CMP3001 - Operating System Project

The Readers-Writers Problem

The readers-writers problem arises when different users try to reach a shared file to either read and write it. To solve this problem, we need to ensure at a time only one writer modifies the file and none other writers can modify and readers can read, on the other hand when a reader is reading, a writer can not change the file but other readers can also read.

All the execution happens on the Main.java file in different classes. *ReadWriteLock* class hass the relevant functions for the read and write operations. *Main* class simulates the readers and writers using functions from *ReadWriteLock* class.

The overview of the ReadWriteLock class:

class ReadWriteLock {

private Semaphore r\_mut = new Semaphore(1);

private Semaphore S = new Semaphore(1);

private int read\_num = 0;

A binary semaphore (mutex) called *r\_mut* is created to keep consistency between readers.

A shared lock, semaphore *S* is created to keep consistency between readers and writers.

An integer variable *read\_num* is created and initially set to 0 to keep the total number of readers.

The implementation of the readLock function:

public void readLock() {

try {

System.out.println("\033[0;31m" + "# A reader is trying to read." + "\033[0m");

r\_mut.acquire();

} catch (Exception e) {}

read\_num++;

if (read\_num == 1) {

try {

S.acquire();

} catch (Exception e) {}

}

r\_mut.release();

}

When called, the function will try to access the file to read.

When it is possible, *r\_mut* is obtained by only one reader at a time instance to keep the consistency of *read\_num*. So when more than one reader tries to access the file at the same time, *read\_num* will be incremented by 1.

When the first reader accesses the file to read *S* is obtained by it so writers cannot access the file unless *S* is released.

Then, *r\_mut* is released so other readers can access the file without waiting.

The implementation of the readUnlock function:

public void readUnlock() {

read\_num--;

if (read\_num == 0) {

S.release();

}

}

This function is implemented to release the shared lock when all the read operations are completed.

When the read operation is finished, *read\_num* is decremented by 1. If no users are reading at that time instance *S* is released. So a writer can start execution or a new reader can read the file.

The implementation of the writeLock function:

public void writeLock() {

try {

System.out.println("\033[0;31m" + "# A writer is trying to write." + "\033[0m");

S.acquire();

} catch (Exception e) {}

}

When it is called, the function will try to access the file to write.

The function tries to acquire *S* lock. If it is accessed by the readers, *S* is set to 0 so a writer can not modify the file. When *S* is 1, a writer accesses the file.

The implementation of the writeUnlock function:

public void writeUnlock() {

S.release();

}

This function is implemented to release the shared lock when all the write operations are completed.

The writer who has access to the file releases the *S* lock. So another writer can start execution or the readers can read the file.

The overview of the Main class:

import java.util.concurrent.Semaphore;

public class Main {

public static void main(String[] args) {

ReadWriteLock RWLock = new ReadWriteLock();

Thread reader1 = new Thread(() -> {

RWLock.readLock();

System.out.println("Reader 1 is reading.");

try{

Thread.sleep(1000);

}

catch(InterruptedException e){}

System.out.println("Reader 1 finished reading.");

RWLock.readUnlock();

});

### Other reader implementations...

Thread writer1 = new Thread(() -> {

RWLock.writeLock();

System.out.println("Writer 1 is writing.");

try{

Thread.sleep(2000);

}

catch(InterruptedException e){}

System.out.println("Writer 1 finished writing.");

RWLock.writeUnlock();

});

### Other writer implementations...

*java.util.concurrent.Semaphore* is imported for the Semaphore implementations in the ReadWriteLock class.

A ReadWriteLock object called *RWLock* is created to coordinate access to a shared resource between multiple threads, ensuring synchronization between **readers** and **writers**.

A thread is created for *reader1*.

When the thread is created readLock function is called. To simulate the read operation the thread waits for a random time. After finishing the reading operation readUnlock is called.

A thread is created for *writer1*.

When the thread is created writeLock function is called. To simulate the write operation the thread waits for a random time. After finishing the writing operation readUnlock is called.

### Previous writer and reader implementations...

writer1.start();

reader1.start();

writer2.start();

reader2.start();

reader3.start();

reader4.start();

writer3.start();

}

}

After the thread implementations are done, all the threads are started by their *.start()* method.

An example output:

# A writer is trying to write.

# A reader is trying to read.

# A writer is trying to write.

Writer 2 is writing.

# A reader is trying to read.

# A reader is trying to read.

# A reader is trying to read.

# A writer is trying to write.

Writer 2 finished writing.

Reader 2 is reading.

Reader 3 is reading.

Reader 4 is reading.

Reader 1 is reading.

Reader 3 finished reading.

Reader 4 finished reading.

Reader 2 finished reading.

Reader 1 finished reading.

Writer 3 is writing.

Writer 3 finished writing.

Writer 1 is writing.

Writer 1 finished writing.

Burak Demirci

2201292