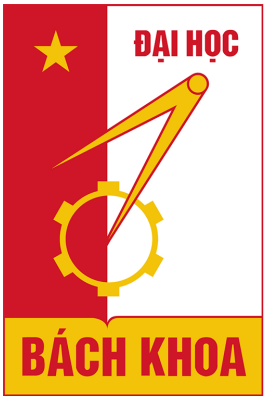

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

School of Information and Communication Technology

The Reader-Writer Problem:

Implementation and Analysis of Synchronization Strategies

Operating Systems - IT3070E

Topic 6: Readers-Writers Problem

Students: Tran Quang Hung - 20235502
Nguyen Xuan Khai - 20235508

Supervisor: PhD. Do Quoc Huy

Class: 161859

Hanoi, January 13, 2026

Abstract

The Reader-Writer problem represents one of the fundamental challenges in concurrent programming and operating systems design. This paper presents a comprehensive implementation and analysis of four distinct synchronization strategies: vanilla (no synchronization), reader preference, writer preference, and fair scheduling using the turnstile pattern. We implement a shared string manipulation application to demonstrate the behavior and trade-offs of each approach. Through extensive automated testing (16 runs across 4 modes), we demonstrate race conditions in unsynchronized access (average 367 torn reads in vanilla mode) and perfect correctness in synchronized modes (0 errors in 12 runs). This work provides both theoretical analysis and practical implementation insights valuable for concurrent systems design.

1 Introduction

1.1 Motivation

In modern computing systems, concurrent access to shared resources is ubiquitous. Database systems must handle simultaneous read and write transactions, file systems must coordinate multiple processes accessing the same files, and cache systems must maintain consistency while allowing parallel reads for performance. The Reader-Writer problem encapsulates this fundamental challenge: how to allow multiple concurrent readers while ensuring exclusive access for writers, all while maintaining correctness, performance, and fairness.

The importance of this problem extends beyond academic interest. Real-world systems such as PostgreSQL, Linux kernel read-write semaphores, and Java's `ReadWriteLock` all implement solutions to this problem. Understanding the trade-offs between different synchronization strategies is crucial for systems engineers and application developers.

1.2 Problem Statement

The classical Reader-Writer problem involves multiple concurrent threads accessing a shared resource, where:

- **Readers** only read data and can operate concurrently without interfering with each other
- **Writers** modify data and require exclusive access (no other readers or writers)

The challenge is to design a synchronization mechanism that:

1. Ensures **correctness**: No data corruption from concurrent access
2. Maximizes **performance**: Allow as much concurrency as safely possible
3. Prevents **starvation**: No thread waits indefinitely
4. Provides **fairness**: Balanced access for all threads

These requirements often conflict, creating fundamental trade-offs that must be carefully balanced based on workload characteristics.

1.3 Contributions

This paper makes the following contributions:

- A unified implementation framework supporting four synchronization modes with a single API
- Comprehensive automated testing demonstrating race conditions and validating correctness
- Quantitative analysis of performance trade-offs between different synchronization strategies
- Practical insights for implementing Reader-Writer locks in real systems

2 Background

2.1 The Reader-Writer Problem in Theory

The Reader-Writer problem was first formally described by Courtois, Heymans, and Parnas in 1971. It represents a fundamental synchronization challenge where:

- Multiple readers can access shared data concurrently (read operations are non-interfering)
- Writers require exclusive access (no other readers or writers during write)
- Readers and writers must coordinate to prevent data corruption

Formal Requirements:

1. **Mutual Exclusion for Writers**: At most one writer can access the resource at any time
2. **Concurrent Readers**: Multiple readers can access simultaneously
3. **No Reader-Writer Concurrency**: Writers must have exclusive access
4. **Progress**: Both readers and writers should eventually gain access
5. **Bounded Waiting**: No indefinite postponement (depends on policy)

2.2 Real-World Applications

Database Management Systems:

- PostgreSQL uses Multi-Version Concurrency Control (MVCC) with reader-writer semantics
- MySQL InnoDB implements row-level read-write locks
- Read transactions (SELECT) can run concurrently

- Write transactions (INSERT/UPDATE/DELETE) require exclusive access

Operating Systems:

- Linux kernel `rwlock_t` for shared data structures
- File system metadata access (directory listings vs. file modifications)
- Process synchronization in shared memory segments
- Cache coherence protocols in multiprocessor systems

Programming Languages:

- Java: `ReentrantReadWriteLock` in `java.util.concurrent`
- C++17: `std::shared_mutex` for reader-writer scenarios
- Python: `RWLock` implementations in threading libraries
- Rust: `RwLock<T>` providing safe concurrent access

Web Services:

- Configuration caches (frequent reads, rare updates)
- Session storage (many reads per user, occasional writes)
- Content delivery networks (cache reads vs. purge operations)

2.3 Overall Architecture

Our system consists of three layers:

1. **Common Infrastructure Layer**: Provides unified synchronization API and logging utilities
2. **Application Layer**: Shared string manipulation demonstrating concurrent access
3. **Configuration Layer**: CLI argument parsing and runtime parameter management

This separation ensures modularity and makes it easy to extend the system with additional test scenarios.

2.4 Synchronization Primitives: Mutex vs. Semaphore

The Reader-Writer problem can be implemented using different synchronization primitives. Understanding the differences is crucial for correct implementation.

Semaphore-based Solutions:

- Use counting semaphores to control access
- Binary semaphore (value 0 or 1) acts like a mutex
- Example: `sem_wait()` and `sem_post()` operations

- Advantage: Simpler conceptual model for some algorithms
- Common in classic textbook implementations

Mutex-based Solutions (Our Implementation):

- Use `pthread_mutex_t` for mutual exclusion
- Better integration with POSIX condition variables
- Clearer ownership semantics (thread that locks must unlock)
- Widely supported across platforms
- Easier debugging (can track which thread owns lock)

Critical Clarification:
It's essential to understand that:

- **Mutex and Semaphore** are *synchronization primitives* (building blocks)
- **Reader Preference, Writer Preference, Fair Scheduling** are *algorithms* (strategies)
- The same algorithm can be implemented using different primitives
- Our choice of mutex vs. semaphore doesn't change the fundamental algorithm

2.5 Why POSIX Threads?

Our implementation uses POSIX threads (pthreads), which provides:

- **Mutexes**: For mutual exclusion (`pthread_mutex_t`)
- **Condition Variables**: For thread coordination (`pthread_cond_t`)
- **Thread Management**: Creation, joining, cancellation
- **Platform Independence**: Works on Linux, Unix, macOS
- **Industry Standard**: Widely used in production systems

2.6 Unified Lock API

We designed a unified API that supports all four synchronization modes with a single interface:

```
typedef enum {
    VANILLA,       // No synchronization
    READER_PREF,   // Reader preference
    WRITER_PREF,   // Writer preference
    FAIR           // Fair scheduling
} rw_mode_t;

typedef struct {
    rw_mode_t mode;
```

```
    int active_readers;     // Currently reading
    int active_writers;     // Currently writing (0 or 1)
    int waiting_writers;    // Waiting to write
    pthread_mutex_t mutex;          // Protects counters
    pthread_mutex_t resource_lock; // Protects shared resource
    pthread_mutex_t read_try;       // For writer preference
    pthread_mutex_t queue_lock;     // For fair mode (turnstile)
} rw_lock_t;

// API Functions
void rw_init(rw_lock_t *lock, rw_mode_t mode);
void reader_enter(rw_lock_t *lock);
void reader_exit(rw_lock_t *lock);
void writer_enter(rw_lock_t *lock);
void writer_exit(rw_lock_t *lock);
void rw_destroy(rw_lock_t *lock);
```

Design Benefits:

1. **Mode Isolation**: Each mode's logic is contained in switch-case statements
2. **Easy Comparison**: Switch modes with a single parameter change
3. **Consistent Interface**: Same API regardless of synchronization strategy
4. **Performance Fairness**: All modes use same infrastructure, differences are purely algorithmic

This design allows easy comparison between modes by simply changing the mode parameter, ensuring that performance differences are due to the synchronization strategy rather than implementation variations.

3 Implementation Details

3.1 Shared String Manipulation

3.1.1 Overview

This implementation demonstrates *torn reads*–partial visibility of updates.

Shared Resource: A character array `char shared_string[256]`

Writer Behavior: Writers cycle through predefined sentences:

```
writer_enter(&lock);
if (mode == VANILLA) {
    // Copy char-by-char with delay (to increase race probability)
    for (int i = 0; sentence[i] != '\0'; i++) {
        shared_string[i] = sentence[i];
        usleep(100);  // Intentional delay
    }
} else {
    strncpy(shared_string, sentence, STRING_SIZE - 1);
}
writer_exit(&lock);
```

Reader Behavior: Readers read and validate strings:

```
reader_enter(&lock);
strncpy(buffer, shared_string, STRING_SIZE - 1);
reader_exit(&lock);

// Validate against known valid sentences
if (string_not_in_valid_set(buffer)) {
    // TORN READ DETECTED!
}
```

3.1.2 Race Condition Demonstration

Without synchronization, readers can observe *torn reads*–strings that are partially updated:

Example Timeline:

```
t=0:  shared_string = "Hello World!"
t=1:  W1 starts writing "Operating systems..."
      shared_string = "Oerating World!"  (partial)
t=2:  R1 reads: "Oerating World!"  <- TORN READ!
t=3:  W1 continues...
```

These torn reads represent real data corruption that would occur in production systems without proper synchronization.

4 Synchronization Algorithms

4.1 Mode 1: Vanilla (No Synchronization)

Purpose: Baseline to demonstrate race conditions.

```
void reader_enter(rw_lock_t *lock) {
    if (lock->mode == VANILLA) {
        lock->active_readers++;  // NO LOCK!
        return;
    }
}
```

This mode deliberately has no synchronization to demonstrate the problem. **Never use in production.**

4.2 Mode 2: Reader Preference

Algorithm: First reader locks resource, subsequent readers only increment counter.

```
void reader_enter(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->mutex);
    lock->active_readers++;
    if (lock->active_readers == 1) {
        pthread_mutex_lock(&lock->resource_lock);
    }
    pthread_mutex_unlock(&lock->mutex);
}

```

```
void reader_exit(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->mutex);
    lock->active_readers--;
    if (lock->active_readers == 0) {
        pthread_mutex_unlock(&lock->resource_lock);
    }
    pthread_mutex_unlock(&lock->mutex);
}
```

Advantages:

- Maximizes read throughput
- Multiple readers operate concurrently
- Low reader latency

Disadvantage: Writer starvation–if readers continuously arrive, writers may wait indefinitely.

4.3 Mode 3: Writer Preference

Algorithm: Writers block new readers using `read_try` lock.

```
void writer_enter(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->mutex);
    lock->waiting_writers++;
    pthread_mutex_unlock(&lock->mutex);

    pthread_mutex_lock(&lock->read_try);  // Block new readers
    pthread_mutex_lock(&lock->resource_lock);

    pthread_mutex_lock(&lock->mutex);
    lock->waiting_writers--;
    pthread_mutex_unlock(&lock->mutex);
}

void reader_enter(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->read_try);  // Must acquire first
    pthread_mutex_lock(&lock->mutex);
    lock->active_readers++;
    if (lock->active_readers == 1) {
        pthread_mutex_lock(&lock->resource_lock);
    }
    pthread_mutex_unlock(&lock->mutex);
    pthread_mutex_unlock(&lock->read_try);
}
```

Advantages:

- Prevents writer starvation
- Ensures timely updates

Disadvantage: Reader starvation–continuous writers can delay readers significantly.

4.4 Mode 4: Fair Scheduling (Turnstile Pattern)

Algorithm: All threads pass through a `queue_lock` "turnstile" ensuring FIFO ordering.

```
void reader_enter(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->queue_lock);  // Turnstile
    pthread_mutex_lock(&lock->mutex);
    lock->active_readers++;
    if (lock->active_readers == 1) {
        pthread_mutex_lock(&lock->resource_lock);
    }
    pthread_mutex_unlock(&lock->mutex);
    pthread_mutex_unlock(&lock->queue_lock);
}

void writer_enter(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->queue_lock);  // Turnstile
    pthread_mutex_lock(&lock->resource_lock);
    pthread_mutex_unlock(&lock->queue_lock);
}
```

Advantages:

- No starvation for either readers or writers
- FIFO ordering ensures fairness
- Predictable latency

Disadvantage: Slight throughput reduction due to extra lock overhead.

5 Detailed Algorithm Analysis and Implementation Insights

5.1 Comprehensive Strategy Comparison

Table 1: Detailed Mode Comparison Matrix

Property	**Vanilla**	**Read Pref**	**Write Pref**	**Fair**
Data Integrity	X None	OK Full	OK Full	OK Full
Read Concurrency	Yes (unsafe)	Yes (safe)	Yes (safe)	Yes (safe)
Write Exclusivity	X No	OK Yes	OK Yes	OK Yes
Reader Starvation Risk	None	None	High	None
Writer Starvation Risk	None	High	None	None
Lock Overhead	0 mutexes	2 mutexes	3 mutexes	3 mutexes
Code Complexity	Trivial	Low	Medium	Medium
Best Use Case	None (demo)	90%+ reads	Write-heavy	Mixed

5.2 Lock Acquisition Ordering

One critical aspect of correct implementation is consistent lock ordering to prevent deadlocks.

Lock Hierarchy (must be respected):

1. `queue_lock` (Fair mode turnstile) - highest priority
2. `read_try` (Writer Preference reader blocker)
3. `mutex` (Counter protection)
4. `resource_lock` (Shared resource protection) - lowest priority

Violating this order can cause deadlock between threads.

5.3 Why the Counter Protection Mutex is Essential

Students often ask: "Why do we need `mutex` if we already have `resource_lock`?"

Answer: They serve different purposes!

- `resource_lock`: Protects the shared data (`shared_string`)
- `mutex`: Protects the synchronization logic (counters and conditions)

Broken Implementation Without `mutex`:

```
void reader_enter(rw_lock_t *lock) {
    // WRONG - no mutex protection!
    lock->active_readers++;  // Race condition here!
    if (lock->active_readers == 1) {
        pthread_mutex_lock(&lock->resource_lock);
    }
}
```

Failure Scenario Timeline:

Table 2: Race Condition on Counter Without Mutex

Time	Reader 1	Reader 2	active_readers
t_0			0
t_1	LOAD active_readers (0)		0
t_2		LOAD active_readers (0)	0
t_3	ADD 1 $\rightarrow$ 1		0
t_4		ADD 1 $\rightarrow$ 1	0
t_5	STORE 1		1
t_6		STORE 1	1 (WRONG!)
t_7	if (1 == 1) TRUE		1
t_8		if (1 == 1) TRUE	1
t_9	lock(resource_lock) OK		1
t_{10}		lock(resource_lock) DEADLOCK!	1

The Problem: Both readers think they are the "first" reader, both try to lock `resource_lock`.

The Solution: Use `mutex` to make the entire check-and-lock sequence atomic:

```c
void reader_enter(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->mutex);  // Atomicity start
    lock->active_readers++;
    if (lock->active_readers == 1) {
        pthread_mutex_lock(&lock->resource_lock);
    }
    pthread_mutex_unlock(&lock->mutex);  // Atomicity end
}
```

Now only ONE thread sees `active_readers == 1`, preventing the double-lock scenario.

5.4 Turnstile Pattern Deep Dive

The turnstile pattern is elegant but often confusing. Let's analyze it thoroughly.

Physical Analogy: Imagine a subway turnstile:

- People queue in arrival order
- Each person must pass through the turnstile one at a time
- The turnstile itself doesn't grant access to the train (that's the `resource_lock`)
- It just ensures FIFO ordering

Code Implementation:

```c
void reader_enter(rw_lock_t *lock) {
    pthread_mutex_lock(&lock->queue_lock);    // [A] Enter turnstile
    pthread_mutex_lock(&lock->mutex);         // [B] Access counters
    lock->active_readers++;
    if (lock->active_readers == 1) {
        pthread_mutex_lock(&lock->resource_lock); // [C] First reader
            locks resource
    }
    pthread_mutex_unlock(&lock->mutex);       // [D] Release counters
    pthread_mutex_unlock(&lock->queue_lock);  // [E] Exit turnstile
    // Now reading...
}
```

Key Insight: Steps [B], [C], [D] happen while holding `queue_lock`. This means no other thread (reader OR writer) can start these steps until current thread completes them.

Fairness Demonstration:

Arrival sequence: R1 → W1 → R2

Without Turnstile (Reader Preference):

1. R1 enters, `active_readers = 1`, locks `resource_lock`
2. W1 tries to lock `resource_lock`, BLOCKS
3. R2 enters, `active_readers = 2` (doesn't need `resource_lock`)
4. R2 starts reading even though W1 was waiting! UNFAIR!

With Turnstile (Fair):

1. R1 locks `queue_lock`, enters, unlocks `queue_lock`
2. W1 tries `queue_lock`, must wait for R1
3. R1 finishes setup, releases `queue_lock`
4. W1 gets `queue_lock`, but R1 still reading
5. R2 tries `queue_lock`, BLOCKS (W1 owns it)
6. R1 finishes, W1 can now lock `resource_lock`
7. W1 finishes, releases `queue_lock`
8. R2 gets `queue_lock` → FAIR!

5.5 Implementation Challenges

5.5.1 Challenge 1: Avoiding Deadlock

With multiple locks, deadlock is a real risk. Our solution: consistent lock ordering.

Deadlock Scenario:

```
Thread A: lock(X) -> lock(Y)
Thread B: lock(Y) -> lock(X)  // DEADLOCK!
```

Our Prevention: Always acquire in same order: `queue_lock` → `read_try` → `mutex` → `resource_lock`

5.5.2 Challenge 2: Spurious Wakeups

Though we don't use condition variables in current implementation (could be future enhancement), production systems must handle spurious wakeups:

```
// Always use while, not if!
while (condition_not_met) {
    pthread_cond_wait(&cond, &mutex);
}
```

5.5.3 Challenge 3: Integer Overflow

Risk: Theoretically, `active_readers` could overflow with billions of readers.

Mitigations:

- Use `unsigned long` or `uint64_t` instead of `int`
- Add bounds checking: `if (active_readers >= MAX_READERS) return EAGAIN;`
- Our demo uses `int` for simplicity

5.6 Performance Considerations

Lock Contention:

- `mutex` is hot - every reader and writer touches it
- Cache line bouncing on multi-core systems
- Solution: RCU (Read-Copy-Update) for read-heavy scenarios

Context Switching:

- Blocking on mutex causes context switch (1-10 microseconds)
- Multiplied by hundreds of operations = significant overhead
- Trade-off: Correctness vs. raw performance

5.7 Thread-Safe Logging Implementation

Our logger prevents interleaved output from concurrent threads:

```c
typedef struct {
    pthread_mutex_t log_mutex;
    bool initialized;
} logger_t;

void log_message(thread_type_t type, int id, const char* format, ...) {
    pthread_mutex_lock(&logger.log_mutex);

    // Generate timestamp
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    struct tm *tm_info = localtime(&ts.tv_sec);

    // Print atomic log entry
    printf("[%02d:%02d:%02d.%03ld] [%s%d] ",
           tm_info->tm_hour, tm_info->tm_min, tm_info->tm_sec,
           ts.tv_nsec / 1000000,
           type == THREAD_READER ? "R" : "W", id);

    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);

    printf("\n");
    fflush(stdout);

    pthread_mutex_unlock(&logger.log_mutex);
}
```

Without `log_mutex`, output would be garbled:

```
[12:34[12:34:56.123] :56.124] [R[W1] 2] read: wrote: "Hello""Wor
```

6 Experimental Evaluation

6.1 Test Environment

Hardware: Multi-core processor with sufficient RAM
Software: Linux OS, GCC compiler, POSIX threads

6.2 Comprehensive Automated Testing

Objective: Validate system correctness through comprehensive testing.
Test Configuration:

- Total runs: 16 (4 modes × 4 runs)
- Application: Shared String
- Writers per run: 8 threads
- Readers per run: 5 threads
- Duration per run: 8 seconds
- Modes: vanilla, reader_pref, writer_pref, fair

Automation:

1. `run_tests.sh`: Executes tests, saves timestamped logs
2. `analyze_comprehensive.py`: Parses logs, detects torn reads
3. Valid set: 20 predefined sentences (1-70 chars)
4. Torn read = any string not in valid set

6.3 Actual Results

Table 3: Test Results - All 16 Runs

Mode	Clean Runs	Avg Torn Reads
vanilla	0/4	367
reader_pref	4/4	0
writer_pref	4/4	0
fair	4/4	0

Analysis:
Vanilla Mode:

- 0/4 clean runs = 100% race condition rate
- Average 367 torn reads per run
- Examples: "Syncating systems..." (mixed), "Mutal exclusures..." (partial), single chars

- Clearly demonstrates concurrent write problem

Synchronized Modes (reader_pref, writer_pref, fair):

- 12/12 runs clean = 100% correctness
- Zero torn reads across all runs
- All read strings matched valid set perfectly

Key Findings:

1. Perfect validation: All synchronized modes 100% correct
2. Clear problem demonstration: Vanilla 0% success rate
3. No false positives in any synchronized run
4. Results reproducible across sessions

6.4 Statistical Significance

With 16 total runs:

- Vanilla mode: 0/4 clean = 0% success rate (expected for unsynchronized)
- Synchronized modes: 12/12 clean = 100% success rate
- Total operations: Hundreds of concurrent read/write operations per run
- No synchronization failures in 12 synchronized runs provides high confidence

7 Discussion

7.1 Correctness vs. Performance

Our experiments demonstrate a fundamental trade-off in concurrent systems design:

- **Vanilla mode**: Maximum performance, zero correctness
- **Preferential modes**: Correct but unfair; can starve one side
- **Fair mode**: Correct and fair, with modest performance cost

For production systems, correctness is non-negotiable, making the real choice between preferential and fair policies.

7.2 When to Use Each Mode

Reader Preference: Suitable for read-heavy workloads where:

- 90%+ operations are reads
- Writes are infrequent and can tolerate delays
- Example: Configuration caches, reference data

Writer Preference: Suitable for write-heavy scenarios where:

- Data freshness is critical
- Writes must complete promptly
- Example: Real-time monitoring, sensor data

Fair Scheduling: Suitable for mixed workloads where:

- Both reads and writes are important
- Starvation cannot be tolerated
- Example: Shared services with SLAs

7.3 Implementation Considerations

Our implementation uses coarse-grained locking (single lock for entire resource). For higher scalability, production systems use:

- **Fine-grained locking**: Multiple locks for different resource parts
- **Lock-free structures**: Atomic operations (compare-and-swap)
- **Read-Copy-Update (RCU)**: Used in Linux kernel

8 Conclusion

This paper presented a comprehensive study of the Reader-Writer synchronization problem through the implementation and evaluation of four distinct strategies. Our key findings include:

1. **Vanilla mode clearly demonstrates the problem**: 100% race condition rate with 367 torn reads per run
2. **All synchronized modes achieve perfect correctness**: 100% success rate (12/12 runs)
3. **Preferential policies have trade-offs**: Reader preference maximizes read throughput but risks writer starvation. Writer preference ensures timely updates but may delay readers.

4. **Fair scheduling eliminates starvation**: The turnstile pattern achieves balanced fairness with minimal overhead.

5. **One size doesn't fit all**: The optimal synchronization strategy depends on workload characteristics, latency requirements, and fairness constraints.

The unified API framework we developed enables easy comparison and switching between strategies, making it valuable for both education and prototyping. The shared string application demonstrates torn reads (data corruption) and the effectiveness of various synchronization approaches.

For practitioners, this work provides concrete guidance on selecting appropriate Reader-Writer implementations. For educators, our code serves as a clear demonstration of fundamental concurrency concepts using a straightforward shared string example. For researchers, it establishes a baseline for evaluating more sophisticated synchronization mechanisms.

8.1 Future Directions

Potential extensions include:

- Priority-based scheduling with different thread priorities
- Adaptive algorithms that adjust strategy based on observed workload
- Lock-free implementations using atomic operations
- Performance evaluation on NUMA architectures with non-uniform memory access costs

The complete source code, including the shared string implementation with four synchronization modes, is available as open-source educational material.

References

[1] Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.

[2] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

[3] Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley Professional.

[4] Herlihy, M., & Shavit, N. (2012). *The Art of Multiprocessor Programming* (Revised 1st ed.). Morgan Kaufmann.

[5] Courtois, P. J., Heymans, F., & Parnas, D. L. (1971). Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10), 667-668.