# BAHCESEHIR UNIVERSITY FACULTY OF ENGINEERING, COMPUTER ENGINEERING

January 06, 2024

**Operating Systems Course Project:**

**The Reader Writer Problem**

**by**

Ali Kaan Karagözgil 2102997

**Abstract:**

In computer science, the Readers-Writers problem is a classic scenario demonstrating the necessity for efficient concurrent access management to a shared resource. This problem involves enabling multiple readers to simultaneously read a resource while ensuring writers have exclusive access when making modifications. The challenge is to create a synchronization mechanism that optimizes efficiency for readers but maintains data integrity during write operations. Our solution uses semaphores, a type of synchronization tool that controls access through a count of available permits, to manage the access to the resource. Semaphores ensure that while many readers can access the data at the same time, any writer obtains exclusive access, effectively preventing readers from accessing the resource during modifications. This project focuses on implementing such a semaphore-based Read-Write-Lock mechanism, ensuring exclusive access for writers, concurrent reading for multiple readers, and overall system stability and performance.

**BAHCESEHIR UNIVERSITY – 2023**

**Analysis of My Solution**

**Class Overview: ReadWriteLock**

The ReadWriteLock class is designed to manage access to a shared resource by multiple reader threads and writer threads. It uses a semaphore S to ensure that while multiple readers can access the shared resource concurrently, writers have exclusive access.

**Analysis of Methods:**

1. **readLock()**

**Functionality:**

* The readLock() method is called when a reader thread wants to read from the shared resource.
* It checks if reader\_count is zero, indicating it's the first reader. If so, it acquires the semaphore S, blocking writers from accessing the shared resource.
* It increments reader\_count, indicating a new reader has started reading.

**Rationale:**

* The first reader needs to block writers to ensure no writes occur during reading. Subsequent readers don't need to acquire S again since concurrent reads are allowed.
* It's synchronized to ensure that checking and modifying reader\_count and potentially acquiring S happen atomically to avoid race conditions.

1. **writeLock()**

**Functionality:**

* The writeLock() method is called by a writer thread that intends to write to the shared resource.
* It acquires the semaphore S, ensuring exclusive access to the shared resource as both readers and writers are blocked until it releases S.

**Rationale:**

* Writers need exclusive access to prevent simultaneous reads or writes, which could lead to data corruption or inconsistency.

1. **readUnLock()**

**Functionality:**

* The readUnLock() method is called when a reader thread finishes reading the shared resource.
* It decrements reader\_count. If reader\_count is zero (indicating it was the last reader), it releases the semaphore S, allowing writers to proceed.

**Rationale:**

* The last reader out should release the semaphore S to allow waiting writers to proceed. This ensures that once all readers are done, writers get a chance to write.
* It's synchronized to ensure that checking and modifying reader\_count and potentially releasing S happen atomically to maintain consistency.

1. **writeUnLock()**

**Functionality:**

* The writeUnLock() method is called by a writer thread after it finishes writing to the shared resource.
* It releases the semaphore S, allowing other readers or writers to acquire it and proceed.

**Rationale:**

* Once a writer is done writing, it must release S to open up the resource for other waiting readers or writers. This ensures fairness and prevents starvation.

**Flow Chart of the Solution**

A diagram of a software flowchart

Description automatically generated

**Conclusion**

Each method in the ReadWriteLock class plays a crucial role in ensuring the correct synchronization behavior as per the Readers-Writers problem. The use of a semaphore (S) effectively manages the mutual exclusion and concurrency requirements. The reader\_count variable tracks the number of active readers, ensuring that the semaphore is released correctly and efficiently. This implementation allows for multiple readers to read concurrently while ensuring writers have exclusive access to the shared resource, thus maintaining data integrity and preventing conflicts. This analysis should provide a clear understanding of the purpose and functionality of each method, contributing to the overall effectiveness of the Readers-Writers problem solution.