Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Programmazione parallela con i Java Thread – classi per la gestione della concorrenza nel package java.util.concurrent

Classi per la programmazione concorrente, disponibili nel package java.util.concurrent.

Abbiamo già visto la classe **Semaphore**.

Il costruttore **Semaphore(int n)** costruisce un semaforo con **n** permessi. Se n = 1 si ha un semaforo binario.

I metodi **acquire()** e **release()** acquisiscono o rilasciano un permesso.

Quando non ci sono più permessi un thread rimane in attesa al semaforo.

Interface BlockingQueue

L'interfaccia **BlockingQueue** definisce una coda sincronizzata:

- Quando la coda è piena, il thread che cerca di scrivere nella coda con il metodo **put()** viene messo in attesa.
- Quando la coda è vuota, il thread che cerca di leggere dalla coda con il metodo **take()** viene messo in attesa.

Con questa classe è molto semplice realizzare uno schema produttore-consumatore.

java.util.concurrent

Interface BlockingQueue<E>

Type Parameters:

E - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>, Queue<E>

All Known Subinterfaces:

BlockingDeque<E>, TransferQueue<E>

All Known Implementing Classes:

ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue

BlockingQueue è un'interfaccia: ci sono diverse classi che la implementano. Ad esempio:

```
BlockingQueue<Integer> c =
                    new ArrayBlockingQueue<Integer>(5);
```

Integer: tipo degli elementi della coda
5: capacità della coda

```
class Producer extends Thread {
   private BlockingQueue<Integer> bq;
   private int num;

   public Producer(BlockingQueue<Integer> c, int num) {
      bq = c;
      this.num = num;
   }

   public void run() {
      for (int i = 0; i < 10; i++) {
            try {
                  bq.put(num*10+i);
                  System.out.println("Producer #" + this.num
                                          + " put: " + (num*10+i));
                  sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) { }
      }
   }
}
```

Interface Lock e Condition

In Java ogni oggetto ha un lock implicito. È anche possibile usare dei lock espliciti mediante l'interfaccia **Lock**. **Lock** fornisce una funzionalità analoga con il metodo **newCondition()** che restituisce un oggetto di tipo **Condition**.

L'interfaccia **Condition** fornisce i metodi:

- **await()** analogo alla wait() di Object
- **signal()** analogo alla notify() di Object
- **signalAll()** analogo alla notifyAll() di Object

```
Lock lock = new ReentrantLock();
Condition notFull  = lock.newCondition();
Condition notEmpty = lock.newCondition();
.....
lock.lock(); // ottiene il lock
.....
notFull.await(); // il thread viene messo in attesa
                 //  sulla condizione notFull
.....
notEmpty.signal(); // viene risvegliato un thread
                   // in attesa sulla condizione notEmpty
.....
lock.unlock();  // rilascia il lock
```

ReentrantLock: implementazione dell'interface Lock

Lock può essere usato per realizzare un **bounded buffer**, con due metodi **put()** e **take()**.

I due metodi si sincronizzano su un lock esplicito, con due condizioni **notFull** e **notEmpty**.

Quando il buffer è pieno, un thread che cerca di scrivere nel buffer viene messo in attesa sulla condizione **notFull**.

Quando il buffer è vuoto, un thread che cerca di leggere dal buffer viene messo in attesa sulla condizione **notEmpty**.

Quando un thread finisce di scrivere fa una **signal()** su **notEmpty**.

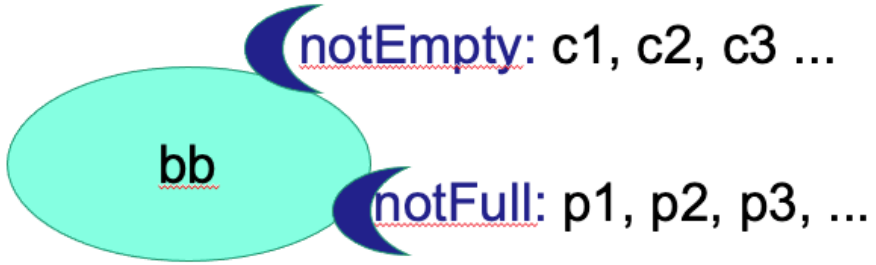
Quando un thread finisce di leggere fa una **signal()** su **notFull**.

Es. produttore-consumatore:

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final int[] items = new int[10];
   int putptr, takeptr, count;

   public void put(int x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
          notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
```



```
public int take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      int x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Nell'esempio precedente non è necessario usare la **signalAll()** anche se ci sono molti produttori e molti consumatori.

Infatti i thread in wait su **notFull** sono tutti produttori e non c'è differenza fra svegliarne uno o un altro.

Analogamente i thread in wait su **notEmpty** sono tutti consumatori.

Viceversa nella implementazione con la classe **CubbyHole** c'è una sola coda di wait in cui vengono inseriti sia produttori che consumatori. Facendo la **notify()** invece della **notifyAll()** si potrebbe svegliare un thread che non è in grado di proseguire, causando una situazione di deadlock.

Lettori e scrittori

Interface **ReadWriteLock** specifica due metodi:

Lock readLock()

Lock writeLock()

Il primo metodo restituisce un lock che può essere tenuto simultaneamente da più lettori, purché non ci siano scrittori.

Il secondo metodo restituisce un lock esclusivo.

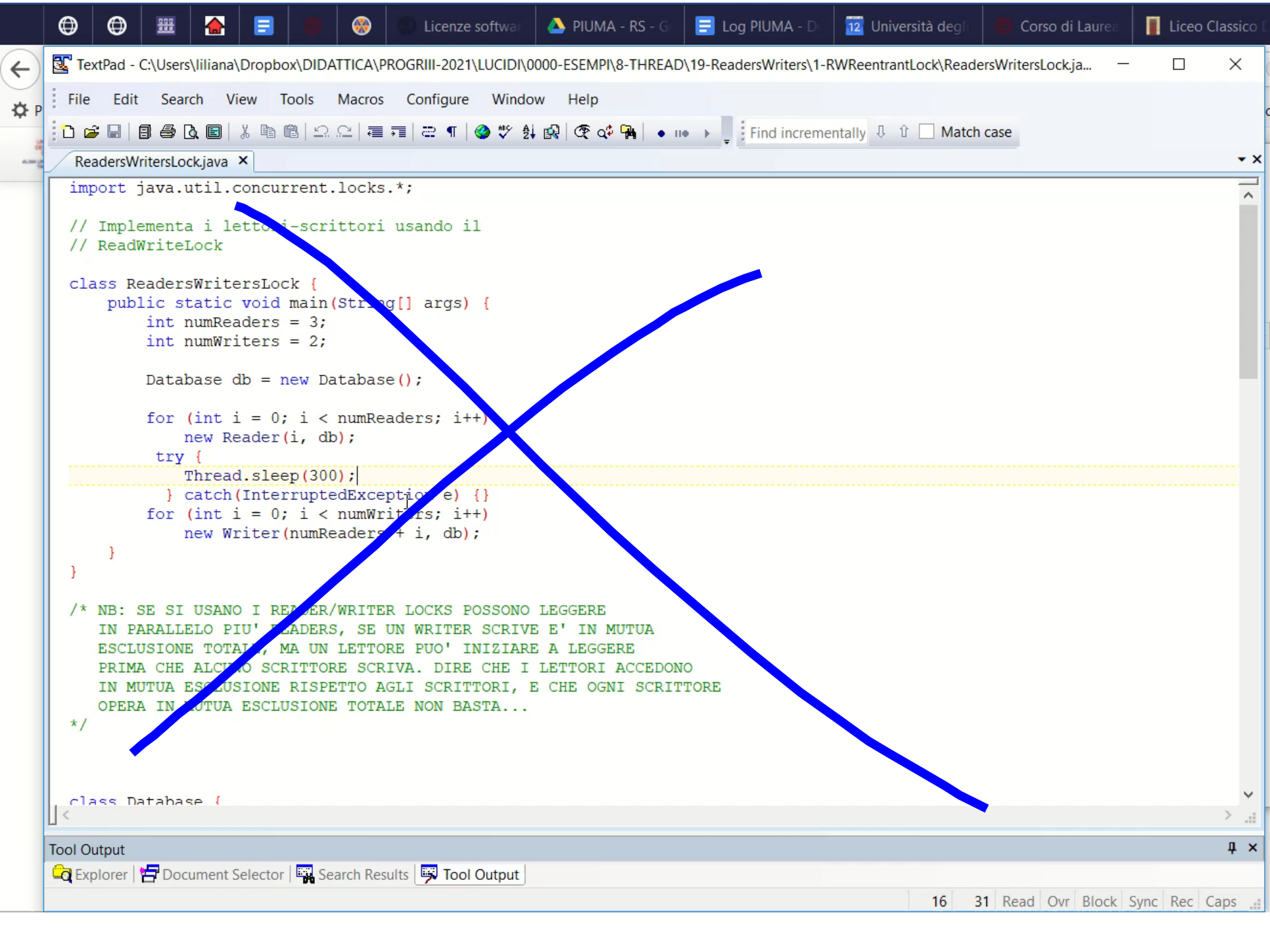
Con questi due lock è facile definire una struttura dati con un metodo read() che usa il primo lock e un metodo write() che usa il secondo.

Esempio: ReadersWritersLock - I

```
class Database {
        private ReadWriteLock rwl = new ReentrantReadWriteLock();
        private Lock rl = rwl.readLock();
        private Lock wl = rwl.writeLock();

        public void read(int i)  {      // i rappresenta l'ID del thread reader
                rl.lock();
                .......
                rl.unlock();
        }
        public void write(int i)  {     // i rappresenta l'ID del thread writer
                wl.lock();
                .......
                wl.unlock();
        }
}
```

Questo garantisce la mutua esclusione ma un lettore potrebbe leggere prima che qualche scrittore scriva → vd esempio


TextPad - C:\Users\liliana\Dropbox\DIDATTICA\PROGRIII-2021\LUCIDI\0000-ESEMPI\8-THREAD\19-ReadersWriters\1-RWReentrantLock\ReadersWritersLock.ja...
File Edit Search View Tools Macros Configure Window Help
Find incrementally
Match case
ReadersWritersLock.java
import java.util.concurrent.locks.*;
// Implementa i lettori-scrittori usando il
// ReadWriteLock
class ReadersWritersLock {
public static void main(String[] args) {
int numReaders = 3;
int numWriters = 2;
Database db = new Database();
for (int i = 0; i < numReaders; i++)
new Reader(i, db);
try {
Thread.sleep(300);
} catch(InterruptedException e) {}
for (int i = 0; i < numWriters; i++)
new Writer(numReaders + i, db);
}
}
/* NB: SE SI USANO I READER/WRITER LOCKS POSSONO LEGGERE
IN PARALLELO PIU' READERS, SE UN WRITER SCRIVE E' IN MUTUA
ESCLUSIONE TOTALE, MA UN LETTORE PUO' INIZIARE A LEGGERE
PRIMA CHE ALCUNO SCRITTORE SCRIVA. DIRE CHE I LETTORI ACCEDONO
IN MUTUA ESCLUSIONE RISPETTO AGLI SCRITTORI, E CHE OGNI SCRITTORE
OPERA IN MUTUA ESCLUSIONE TOTALE NON BASTA...
*/
class Database {
Tool Output
Explorer
Document Selector
Search Results
Tool Output
16 31 Read Ovr Block Sync Rec Caps

Esempio: ReadersWritersLock - II

Come si resolve il problema della lettura di dati inesistenti?

Usando le Condition per mettere in attesa i reader che tentano di leggere quando non ci sono dati disponibili, come fatto negli esempi precedenti (notFull, notEmpty).

Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.