**Esercizio 1**

Ho modificato la classe S4Esercizio1 muovendo l’implementazione dei metodi pubblici **increment** e **getValue** in vari metodi interni privati. Per ogni soluzione (implicit, write, read-write) ho implementato il corrispondente metodo interno.

// Lock Espliciti  
private final static Lock *lock* = new ReentrantLock();  
// RW Lock  
private final static ReentrantReadWriteLock *rwlock* = new ReentrantReadWriteLock();  
private final static Lock *readlock* = *rwlock*.readLock();  
private final static Lock *writelock* = *rwlock*.writeLock();  
  
public static long increment() {  
 //return incrementExp();  
 return *incrementImplicit*();  
 //return incrementRWExp();  
}  
public static long getValue() {  
 //return getValueExp();  
 return *getValueImplicit*();  
 //return getValueRWExp();  
}

Come da aspettative l’implementazione più veloce è quella con I read-write lock, visto che ho molte piu’ lettura (10 worker con frequenza alta) che scritture (3 writer con frequenza piu’ bassa)

|  |  |
| --- | --- |
| **Tipo** | **Comportamento** |
| RW lock | Simulation took: **8625** ms |
| W lock | Simulation took: 9281 ms |
| Implicit lock | Simulation took: 9707 ms |

**Esercizio 2**

Ho modificato il metodo resetIfAbove della classe Sensore passando al metodo **compareAndSet** il valore corrente del counter. Considerando che dopo il controllo della soglia il valore del counter puo’ essere stato cambiato da un altro thread, invece che fare una secca **set** uso la compareAndSet. Col risultato della compareAndSet so se il counter e’ stato effettivamente cambiato (compareAndSet ha ritornato false), in questo caso rieseguo la procedura (iterazione nel while).

private boolean resetIfAbove() {  
 while(true) {  
 int currentAmount = S4Esercizio2.*counter*.get();  
 if(currentAmount < soglia) {  
 return false;  
 }  
 if(S4Esercizio2.*counter*.compareAndSet(currentAmount, 0)) {  
 return true;  
 }  
 }  
 }

**Esercizio 4**

class Coordinate {  
 // MDS: lat e lon, membri pubblici non incapsulati di un oggetto condiviso  
 public double lat = 0.0;

// MDS: update non sincronizzato di lat e lon  
// i due valori vengono aggiornati separatamente  
// il main thread potrebbe leggere un dato parzialmente aggiornato  
S4Esercizio4.*curLocation*.lat = ThreadLocalRandom.*current*().nextDouble(-90.0, +90.0);

public class S4Esercizio4 {  
 // MDS: dovrebbe essere volatile per assicurare corretta visibilitac'  
 static boolean *completed* = false;

// MDS: il main thread non cambia lo stato dell’oggetto condiviso Coordinate

// basterebbe un read lock   
 *lock*.lock();

Ho creato una classe immutable CoordinateImmutable:

final class CoordinateImmutable {  
 private final double lat;  
 private final double lon;  
  
 CoordinateImmutable(double lat, double lon) {  
 this.lat = lat;  
 this.lon = lon;  
 }public double distance(final CoordinateImmutable from) {

L’aggiornamento della posizione corrente avviene creando un nuovo CoordinateImmutable a cui passo gia’ le coordinate. Lo stato non viene piu’ modificato. Al prossimo giro creero’ una nuova istanza di CoordinateImmutable

while (!S4Esercizio4fix.*completed*) {  
 S4Esercizio4fix.*writeLock*.lock();  
 try {  
 double lat = ThreadLocalRandom.*current*().nextDouble(-90.0, +90.0);  
 double lon = ThreadLocalRandom.*current*().nextDouble(-180.0, +180.0);  
 S4Esercizio4fix.*curLocation* = new CoordinateImmutable(lat, lon);  
 } finally {  
 S4Esercizio4fix.*writeLock*.unlock();  
 }

Il main thread usa read lock, mentre il gpsThread usa writeLock. Avendo solo 2 thread non c’e’ un reale vantaggio.

public class S4Esercizio4fix {  
 static volatile boolean *completed* = false;  
 static CoordinateImmutable *curLocation* = null;  
 static ReadWriteLock *rwLock* = new ReentrantReadWriteLock();  
 static Lock *readLock* = *rwLock*.readLock();  
 static Lock *writeLock* = *rwLock*.writeLock();  
 //static Lock lock = new ReentrantLock();