Readers Writers Problem

- The readers-writers problem is a classical synchronization problem where an object or a shared resource can be modified or read simultaneously by concurrent threads.
- Some of these threads are readers they only want to read the data from the object and some of the threads are writers they want to write into the object.

In the Readers Writers Problem:

- Many readers are allowed to access the resource at the same time.
- When a reader is accessing the resource, no writers can modify it at the same time.
- Only one Writer is allowed to access the resource at any moment in time.
- When a writer is modifying the resource, no-one else (reader or writer) can access it at the same time.

In first solution we initialized:

- readCount initialized to 0 // number of Readers allowed to read simultaneously.
- Semaphore readerLock // control access to readers counter (initialized to 1)
- Semaphore writeLock // control write access (initialized to 1)

Reader() **method**:
In the code, readerLock and writeLock are semaphores that are initialized to 1. Also, ReaderCount is a variable that is initialized to 0. The readerLock semaphore ensures mutual exclusion and writeLock handles the writing.
If a reader wants to read the object, acquire operation is performed on ReaderLock and readerCount is incremented. And if it was the first reader acquire operation is performed on writeLock to block any writer from accessing the object while reading and then release the ReaderLock. When a reader finishes, he acquire the readerLock again and decreament the count. And if it was the last reader release operation is performed on writeLock to let a writer and then release readerLock.

```
Reader(){

Repeat
ReaderLock.acquire;
ReaderCount ++;
If(ReaderCount == 1) writeLock.acquire;
ReaderLock.realease;

//read resource here

ReaderLock.acquire;
ReaderCount--;
If(ReaderCount == 0) writeLock.release;
ReaderLock.realease;

unitll done
}
```

Writer() method:

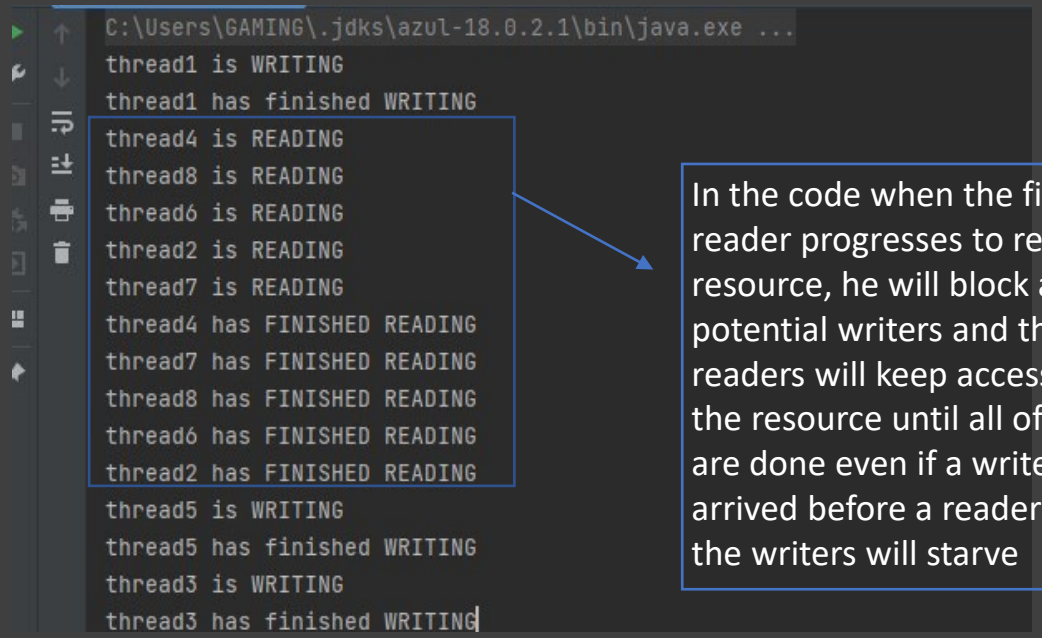
If a writer wants to access the resource, acquire operation is performed on writeLock to make sure that no other writers can modify the object at the same time.

When a writer is done writing into the object, release operation is performed on writeLock to let other writer to modify the resource.

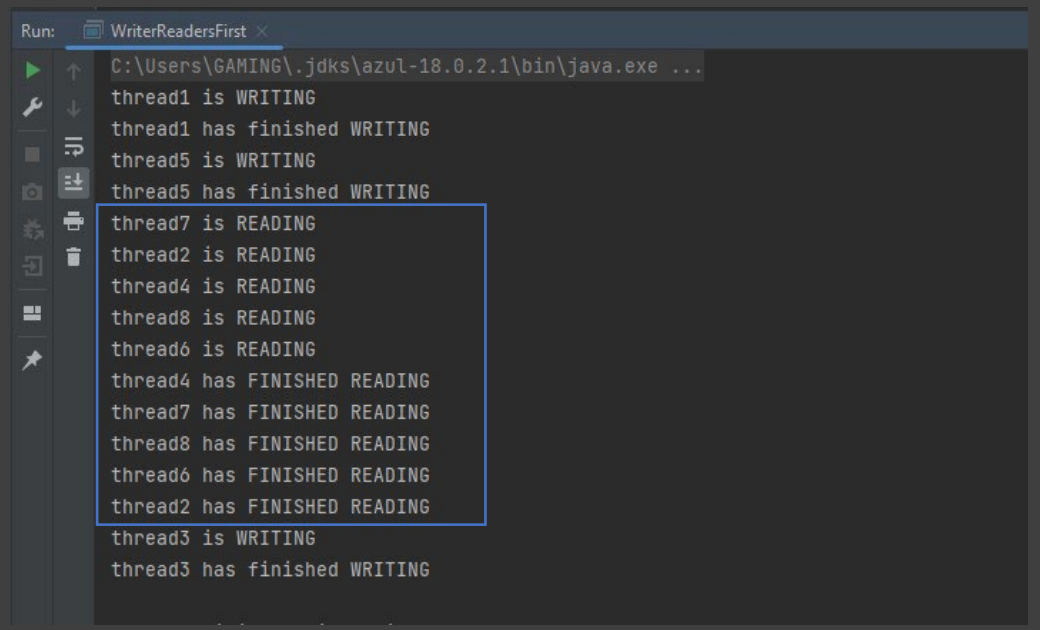
```
Writer(){

Repeat

        WriteLock.acquire;  //mutal execution

    // write resource here

        writeLock.release;

untill done


}
```

Starvation example :

- But this solution has a downside that is the starvation of the Writers: a Writer thread does not have a chance to execute while any number of Readers continuously entering and leaving the working area. This solution obviously favors readers and in there is no guarantee that a writer process does not starve.



In the code when the first reader progresses to read the resource, he will block any potential writers and the readers will keep accessing the resource until all of them are done even if a writer arrived before a reader so the writers will starve



Reader (){:

```
acquire(readLock) ;
acquire(mutex2) ;
readCount++;
if(readcount == 1) acquire(writeLock) ;
release(mutex2) ;
release(readLock) ;

//Read Resource Here

acquire(mutex2) ;
readCount--;
if(readCount == 0) release(writeLock) ;
release(mutex2)
}
```

Reader() **method**:

In the code, readLock and writeLock are semaphores that are initialized to 1. Also, readCount is a variable that is initialized to 0. The mutex2 semaphore ensures mutual exclusion and writeLock handles the writing.

If a reader wants to read the object, acquire operation is performed on readLock and is performed also on mutex2 and readCount is incremented. And if it was the first reader acquire operation is performed on writeLock to block any writer from accessing the object while reading and then release the mutex2 lock and ReadLock. When a reader finishes, he acquire the mutex2 again and decreament the count. And if it was the last reader release operation is performed on writeLock to let a writer and then release mutex2.

In second solution we initialized :

-readCount initialized to 0 // number of Readers allowed to read simultaneously.
-writeCount initialized to 0 // number of Writers allowed to write simultaneously.
-Semaphore *mutex2* // Control access to readers counter (initialized to 1)
-Semaphore mutex // Control access to writers counter (initialized to 1)
-Semaphore readLock // Control read access (initialized to 1)
-Semaphore writeLock // Control write access (initialized to 1)

```
writer() {
 acquire(mutex) ;
 writeCount++;
if(writeCount == 1) acquire(readLock) ;
 release (mutex) ;

acquire(writeLock) ;
 //  write resource here
 release(writeLock) ;

acquire(mutex)
writeCount--;
If(writeCount == 0) realease (readLock) ;
release(mutex) ;
}
```

Writer() method:
If a writer wants to write the object, acquire operation is performed on mutex and writeCount is incremented. And if it was the first writer acquire operation is performed on readLock to block any reader from accessing the object while writing and any coming readers will wait untill the last writer finishes so they can read and then release the mutex. When a writer finishes, he acquire the mutex again and decreament the count. And if it was the last writer release operation is performed on readLock to let readers enter and then release the mutex.

Starvation example :

- this solution has a downside also that is the starvation of the Writers: a Reader thread does not have a chance to execute while any number of Writers continuously entering and leaving the working area. This solution obviously favors Writers and in there is no guarantee that a reader process does not starve.

```
C:\Users\GAMING\.jdks\azul-18.0.2.1\bin\java.exe ...
thread12 is READING
thread12 has FINISHED READING
thread3 is WRITING
thread3 has finished WRITING
thread10 is WRITING
thread10 has finished WRITING
thread9 is WRITING
thread9 has finished WRITING
thread2 is WRITING
thread2 has finished WRITING
thread5 is WRITING
thread5 has finished WRITING
thread11 is WRITING
thread11 has finished WRITING
thread8 is READING
thread6 is READING
thread7 is READING
thread1 is READING
thread4 is READING
thread6 has FINISHED READING
thread8 has FINISHED READING
thread4 has FINISHED READING
thread1 has FINISHED READING
thread7 has FINISHED READING
```

In the code when the first writer progresses to write the resource, he will block any potential readers and the writers will keep accessing the resource until all of them are done even if a reader arrived before a writer so the readers will starve

```
C:\Users\GAMING\.jdks\azul-18.0.2.1\bin\java.exe ...
thread3 is WRITING
thread3 has finished WRITING
thread11 is WRITING
thread11 has finished WRITING
thread2 is WRITING
thread2 has finished WRITING
thread9 is WRITING
thread9 has finished WRITING
thread10 is WRITING
thread10 has finished WRITING
thread5 is WRITING
thread5 has finished WRITING
thread12 is READING
thread8 is READING
thread4 is READING
thread6 is READING
thread7 is READING
thread1 is READING
thread6 has FINISHED READING
thread1 has FINISHED READING
thread8 has FINISHED READING
thread4 has FINISHED READING
thread7 has FINISHED READING
thread12 has FINISHED READING
```

Dead lock example : (Base use of semaphores)

If we did lock the resource when a writer progresses to Write the resource and after finishing the writing we didn't release that lock, a DeadLock will happen cause readers will keep waiting till the writer releases the lock (which will not happen) and also the writer will get stuck after he finishs writing cause he can't do anything else without releasing the lock

```
2 usages
static class Write implements Runnable {
    @Override
    public void run() {
        try {
            writeLock.acquire();
            System.out.println(Thread.currentThread().getName() + " is WRITING");
            Thread.sleep( millis: 2500);
            System.out.println(Thread.currentThread().getName() + " has finished WRITING");
            //writeLock.release();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
C:\Users\GAMING\.jdks\azul-18.0.2.1\
thread1 is WRITING
thread1 has finished WRITING
```

Writer got stuck after he finishes and readers will keep waiting

Real Life application:

Bank account is a real-life application when readers writers problem can occur.
Readers are not changing account data just reading data from it, ex: Checking your account balance
Writers are modifying account data and doing updates; ex: Despoit from your account or Doing withdrawal operation
So, we can't check our balance while doing despoit or withdrawal
Also we can't take withdrawl money from account while updating to it.

I created Account Class with three methods to implement this application on readers witers problem:

-getBalance
-despoit
-withdrawal