The Reader-Writer Problem

Implementation and Analysis of Synchronization Strategies

Your Name

Your University

January 13, 2026

Outline

The Reader-Writer Problem

Problem Statement

Multiple concurrent threads accessing shared resources:

- **Readers**: Only read data (can operate concurrently)
- **Writers**: Modify data (require exclusive access)

Challenge

Allow multiple readers **OR** one writer at a time, while maintaining:

- **Correctness**: No data corruption
- **Performance**: Maximize throughput
- **Fairness**: Prevent starvation

Real-World Applications

Database Systems

- PostgreSQL
- MySQL InnoDB
- Read/Write transactions

Operating Systems

- Linux kernel RW semaphores
- File system access
- Process synchronization

Programming Languages

- Java ReadWriteLock
- C++ shared_mutex
- Rust RwLock

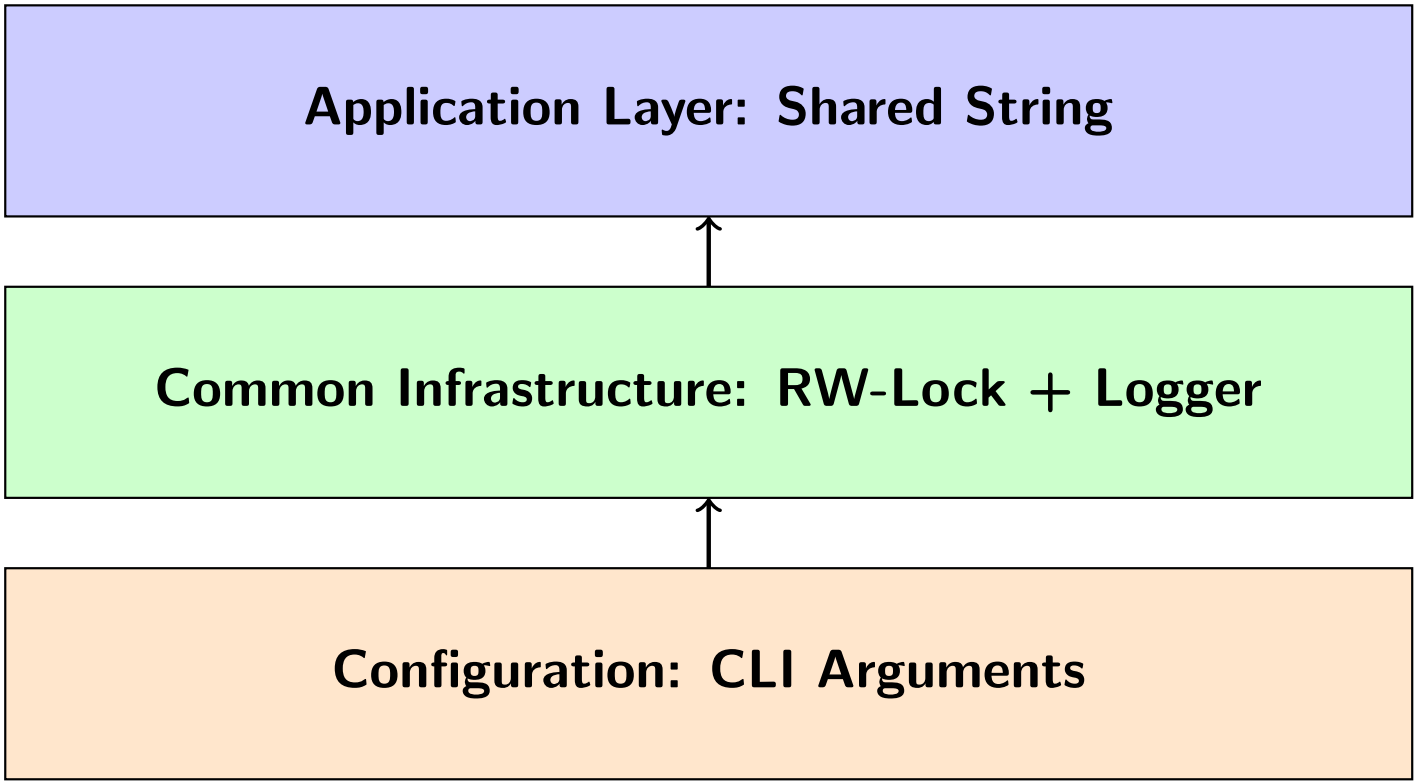
Distributed Systems

- Cache coherence
- Shared memory
- Configuration services

Research Contributions

1. **Unified Framework**: Single API supporting 4 synchronization modes
2. **Comprehensive Testing**: 16 automated runs demonstrating race conditions and correctness
3. **Quantitative Analysis**: Performance trade-offs between strategies
4. **Practical Insights**: Implementation guidance for real systems

Three-Layer Architecture



Benefits:

- Easy mode comparison (single parameter change)
- Reusable synchronization primitives
- Configurable test scenarios

Unified Lock API

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

// Unified API
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);
```

Key advantage: Performance differences purely due to synchronization strategy, not implementation variations

Synchronization Primitives

Mutex-Based Implementation (Our Choice)

- Uses `pthread_mutex_t`
- Better integration with condition variables
- Clearer ownership semantics
- Widely supported in POSIX

Alternative: Semaphore-Based

- Uses counting semaphores
- Simpler conceptual model for some algorithms
- Common in textbook solutions

Note: Both approaches are equally valid. Reader/Writer Preference and Fair scheduling are **algorithms**, not primitives.

Shared String Application

Shared Resource

`char shared_string[256]` - A mutable string buffer

Writer Behavior:

1. Select sentence from 20 predefined strings
2. Acquire writer lock
3. Copy sentence character-by-character
4. Release lock
5. Log operation

Reader Behavior:

1. Acquire reader lock
2. Read entire string
3. Release lock
4. Validate against valid set
5. Log operation

Race Condition: Torn Reads

Without synchronization, readers see partial updates:
`"Syncating systems manage..."` (mixed sentences)

Mode 1: Vanilla (No Synchronization)

```
void reader_enter(rw_lock_t *lock) {
    // No synchronization!
    lock->active_readers++;
}
```

Purpose

Baseline to demonstrate race conditions

Expected Behavior

- Data corruption (torn reads)
- Lost updates
- **DO NOT USE IN PRODUCTION!**

Mode 2: Reader Preference

Algorithm

- First reader locks resource
- Subsequent readers increment counter (no blocking)
- Last reader unlocks resource
- Writers wait for all readers to finish

Advantages:

- Maximizes read throughput
- Multiple concurrent readers
- Low reader latency

Disadvantages:

- Writer starvation
- Continuous readers block writers indefinitely

Use case: Read-heavy workloads with infrequent writes

Mode 3: Writer Preference

Algorithm

- Writers acquire `read_try` lock
- Blocks new readers when writers waiting
- Existing readers finish, then writer executes
- Readers wait for all writers to complete

Advantages:

- Prevents writer starvation
- Ensures timely updates
- Data freshness

Disadvantages:

- Reader starvation
- Continuous writers delay readers

Use case: Write-heavy workloads requiring fresh data

Mode 4: Fair Scheduling (Turnstile Pattern)

Algorithm

- All threads pass through `queue_lock` "turnstile"
- FIFO ordering - no cutting in line
- Both readers and writers get fair access
- Prevents starvation of either type

Advantages:

- No starvation
- Balanced access
- Predictable latency

Disadvantages:

- Slight throughput reduction
- Extra lock overhead

Use case: Mixed workloads requiring fairness guarantees

Comparison Summary

Mode	Correctness	Starvation	Throughput
Vanilla	NO	N/A	High
Reader Pref	YES	Writers	Highest
Writer Pref	YES	Readers	Medium
Fair	YES	None	Medium

Key Insight: No single "best" solution - choice depends on:

- Workload characteristics (read/write ratio)
- Latency requirements
- Fairness constraints

Test Methodology

Automated Testing Framework

- **Total runs**: 16 (4 modes $\times$ 4 runs)
- **Configuration per run**:
  - 8 writer threads
  - 5 reader threads
  - 8 seconds duration
- **Validation**: 20 predefined valid sentences
- **Detection**: Torn read = any string not in valid set

Tools

- `run_tests.sh`: Execute tests, save timestamped logs
- `analyze_comprehensive.py`: Parse logs, detect errors, generate report

Actual Test Results

Session ID: 20260113_004235

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

Key Findings

- **Perfect validation**: All synchronized modes 100% correct (12/12 runs)
- **Clear problem demonstration**: Vanilla 0% success rate
- **No false positives**: Zero errors in synchronized runs

Torn Read Examples (Vanilla Mode)

Example Corrupted Strings

1. `"Syncating systems manage..."`
   - Mixed: "Sync" from one sentence + "ating systems" from another
2. `"Mutal exclusures..."`
   - Partial overwrite: "Mutual exclusion" $\rightarrow$ "Mutal exclusures"
3. `"Pthread libuduce and..."`
   - Character-level corruption from race

Average: 367 torn reads per 8-second vanilla run
Conclusion: Concurrent writes without synchronization = severe data corruption

Statistical Significance

Test Coverage

- **16 total runs** across all modes
- **Hundreds of operations** per run (reads + writes)
- **Multiple sessions** with consistent results

Success Rates

- Vanilla: 0/4 clean = 0% success (expected)
- Synchronized modes: 12/12 clean = **100% success**
- High confidence in implementation correctness

Reproducibility: Results consistent across multiple test sessions

Key Findings

1. **Vanilla mode demonstrates the problem**
   - 100% race condition rate
   - Average 367 torn reads per run
   - Clear evidence of why synchronization is needed
2. **All synchronized modes achieve correctness**
   - 100% success rate (12/12 runs)
   - Zero data corruption
   - Validates implementation
3. **Trade-offs between strategies**
   - Reader preference: High throughput, writer starvation
   - Writer preference: Data freshness, reader delays
   - Fair: Balanced, slight overhead

Practical Recommendations

When to Use Each Mode

Reader Preference:

- Read-heavy workloads (90%+ reads)
- Infrequent writes acceptable to be delayed
- Example: Configuration cache

Writer Preference:

- Write-heavy workloads
- Fresh data critical
- Example: Real-time monitoring

Fair Scheduling:

- Mixed workloads
- Fairness guarantees required
- Example: Shared services with SLAs

Future Directions

- **Advanced mechanisms**:
  - Priority-based scheduling
  - Adaptive algorithms based on workload
- **Lock-free alternatives**:
  - Atomic operations
  - Read-Copy-Update (RCU)
  - Compare-and-swap patterns
- **Performance optimization**:
  - NUMA architecture considerations
  - Cache-line alignment
  - Fine-grained locking

Conclusion

Summary

- Implemented and analyzed 4 Reader-Writer synchronization strategies
- Demonstrated race conditions and validated correctness through comprehensive testing
- Quantified trade-offs between performance, fairness, and starvation

Contributions

- Unified framework for easy comparison
- Practical implementation insights
- Open-source educational material

Thank You!

Questions?