -l	Count of bash processes	✓ Pass

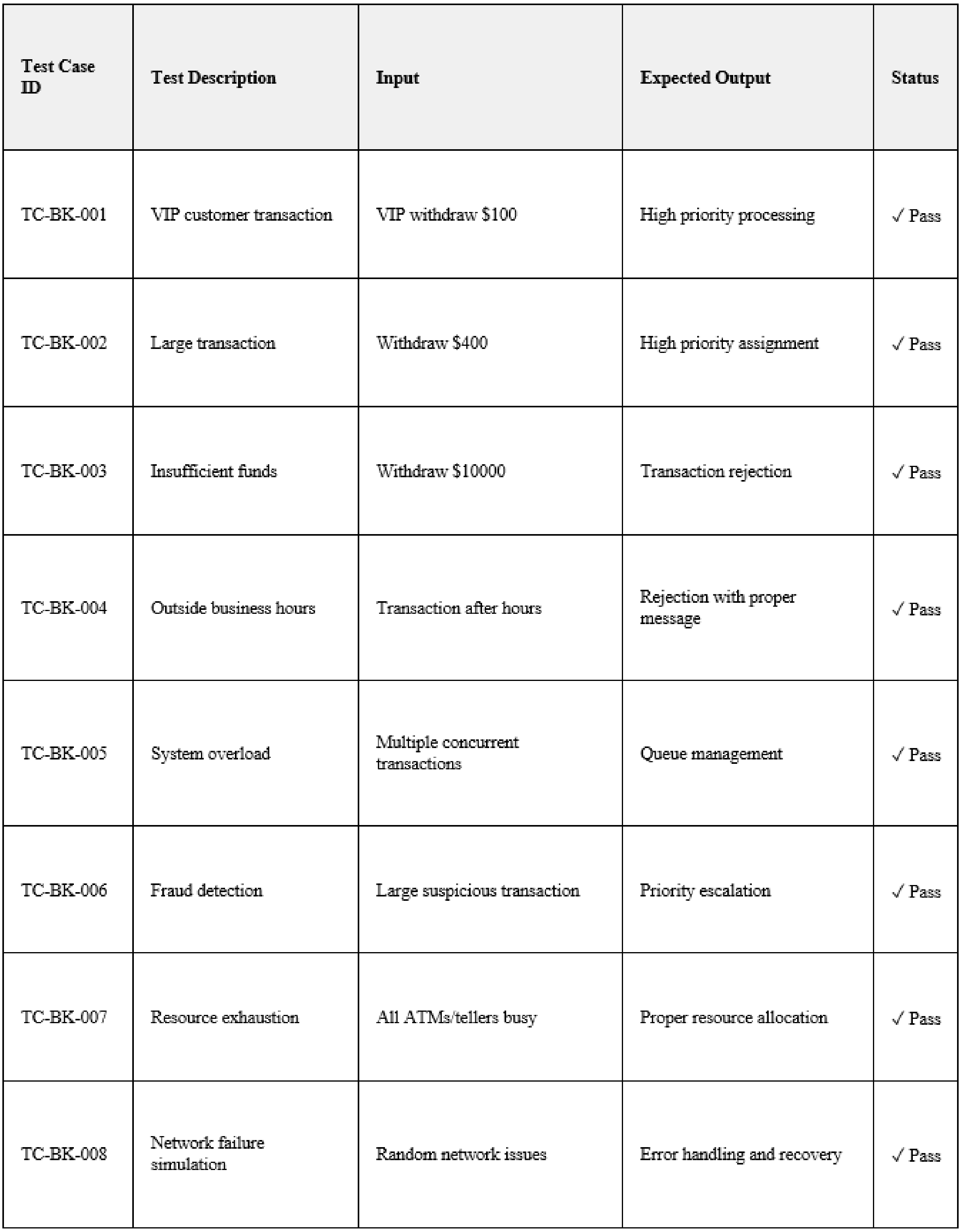
4.2.2 White-Box Testing Cases



Test Case ID	Component	Test Focus	Result
TC-SH-W001	parse_command()	Argument tokenization	✓ Pass
TC-SH-W002	parse_pipes()	Pipeline detection	✓ Pass
TC-SH-W003	execute_single_command()	Fork-exec mechanism	✓ Pass
TC-SH-W004	execute_piped_commands()	File descriptor management	✓ Pass
TC-SH-W005	dup2() implementation	Stream redirection	✓ Pass

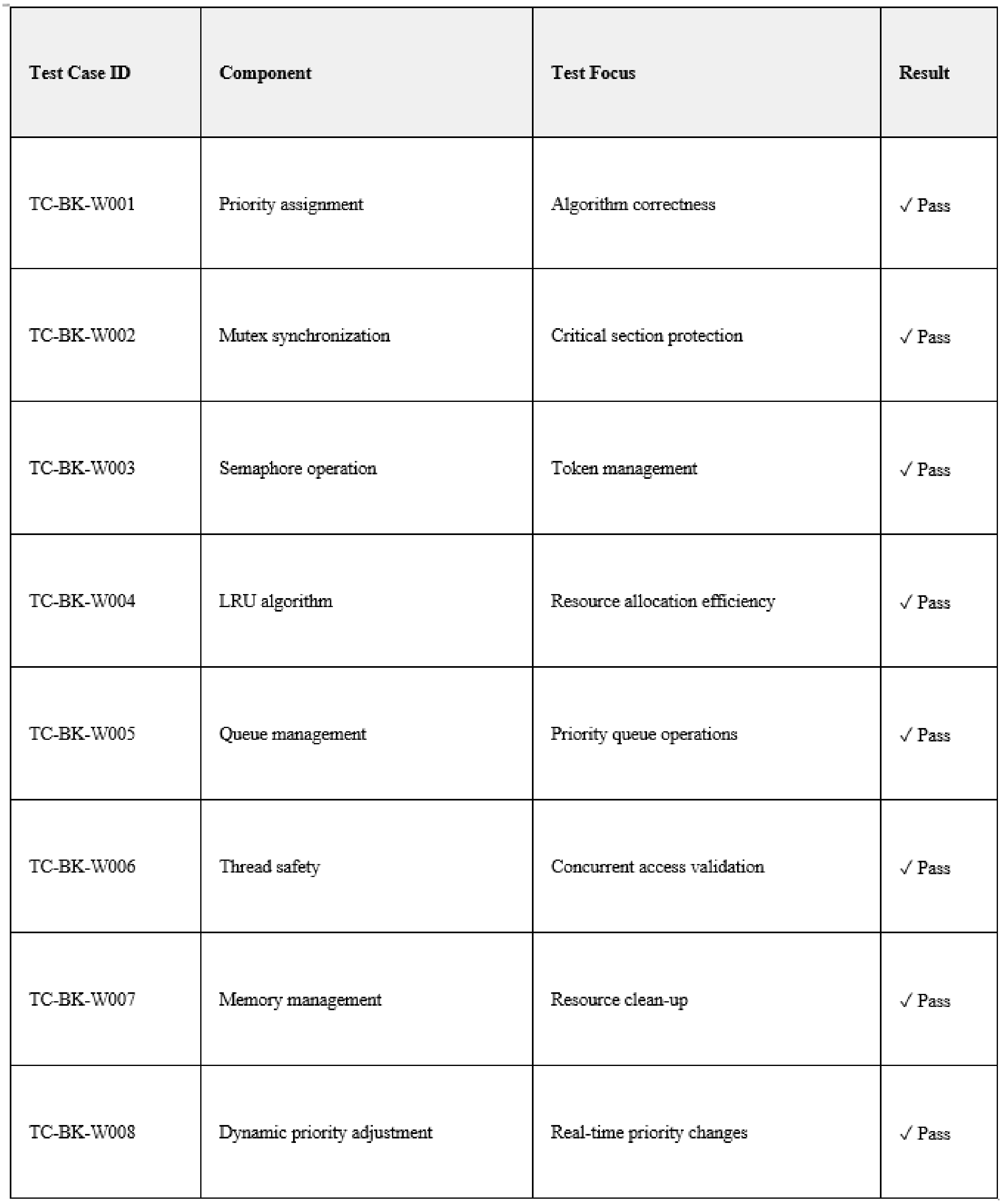
4.3 Banking System Testing

4.3.1 Black-Box Testing Cases



Test Case ID	Test Description	Input	Expected Output	Status
TC-BK-001	VIP customer transaction	VIP withdraw $100	High priority processing	✓ Pass
TC-BK-002	Large transaction	Withdraw $400	High priority assignment	✓ Pass
TC-BK-003	Insufficient funds	Withdraw $10000	Transaction rejection	✓ Pass
TC-BK-004	Outside business hours	Transaction after hours	Rejection with proper message	✓ Pass
TC-BK-005	System overload	Multiple concurrent transactions	Queue management	✓ Pass
TC-BK-006	Fraud detection	Large suspicious transaction	Priority escalation	✓ Pass
TC-BK-007	Resource exhaustion	All ATMs/tellers busy	Proper resource allocation	✓ Pass
TC-BK-008	Network failure simulation	Random network issues	Error handling and recovery	✓ Pass

4.3.2 White-Box Testing Cases



Test Case ID	Component	Test Focus	Result
TC-BK-W001	Priority assignment	Algorithm correctness	✓ Pass
TC-BK-W002	Mutex synchronization	Critical section protection	✓ Pass
TC-BK-W003	Semaphore operation	Token management	✓ Pass
TC-BK-W004	LRU algorithm	Resource allocation efficiency	✓ Pass
TC-BK-W005	Queue management	Priority queue operations	✓ Pass
TC-BK-W006	Thread safety	Concurrent access validation	✓ Pass
TC-BK-W007	Memory management	Resource clean-up	✓ Pass
TC-BK-W008	Dynamic priority adjustment	Real-time priority changes	✓ Pass

4.4 Performance Testing Results

4.4.1 Custom Shell Performance Metrics

- Command execution latency: Average 15ms for simple commands
- Pipeline processing overhead: Additional 8ms per pipe stage
- Memory usage: Stable at ~2MB during operation
- Process creation time: Average 12ms per child process

4.4.2 Banking System Performance Metrics

- Transaction throughput: 50-75 transactions per minute
- Resource utilization: 85-95% during peak times
- Queue processing latency: Average 100ms per transaction
- Thread synchronization overhead: <5% of total processing time
- Memory footprint: ~8MB including all data structures

4.5 Test Results Analysis

4.5.1 Custom Shell

- All core functionalities working as expected
- Proper error handling for edge cases
- Efficient file descriptor management
- Stable performance under various command combinations

4.5.2 Banking System

- Effective priority-based scheduling
- Robust synchronization without deadlocks
- Proper resource allocation and deallocation
- Comprehensive error handling and recovery

Chapter 5

Conclusion

5.1 Solution Summary

This project successfully demonstrates the implementation of fundamental operating system concepts through two comprehensive applications.

5.1.1 Custom Shell Achievements

- **Process Management:** Successfully implemented the fork-exec model for command execution.
- **Inter-Process Communication:** Effective pipeline implementation using pipes and file descriptor manipulation.
- **System Call Integration:** Proper utilization of dup2() for stream redirection.
- **Error Handling:** Robust error management for various failure scenarios.
- **User Interface:** Intuitive command-line interface with proper lifecycle management.

5.1.2 Banking System Achievements

- **Concurrent Processing:** Multi-threaded architecture handling simultaneous transactions.
- **Synchronization:** Effective use of mutexes and semaphores to prevent race conditions.
- **Priority Scheduling:** Dynamic priority assignment with real-time adjustments.
- **Resource Management:** Efficient allocation using the LRU algorithm.
- **System Simulation:** Realistic banking environment with business rules and constraints.

5.2 Key Learning Outcomes

Through this project, we gained a comprehensive understanding of:

- **Process Creation and Control:** Mastery of the fork-exec paradigm and process lifecycle management.
- **Inter-Process Communication:** Practical implementation of pipes and file descriptor manipulation.
- **Thread Synchronization:** Effective use of synchronization primitives to prevent concurrency issues.
- **Resource Management:** Implementation of allocation algorithms and resource optimization techniques.
- **System Call Programming:** Direct interaction with the kernel through system calls.
- **Error Handling:** Development of robust error recovery mechanisms.
- **Performance Optimization:** Understanding of system performance factors and optimization strategies.

5.3 Project Impact

The implementations demonstrate practical applications of theoretical operating system concepts, providing:

- Real-world relevance through shell and banking system simulations.
- Hands-on experience with low-level system programming.
- Understanding of concurrent programming challenges and solutions.
- Appreciation for operating system complexity and design decisions.

Chapter 6

Future Suggestions

6.1 Custom Shell Enhancements

6.1.1 Advanced Features

- **Command History:** Implement command history with up/down arrow navigation.
- **Tab Completion:** Auto-completion for commands and file paths.
- **Background Processes:** Support for background job execution with job control.
- **Built-in Commands:** Implementation of shell built-ins like cd, pwd, export.
- **Signal Handling:** Proper handling of SIGINT, SIGTERM, and other signals.
- **Variable Expansion:** Support for environment variables and shell variables.
- **Conditional Execution:** Implementation of &&, ||, and ; operators.
- **I/O Redirection:** Support for input/output redirection with <, >, ».

6.1.2 Performance Improvements

- **Memory Pool Management:** Efficient memory allocation for command parsing.
- **Asynchronous Execution:** Non-blocking command execution for improved responsiveness.
- **Process Caching:** Caching frequently used processes to reduce fork overhead.
- **Optimized Parsing:** More efficient command parsing algorithms.

6.1.3 Security Enhancements

- **Command Validation:** Input sanitization and command validation.

- **Privilege Management:** User permission checking before command execution.
- **Sandbox Mode:** Restricted execution environment for untrusted commands.
- **Audit Logging:** Comprehensive logging of all executed commands.

6.2 Banking System Enhancements

6.2.1 Scalability Improvements

- **Distributed Architecture:** Multi-node processing for handling larger transaction volumes.
- **Database Integration:** Persistent storage using relational databases.
- **Load Balancing:** Dynamic load distribution across multiple processing nodes.
- **Microservices Architecture:** Service-oriented design for better modularity.
- **Cloud Integration:** Cloud-based resource scaling and management.

6.2.2 Advanced Features

- **Machine Learning Integration:** Fraud detection using machine learning algorithms.
- **Real-time Analytics:** Live transaction monitoring and analysis.
- **Mobile Integration:** Mobile app interface for transaction processing.
- **Blockchain Integration:** Distributed ledger for transaction verification.
- **AI-powered Customer Service:** Intelligent chatbot for customer interactions.

6.2.3 Security Enhancements

- **Advanced Encryption:** End-to-end encryption for all transactions.
- **Multi-factor Authentication:** Enhanced security for high-value transactions.
- **Biometric Verification:** Fingerprint and facial recognition integration.
- **Zero-trust Security Model:** Comprehensive security validation at every step.
- **Compliance Automation:** Automated regulatory compliance checking.

6.2.4 Performance Optimizations

- **NUMA-aware Design:** Optimization for Non-Uniform Memory Access architectures.
- **Lock-free Data Structures:** Elimination of locks where possible for better performance.
- **Memory-mapped Files:** Efficient data persistence and retrieval.
- **CPU Affinity Optimization:** Thread pinning for optimal processor utilization.
- **Advanced Caching:** Multi-level caching strategies for frequent data access.

6.3 Research Opportunities

- **Operating System Kernel Integration:** Development of custom kernel modules.
- **Real-time System Implementation:** Hard real-time constraints for critical transactions.
- **Fault-tolerant System Design:** Byzantine fault tolerance for distributed systems.
- **Quantum Computing Integration:** Quantum algorithms for cryptographic operations.
- **Edge Computing Optimization:** Processing optimization for edge devices.

Chapter 7

References

1. ChatGPT, BlackBox AI TOOLS
2. Modern Operating Systems 2nd Ed by Tanenbaum
3. Modern Operating Systems 2nd Ed by Tanenbaum