Concorrenza

Ver 07022023

Accesso a variabili condivise

```
public class Counter {

    private int counter;

    public void increment() {
                counter++;
    }

    public int getValue() {
           return counter;
    }

}
```

Data una variabile **contatore** di tipo **Counter** condivisa tra due thread *A* e *B* che invocano ripetutamente il metodo **increment()** dell'oggetto condiviso **contatore**.

Cosa succede se thread A e thread B invocano contemporaneamente **contatore.increment()**?

Ci aspetteremmo che il valore iniziale di **contatore** sia incrementato di due unità.

Potrebbe succedere che invece la variabile risulti incrementata solo di una unità. Il risultato sarebbe errato!!

Cerchiamo prima di capire il motivo

```
public class Incrementer extends Thread {

    private final Counter counter;
    private final int incrementValue;

    public Incrementer(Counter counter, int incrementValue)    {
        this.counter = counter;
        this.incrementValue = incrementValue;
    }

    public void run() {
        for (int i = 0; i < incrementValue; i++) {
            counter.increment();
        }
    }

}
```

Qui la classe *Incrementer* di tipo *Thread* che esegue gli incrementi sull'oggetto di tipo *Counter* che viene passato nel costruttore.

```
*
     Counter counter = new Counter();

     int incrementValue1 = 200_000_000;
     int incrementValue2 = 100_000_000;

     Incrementer threadA = new Incrementer(counter, incrementValue1);
     Incrementer threadB = new Incrementer(counter, incrementValue2);

     threadA.start();
     threadB.start();

     threadA.join();
     threadB.join();

     int counterValue = counter.getValue();

     System.out.println("SUM VALUE: " +  counterValue + "- SHOULD BE: " + (incrementValue1 +
incrementValue2));
```

Alcuni output dell'esecuzione:

SUM VALUE: 200.074.966 - SHOULD BE: 300.000.000
SUM VALUE: 201.413.358 - SHOULD BE: 300.000.000
SUM VALUE: 200.757.965 - SHOULD BE: 300.000.000

Come si vede la differenza è notevole tra il risultato ottenuto e quello corretto. A cosa è dovuta questa inconsistenza?

```
// classe Counter
public void increment() {
            counter++;
}
```

Analizziamo il problema: il metodo increment() è composto dal semplice statement conter++, ma anche questa semplice operazione NON è un operazione atomica; è composta da più istruzioni macchina: quando più thread invocano il metodo SULLO STESSO OGGETTO, il risultato potrebbe non essere quello che ci aspettiamo.

Il semplice statement di increment, counter++, può essere decomposto in 3 step:
1) Caricare il valore corrente di counter;
2) Incrementare il valore recuperato di 1;
3) Immagazzinare nella variabile counter il valore incrementato;

Per indicare una operazione all'interno di una sequenza di esecuzione, introduciamo la seguente notazione:

ti.j indica l'operazione j svolta dal thread ti;

Se supponiamo counter vale 5, supponiamo che i thread A e B invocano il metodo increment(), su un oggetto *contatore* di tipo Counter, con questo flusso d'esecuzione: A.1 < B.1 < A.2 < B.2 < A.3 < B.3 il valore di *counter* alla fine dell'esecuzione vale 6 e non come ci si aspetterebbe 7.

Mentre con questo flusso d'esecuzione: A.1 < A.2 < A.3 < B.1 < B.2 < B.3
sempre partendo da un valore di counter = 5 avremmo alla fine dell'esecuzione il valore corretto 7.

SIAMO NELLA SITUAZIONE DI ACCESSO in MODIFICA CONCORRENTE AD UNA VARIABILE anche detta SEQUENZA CRITICA.

L’esempio precedente mostra un tipico problema della programmazione concorrente: per ottenere un risultato corretto alcune sequenze di istruzioni non devono essere mescolate tra loro durante l’esecuzione. Chiameremo una sequenza di istruzioni di questo tipo sequenza critica e chiameremo mutua esclusione la proprietà che vogliamo garantire a tali sequenze.

Mutua esclusione: metodi synchronized

```
public class Counter {

    private int counter;

    public synchronized void increment() {
        counter++;
    }

    public synchronized int getValue() {
        return counter;
    }

}
```

Il **synchronized** aggiunta alla signature del metodo garantisce che un solo thread alla volta può eseguire il metodo: il thread deve acquisire il lock associato all'oggetto per poter eseguire il metodo.

Mutua esclusione: metodi synchronized

```
public class Counter {

     private int counter;

     public synchronized void increment() {
          counter++;
     }

     public synchronized int getValue() {
          return counter;
     }


}
```

Quando il thread invoca un metodo sincronizzato, in automatico, cerca di acquisire il lock associato all'oggetto. Se qualcun altro ha già acquisito il lock deve aspettare in coda, bloccato finché il primo non ha finito; solo a questo punto, quest'ultimo può acquisire il lock e può eseguire quindi il metodo. UNO ALLA VOLTA, per questo si parla di mutua esclusione.

Mutua esclusione: i blocchi synchronized

```
synchronized( ‹object›  ) {
    ‹statements›
}
```

Il blocchi synchronized è il secondo modo per garantire la mutua esclusione: il thread per eseguire il blocco di codice tra {} deve acquisire il lock dell'oggetto tra () e questo garantisce la mutua esclusione: un solo thread alla volta potrà eseguire questo codice. Se ad esempio:

```
//contatore variabile di tipo Counter

synchronized(contatore) {
    contatore.increment();
}
```

Un solo thread alla volta può eseguire questo codice perché per eseguire il codice deve prima acquisire il lock della variabile, in questo caso, *contatore*.

Mutua esclusione: i blocchi synchronized

```
public class Counter {

    private int counter;

    public void increment() {
        synchronized(this) {
            counter++;
        }
    }

    public int getValue() {
        synchronized(this) {
            return counter;
        }
    }

}
```

Esempio di contatore implementato con blocchi synchronized equivalente alla versione con metodi synchronized.

Mutua esclusione: i blocchi synchronized

```
public class Counter {

    private int counter;

    public void increment() {
        synchronized(this) {
            counter++;
        }
    }

    public int getValue() {
        synchronized(this) {
            return counter;
        }
    }

}
```

Dati due thread A e B, possono eseguire contemporaneamente il blocco che incrementa il *counter*?

NO NON POSSONO: uno solo dei due thread prende il lock dell'oggetto, l'altro thread deve aspettare finché il primo thread non ha terminato l'esecuzione del blocco di codice. Solo a questo punto, il secondo thread può eseguire anche lui lo stesso blocco del codice.

Mutua esclusione: i blocchi synchronized

```
public class Counter {

    private int counter;

    public void increment() {
        synchronized(this) {
            counter++;
        }
    }

    public int getValue() {
        synchronized(this) {
            return counter;
        }
    }

}
```

Dati due thread A e B, possono eseguire contemporaneamente, uno il blocco synchronized che incrementa il *counter* e l'altro quello che ritorna il valore di *counter*?

NO NON POSSONO: uno solo dei due thread prende il lock dell'oggetto, l'altro thread deve aspettare finché il primo thread non ha terminato l'esecuzione del blocco anche se sono due blocchi synchronized differenti (il lock dell'oggetto è unico).

Mutua esclusione: i blocchi synchronized

```
public class Counter {

    private int counter;
    private Object obj1 = new Object();
    private Object obj2 = new Object();

    public void increment() {
        synchronized(obj1) {
            counter++;
        }
    }

    public int getValue() {
        synchronized(obj2) {
            return counter;
        }
    }

}
```

Dati due thread A e B, possono eseguire contemporaneamente, uno il blocco synchronized che incrementa il *counter* e l'altro quello che ritorna il valore di *counter*?

SI POSSONO eseguire contemporaneamente i due blocchi: quando il thread A accede al blocco che incrementa counter acquisisce il lock dell'oggetto obj1, mentre quando thread B accede al blocco che ritorna il valore di counter ottiene il lock da obj2.

Un altro esempio di sequenza critica, può succedere in un if statement.
Consideriamo il seguente statement, che ha lo scopo di evitare l'errore di divisione per zero:

```
if ( A != 0 ) {
    B = C / A;
}
```

Supponiamo che questo codice è eseguito da alcuni thread.
Se la variabile A è condivisa da uno o più thread, e se nulla è fatto per proteggere la sequenza critica, allora è possibile che uno questi thread possa cambiare il valore di A portandolo a 0 nel frattempo che il primo thread ha controllato la condizione A != 0 e si appresta a eseguire la divisione.

Questo significa potrebbe finire a dividere per zero, anche se ha controllato che A sia diversa da 0!
Per fissare il problema delle sezioni critiche, ci deve essere qualche modo per acquisire un accesso esclusivo a una risorsa condivisa.

Utilizzo blocchi synchronized

```
// codice eseguito da thread 1
synchronized(A) {
        if ( A != 0 ) {
                B = C / A;
        }
}
```

```
// codice eseguito da thread 2
synchronized(A) {
        A = 0;
}
```

Non basta che solo uno di questi blocchi sia synchronized perché il codice sia sicuro! Anche il blocco che assegna ad A il valore 0 deve essere synchronized per garantire la mutua esclusione.

Esempi con accesso a variabili condivise da parte di più thread

- Incremento Contatore: accesso condiviso a una variabile contatore incrementata da più thread concorrentemente;
- Trasferimento bonifico: esempio di operazioni di trasferimento valori che rappresentano conti bancari in modo concorrente;
- Calcolo dei Numeri Primi: utilizzo di più thread per eseguire il calcolo sul numero di numeri primi in un certo range di interi;

I Deadlock (stallo)

La sincronizzazione può aiutare a prevenire le **race condition**, ma introduce la possibilità di un altro tipo di errore, il **deadlock** o **stallo**. Un deadlock avviene quando un thread continua ad aspettare una risorsa che non gli arriverà mai.

In una cucina, un deadlock può succedere se due cuochi vogliono contemporaneamente misurare una tazza di latte. Il primo cuoco prende il misurino e il secondo cuoco prende il latte. Il primo cuoco ha bisogno del latte, ma non può averlo perché ce l'ha il secondo cuoco. Il secondo cuoco ha bisogno del misurino, ma non lo può ottenere perché ce l'ha il primo.

Nessun cuoco può continuare e niente di più può essere fatto. Questo è il deadlock.

I Deadlock

Esattamente la stessa cosa può succedere in un programma, per esempio se ci sono due thread (come i due cuochi) entrambi dei quali deve ottenere i lock su gli stessi due oggetti (come il latte e il misurino) prima che possano procedere.

I deadlock possono capitare facilmente a meno che grande attenzione non è presa per evitarli.

La situazione più elementare di deadlock si crea quando due thread t1 e t2 bloccano due risorse A e B e raggiungono una situazione nella quale t1 ha bloccato A e attende di bloccare B mentre t2 ha bloccato B e attende di bloccare A.

Vedi Esempio Deadlock.