📚 The Reader–Writer Problem

> “If a writer edits the book while readers are still reading it…
> chaos follows — even faster than on Reddit.”

💡 Scenario

- Shared **data (a file, DB, memory segment)** accessed by multiple threads.
- **Readers** can access concurrently (no changes).
- **Writers** need exclusive access.

Goals:

✅ Allow concurrent reading.

✅ Only one writer at a time.

✅ Prevent starvation.

Base Setup

```
import threading, time, random

data = 0
readcount = 0
lock = threading.Lock()
rw_mutex = threading.Lock()
```

⚠️ Attempt #1 – The Naïve Version

```
def reader():
    global data
    while True:
        print(f"Reader sees {data}")
        time.sleep(random.uniform(0.3, 0.8))

def writer():
    global data
    while True:
        data += 1
        print(f"Writer updates data to {data}")
        time.sleep(random.uniform(0.5, 1.0))
```

😱 Race condition city:

- Readers may read while writer is changing `data`.
- Multiple writers may update simultaneously.

🔒 Attempt #2 – One Lock to Rule Them All

```
def reader():
    while True:
        with lock:
            print(f"Reader sees {data}")
        time.sleep(random.uniform(0.3, 0.8))

def writer():
    global data
    while True:
        with lock:
            data += 1
            print(f"Writer updates data to {data}")
        time.sleep(random.uniform(0.5, 1.0))
```

✅ Safe

❌ No concurrency at all — readers block each other unnecessarily.

> "You've solved chaos by banning everyone from the library."

📖 Attempt #3 – Reader-Priority Solution

Allow multiple readers but lock out writers when any reader is active.

```
readcount = 0
mutex = threading.Lock()      # protects readcount
rw_mutex = threading.Lock()   # writer exclusion

def reader(id):
    global readcount
    while True:
        with mutex:
            readcount += 1
            if readcount == 1:
                rw_mutex.acquire()  # first reader blocks writers
        print(f"👓 Reader {id} reads {data}")
        time.sleep(random.uniform(0.3, 0.8))
        with mutex:
            readcount -= 1
            if readcount == 0:
                rw_mutex.release()  # last reader unblocks writers
        time.sleep(random.uniform(0.2, 0.6))

def writer(id):
    global data
    while True:
        rw_mutex.acquire()
        data += 1
        print(f"✏️ Writer {id} updates data to {data}")
```

✍️ Attempt #4 – Writer-Priority Solution

Writers wait for readers, but new readers must wait if a writer is waiting.

```
readcount = 0
writecount = 0

mutex = threading.Lock()
rw_mutex = threading.Lock()
readTry = threading.Lock()
resource = threading.Lock()

def reader(id):
    global readcount
    while True:
        readTry.acquire()
        mutex.acquire()
        readcount += 1
        if readcount == 1:
            resource.acquire()
        mutex.release()
        readTry.release()

        print(f"👓 Reader {id} reads {data}")
        time.sleep(random.uniform(0.3, 0.8))

        mutex.acquire()
        readcount -= 1
        if readcount == 0:
            resource.release()
        mutex.release()
        time.sleep(random.uniform(0.2, 0.6))

def writer(id):
    global data, writecount
    while True:
        rw_mutex.acquire()
        writecount += 1
        if writecount == 1:
            readTry.acquire()
        rw_mutex.release()

        resource.acquire()
        data += 1
        print(f"✏️ Writer {id} updates data to {data}")
        resource.release()
        rw_mutex.acquire()
        writecount -= 1
```

🧠 Summary

Approach	Concurrency	Fairness	Notes
Naïve	❌	❌	Chaos
One Lock	❌	✅	Overly safe
Reader-Priority	✅	❌	Writers may starve
Writer-Priority	✅	✅	Balanced and fair

Main Function

```
def main():
    threads = []
    for i in range(3):
        t = threading.Thread(target=reader, args=(i,), daemon=True)
        threads.append(t)
        t.start()
    for i in range(2):
        t = threading.Thread(target=writer, args=(i,), daemon=True)
        threads.append(t)
        t.start()
    time.sleep(10)
    print("Simulation complete.")
```

🧩 OS Concepts

OS Idea	Analogy
Reader	Process doing `read()`
Writer	Process doing `write()`
Critical Section	Shared memory region
Semaphore	Access permit
Starvation	A writer never gets CPU time

🏁 Takeaway

> “The Reader–Writer problem teaches one hard truth:
> **Sometimes fairness means making everyone wait a little.”**