# Operating Systems – Practical Exam

© HZ  
PhD Hubert Zarzycki  
All rights reserved.

## 1. Introduction

In multiprogramming environments, multiple processes often need simultaneous access to shared data or system resources. Efficient resource management ensures data consistency, prevents data races, and avoids system inefficiencies such as deadlock or starvation. This section presents the Readers-Writers Problem, a classic example of synchronization in concurrent systems.

## 2. Readers-Writers Problem Definition

The Readers-Writers Problem addresses synchronization between concurrent readers and writers accessing shared data. The challenge lies in permitting multiple readers to read simultaneously while ensuring that writers have exclusive access to write. This coordination is necessary to prevent inconsistencies caused by concurrent modifications.

A correct solution must satisfy the following constraints:

• Readers-Readers compatibility: Multiple readers can read simultaneously.

• Writer exclusion: Only one writer can write at a time, and no readers should read during writing.

• Fairness / Avoid starvation: Neither readers nor writers should suffer indefinite postponement.

Diagram Description: Readers-Writers Problem (General Overview)

• Reader/Writer Nodes: Represent threads that attempt to access a shared resource.

• Request Access: All threads request entry into the critical section.

• Decision Point: Logic checking whether a thread can proceed.

• Critical Sections: Read or write access.

• Blocked Queue: Waiting threads.

• Release Access: When done, thread exits critical section.

• Shared Resource Block: Represents internal synchronization logic.

## 3. Readers-Writers Problem Variants

Three classical variants:

• Reader Preference – readers have priority, writers can starve.

• Writer Preference – writers have priority, readers may wait.

• Fairness – bounded waiting, prevents starvation for both.

Each variant provides different trade-offs in concurrency systems.

## 3.1. Reader Preference

Allows many readers if no writer is active. New readers can continue entering even if a writer is waiting. This can starve writers in read-heavy scenarios.

• Synchronization: mutex + counter.

• Use Case: optimal in read-heavy systems.

## 3.2. Writer Preference

If a writer is waiting, no new readers are allowed to start.

• Synchronization: semaphores, counters or flags for writers.

• Use Case: write-intensive systems.

## 3.3. Fairness

Both readers and writers are served fairly in order of arrival.

• Synchronization: queues, tickets.

• Ensures bounded waiting.

• Requires complex logic.

## 4. Synchronization Primitives

Semaphores and mutexes:

• Semaphore: Integer with wait()/signal().

• Mutex: Binary semaphore.

Goals:

• Mutual exclusion for writers.

• Multiple readers allowed.

• No interleaving of read/write.

## 5. Python Implementation

Below is a simplified version of the base Readers-Writers implementation in Python.

import threading  
import time  
import random  
  
shared\_data = 0  
read\_count = 0  
read\_count\_lock = threading.Lock()  
resource\_lock = threading.Lock()  
  
def reader(reader\_id):  
 global read\_count  
 with read\_count\_lock:  
 read\_count += 1  
 if read\_count == 1:  
 resource\_lock.acquire()  
 print(f"[Reader-{reader\_id}] Reading value: {shared\_data}")  
 time.sleep(random.uniform(0.1, 0.5))  
 with read\_count\_lock:  
 read\_count -= 1  
 if read\_count == 0:  
 resource\_lock.release()  
 print(f"[Reader-{reader\_id}] Done reading.")  
  
def writer(writer\_id):  
 global shared\_data  
 resource\_lock.acquire()  
 shared\_data += 1  
 print(f"[Writer-{writer\_id}] Wrote value: {shared\_data}")  
 time.sleep(random.uniform(0.2, 0.6))  
 resource\_lock.release()  
 print(f"[Writer-{writer\_id}] Done writing.")  
  
def main():  
 reader\_threads = []  
 writer\_threads = []  
 for i in range(5):  
 t = threading.Thread(target=reader, args=(i,))  
 reader\_threads.append(t)  
 t.start()  
 for i in range(2):  
 t = threading.Thread(target=writer, args=(i,))  
 writer\_threads.append(t)  
 t.start()  
 for t in reader\_threads + writer\_threads:  
 t.join()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

## 6. Sample Analyses and Conclusion

6. Sample Analyses and Conclusion

Running the algorithm with different variants and parameters:

Scenario 1: Reader-Preference

Readers consistently gain access before writers, leading to writer starvation in heavy read load.

Scenario 2: Writer-Preference

Writers proceed without delay; readers must wait.

Scenario 3: Balanced queue (Fair solution)

Readers and writers alternate fairly. No starvation observed.

The Readers-Writers Problem is a foundational case in resource synchronization. Practical mastery of its variations ensures readiness for real-world systems programming. Through experimentation, students grasp the consequences of prioritization, synchronization, and thread-safe design—key concepts in robust operating system design.

## 7. Preparation for Task Execution

7. Preparation for Task Execution

1. Log in to the Linux system and open the terminal. If there is no Linux system on the computer, you can use the remote Linux with administrator rights available at:

https://copy.sh/v86/?profile=linux26 or https://bellard.org/jslinux/ (Copy.sh or JS/Linux).

2. In your home directory, create code file named exam\_RW\_solution.py:

touch ~/exam\_RW\_solution.py

3. Save the code in exam\_RW\_solution.py

4. Open a terminal or command prompt.

5. Navigate to the folder containing the scripts using:

cd ~/

6. Run the script using:

python3 exam\_RW\_solution.py

7. Observe the output.

## 8. Practical Exam – Readers-Writers Problem

8. Practical Exam – Readers-Writers Problem

Note: Open a terminal in Linux (Ubuntu). Complete the following tasks, recording observations in the source code. Save code files with exemplary results. The results should be documented as comments within the code. If applicable, add short descriptions of the results obtained after running each task. Use only standard Python libraries (no external packages required). Ensure compatibility with Python 3 and standard Linux environments (Ubuntu).

Course: Operating Systems

Environment: Ubuntu Linux (Python, no admin privileges)

Duration: 1 hour

Language: Python 3

Instructions:

• All answers must be saved in a single Python source file (exam\_RW\_solution.py)

• At the beginning of your file, include a comment in the following format:

# Month of birth: 7

# First letter of first name: K

# Index last digit: 3

# Day of birth: 12

Each student must complete four tasks (1, 2, 3, 4) assigned according to:

• Task 1: Based on month of birth (reader/writer variant)

• Task 2: Based on first letter of first name (logging/debugging)

• Task 3: Based on last digit of index number (feature extension)

• Task 4: Based on day of birth (even/odd) (performance/testing)

Use Readers-Writers Base Template as version to be modified.

Create only your assigned subtasks with software variants. Creating more variants or other unassigned subtasks will result in a negative grade on the practical exam.

## Task Assignments Overview

Task 1 – Variant Implementation (Assigned by Month of Birth):

1–4 -> 1a. Reader-Preference Algorithm

5–8 -> 1b. Writer-Preference Algorithm

9–12 -> 1c. Fair Algorithm with Queue

Each version must:

• Use threading and synchronization (e.g., threading.Lock(), threading.Semaphore())

• Allow simulation with multiple readers/writers

• Clearly identify entry/exit of critical section via console print statements

Task 2 – Logging, Timestamps, and Debugging Enhancements (Assigned by First Letter of First Name):

A–G -> 2a. Add timestamped logs

H–O -> 2b. Track and display waiting times

P–Z -> 2c. Log to external file

Task 3 – Functionality Extension (Assigned by Last Digit of Index Number):

0–1 -> 3a. Add priority boost

2–3 -> 3b. Timeout mechanism

4–5 -> 3c. Role switch

6–7 -> 3d. Limit concurrent readers

8–9 -> 3e. Resource lock delay

Task 4 – Testing, Performance and Evaluation (Assigned by Day of Birth Even/Odd):

Even -> 4a. Compare performance

Odd -> 4b. Stress test

## Submission Guidelines

Expected Submission Structure (Sample File Layout):

# Month of birth: 5

# First letter of first name: M

# Index last digit: 7

# Day of birth: 21

# Task 1b: Writer-Preference Implementation

# [code here...]

# Task 2a: Add timestamped logs

# [code here...]

# Task 3d: Limit concurrent readers

# [code here...]

# Task 4b: Stress testing and deadlock detection

# [code, log samples, observations...]

Tools and Libraries:

• threading, time, random, logging (standard Python 3)

• Linux CLI tools: time, htop, watch, ps, etc.

• Suggested log file name: rw\_log\_<your\_index>.log

Final Notes:

• Solutions must be original and not copied.

• If a feature requires additional code structure (e.g., a priority queue), justify it in a comment.

• Include clear section comments for each task.

• Output should be readable and allow for verification.

After completing all tasks:

• Submit .py files (e.g., exam\_RW\_solution.py) with clearly commented experiments.

• Include a short paragraph summarizing your observations and conclusions from each task.

• Be prepared to present results live to the instructor during the laboratory session.

## Literature

Literature:

- A. Silbershatz, J.L. Peterson, P.B. Galvin – Operating System Concepts

- A.S. Tannenbaum – Modern Operating Systems

- https://tldp.org/ – The Linux Documentation Project

- https://linuxjourney.com/

- https://linuxcommand.org/

- https://ubuntu.com/tutorials/command-line-for-beginners#1-overview

- https://linuxsurvival.com/linux-tutorial-introduction/

- William Shotts – The Linux Command Line: A Complete Introduction

- Ward B. – How Linux Works. What Every Superuser Should Know

- Richard Blum, Christine Bresnahan – Linux Command Line and Shell Scripting Bible

- Daniel J. Barrett – Linux Pocket Guide

- Cameron Newham – Learning the Bash Shell

- Christopher Negus – Linux Bible