POLITECNICO DI MILANO

5. Concorrenza e Thread (2)

Informatica 3

Andrea Mocci, mocci@elet.polimi.it

Esercizio 1

- Si consideri in Java un oggeto condiviso, esemplare di una classe Pippo che contiene una variabile intera che deve essere sempre maggiore o uguale a zero.
- L'oggetto è accessibile in modo concorrente da diversi threads mediante le seguenti operazioni:
  - inc(x) : incrementa di x il valore della variabile condivisa;
  - dec(x) : decrementa di x il valore della variabile condivisa.

Esercizio 1

- Si tratteggi la classe Pippo nei seguenti casi;
- Per il decremento, si considerino i due seguenti casi:

1. Il decremento di x può essere fatto solo se il valore della variabile condivisa prima del decremento è positiva, altrimenti il thread viene sospeso;
2. Il decremento può essere fatto solo se il valore della variabile condivisa dopo il decremento non diventa negativa, altrimenti il thread viene sospeso;

Esercizio 1

- Per quanto riguarda l'incremento, si considerino i seguenti casi:

3. L'incremento può essere sempre fatto;
4. L'incremento può essere fatto solo se il valore della variabile condivisa prima dell'incremento non supera una certa soglia (definita a priori);
5. L'incremento può essere fatto solo se il valore della variabile condivisa dopo l'incremento non supera la soglia di cui sopra.

Esercizio 1 (soluzione)

- Versione più semplice: 1 + 3

```
/* 1) Decremento solo se valore > 0
 * 3) Incremento puo' sempre essere fatto */
public class Pippo {
   private int value = 0;
   public synchronized void dec(int x) {
    while(value <= 0) {
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    value -= x;
   }
   public synchronized void inc(int x) {
    value += x; notifyAll();
   }
}
```

Esercizio 1 (soluzione)

- Versione con requisiti 2 + 3

```
/* 2) Decremento solo se valore dopo decremento >= 0
 * 3) Incremento puo' sempre essere fatto */
public class Pippo {
   private int value = 0;
   public synchronized void dec(int x) {
    while(value – x <= 0) {
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    value -= x;
   }
   public synchronized void inc(int x) {
    value += x; notifyAll();
   }
}
```

Esercizio 1 (soluzione)

- Versione con requisiti 1 + 4

```
/* 1) Decremento solo se valore >= 0
 * 4) Incremento solo se valore prima dell'incremento non
   supera la soglia */
public class Pippo {
   private int value = 0;
   private int soglia = 0;
   public Pippo (int soglia) {
     this.soglia = soglia;
   }

   public synchronized void inc(int x) {
    value += x; notifyAll();
   }
```

Esercizio 1 (soluzione)


```
    /* 4) Incremento solo se valore prima dell'incremento non
    supera la soglia */
    public synchronized void inc(int x) {
     if (value > soglia)
              return;
         value += x;
         notifyAll();
    }
}
```

Esercizio 1 (soluzione)

- Versione con requisiti 1 + 5

```
/* 1) Decremento solo se valore >= 0
 * 5) Incremento solo se valore dopo l'incremento non supera
   la soglia */
public class Pippo {
   private int value = 0;
   private int soglia = 0;
   public Pippo (int soglia) {
     this.soglia = soglia;
   }

   public synchronized void inc(int x) {
    value += x; notifyAll();
   }
```

Esercizio 1 (soluzione)


```
    /* 3) Incremento solo se valore dopo l'incremento non
    supera la soglia */
    public synchronized void inc(int x) {
     if (value + x > soglia)
              return;
         value += x;
         notifyAll();
    }
}
```

Esercizio 2

- Si consideri un programma concorrente in Java in cui più thread possono accedere ad un oggetto ST di tipo Stack (una pila) condiviso.
- Per tale tipo sono definiti due metodi, push e pop, di tipo synchronized.
- Push ha come parametro una lista di elementi, che vengono inseriti in cima alla pila.
- Pop ha un parametro intero k, e il suo effetto è quello di prelevare una lista di k elementi dalla cima della pila, restituendola al chiamante.

Esercizio 2

- Si supponga che lo Stack abbia una capacità illimitata;
- Occorre invece che i threads che cercano di eseguire l'operazione pop possano sospendersi nel caso in cui lo stack non contenga il numero voluto di elementi.

Esercizio 2 (soluzione)


```
import java.util.Stack;
import java.util.List;
import java.util.ArrayList;

public class SharedStack {
    private Stack st = new Stack();

    public synchronized void push(List l) {
        for (Object o: l) {
            this.st.push(o);
        }
        notifyAll();
    }
…
```

Esercizio 2 (soluzione)


```
public synchronized List pop(int k) {
        while(this.st.size() < k)
            wait();

        List res = new ArrayList();

        while(res.size() < k)
          l.add(this.st.pop());
    }
}
```

Esercizio 3

- Si definiscano in Java due tipi di thread:
  - Il tipo Conta, che stampa interi da 1 a 10, aspettando 0.5 sec ad ogni passo e , prima di terminare, segnala la fine del conteggio.
  - Il tipo Attende, che alla creazione si mette in attesa di un segnale da un thread, fornito alla creazione dell'oggetto, e poi termina.

- Si definisca un programma che faccia partire due tread, C di tipo Conta e A di tipo Attende, in modo che A si metta in attesa di C.

Esercizio 3 (soluzione)


```
class Conta extends Thread {
  public void run() {
      for (int i = 0; i <= 10; i++) {
        System.out.println(i);
        Thread.sleep(500);
      }
      synchronized(this) {
        notify();
      }
  }
}
```

Esercizio 3 (soluzione)


```
class Attende extends Thread {
  private Conta conta;

  public Attende(Conta conta) {
    this.conta = conta;
  }

  public void run() {
    System.out.println("Attende si mette in attesa");
    synchronized(conta) {
      conta.wait();
    }
    System.out.println("Attende non attende piu' e
   termina");
  }
}
```

Esercizio 3 (soluzione)


```
class Programma {
  public static void main(String[] args) {
    Conta conta = new Conta();
    Attende attende = new Attende(conta);
    conta.start();
    attende.start();
  }
}
```

Esercizio 4

- Si considerino due thread Java X e Y e due oggetti istanze di classi rispettivamente A e B.
- La classe A offre due metodi pubblici synchronized, ma1 e ma2; l'oggetto di classe B offre due metodi publlici synchronized mb1 e mb2.
- Si supponga che ma1 invochi mb2 e che mb1 invochi ma2.

Esercizio 4 (soluzione)

- 1) Si tratteggi il codice Java relativo alla classe A.

```
class A {
  B b;
  public setB(B b) {
    this.b = b;
  }
  synchronized public void ma1() {
    ...
    b.mb2();
  }
  synchronized public void ma2() { ... }
}
```

Esercizio 4 (soluzione)

- 1) Si tratteggi il codice Java relativo alla classe B.

```
Class B {
  A a;
  public setA(A a) {
    this.a = a;
  }
  synchronized public void mb1() {
    ...
    a.ma2();
  }
  synchronized public void mb2() { ... }
}
```

Esercizio 4

- 2) Quale grave situazione di errore può capitare se X chiama ma1 sull'oggetto di classe A e Y chiama mb1 sull'oggetto di classe B?
- Mostrare con un esempio l'insorgere della situazione di errore.

Esercizio 4 (soluzine)

- Supponiamo che vengano creati due oggetti x : A e y : B. Supponiamo anche che siano state eseguite le istruzioni x.setB(y) e y.setA(x).
- Supponiamo che il thread X invochi il metodo x.ma1(), e che l'esecuzione del thread si fermi prima dell'invocazione di b.mb2(). X ha acquisito il lock su x.
- Supponiamo che l'esecuzione passi al thread Y che prova ad invocare mb1(). Y acquisisce il lock su y ma, quando prova ad eseguire ma2() su x, non riesce perchè il lock è acquisito da X.
- X, comunque, non riesce ad acquisire il lock su y:
- Si ha deadlock

Esercizio 4

- Per eliminare la situazione di deadlock si potrebbe ricorrere alla seguente soluzione, che viene solo tratteggiata sommariamente.
- Eliminare il modificatore synchronized dai metodi di A e B;
- Introdurre un oggetto di classe C che funge da monitor ed esporta due metodi, mc1 ed mc2, che si incaricano di chiamare rispettivamente ma1 e mb1.
- Descrivere questa soluzione in maniera sufficiententemente precisa, evidenziando il perchè non si verifica deadlock.

Esercizio 4 (soluzione)

- Si supponga di aver eliminato synchronized dalle signature dei metodi di A e B
- Classe C:

```
class C {
  A a; B b;

  public C(A a, B b) {
    this.a = a; this.b = b;
  }

  synchronized void mc1() { a.ma1(); }
  synchronized void mc2() { b.mb1(); }
}
```

Esercizio 4 (soluzione)

- Perchè si risolve il deadlock:
  - Se X è il primo thread ad essere eseguito, Y non può invocare mc2() fino a quando X ha rilasciato il lock; X può quindi invocare indirettamente prima ma1() e poi mb2() senza che Y possa invocare mb1().
  - Stesso discorso vale viceversa se l'esecuzione parte da Y.

Esercizio 5

- Una agenzia gestisce N cartelloni pubblicitari.
- I clienti possono effettuare delle richieste di affissione specificando la durata dell'affissione in giorni e il cartellone su cui effettuarla.
- Se il cartellone è libero la richiesta viene immediatamente evasa, altrimenti verrà evasa non appena possibile.

Esercizio 5

- Modellizzare gli N cartelloni come una risorsa condivisa in Java, con le seguenti funzionalità:
- Deve essere possibile specificare il numero di cartelloni N alla creazione;
- Deve essere disponibile un metodo request() che consenta di specificare il codice cliente, il cartellone desiderato e la durata dell'affissione;
- Deve essere disponibile un metodo update() che aggiorna lo stato corrente dei cartelloni e rimuove le richieste scadute.
- Si supponga che il metodo update venga chiamato dal gestore del sistema una sola volta alla fine della giornata.

Esercizio 5 (soluzione)


```
public class Boards {
  ArrayList<Boolean> free;
  ArrayList<String> id;
  ArrayList<Integer> days;

  public Boards(int nboards) {
    /* construct fields */
    for (int i = 0; i < nboards; i++) {
      free.add(true);
      days.add(0);
      id.add("");
    }
  }
}
```

Esercizio 5 (soluzione)


```
…
  synchronized void request (int boardnum,
    String id, int days) {
    while(!free.get(boardnum)) {
      try { wait(); } catch(InterruptedException e) {}
    }

    this.free.set(boardnum, false);
    this.id.set(boardnum, id);
    this.days.set(boardnum, days);
  }
```

Esercizio 5 (soluzione)


```
synchronized void update() {
   boolean freed = false;
   for (int i = 0; i < free.size(); i++) {
     if (!free.get(i)) {
       days.set(i, days.get(i) - 1);
       if (days.get(i) == 0) {
         free.set(i, true);
         freed = true;
       }
     }
   }
   if (freed)
     notifyAll();
}
```

Esercizio 5

- Rispondere alla seguente domanda:
- Le richieste non evase immediatamente vengono gestite nell'ordine in cui sono ricevute?
- In caso affermativo giustificare la risposta; in caso negativo, spiegare come andrebbe modificata la soluzione.

Esercizio 5 (soluzione)

- Come in altri casi, la soluzione non garantisce che le richieste vengano evase nell'ordine di arrivo.
- Per ottenere tale risultato, si può usare una coda che tenga traccia dell'ordine delle richieste;
- Ad ogni risveglio, il thread risvegliato controlla di essere il primo nella coda, e se si, potrà procedere all'utilizzo del cartellone.