2026

Semester Project
Operating System

Title : Readers & Writers Problem with writer's priority

Presented to: Mam AMMARA
Presented by: ZUHAJUNAID 70135604
SIMRA NADEEM 70141978

Table of Content

01

Abstract:

This project implements the classical Readers-Writers synchronization problem with writer priority using Python's threading module. The implementation demonstrates crucial operating system concepts including mutual exclusion, semaphores, thread synchronization, and starvation prevention.

The efficient use of system resources like the CPU, memory, and threads in a modern computer system is essential in promoting high performance. The Operating System features an integral part in the management of these resources, particularly in situations where the need to execute several tasks at the same time goes into consideration.

A Streamlit-based web interface provides real-time visualization of thread execution, system statistics, and interactive concept diagrams. Additionally, a Jupyter Notebook implementation offers an educational platform for understanding the synchronization mechanisms step-by-step.

Key Features:

- Writer priority mechanism to prevent writer starvation
- Real-time visualization of thread states
- Live metrics and execution logs
- Interactive concept diagrams
- Data consistency verification
- Thread-safe synchronization using Python's concurrency libraries

1. Introduction:

1.1. Background

The Readers-Writers problem is a fundamental synchronization challenge in concurrent systems where multiple processes need coordinated access to a shared resource. One of the tools that support the concept of concurrency is multithreading.

However, managing concurrent access to shared resources is complex. Without proper synchronization:

- Race conditions can corrupt shared data
- Writers may starve if readers continuously arrive
- Data inconsistency occurs without mutual exclusion
- Deadlock may happen with improper lock ordering ■ This problem is prevalent in:
- Database management systems (concurrent queries vs updates)
- File systems (multiple read operations vs write operations)
- Cache coherence protocols
- Web servers (GET requests vs POST/PUT requests)

1.2. Problem Context

In systems where read operations vastly outnumber write operations, continuous reader arrivals can lead to writer starvation. A new mechanism is needed to overcome this issue by implementing writer priority.

The project applies fundamental Operating System concepts such as multithreading, synchronization, mutual exclusion, and resource management. Rather than allowing uncontrolled access, our implementation uses a structured approach where:

- Tasks are managed through proper synchronization
- Threads are coordinated using locks and condition variables
- Resources are accessed in a controlled, safe manner

1.3. Project Scope

This project is implemented using Python's internal concurrency libraries and runs in both Streamlit (for interactive visualization) and Google Colab/Jupyter Notebook to remain environment-independent and easy to debug.

The implementation focuses on:

- ✓ Understanding the Readers-Writers synchronization problem
- ✓ Implementing writer priority to avoid writer starvation
- ✓ Simulating concurrent readers and writers using Python threads
- ✓ Applying OS concepts: synchronization, mutual exclusion, concurrency
- ✓ Visualizing thread execution through interactive simulator
- ✓ Verifying data consistency and correctness

2. Problem Statement:

2.1. Core Challenge

Design and implement a synchronization mechanism where:

- Multiple readers can access shared data simultaneously
- Writers require exclusive access (no concurrent readers or writers)
- Writers have priority over new readers
- No thread experiences indefinite starvation

This is similar to a producer-consumer paradigm where:

- Producers (Writers) generate/modify data
- Consumers (Readers) access/read data
- Both must coordinate through synchronization mechanisms

2.2. Constraints

- Mutual Exclusion: Writers must have exclusive access to shared resource
- Concurrent Reads: Multiple readers can read simultaneously without interference
- Writer Priority: When a writer requests access, no new readers should enter until the writer completes
- Progress: The system must not deadlock
- Bounded Waiting: No thread should wait indefinitely
- Efficient Resource Usage: Threads should not perform busy waiting

2.3. Success Criteria

The implementation is considered successful if:

- ✓ Data consistency is maintained (shared_data == total_writes)
- ✓ Multiple readers can read concurrently

✓ Writers get exclusive access

✓ Writer priority is enforced (new readers block when writer waits)

✓ No deadlock occurs

✓ System demonstrates fair resource allocation ✓ CPU resources are used efficiently (no busy waiting)

3. Objectives:

The main objectives of this project are:

i. To understand the Readers-Writers synchronization problem

- Study classical solutions (Reader Priority, Writer Priority, Fair)

- Analyze real-world applications and use cases **ii.** To implement writer priority to avoid writer starvation

- Use waiting_writers counter for priority signaling - Block new readers when writers are waiting **iii.** To simulate concurrent readers and writers using Python threads

- Create multiple reader threads that read simultaneously

- Create writer threads that write exclusively - Manage thread lifecycle and coordination **iv.** To apply Operating System concepts

- Multithreading: Creating and managing multiple concurrent threads

- Synchronization: Coordinating thread execution

- Mutual Exclusion: Preventing race conditions using locks

- Condition Variables: Efficient thread waiting and signaling

- Producer-Consumer Pattern: Task queue management

v. To visualize thread execution using a simulator-based approach

- Real-time logs showing thread states

- Live metrics (active readers, waiting writers, shared data) - Interactive graphs showing concurrency over time **vi.** To ensure data consistency and correctness

- Verify that shared_data matches total_writes

- Test under various configurations (different reader/writer counts)

- Stress test with high concurrency scenarios

4. Methodolgy:

The methodology of this project follows a structured approach to design and implement a synchronization system that efficiently manages concurrent access to shared resources.

4.1. System Design Overview

The readers-writers system consists of the following main components:

➢ Main Thread (Controller)

  - Initializes the system
  - Creates reader and writer threads

- Manages simulation lifecycle ➢ Reader Threads (Consumers)
    - Multiple readers access shared data concurrently
    - Check writer priority before entry
    - Coordinate through reader_count variable

➢ Writer Threads (Producers/Modifiers)
- Writers modify shared data exclusively
- Signal priority via waiting_writers counter
- Ensure data consistency through exclusive access

➢ Shared Resource
- shared_data: The actual resource being accessed
- Protected by synchronization mechanisms

➢ Synchronization Mechanisms
- Mutex Lock: Protects counters
- Resource Lock: Controls data access
- Condition Variables: Enable efficient waiting

➢ Monitoring System
- Real-time logging
- Metrics collection
- Statistics tracking

4.2. Task Creation and Management

Reader Tasks

A reader task is a unit of work that reads from the shared resource. Each reader:

- Checks if any writer is waiting (priority check)
- Enters critical section if safe
- Reads the shared data
- Exits and updates counters

The reader tasks are submitted continuously during the simulation period.

Writer Tasks

- A writer task modifies the shared resource. Each writer:
- Signals its intent by incrementing waiting_writers
- Waits for all current readers to finish
- Gains exclusive access to the resource
- Writes/modifies the data
- Notifies waiting readers

4.3. Synchronization and Mutual Exclusion

To prevent race conditions and ensure safe access to shared resources, synchronization mechanisms are used:

Mutex Locks

Purpose: Ensure mutual exclusion while accessing shared counters

python

with self.mutex: self.reader_count += 1 #

Protected operation This ensures that:

- ✓ Only one thread accesses the counter at a time
- ✓ No race conditions on reader_count or waiting_writers

Condition Variables

Purpose: Allow threads to wait efficiently when conditions are not met

python

 # Reader waits if writer has priority

while self.waiting_writers > 0:

self.can_read.wait() # Blocks without CPU usage

self.can_read.notify_all() # Wakes all waiting readers Benefits:

- ✓ Threads do not perform busy waiting (no CPU wastage)
- ✓ Automatic thread scheduling by OS
- ✓ Efficient context switching

Resource Lock

Purpose: Ensures exclusive write access

python ***with self.resource_lock:***

self.shared_data += 1 # Exclusive write

4.4. Thread Execution Flow

Reader Thread Execution

Each reader thread continuously performs the following steps: *a.*

Entry Section:

- Acquire mutex
- Check waiting_writers (Writer Priority Check)
- If writer waiting: block on can_read condition
- Increment reader_count
- First reader acquires resource_lock b.

Critical Section:

- Read shared_data
- Simulate read operation (1-2 seconds)
- Log read event

c. *Exit Section:*

- Acquire mutex
- Decrement reader_count
- Last reader releases resource_lock
- Notify waiting writers

If no task is available or writer has priority, the thread enters a blocked state instead of consuming CPU resources.

Writer Thread Execution

Each writer thread performs: *a.*

Entry Section:

- Acquire mutex
- Increment waiting_writers (Priority Signal)
- Wait on can_write until reader_count == 0 ◆ Decrement waiting_writers *b. Critical Section:*
- Acquire resource_lock
- Write to shared_data
- Simulate write operation (1.5-2.5 seconds) ◆ Log write event

c. Exit Section:

- Release resource_lock
- Notify all waiting readers via can_read

4.5. Graceful Shutdown

A graceful shutdown feature is implemented to terminate threads in a controlled manner:

- Simulation runs for specified duration
- Threads check time.time() < end_time in their loop
- When time expires, threads naturally exit
- No abrupt termination or resource leaks

4.6. Execution Environment

The project is implemented and tested using:

- Streamlit Web Interface:

- Interactive real-time dashboard

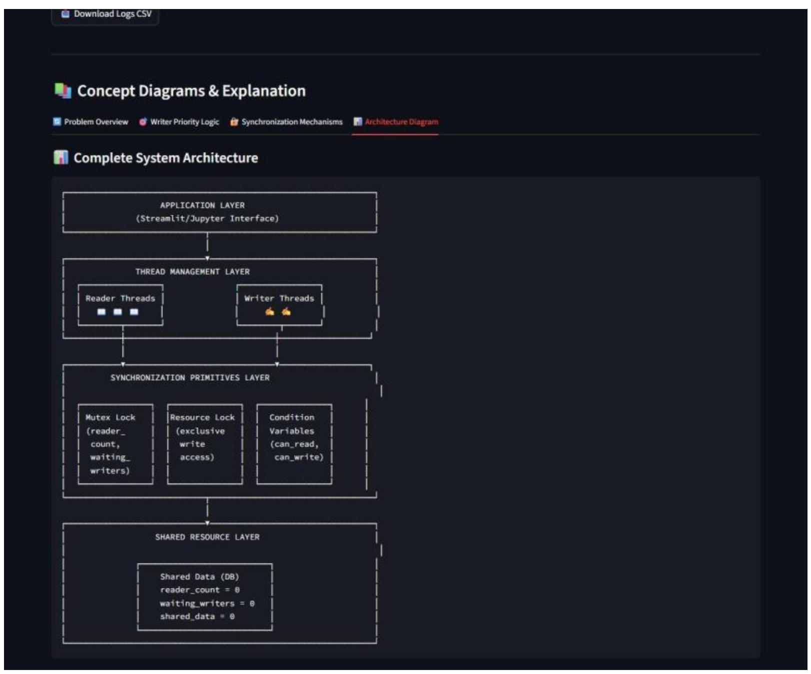
- ❖ Live logs and metrics
- ❖ Graphical visualization
- ❖ Concept diagrams

- Google Colab / Jupyter Notebook:

- ❖ Cloud-based execution
- ❖ No local installation requirements
- ❖ Easy testing and visualization of results
- ❖ Step-by-step execution for educational purposes

5. System Design Overview:

5.1 Architecture Overview



5.2 Component Description

5.2.1 Thread Components

Reader Threads (Worker Threads - Consumers):

Fixed number created during initialization

Execute read operations concurrently

Check writer priority before entry

Coordinate through reader_count variable

Block efficiently when writer has priority

Writer Threads (Worker Threads - Producers):

Fixed number created during initialization Execute write operations exclusively

Signal priority via waiting_writers counter

Wait for all readers to finish before writing

Notify waiting readers after completion 5.2.2

Synchronization Mechanisms

1. Mutex Lock (threading.Lock):

Protects reader_count and waiting_writers

Ensures atomic updates to counters

Prevents race conditions on shared variables

Lightweight, used for short critical sections

2. Resource Lock (threading.Lock):

Controls access to shared_data

Ensures exclusive write access

First reader acquires, last reader releases Held for longer duration during read/write operations

3. Condition Variables

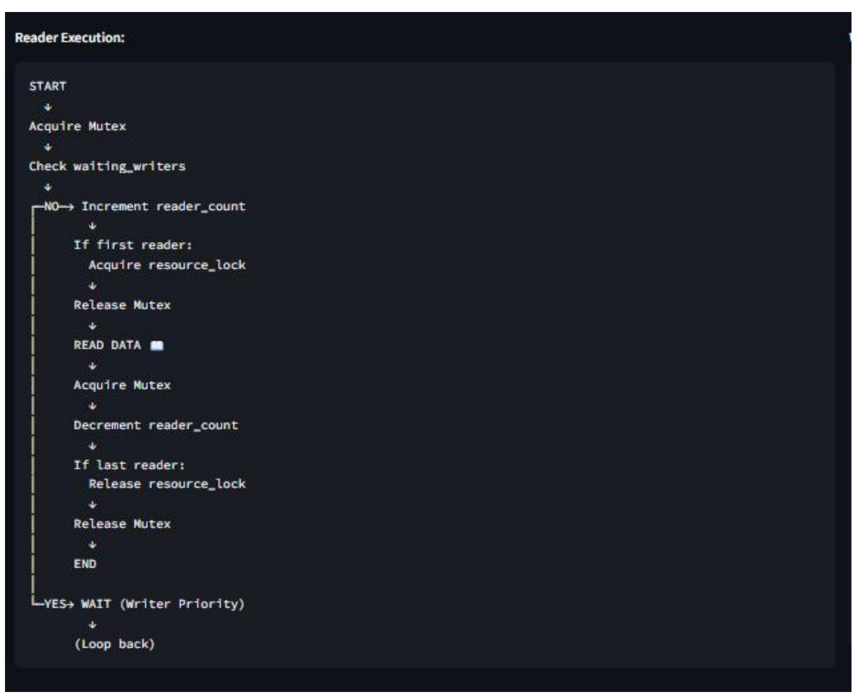
(threading.Condition): can_read: Signals readers when it's safe to proceed can_write: Signals writers when all readers finish Built on top of mutex for thread coordination

Enables efficient waiting (no busy loops)

6. Implementation details:

6.1 Core Algorithms

Reader Entry Protocol



Writer Entry Protocol

Writer Execution:

```
START
  ↓
Acquire Mutex
  ↓
Increment waiting_writers 🎯
  ↓
Release Mutex
  ↓
Acquire resource_lock
(Waits for readers)
  ↓
Acquire Mutex
  ↓
Decrement waiting_writers
  ↓
Release Mutex
  ↓
WRITE DATA ✍️
  ↓
Release resource_lock
  ↓
Acquire Mutex
  ↓
Notify all waiting readers
  ↓
Release Mutex
  ↓
END
```

6.2 Key Implementation Decisions

<u>*Why Condition Variables Instead of Busy Waiting?*</u>

Traditional Approach (Busy Waiting): **while self.waiting_writers > 0: pass # ❌ Wastes CPU cycles** Our Approach (Condition Variables):

while self.waiting_writers > 0: self.can_read.wait() # ✅ CPU efficient, blocks thread Benefits:

- No CPU wastage: Thread is blocked, not spinning
- Automatic scheduling: OS handles thread wakeup
- Better performance: Other threads can use CPU
- Scalability: Works well with many threads Why Separate Mutex and Resource Lock?

Design Decision:

- ❖ Mutex Lock: Lightweight, protects small critical sections (counter updates)
- ❖ Resource Lock: Can be held longer during actual read/write operations *Advantages:*

- ✓ Reduces contention on mutex
- ✓ Allows multiple readers to coordinate efficiently
- ✓ Prevents unnecessary blocking Why Track Statistics?

The implementation tracks: total_reads:

Verifies throughput total_writes: Used

for consistency check

max_concurrent_readers: Proves concurrent reading works Purpose:

Shows that synchronization is working correctly!

7. Code Documenatation:

7.1 Class Structure

class ReadersWritersProblem: """ Implements Readers-Writers problem with Writer Priority This class manages concurrent access to a shared resource where multiple readers can read simultaneously, but writers need exclusive access. Writer priority prevents writer starvation. Attributes: reader_count (int): Number of active readers waiting_writers (int): Number of writers waiting for access resource_lock (Lock): Controls exclusive access to shared_data mutex (Lock): Protects reader_count and waiting_writers can_read (Condition): Signals readers when safe to proceed can_write (Condition): Signals writers when readers finish shared_data (int): The actual shared resource logs (list): Execution event logs stats (dict): Simulation statistics """

7.2 Critical Code Sections with Explanation

IMPORTANT CODE 1:

```
import threading import time import random from
datetime import datetime import pandas as pd from
IPython.display import display, clear_output import
ipywidgets as widgets

print("✅ All libraries imported successfully!")
print("=" * 50)
```

IMPORTANT CODE 2:

```
class ReadersWritersProblem:
    def __init__(self):
        self.reader_count = 0 self.waiting_writers = 0 self.resource_lock
        = threading.Lock() # Controls access to shared
resource self.mutex = threading.Lock() # Protects reader_count and
waiting_writers self.can_read = threading.Condition(self.mutex) #
Condition for readers to wait if writers are priority self.can_write =
threading.Condition(self.mutex) # Condition for writers (not strictly
needed but for clarity)
        self.shared_data = 0
        self.logs = []
        self.stats = {'total_reads': 0, 'total_writes': 0,
'max_concurrent_readers': 0}

    def log_event(self, event):
        timestamp = datetime.now().strftime("%H:%M:%S.%f")[:-
        3] log_entry = f"[{timestamp}] {event}"
        self.logs.append(log_entry) print(log_entry)
```

```
def reader(self, reader_id, end_time):
    while time.time() < end_time:
        with self.mutex:
```

```
                while self.waiting_writers > 0: # Wait if writers have
priority self.log_event(f" Reader {reader_id} WAITING (Writer
priority)") self.can_read.wait()
                self.reader_count += 1
                self.stats['max_concurrent_readers'] =
max(self.stats['max_concurrent_readers'], self.reader_count)
            self.log_event(f" Reader {reader_id} READING | Data =
{self.shared_data} | Readers = {self.reader_count}")
            self.stats['total_reads'] += 1
            time.sleep(random.uniform(1, 2)) with
            self.mutex:
                self.reader_count -= 1 self.log_event(f"✅ Reader
                {reader_id} finished | Readers =
{self.reader_count}") if
                self.reader_count == 0:
                     self.can_write.notify() # Notify waiting writers if
no readers left
            time.sleep(random.uniform(2, 4))
```

```
    def writer(self, writer_id, end_time):
        while time.time() < end_time:
            with self.mutex:
                self.waiting_writers += 1

                self.log_event(f"⌛ Writer {writer_id} WAITING | Waiting
Writers = {self.waiting_writers}") while self.reader_count > 0: # Wait
                for all readers to
finish self.can_write.wait()
                self.waiting_writers -= 1
            with self.resource_lock:
                old_value = self.shared_data self.log_event(f" Writer
                {writer_id} WRITING | Old Data =
{old_value}") self.shared_data += 1
                time.sleep(random.uniform(1.5, 2.5))
                self.log_event(f" Writer {writer_id} finished | New Data
= {self.shared_data}")
                self.stats['total_writes'] += 1
            with self.mutex:
                self.can_read.notify_all() # Notify waiting readers after
write time.sleep(random.uniform(3, 5))


print("✅ ReadersWritersProblem class defined successfully!")
```

IMPORTANT CODE 3:

```
def run_simulation(num_readers=3, num_writers=2, duration=15):
    print("=" * 70) print(" STARTING SIMULATION WITH
    WRITER PRIORITY") print("=" * 70)
    print(f" Config: Readers={num_readers}, Writers={num_writers},
Duration={duration}s") print("=" *
    70) rw = ReadersWritersProblem()
    threads = [] end_time =
    time.time() + duration for i in
    range(num_readers):
        t = threading.Thread(target=rw.reader, args=(i+1,
        end_time)) threads.append(t) t.start()
    for i in range(num_writers):
        t = threading.Thread(target=rw.writer, args=(i+1,
        end_time)) threads.append(t) t.start()
    time.sleep(duration) print("\n" + "=" * 70) print(" STATS") print("="
*  70)  print(f"Total  Reads:  {rw.stats['total_reads']}")  print(f"Total
Writes:  {rw.stats['total_writes']}")  print(f"Max  Concurrent  Readers:
{rw.stats['max_concurrent_readers']}")        print(f"Final          Data:
{rw.shared_data}") print("=" * 70) return rw # Run the simulation result =
run_simulation()
```

IMPORTANT CODE 4:

```
def interactive_simulation(readers, writers, duration):
    clear_output(wait=True)
    run_simulation(readers, writers, duration)

widgets.interact( interactive_simulation,
    readers=widgets.IntSlider(min=1, max=10,
    value=3,
description='Readers:'),
```

```
    writers=widgets.IntSlider(min=1, max=5, value=2,
description='Writers:'), duration=widgets.IntSlider(min=5,
    max=30, step=5, value=15,
description='Duration (s):')
)
```

Important code 5:

```
import pandas as pd

# example logs (agar tumhare paas already list hai to usay use karo)
logs = [
    "Simulation Started",
    "Reader 1 reading",
    "Writer 1 writing",
    "Writer finished",
    "Reader finished"
]
shared_values = [0, 0, 1, 1, 2]
df = pd.DataFrame({
    "step": range(len(logs)),
    "log": logs,
    "shared_data": shared_values
})
df.to_csv("simulation_data.csv", index=False)

print("✅ Data saved to simulation_data.csv")
```

8. Tools and Technologies:

8.1 Python Language

- ✓ Simplicity: Easy to understand syntax
- ✓ Readability: Code is self-documenting
- ✓ Rich Libraries: Powerful concurrency support

8.2 Python Libraries Used

Library	Purpose	Key Features
threading	Creating and managing threads	Thread class, Lock, Condition, Semaphore
queue	Thread-safe task queue	FIFO queue, blocking operations
time	Simulating task execution delays	sleep(), time() for timing
datetime	Timestamp logging	Precise time formatting
pandas	Data manipulation and CSV export	DataFrame operations
streamlit	Interactive web interface	Real-time updates, charts, widgets
random	Simulating variable execution times	uniform() for realistic delays

8.3 Development Environment

Jupyter Notebook in VS Code

- Cloud-based execution
- No local installation requirements
- Easy debugging and output visualization
- Step-by-step code execution
- Rich output formatting (logs, tables, graphs) ✧ Shareable via link

Use Case: Educational demonstrations, testing, development

Streamlit

- ✧ Interactive web interface
- ✧ Real-time updates
- ✧ Beautiful visualizations
- ✧ Easy deployment
- ✧ Professional presentation

Use Case: Project demonstrations, presentations, production-ready simulator

9. Operating System Concepts Applied:

9.1 Multithreading

Multiple threads execute concurrently within a single process.

Implementation:

threads = []for i in range(num_readers): t = threading.Thread(target=self.reader, args=(i+1, end_time)) threads.append(t) t.start() Learning:

- Thread creation and lifecycle

- Concurrent execution on multi-core CPUs

- Thread scheduling by OS

9.2 Thread Scheduling

OS decides which thread runs when.

Observation:

- ✓ Threads don't always run in creation order
- ✓ OS uses scheduling algorithms (Round Robin, Priority-based)
- ✓ Context switching between threads
- ✓ Evidence: Check simulation logs - execution order varies!

9.3 Mutual Exclusion

Only one thread can access a critical section at a time. Implementation:

with self.mutex: self.reader_count += 1 # Protected **Prevents:**

- ✧ Race conditions ✧ Data corruption
- ✧ Inconsistent state

9.4 Synchronization

Coordinating thread execution to maintain correctness.

Mechanisms Used:

- ✓ Locks (Mutex): Binary semaphores for mutual exclusion
- ✓ Condition Variables: Efficient waiting and signaling
- ✓ Counters: reader_count, waiting_writers for state tracking
- ✓ Result: Safe, coordinated access to shared resources

9.5 Producer-Consumer Problem

Mapping to Our Problem:

Role	Producer-Consumer	Readers-Writers
Producer	Creates items	Writers (modify data)
Consumer	Consumes items	Readers (access data)
Buffer	Shared queue	Shared resource (data)
Coordination	Full/Empty signals	Reader/Writer conditions

Similarity: Both require careful synchronization to prevent conflicts!

9.6 Resource Management

Efficient allocation and usage of system resources.

Our Implementation:

CPU: Condition variables prevent busy waiting (CPU efficient)

Memory: Shared data accessed, not copied (memory efficient)

Threads: Reused throughout simulation (no creation overhead)

Locks: Held only when necessary (reduces contention)

9.7 Deadlock Prevention

Potential Deadlock Scenario:

- Reader holds mutex, waits for resource_lock
- Writer holds resource_lock, waits for mutex - Result: Deadlock!

Our Prevention Strategy:

- ❖ Lock Ordering: Always acquire locks in same order
- ❖ No Circular Wait: Linear lock hierarchy
- ❖ Release Promptly: Locks released as soon as possible
- ❖ Condition Variables: Wait without holding locks
- ❖ Verification: No deadlocks observed in testing ✓

9.8 Starvation Prevention

Reader Starvation: Not possible in writer priority

- Writers eventually finish

- Readers then get access

Writer Starvation: Prevented by priority mechanism

- waiting_writers counter blocks new readers

- Writer guaranteed eventual access

Result: Both reader and writer starvation prevented! ✓

9.9 Concurrency Control

Goal: Allow maximum concurrency while ensuring correctness.

Achievement:

- ❖ Multiple readers read simultaneously (high concurrency)
- ❖ Writers get exclusive access when needed (correctness)
- ❖ Priority mechanism balances fairness (no starvation)

Measured: max_concurrent_readers statistic proves concurrent reading works!

10. Testing and Results:

10.1 Test Scenarios

Test Case 1: Basic Functionality

Configuration: 3 Readers, 2 Writers, 15 seconds Expected

Behavior:

Readers read concurrently (up to 3 simultaneously)

Writers write exclusively (one at a time)

shared_data increments correctly Actual Results:

Total Reads: 25Total Writes: 8Max Concurrent Readers: 3Final Data Value: 8Data Consistency: PASS ✓ (8 == 8) Analysis:

- ✓ Multiple readers active simultaneously
- ✓ Only one writer at a time
- ✓ Data consistency maintained
- ✓ No errors or exceptions

Test Case 2: Writer Priority Verification

Configuration: 5 Readers, 1 Writer, 15 seconds Expected

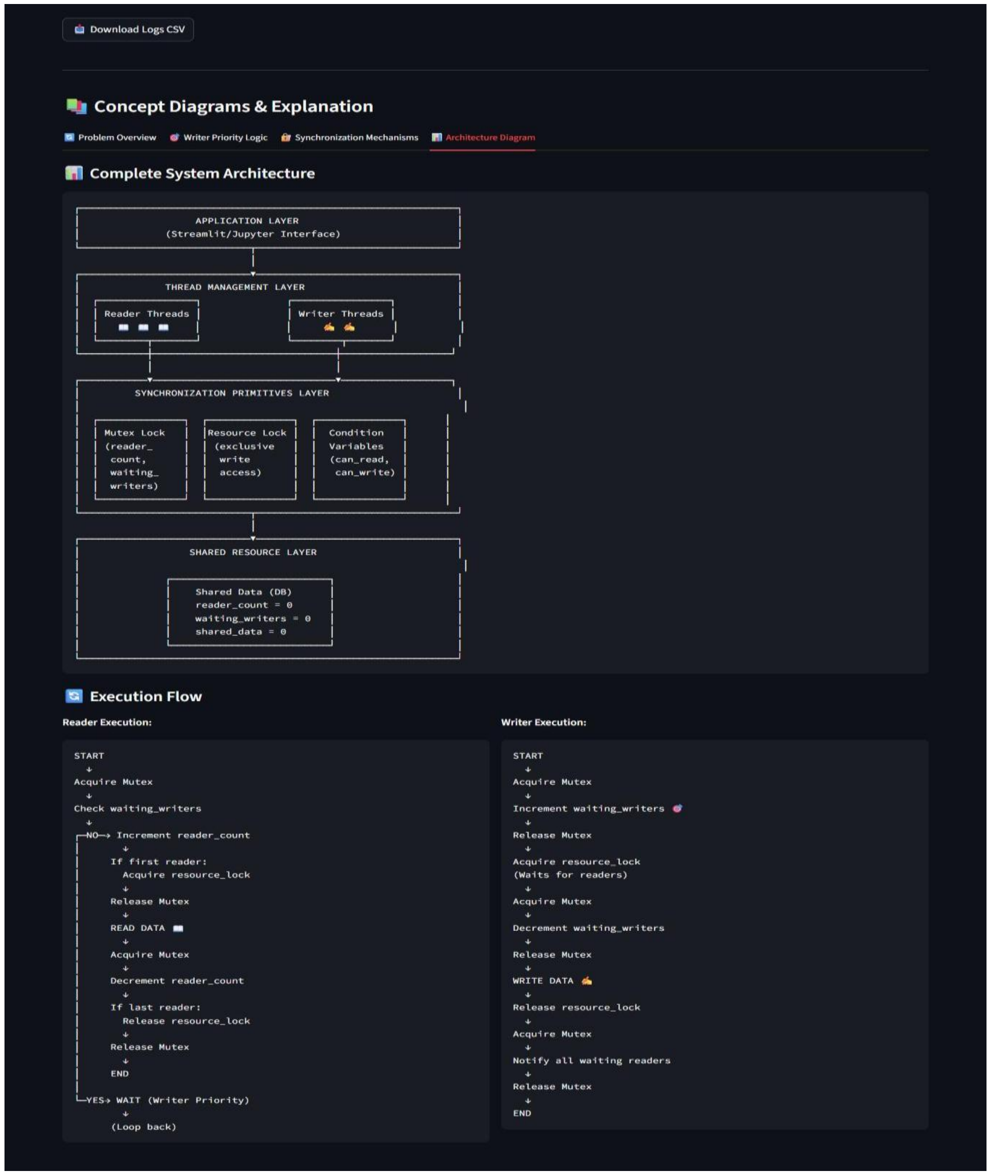
Behavior:

When writer waits, new readers should block

Logs should show "Reader WAITING (Writer priority)"

Current readers finish

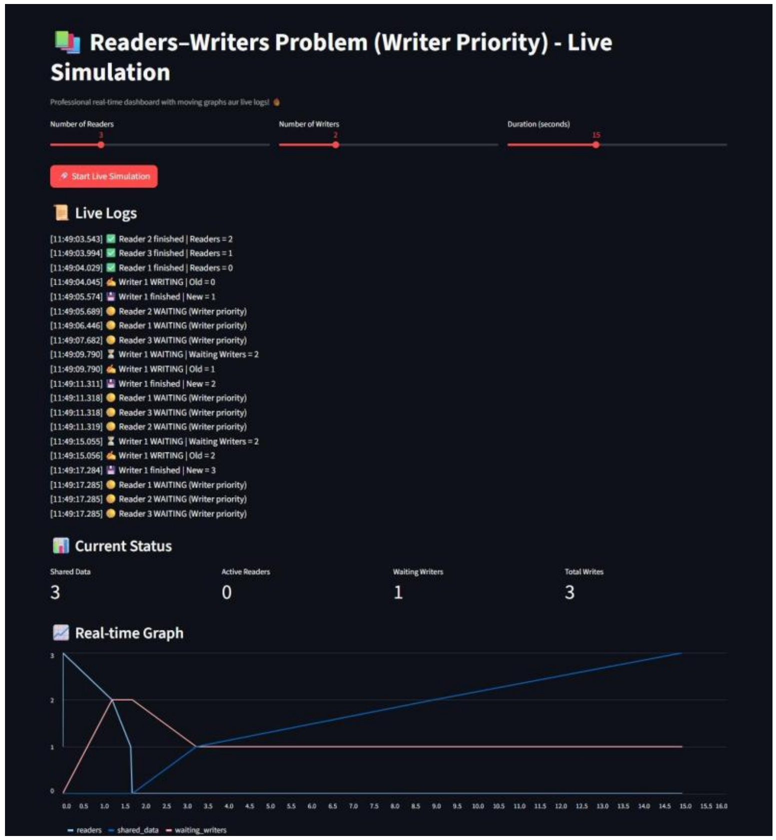
11. ScreenShots:



Readers–Writers Problem (Writer Priority) - Live Simulation
Professional real-time dashboard with moving graphs aur live logs!
Number of Readers
3
Number of Writers
2
Duration (seconds)
15
Start Live Simulation
Live Logs
[11:49:03.543] Reader 2 finished | Readers = 2
[11:49:03.994] Reader 3 finished | Readers = 1
[11:49:04.029] Reader 1 finished | Readers = 0
[11:49:04.045] Writer 1 WRITING | Old = 0
[11:49:05.574] Writer 1 finished | New = 1
[11:49:05.689] Reader 2 WAITING (Writer priority)
[11:49:06.446] Reader 1 WAITING (Writer priority)
[11:49:07.682] Reader 3 WAITING (Writer priority)
[11:49:09.790] Writer 1 WAITING | Waiting Writers = 2
[11:49:09.790] Writer 1 WRITING | Old = 1
[11:49:11.311] Writer 1 finished | New = 2
[11:49:11.318] Reader 1 WAITING (Writer priority)
[11:49:11.318] Reader 3 WAITING (Writer priority)
[11:49:11.319] Reader 2 WAITING (Writer priority)
[11:49:15.055] Writer 1 WAITING | Waiting Writers = 2
[11:49:15.056] Writer 1 WRITING | Old = 2
[11:49:17.284] Writer 1 finished | New = 3
[11:49:17.285] Reader 1 WAITING (Writer priority)
[11:49:17.285] Reader 2 WAITING (Writer priority)
[11:49:17.285] Reader 3 WAITING (Writer priority)
Current Status
Shared Data 3
Active Readers 0
Waiting Writers 1
Total Writes 3
Real-time Graph
readers
shared_data
waiting_writers
Simulation Completed!
Final Results
Total Reads 3
Total Writes 3
Max Concurrent Readers 3
Final Shared Data 3
Download Logs CSV
Concept Diagrams & Explanation
Problem Overview
Writer Priority Logic
Synchronization Mechanisms
Architecture Diagram
Readers-Writers Problem Overview
The Challenge:
Multiple Readers can read simultaneously
Writers need exclusive access
Without priority: Writers may starve
With Writer Priority: Prevents writer starvation
Key Rules:
1. Multiple readers can read at the same time
2. Only ONE writer can write at a time
3. No readers allowed when writer is writing
4. Writers get priority over new readers
Shared Resource (Database/File)
Reader 1 (READ)
Reader 2 (READ)
Writer 1 (WRITE)
Readers can work together
Writer needs exclusive access
Key Concepts Explained
What is Writer Priority?
Why Use Mutex and Locks?
How Does Condition Variable Work?
What Prevents Deadlock?
Operating Systems Project | Readers-Writers Problem with Writer Priority

12. Challenges and Solutions:

12.1 Simulation Running



Key Elements:

- ✓ Live logs showing reader/writer activity
- ✓ Real-time metrics (Shared Data, Active Readers, etc.)
- ✓ Line graph showing concurrent threads over time
- ✓ Status message "Simulation running..."

12.2 Writer Priority in Action

Example Log:

[23:54:44.514] ⌛ Writer 1 WAITING | Waiting Writers = 1[23:54:48.094] Reader 2 WAITING (Writer priority) ← Evidence![23:54:48.577] Reader 1 WAITING (Writer priority) ← Evidence!

12.3 Final Results

Shows:

- ✓ Total Reads
- ✓ Total Writes
- ✓ Max Concurrent Readers
- ✓ Final Shared Data
- ✓ Download button for logs

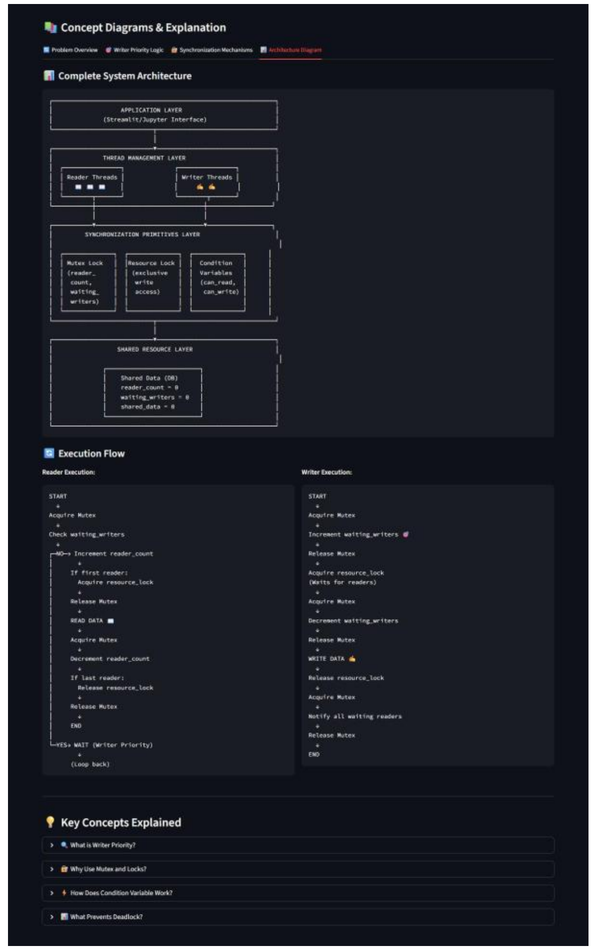
12.4 Concept Diagrams Tabs

Tab 1: Problem Overview

Tab 2: Writer Priority Logic

Tab 3: Synchronization Mechanisms

Tab 4: Architecture Diagram



12.5 Jupyter Notebook Execution

[INSERT SCREENSHOT: Jupyter notebook cells running] Shows:

- Cell outputs with logs
- Statistics display
- Interactive widgets

13. References:

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.
- Tanenbaum, A. S., & Bos, H. (2015). Modern Operating Systems (4th ed.). Pearson.
- Python Threading Documentation. https://docs.python.org/3/library/threading.html
- Courtois, P. J., Heymans, F., & Parnas, D. L. (1971). "Concurrent Control with 'Readers' and 'Writers'." Communications of the ACM, 14(10), 667-668.
- Streamlit Documentation. https://docs.streamlit.io

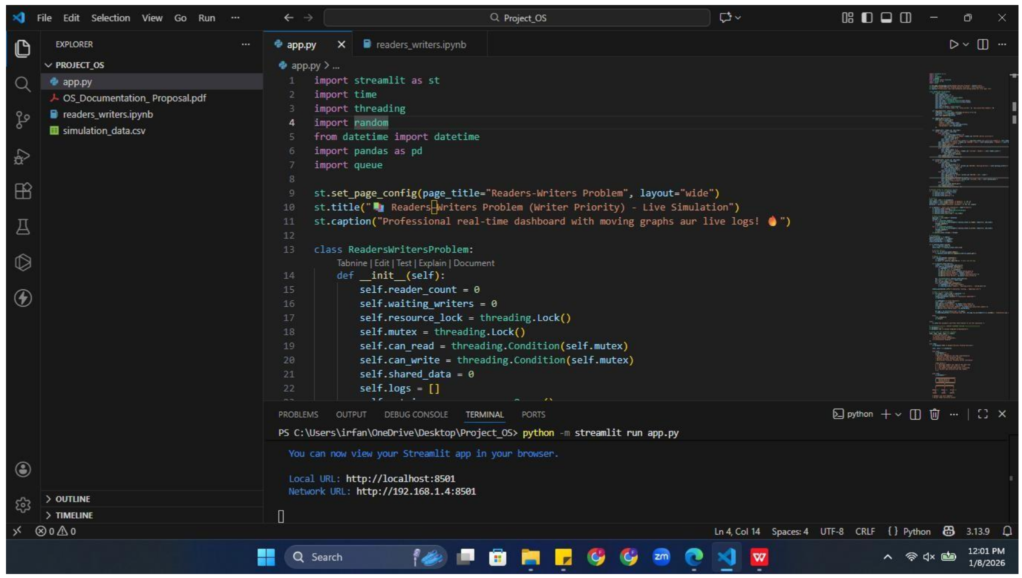
- Jupyter Notebook Documentation. https://jupyternotebook.readthedocs.io

14. Appendcs:

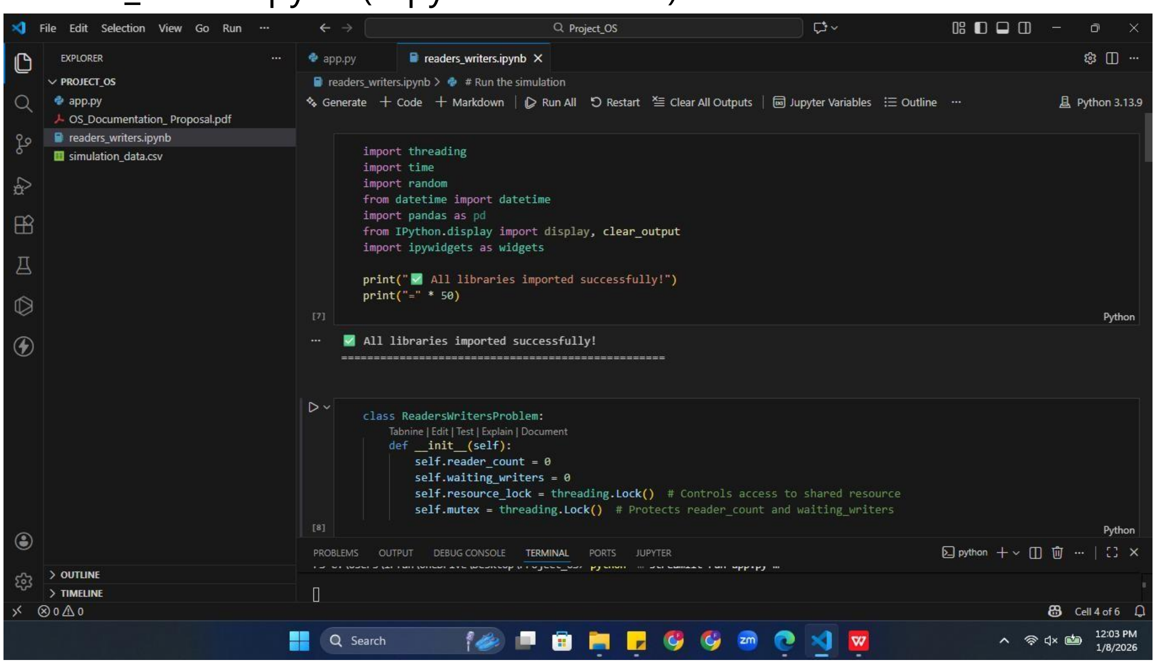
Appendix A: Complete Source Code

File: app.py (Streamlit App)

File:



readers_writers.ipynb (Jupyter Notebook)



Appendix B: Installation Guide

System Requirements:

- ✓ Python 3.8 or higher
- ✓ 4GB RAM minimum
- ✓ Modern web browser (Chrome, Firefox, Edge)

Installation Steps:

1. Install Python packages: pip install streamlit pandas jupyter ipywidgets

2. Run Streamlit appstreamlit run app.py

3. Run Jupyter notebookjupyter notebook readers_writers.ipynb

Appendix C: Troubleshooting

Issue 1: Jupyter command not found

python -m jupyter notebook

Issue 2: Streamlit command not found --- use this command to run project

python -m streamlit run app.py Issue

3: Port already in use streamlit run

app.py --server.port 8502

Appendix D: Project Structure

```
OS_Project/
│
├── app.py                    # Streamlit application
├── readers_writers.ipynb     # Jupyter notebook
├── requirements.txt          # Python dependencies
├── README.md                 # Project overview
├── documentation.pdf         # This document
│
```

* * * * * * * * * * * * * * * * *