Student: Irakli Kelbakiani

Université de Neuchâtel

Concurrency: Multi-core Programming and Data Processing

Homework 2

Student: Irakli Kelbakiani

Email: Irakli.Kelbakiani@students.unibe.ch

Student: Irakli Kelbakiani

Assignment 2:

March 31,2022

Github repo: https://github.com/KelbakianiIrakli/Multi-core-Programming/tree/concurrency-hw2/src/hw2

Assignment 2

Exercise 2.1

Implement Peterson's generalized algorithm (the "Filter" algorithm presented in the course) and use it in the "Counter" program seen in the labs to protect the shared integer.
As a reminder, the program should take as arguments two integers, T and N, and forks T threads that modify a shared volatile integer as follows. Even threads (i.e., threads 0, 2, 4…) will increment the integer N times while odd threads (i.e., threads 1, 3, 5…) will decrement it N times. The program will print the final value of the integer (which should be 0 if T is even, or N otherwise).
Execute the program with N=10'000'000 and T={2,4,8,16}. Report execution times.
HINT: for the level and victim arrays, you are advised to use the AtomicIntegerArray *type (declaring a simple integer array as* volatile *will not make the array elements volatile, but only the reference).*
Please, see the file Exercise2_1.java
The results:

T=2	*T=4*	*T=8*	*T=16*
10718 ms	26975 ms	78645 ms	*525210*

Exercise 2.2

1. Read-write lock

A read-write lock allows either a single writer or multiple readers to execute in a critical section. Provide an implementation of a read-write lock in Java. You can use synchronized methods and the wait/notify mechanism if you wish. The class should provide the 4 methods lockRead(), unlockRead(), lockWrite(), and unlockWrite(). This implementation does not need to be FIFO, starvation-free, nor reentrant.
HINT: you might want to keep track of the number of readers and writers.

Please, see the file Exercise2_2_1.java

2. Starvation-free read-write lock

Try to make the read-write lock starvation free for writes (a writer cannot be blocked

forever by readers continuously requesting and acquiring the lock).
Please, see the file Exercise2_2_2.java

3. FIFO and reentrant read-write lock (optional)

Try to make the read-write lock FIFO and reentrant.

Please, see the file Exercise2_2_3.java

Exercise 2.3

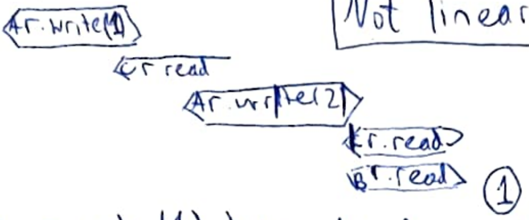
Are the following histories linearizable or sequentially consistent? Explain your answers and write the equivalent linearizable/sequential consistent histories where applicable.

1. Read/write register

a. Concurrent threads A, B, C, register r.

```
A: r.write(1)
C: r.read()
A: r:void
A: r.write(2)
C: r:2
C: r.read()
B: r.read()
A: r:void
C: r:1
A: r.write(1)
B: r:1
A: r:void
```

Not linearizable



A: r.write(1) happened before A: r.write(2) as A: r:void response was received. As C: r:2 read 2 already, it is impossible for C to read 1 again.

Sequentially consistent!

1 A: r.write(1)
2 A r.write(2)
3 C. r read()
4 C. r:2
5 A r.write(1)
6. C.read()
7. B.read()
8 .r:1
9.r:1

b. Concurrent threads A, B, C, register r.

```
A: r.write(1)
B: r.read()
A: r:void      r.write(1) finished
A: r.write(2)
A: r:void      r.write(2) finished
A: r.write(1)
B: r:1
C: r.read()
A: r:void
C: r:2
```

Not linearizable

r:1 is already written as B read 1, so no chance for thread C to read 2 as 1 was already written and read, and 2 was never written afterwards

$$G = \begin{bmatrix} A & r.write(1) \\ B & r.read \\ B & r:1 \\ A & r.write\ 2 \\ C. & r.read() \\ C. & 2 \\ A. & r.write(1) \end{bmatrix}$$

sequentially consistent

2. Stack

We have the following operations:

- push(x) pushes element x on the stack, returns void;
- pop() retrieves an element from the stack;
- empty() returns true if stack is empty and false otherwise.

a. Concurrent threads A, B, C, stack s.

```
C: s.empty()
A: s.push(10)
B: s.pop()
A: s:void
A: s.push(20)
B: s:10
A: s:void
C: s:true
```

Not Linearisable. C s.empty returned true after A:s.push(10), A:s.push(20) returned s.void s.void. s-, C: s:true could not return true ~~until~~ both values wouldn't get poped out.

Sequentially consistent

G = C: s.empty
C: s.true
A s.push(10)
A s.void
B. s.pop
B: s.10
A. s.push(20

b. Concurrent threads A, B, C, stack s.

```
A: s.push(10)
B: s.push(10)
A: s:void
A: s.pop()
B: s:void
B: s.empty()
A: s:10
B: s:true
A: s.pop()
A: s:10
```

Not Linearisable, B s.empty() could not return S:true until S.pop wouldn't return

Sequentially consistent

G = A. s.push(10)
A. s.void
A. s.pop
A. s:10
B. s.push
A. s.pop()
A. s:10
B s.empty
B. s true

3. Queue

We have the following operations:

- enq(x) inserts element x in the queue, returns void;
- deq() retrieves an element from the queue.

a. Concurrent threads A, B, C, queue q.

```
A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
A: q.deq()
C: q.deq()
A: q:y
C: q:y
```

Neither linearizable NOR sequential.

y cannot be dequed before X, First in First OUT, X was enqued first and also y cannot be dequed twice