Esercizi 5-6-2023

Antonio Petrillo

June 2, 2023

Contents

1 Mistery Thread 8 (2020-2-27)

1.1 Code

```
public class A {

    private volatile int n;

    public int incrementAndGet() {
        return ++n;
    }

    public static void main(String[] args) {
        A a = new A();
        A b = new A();
        Thread t1 = new Thread(() -> System.out.println(a.incrementAndGet()));
        Thread t2 = new Thread(() -> System.out.println(b.incrementAndGet()));
        Thread t3 = new Thread(() -> System.out.println(a.incrementAndGet()));
        t1.start();
        t2.start();
        t3.start();
    }

}
```

1.2 Possibili Outputs

count	1st print	2nd print	3rd print
1	1	1	1
2	1	2	3
3	2	1	3
4	1	3	2
5	2	3	1
6	3	1	2
7	3	2	1
8	1	2	2
9	2	1	2
10	2	2	1
11	1	1	2
12	1	2	1
13	2	1	1

2 Missing synch 3 (2019-3-19)

2.1 Snippet

```
class MyThread extends Thread {
    public void run() {
        final int n = a.length - 1;
        // --- 1 ---
        for (int i = 0; i < n >> 1; i++ ){
            // --- 2 ---
            // swap a[i] e a[n - i]
            int temp = a[i];
            a[i] = a[n - i];
            a[n - i] = temp;
            // --- 3 ---
        }
        // --- 4 ---
    }
}
```

2.2 Esercizio

Il main lancia due istanze di MyThread con l'intento di trovare l'array inalterato. Scegliere le risposte che rendono thread-safe ed asente da race

condition l'esecuzione.

2.3 Possibili risposte

1. ☐ Non aggiungere nulla
2. ☐ Aggiungere `synchronized` al run
3. ☐ 1 => `synchronized(this) {` AND 4 => `}`
4. ☐ 1 => `synchronized {` AND 4 => `}`
5. ☒ 1 => `synchronized(a) {` AND 4 => `}`
6. ☒ 1 => `synchronized(MyThread.class) {` AND 4 => `}`
7. ☐ 1 => `a.wait()` AND 4 => `a.notify()`
8. ☐ 2 => `synchronized(this) {` AND 3 => `}`
9. ☐ 2 => `synchronized(a[i]) {` AND 3 => `}`
10. ☒ 2 => `synchronized(a) {` AND 3 => `}`
11. ☒ 2 => `synchronized(MyThread.class) {` AND 3 => `}`

2.4 Non thread safe

2.4.1 A

Di certo non va bene dato che piú thread lavorano (lettura/scrittura) sulle celle dell'array possono esserci race condition sui valori. É necessario rendere run mutuamente esclusivo con se stesso

2.4.2 B

Dalla traccia viene specificato che vengono create due istanze diverse di MyThread che quindi hanno due monitor diversi. Se due MyThread chiamano run, questi acquisiscono il monitor della propria istanza ma non impediscono all'altro di accedere e modificare l'array.

2.4.3 C

Stessa ragione di B.

2.4.4 D

Sintassi errata, va specificato l'oggetto da cui assumere il monitor, non compila

2.4.5 G

`a.wait()` dovrebbe trovarsi in un try/catch, questo codice non compila. Inoltre qualora vi fosse il try/catch lancerebbe un'eccezione a runtime dato che il metodo non acquisice mai il lock del monitor dell'array `a`. Se supponiamo anche che il metodo acquisice anche il monitor di `a` andrebbe in attesa infinita. Sia il primo che il secondo thread effettuerebbero `a.wait()` senza controllare nessuna condizione, questo implica che i thread non possono essere svegliati.

2.4.6 H

Stessa ragione di B e C.

2.4.7 I

L'array é definito nel seguente modo `int[] a`, quindi di per se `a` possiede un monitor, ma le varie celle che lo compongono no dato che sono di tipo primitivo. Con queste modifiche il programma non compilerebbe.

2.5 Thread safe

2.5.1 E

Ora il run acquisice il monitor dell'array impedendo quindi ad altre istanze di modificarlo. Posizionare peró le istruzioni a riga `1` e riga `4` annulla qualsiasi beneficio del parallelismo dato che deve prima terminare il run di un thread per poi poter eseguire l'altro.

2.5.2 F

Dato che tutte le istanze di MyThread sfruttano lo stesso monitor non puó succedere che piú istanze di MyThread si trovino nella zona critica contemporamente. In pratica questa é una soluzione ai problemi di B, C e H. Per un problema del genere acquisire il monitor di `MyThread.class` é sconsigliato dato che é unico in tutta l'esecuzione e non sappiamo se un client puó acquisirlo da qualche parte.

2.5.3 J

Funziona per lo stesso motivo di E, solo che ora il lock viene acquisito prima dell'accesso alle caselle, permettendo quindi un interleaving fra i thread. Questa é la soluzione migliore secondo me.

2.5.4 K

Funziona per le stesse motivazioni di F, e per le stesse ragioni di J (ovvero che permette interleaving), é preferibile ad F. Lo svantaggio é che acquisisce sempre il lock del suo oggetto classe.

3 Missing synch 3 (2021-10-26)

3.1 Snippet

```
public class MyThread extends Thread {

    public void run() {
        // 1
        for (int i = 0; i < a.length; i++) {
            // 2
            if (a[i] > b[i]) {
                int temp = b[i];
                b[i] = a[i];
                a[i] = temp;
            }
            // 3
        }
        // 4
    }
}
```

3.2 Esercizio

Le istanze di MyThread condividono due array `int[]` a e b. Quali delle seguenti istruzioni rendono thread safe il codice?

3.3 Possibili risposte

1. ☐ Non aggiungere nulla

2. ☐ 1 => `synchronized(this) {` AND 4 => `}`
3. ☒ 1 => `synchronized(MyThread.class) {` AND 4 => `}`
4. ☐ 1 => `synchronized {` AND 4 => `}`
5. ☒ 1 => `synchronized(a) {` AND 4 => `}`
6. ☒ 1 => `synchronized(b) {` AND 4 => `}`
7. ☐ 2 => `synchronized(this) {` AND 3 => `}`
8. ☐ 2 => `synchronized(a[i]) {` AND 3 => `}`
9. ☒ 2 => `synchronized(b) {` AND 3 => `}`
10. ☐ 2 => `a.wait()` AND 3 => `a.notifyAll()`

3.4 Non Thread safe

3.4.1 A

Piú thread condividono `a` e `b` quindi vanno coordinati.

3.4.2 B

Acquisisre il monitor di una istanza del thread non impedisce alle altre istanze di accedere alla zona critica.

3.4.3 D

Non compila, nel blocco synchronized va specificato l'oggetto di cui prendere il monitor.

3.4.4 G

Non funziona per lo stesso motivo id B.

3.4.5 H

L'array possiede un monitor essendo un oggetto ma il tipo primitivo `int` no, questa soluzione non compila.

3.4.6 J

Il thread non acquisice il monitor di `a` quindi lancerá un errore a runtime nel momento in cui `wait()` cercherá di lasciare il lock.

3.5 Thread safe

3.5.1 C

Funziona dato che tutte le istanze di thread sono *sincronizzate* sullo stesso monitor. Posizionare le istruzioni a riga 1 e 4, comporta che l'esecuzione di un thread avviene solo dopo che il thread che ha prima acquisito il lock lo rilasci, questa soluzioni in pratica annulla il parallelismo.

3.5.2 E

Tutti i thread sono *sincronizzati* sullo stesso monitor quindi non puó capitare che piú thread si trovino nella sezione critica contemporamente. Come la C, questa soluzione annulla del tutto il parallelismo.

3.5.3 F

Come E, quindi anche con gli stessi problemi, solo che acquisice il monitor di `b`.

3.5.4 I

Concettualmente come E ed F con la differenza che ora le istruzioni sono a riga 2 e 3 permettendo un interleaving fra i thread. Questa é la soluzione migliore.