Azzolini Riccardo

Correttezza dei programmi concorrenti – Problemi tipici e come evitarli

1 Cooperazione fra thread

`synchronized` permette a un thread di modificare in modo sicuro dei dati condivisi, che potranno essere letti da un altro thread. Spesso, però, l'altro thread ha bisogno di sapere che tali dati sono stati modificati. In questo caso, l'uso di parti di codice `synchronized` non è sufficiente: siccome ciclare su un dato condiviso finché esso non cambia valore non è una soluzione accettabile, servono invece dei meccanismi che mettano in attesa un thread, e che lo risveglino quando le condizioni sono cambiate. A tale scopo, Java mette a disposizione i metodi `wait`, `notify` e `notifyAll` della classe `Object` (che sono quindi invocabili su qualsiasi oggetto).

Nota: Un thread può chiamare questi metodi su un oggetto solo quando detiene il lock di tale oggetto, cioè all'interno di un metodo/blocco `synchronized` sull'oggetto: chiamarli in un contesto diverso genera un'eccezione.

1.1 Metodo `wait`

Quando un thread chiama `wait`, esso rilascia il lock sull'oggetto, e va in attesa. In seguito, quando il thread verrà risvegliato (mediante una chiamata `notify` o `notifyAll`), riacquisirà anche il lock:

```
synchronized (this) { // lock
    try {
        this.wait(); // unlock
        // lock
    } catch (InterruptedException e) {}
} // unlock
```

Un thread che esegue una `wait` viene messo in un'apposita lista, chiamata **wait list**, contenente tutti i thread che hanno invocato `wait` sullo stesso oggetto. Poi, quando esso dovrà essere risvegliato, verrà spostato dalla wait list alla **ready list** (la lista dei thread schedulabili). Ciò è simile alla gestione dei thread che fanno richieste di I/O su dispositivi non pronti (ma, ovviamente, cambiano gli eventi che provocano l'attesa e il risveglio).

`wait` si usa solitamente quando un thread determina, dopo essere entrato in una sezione `synchronized`, che non ci sono le condizioni necessarie per proseguire, e quindi deve attendere finché esse non diventano verificate (a causa di modifiche apportate all'oggetto da parte di altri thread).

1.2 Metodi `notify` e `notifyAll`

Una chiamata `notify` su un oggetto sposta un thread *qualsiasi* (scelto in modo non deterministico) dal wait set dell'oggetto al ready set (cioè, in pratica, lo risveglia, nel senso che lo rende nuovamente schedulabile).

Invece, invocando `notifyAll` su un oggetto si spostano *tutti* i thread dal suo wait set al ready set.

Quale di questi due sia il metodo corretto da utilizzare varia in base alle circostanze. Ad esempio:

- se un insieme di thread attendono la fine di un determinato compito, quando esso termina si può usare `notifyAll` per sbloccarli tutti;
- se un insieme di thread attendono di entrare in un blocco esclusivo, è necessario risvegliarne solo uno alla volta, mediante `notify`.

2 Monitor

Si definisce **monitor** una struttura dati i cui metodi sono *tutti* `synchronized`, in modo che, in un dato momento, un solo thread possa eseguirne i metodi. In altre parole, un oggetto di questo tipo può essere "usato" da un solo thread alla volta.

Il concetto di monitor è stato ideato come costrutto di sincronizzazione per linguaggi ad alto livello (ben prima dell'esistenza di Java). Esso è infatti equivalente ai semafori:

- Si può creare un monitor usando un semaforo, dotando ciascuna istanza della classe di un proprio semaforo, e inserendo, in ogni metodo, una chiamata `acquire` all'inizio e una `release` alla fine. In Java, questo è esattamente ciò che si ottiene dichiarando i metodi come `synchronized`.
- Si può definire un semaforo usando un monitor:

```
public class Semaphore {
    private int value;

    public Semaphore(int init) {
        value = init;
    }
```

```
    public synchronized void acquire() throws InterruptedException {
        while (value == 0) {
            wait();
        }
        value--;
    }

    public synchronized void release() {
        value++;
        notify();
    }
}
```

In particolare, Java non fornisce i semafori come costrutto di base del linguaggio, ma essi sono invece disponibili tramite la classe `java.util.concurrent.Semaphore`.

3 Coordinazione di più thread

Uno dei problemi ricorrenti nell'ambito della programmazione concorrente è l'esigenza di coordinare le attività svolte da più thread. Ad esempio, può essere necessario che due o più thread eseguano un determinato compito a turno, alternandosi in un ordine *controllato dall'applicazione* (e non dallo scheduler).

4 Produttore-consumatore

Un altro problema tipico è quello del **produttore-consumatore**, detto anche problema del **buffer limitato**:

- un thread, il produttore, deposita dati in una zona di memoria condivisa (il buffer);
- un altro thread, il consumatore, preleva dati dal buffer;
- il buffer ha capacità limitata.

Il problema è assicurare che:

- il produttore non cerchi di inserire nuovi dati quando il buffer è pieno (esso deve invece attendere il consumatore crei uno spazio, cioè estragga un elemento);
- il consumatore non cerchi di estrarre dati quando il buffer è vuoto (in questo caso, esso si deve mettere in attesa finché il produttore non rende disponibili altri dati).

5 Deadlock

Quando si garantisce a un processo l'accesso esclusivo a una risorsa (ad esempio tramite mutua esclusione), si evitano le race condition, ma possono invece crearsi situazioni problematiche di **deadlock** (**stallo**).

Ci sono quattro condizioni necessarie affinché un deadlock si verifichi. Esse sono generalmente espresse in termini di risorse assegnate a un thread, e sono:

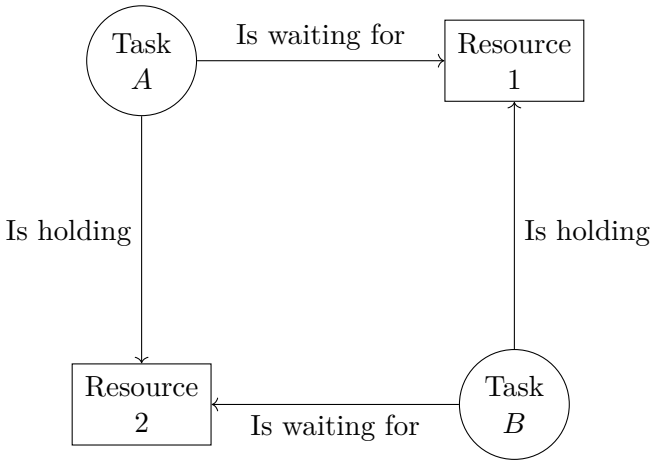
Mutual exclusion: solo un'attività concorrente per volta può utilizzare una risorsa (in altre parole, essa non è condivisibile simultaneamente).

Hold and wait: esistono attività concorrenti che sono in possesso di alcune risorse, e intanto aspettano che si liberino altre risorse da acquisire.

Assenza di preemption sulle risorse: una risorsa può essere rilasciata solo volontariamente da un'attività concorrente (non può essere tolta con la forza).

Circular wait: esiste una catena circolare di attività concorrenti tale che ognuna di esse mantenga bloccate delle risorse che, contemporaneamente, vengono richieste dalle attività successive. In altre parole, esiste un gruppo di attività (thread) $\{t_0, t_1, \ldots, t_n\}$ nel quale t_0 è in attesa di una risorsa occupata da t_1, t_1 attende una risorsa occupata da t_2, ecc., e t_n sta aspettando una risorsa di t_0.

Raffigurazione di un (caso semplice di) deadlock



Quando si verificano queste condizioni, siccome tutti i thread sono in attesa, nessuno potrà mai creare l'evento di sblocco (cioè rilasciare una risorsa che serve ad altri, dopo aver finito di usarla), quindi l'attesa si protrae all'infinito.

5.1 Esempio

```
import java.util.concurrent.ThreadLocalRandom;

public class LockObjects implements Runnable {
    private Object first, second;

    public LockObjects(Object first, Object second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            Thread.sleep(ThreadLocalRandom.current().nextInt(10, 100));
            System.out.println("Running");

            synchronized (first) {
                System.out.println("First item locked");
                Thread.sleep(
                    ThreadLocalRandom.current() .nextInt(10, 100)
                );

                synchronized (second) {
                    System.out.println(
                        "Lock worked: " + first + ", " + second
                    );
                }
            }
        } catch (InterruptedException e) {}
    }
}

public class MakeDeadlock {
    public static void main(String[] args) {
        Object a = new Object();
        Object b = new Object();
        Thread t1 = new Thread(new LockObjects(a, b));
        Thread t2 = new Thread(new LockObjects(b, a));
        t1.start();
        t2.start();
    }
}
```

In questo esempio, il thread `t1` cerca di ottenere prima il lock sull'oggetto `a`, e poi il lock su `b`, mentre `t2` vuole ottenere i due lock nell'ordine inverso (prima su `b`, poi su `a`). Allora, se uno dei due thread venisse interrotto dallo scheduler mentre ha acquisito il primo lock, ma non ancora il secondo, si ha una situazione di stallo:

```
Running
First item locked
Running
First item locked
<deadlock>
```

Siccome, però, il verificarsi dei deadlock è non deterministico, è assolutamente possibile che uno dei due thread riesca ad acquisire entrambi i lock prima dell'altro. In tal caso, il programma funziona correttamente, e produce un output del tipo:

```
Running
First item locked
Lock worked: java.lang.Object@7c3386c7, java.lang.Object@3a1db8d6
Running
First item locked
Lock worked: java.lang.Object@3a1db8d6, java.lang.Object@7c3386c7
```

5.2 Modi per gestire il deadlock

Esistono due principali approcci per affrontare le situazioni di deadlock:

Deadlock prevention: si previene il verificarsi di deadlock, facendo in modo che almeno una delle quattro condizioni necessarie non si verifichi mai.

Deadlock removal: non si previene il deadlock, ma lo si risolve quando ci si accorge che è avvenuto (ad esempio, rendendo le risorse *preemptible*, cioè consentendo a un thread di sottrarle forzatamente a un altro).

Questa seconda opzione è più complicata, in quanto:

- serve un modo di accorgersi del verificarsi di un deadlock;
- quando viene negata una risorsa a un thread (per risolvere un deadlock), esso deve essere riportato nello stato in cui si trovava prima di tentare l'acquisizione.

Per questo, essa viene utilizzata soprattutto in ambiti particolari, quali ad esempio i database (che dispongono di meccanismi per riportare una transazione a uno stato precedente).

5.2.1 Esempi di deadlock prevention

- Una possibile strategia di deadlock prevention è eliminare la *hold and wait*, cioè fare in modo di non detenere mai una risorsa mentre si è in attesa di un'altra. Per esempio, il codice

  ```
  synchronized (a) {
      // ...
      synchronized (b) {
          // ...
      }
  }
  ```

  dovrebbe essere trasformato in:

  ```
  synchronized (a) {
      // ...
  }
  synchronized (b) {
      // ...
  }
  ```

  Questa soluzione è facile da applicare, ma non sempre possibile: se, ad esempio, l'elaborazione da effettuare coinvolge sia `a` che `b`, bisogna per forza avere contemporaneamente i lock su entrambi, e ciò comporta inevitabilmente l'acquisizione di uno dei due lock mentre si detiene già l'altro.

- Un'altra opzione è eliminare la circolarità, ordinando le risorse e richiedendo che l'acquisizione dei lock segua sempre tale ordine: ad esempio, se A precede B, e un thread vuole accedere a entrambe queste risorse, esso deve prima acquisire il lock su A, e poi, una volta che lo detiene, può richiedere anche quello su B.

  Talvolta, però, un thread può scoprire che ha bisogno di A solo dopo aver acquisito B e fatto un po' di elaborazioni: in tal caso, esso dovrebbe "disfare" le elaborazioni eseguite su B e rilasciare la risorsa, ottenere l'accesso ad A e B nell'ordine corretto, e infine ripetere le elaborazioni (il che potrebbe essere difficile o impossibile).