Advanced Operating Systems

AOS Project Report

Simulation and Analysis of the Readers-Writers Problem

Authors:

Khelifi Ayyoub

Chaalal Djewed

Department of Computer Science

USTO

December 12, 2025

Abstract

This report examines the classical Readers-Writers problem in concurrent programming. We explore the synchronization requirements needed to maintain data integrity when multiple threads access shared resources. Three synchronization strategies are analyzed: Reader Priority, Writer Priority, and Fair (First-Come-First-Served). We present an implementation using Python's `threading` and `PyQt6` libraries that visualizes thread behavior under each strategy, demonstrating how they prevent race conditions and handle potential starvation scenarios.

Contents

Chapter 1

Introduction

The Readers-Writers problem is a classic synchronization challenge in operating systems and database theory. It models a scenario where multiple threads need access to a shared resource like a database or file. These threads fall into two categories:

- **Readers**: threads that only read data without modifying it. Multiple readers can safely access the resource at the same time.
- **Writers**: threads that modify the data and require exclusive access. No other thread (reader or writer) can access the resource while a writer is active.

The main challenge is coordinating these accesses to prevent race conditions and data inconsistency, while also maximizing concurrency and avoiding starvation (where a thread waits indefinitely). This report presents a simulator that addresses these challenges through different synchronization policies.

Chapter 2

Background and Theory

2.1 Problem Definition

Let R represent the number of active readers and W represent the number of active writers. The critical section must satisfy these constraints:

$$|W| \leq 1 \tag{2.1}$$

$$|W| = 1 \implies |R| = 0 \tag{2.2}$$

$$|R| > 0 \implies |W| = 0 \tag{2.3}$$

These constraints ensure mutual exclusion for writers while permitting concurrent reader access.

2.2 Synchronization Strategies

2.2.1 Reader Priority

With reader priority, readers are never blocked unless a writer is currently using the shared resource. New readers can always join if other readers are active, even if writers are waiting.

- **Advantage**: Maximizes read throughput.
- **Drawback**: Writers can be starved if readers arrive frequently enough to keep the resource continuously occupied.

2.2.2 Writer Priority

Under writer priority, once a writer requests access, no new readers can start reading. All waiting writers must complete before new readers are admitted.

- **Advantage**: Ensures writers get timely access and data stays fresh.
- **Drawback**: Readers can be starved if writers arrive frequently.

2.2.3 Fair Strategy

The fair approach prevents starvation by serving all requests in First-In-First-Out order, regardless of thread type. Each thread waits only for those that arrived before it.

- **Advantage**: No thread can be starved indefinitely.
- **Drawback**: May reduce overall throughput compared to priority-based approaches.

Chapter 3

Implementation

Our solution is implemented in Python using the `threading` module for synchronization and `PyQt6` for the graphical interface.

3.1 Architecture

The implementation uses a class hierarchy built around `ReadWriteLock`, which defines the interface for lock operations.

3.1.1 Base Class: `ReadWriteLock`

This abstract class defines the locking interface:

```
class ReadWriteLock:
    def start_read(self, thread_id): pass
    def end_read(self, thread_id): pass
    def start_write(self, thread_id): pass
    def end_write(self, thread_id): pass
```

3.1.2 Reader Priority Implementation

This implementation uses `threading.Condition` variables. Readers wait only when a writer is actively using the resource:

```
def start_read(self, thread_id):
    with self.mutex:
        # Wait only if a writer is active
        while self.active_writers > 0:
            self.readers_cond.wait()
        self.active_readers += 1
```

Writers must wait for all active readers to finish before gaining access.

3.1.3 Writer Priority Implementation

To give writers priority, readers must wait if any writer is either active or waiting:

```
def start_read(self, thread_id):
    with self.mutex:
        # Block if writers are active OR waiting
        while self.active_writers > 0 or self.waiting_writers > 0:
            self.readers_cond.wait()
        self.active_readers += 1
```

The check for `waiting_writers > 0` ensures queued writers get access before newly arriving readers.

3.1.4 Fair Strategy Implementation (True RW Lock)

The fair strategy is implemented in `TrueFairRWLock` using a FIFO queue to enforce strict ordering. This effectively solves the "Third Readers-Writers Problem."

```
class TrueFairRWLock(ReadWriteLock):
    def __init__(self):
        self.queue = deque()  # Stores (thread_id, type, condition)

    def start_read(self, thread_id):
        with self.lock:
            me = threading.Condition(self.lock)
            self.queue.append((thread_id, 'R', me))
            while True:
                # Enter if no active writer AND I am at the head
                # (or head is reader and I am a consecutive reader)
                if self.can_enter_read(thread_id):
                    self.queue.popleft()
                    break
                me.wait()
```

3.1.5 Adaptive Strategy (Aging)

We implemented an advanced "Adaptive" lock (also known as priority inheritance or aging). By default, it operates in **Reader Priority** to maximize throughput. However, if a Writer waits longer than a defined threshold ($T > 1.0s$), the system dynamically switches to **Strict Writer Priority** to prevent starvation.

```
class AdaptiveRWLock(ReadWriteLock):
    def start_read(self, thread_id):
        while True:
            # If Writer Starving -> Block new readers
            if self.should_panic() and self.waiting_writers > 0:
                 self.condition.wait()
                 continue
            # ... Normal logic ...
```

This hybrid approach empirically offers the highest performance while maintaining a hard guarantee against infinite blocking.

3.2 Thread Simulation

The `GenericWorker` class (extending `QThread`) simulates a thread's lifecycle:

1. **Request**: Call `start_read` or `start_write` and measure wait time.
2. **Critical Section**: Simulate work with sleep and update shared data
3. **Release**: Call `end_read` or `end_write`

Chapter 4

Results and Analysis

The application provides real-time visualization of thread synchronization.

4.1 Interface Components

The interface consists of four main parts:

1. **Control Panel**: Allows dynamic thread creation and strategy selection.
2. **Metrics Dashboard**: Displays real-time performance data:
    - Average and Maximum Wait Times.
    - System Throughput (ops/sec).
    - Total serviced Readers vs Writers.
3. **Visualizer**:
    - Shows a central "Data" node.
    - Reader threads appear as orbiting nodes.
    - Writer threads overlay the center (exclusive access).
4. **Event Log**: A timestamped record of all operations.

4.2 Observed Behavior

Testing revealed the expected behavior for each strategy:

- **Reader Priority**: Continuous addition of readers prevented writers from entering. **Metric observed**: Writer Max Wait Time increased indefinitely while Reader Wait Time remained low.
- **Writer Priority**: Writers took precedence, blocking new readers immediately. **Metric observed**: Reader Max Wait Time spiked during write bursts.
- **Fair Strategy (FIFO)**: Threads accessed the resource in strict arrival order. **Metric observed**: Wait times were distributed evenly relative to queue position; zero starvation occurred.

Chapter 5

Conclusion

The Readers-Writers problem highlights the inherent trade-offs between concurrency and fairness in system design. Our implementation successfully demonstrates these trade-offs through visualization. Reader Priority maximizes read concurrency but risks starving writers. Writer Priority ensures data freshness but can starve readers. The Fair strategy balances both concerns, making it suitable for systems where starvation is unacceptable. The simulation confirms that our Python lock implementations correctly enforce these behaviors using standard synchronization primitives.