Programmazione Concorrente e Parallela - Serie 3

Maura Clerici

10.03.2019

Esercizio 1

L'esercizio descrive il lavoro di 10 lavoratori (Worker) il quale, durante l'esecuzione (run()), si affida a due variabili di controllo per eseguire delle azioni: isRunning (boolean) e finish (int). Esse non sono protette per un'esecuzione parallela da parte di più lavoratori, quindi il loro valore potrebbe risultare "falso" in memoria. Per proteggerle con la tecnica meno dispendiosa per il programma, vengono ridefinite come variabili volatile:

```
class Worker implements Runnable {
    public static volatile boolean isRunning = false;
    public static volatile int finished = 0;
    ...
}
```

Esercizio 2

È stata sviluppata una prima versione senza implementare alcuna protezione. Il contatore da incrementare è stato rappresentato con una variabile globale visibile da tutti, ma quando si esegue il programma i thread non riescono a leggere il suo vero valore. Essi leggono unicamente un loro valore memorizzato in cache.

Versione con variabile volatile

Implementando lo stesso codice e modificando la variabile globale in volatile, i thread riescono a leggerla correttamente, il programma esegue correttamente. Tuttavia, ci sono delle compound action non gestite, sia all'interno della classe Sensore (nel run()) che nel metodo principale.

Versione con variabile atomica

La seconda versione prevede una variabile atomica; il contatore è stato trasformato in:

```
public static AtomicInteger contatore = new AtomicInteger();
```

Di conseguenza, sono stati utilizzati i metodi S3Esercizio2_atomic.contatore.get() per ottenerne il valore, S3Esercizio2_atomic.contatore.set(0) per azzerarlo e S3Esercizio2_atomic.contatore.getAndAdd(n) per incrementarlo di un numero random. Grazie a quest'implementazione si riesce a garantire l'atomicità della risorsa condivisa nel momento in cui essa viene incrementata.

Versione con Explicit Locks

È stata creata una classe per la risorsa condivisa: il contatore. Sono stati definiti i metodi getValore(), setValore() e resetValore() per interagire con il contatore e sono stati protetti con dei locks:

```
class Contatore {
private final Lock lock = new ReentrantLock();
private int valore = 0;

public int getValore() {
    lock.lock();
    try {
        return valore;
    } finally {
        lock.unlock();
    }
}

public void setValore(int valore) {
    lock.lock();
    try {
        this.valore = valore;
    } finally {
        lock.unlock();
    }
}

public void resetValore() {
    lock.lock();
    try {
```

```
            this.valore = 0;
        } finally {
            lock.unlock();
        }
    }
}
```

In questo modo tutte le interazioni con il contatore risultano protette.

Versione con ReadWrite Locks

È stata mantenuta la classe del contatore e sono stati definiti i seguenti locks:

```
private ReadWriteLock lock = new ReentrantReadWriteLock();
private Lock readLock = lock.readLock();
private Lock writeLock = lock.writeLock();
```

In seguito, i metodi della stessa classe sono stati protetti con i lock read e write:

```
public int getValore() {
    readLock.lock();
    try {
        return valore;
    } finally {
        readLock.unlock();
    }
}

public void setValore(int valore) {
    writeLock.lock();
    try {
        this.valore = valore;
    } finally {
        writeLock.unlock();
    }
}

public void resetValore() {
    writeLock.lock();
    try {
        this.valore = 0;
    } finally {
        writeLock.unlock();
    }
}
```