**CMP3001 Operating Systems**

**Project Report Document**

**Enes Burak Çetin – 2104091**

**1. Introduction And Problem Description**

In this project we were supposed to provide a solution to readers-writers problem in which several processes (readers and writers) are trying to access shared variables.

Two operations on the semaphore were allowed; **acquire()** and **release()** (they correspond wait and signal functions)

The problem involves two types of threads: readers and writers. Readers can access the resource simultaneously, but writers require exclusive access. If a writer is writing, no other reader or writer should be allowed to access the resource. The challenge is to ensure that readers do not block each other, while writers are given priority over readers when necessary.

**2. Solution Explanation**

This solution uses three semaphores to manage access to the resource:  
**-** **Semaphore 'S':** This protects the readersCount and writersCount variables to ensure that they are updated safely.  
**- Semaphore 'readerBlock':** This blocks new readers if a writer is currently writing.  
**- Semaphore 'writerBlock':** This ensures that writers proceed one by one into the critical section and blocks other writers from entering while one is writing.

The solution defines the following functions to manage locking and unlocking of the resource:  
**- readLock():** This function allows a reader to acquire access to the shared resource. It ensures that the first reader blocks writers and other readers can enter concurrently.  
**- writeLock():** This function allows a writer to acquire exclusive access to the resource. It ensures that no other writer or reader can enter the critical section while the writer is writing.  
**- readUnLock():** This function releases the reader's lock and, if it is the last reader, allows writers to proceed.  
**- writeUnLock():** This function releases the writer's lock and allows new readers or writers to enter.

All functions are explained in detail below.

**readLock() function:**

1. public void readLock() {

2. try {

3.          readerBlock.acquire(); // Prevents readers from entering if a writer is waiting or writing.

4.         S.acquire(); // Enter critical section for counter manipulation.

5.         readersCount++; // Increment the number of active readers.

6.          if (readersCount == 1) { // If this is the first reader.

7.              writerBlock.acquire(); // First reader blocks writers from entering.

8.         }

9.      } catch (InterruptedException e) {

10.         e.printStackTrace();

11.     } finally {

12.         S.release(); // Exit critical section.

13.         readerBlock.release(); // Allow other readers to acquire the read lock.

14.     }

15.   }

**writeLock() function:**

1. public void writeLock() {

2.         try {

3.             System.out.println(Thread.currentThread().getName() + " is waiting to write.");

4.             S.acquire(); // Acquires the mutex to safely update writersCount.

5.             if (writersCount == 0) { // If this is the first writer waiting.

6.                 readerBlock.acquire(); // First writer blocks new readers from entering.

7.             }

8.             writersCount++; // Increment the number of writers waiting or writing.

9.             S.release(); // Release the mutex.

10.             writerBlock.acquire(); // Acquire the lock to enter the critical section for writing.

11.             System.out.println(Thread.currentThread().getName() + " has entered the critical section to write.");

12.         } catch (InterruptedException e) {

13.             e.printStackTrace();

14.         }

15.     }

**readUnLock() function:**

1.    public void readUnLock() {

2.         try {

3.             S.acquire(); // Enter critical section for counter manipulation.

4.             readersCount--; // Decrement the number of active readers.

5.             if (readersCount == 0) { // If this was the last reader.

6.                 writerBlock.release(); // Last reader unblocks writers.(Allow writers to enter)

7.             }

8.         } catch (InterruptedException e) {

9.             e.printStackTrace();

10.         } finally {

11.             S.release(); // Exit critical section.

12.         }

**writeUnLock() function:**

1.    public void writeUnLock() {

2.         try {

3.             System.out.println(Thread.currentThread().getName() + " has exited the critical section after writing.");

4.             S.acquire(); // Enter critical section.

5.             writersCount--; // Decrement number of writers waiting or writing.

6.             if (writersCount == 0) { // If no more writers are waiting or writing.

7.                 readerBlock.release(); // Last writer unblocks readers.(Allow readers to enter again.)

8.             }

9.             S.release(); // Exit critical section.

10.

11.             writerBlock.release();

12.         } catch (InterruptedException e) {

13.             e.printStackTrace();

14.         }

15.     }

**3. Implementation Details**

The implementation utilizes Java's concurrency model to manage multiple reader and writer threads. Each thread either acquires a read lock or a write lock depending on its type, performs its task (reading or writing), and then releases the appropriate lock when finished. The program randomly generates a number of readers and writers and simulates their execution. To make it more realistic, sleep functions are called for simulating reading and writing processes.

**4. Sample Test**

In the sample test, a random number of readers (between 2 and 5) and writers (between 1 and 3) are created. Each reader thread acquires the read lock, simulates reading for a short period, and then releases the read lock. Each writer thread acquires the write lock, simulates writing for a longer period, and then releases the write lock.

**5. Conclusion**

This solution demonstrates an effective way to manage access to a shared resource using semaphores, ensuring that multiple readers can access the resource concurrently, while writers are given exclusive access when needed. The solution ensures that no writer is starved, though additional mechanisms could be introduced to guarantee fairness. The use of semaphores provides an efficient way to synchronize multiple threads while avoiding issues like deadlocks or race conditions.

**Some outputs:**

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldumetin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu