OS Project

Topic 6. The Reader-Writer Problem

Tran Quang Hung 20235502
Nguyen Xuan Khai 20235508

Hanoi University of Science and Technology
Supervisor: PhD. Do Quoc Huy

January 13, 2026

Outline

INTRODUCTION

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

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

IMPLEMENTATION

Demonstration

```c
char shared_string[STRING_SIZE];
// Predefined sentences for writers to cycle through
const char* sentences[] = {
    "A",
    "Hello World!",
    "The quick brown fox jumps over the lazy dog.",
    "Operating systems manage hardware and software resources.",
    "X",
    "Synchronization prevents race conditions in concurrent programs.",
    "Readers and writers must coordinate access to shared data.",
    "Race!",
    "Mutual exclusion ensures only one writer at a time.",
    "Pthread library provides powerful threading primitives.",
    "Concurrency bugs are difficult to reproduce and debug consistently.",
    "AB",
    "Memory barriers ensure proper ordering of operations across cores.",
    "Deadlock occurs when threads wait indefinitely for each other.",
    "Test",
    "Lock-free data structures use atomic operations for synchronization.",
    "Thread pools improve performance by reusing worker threads efficiently.",
    "!",
    "Context switching between threads has performance overhead costs.",
    "Critical sections must be kept as short as possible for efficiency."
};
```

Demonstration

```
[00:42:42.170] [W2] wrote: "Thread pools improve performance by reusing worker threads efficiently."
[00:42:42.175] [W5] wrote: "Memory barriers ensure proper ordering of operations across cores."
[00:42:42.179] [W4] wrote: "Mutual exclusion ensures only one writer at a time."
[00:42:42.182] [R3] read: "Lock-free data structures use atomic operat a time."
[00:42:42.183] [R5] read: "Lock-free data structures use atomic operatio time."
[00:42:42.184] [R1] read: "Lock-free data structures use atomic operations for synchss cores."
[00:42:42.186] [W3] wrote: "Lock-free data structures use atomic operations for synchronization."
[00:42:42.187] [R2] read: "Lock-free data structures use atomic operations for synchronization."
[00:42:42.189] [W8] wrote: "Lock-free data structures use atomic operations for synchronization."
[00:42:42.194] [W6] wrote: "Lock-free data structures use atomic operations for synchronization."
[00:42:42.203] [R5] read: "Thread pe data structures use atomic operations for synchronization."
[00:42:42.204] [R4] read: "Thread pools imptructures use atomic operations for synchronization."
[00:42:42.212] [W7] wrote: "Thread pools improve performance by reusing worker threads efficiently."
```

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

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)

SYNCHRONIZATION MODES

Mode 1: Vanilla (No Synchronization)

```c
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
Vanilla	**NO**	N/A
Reader Pref	YES	Writers
Writer Pref	YES	Readers
Fair	YES	None

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

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

EXPERIMENTAL RESULTS

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

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

CONCLUSION

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

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

Thank You for listening!