

Programmazione concorrente & Thread in Java (2)

Corso di Laurea in Informatica, Programmazione Distributa
Delfina Malandrino, dmalandrino@unisa.it
http://www.unisa.it/docenti/delfinamalandrino

1

Organizzazione della lezione

2

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread:
  - I problemi Deadlock
  - Altri problemi
- Conclusioni

2

Organizzazione della lezione

3

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread:
  - I problemi Deadlock
  - Altri problemi
- Conclusioni

3

Un esempio

4

- 5 amici che vogliono dipingere una casa con 5 stanze
- Se le 5 stanze sono uguali in dimensione (ed anche gli amici sono ugualmente capaci e produttivi!) allora finiscono in 1/5 del tempo che ci avrebbe impiegato una sola persona
  - lo speedup ottenuto è 5, pari al numero di amici
- Se 1 stanza è grande il doppio, però, il risultato è diverso
- Il tempo per fare la stanza grande "domina" il tempo delle altre
  - (naturalmente non consideriamo la complicazione di aiutare il poveretto cui è toccata la stanza grande, per l'overhead del coordinamento necessario)

4

La legge di Amdahl

5

- Lo *speedup* S di un programma X è il rapporto tra il tempo impiegato da un processore per eseguire X rispetto al tempo impiegato da *n* processori per eseguire X
- Sia *p* la parte del programma X che è possibile parallelizzare
  - con *n* processori la parte parallela prende tempo p/n mentre la parte sequenziale prende tempo (1 − p)

Legge di Amdahl

Lo speedup che si ottiene eseguendo il programma X su n processori, dove p è la parte di X che si può parallelizzare è:

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

5

La legge di Amdahl

6

Legge di Amdahl

Lo speedup che si ottiene eseguendo il programma X su n processori, dove p è la parte di X che si può parallelizzare è:

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

- La legge di Amdahl viene usata per predire l'aumento massimo teorico di velocità che si ottiene usando più processori

6

La legge di Amdahl

7

- I 5 amici pittori: se le stanze sono 5, di cui 4 valgono 1, mentre 1 stanza è grande il doppio (2*1=2), allora lo speedup che si ottiene è:

$$S = \frac{1}{1 - p + \frac{p}{n}} = \frac{1}{1 - 5/6 + 1/6} = 3$$

- Se gli amici e le stanze fossero 10 e 1 stanza è grande il doppio delle altre si avrebbe lo speedup:

$$S = \frac{1}{1 - p + \frac{p}{n}} = \frac{1}{1 - 10/11 + 1/11} = 5.5$$

7

L'utilizzo di una macchina multiprocessore

8

- La legge di Amdahl ci dice che la parte sequenziale del programma rallenta significativamente qualsiasi speedup che possiamo pensare di ottenere
- Quindi, per velocizzare un programma non basta investire sull'hardware (più processori, più veloci, ..) ma è assolutamente necessario e molto più cost-effective impegnarsi a **rendere la parte parallela predominante rispetto alla parte sequenziale**
  - (fortunatamente per noi informatici!)

8

Organizzazione della lezione

9

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread:
  - i problemi Deadlock
  - Altri problemi
- Conclusioni

9

Comunicazione fra thread…

10

- . . . tipicamente condividendo accesso a:
  - campi (tipi primitivi)
  - campi che contengono riferimenti a oggetti
- Comunicazione molto efficiente
  - rispetto all'usare la rete
- Possibili due tipi di errori:
  1. interferenza di thread
  2. inconsistenza della memoria
- Per risolvere questi problemi, necessaria la sincronizzazione
  - che a sua volta genera problemi di contesa: quando più thread cercano di accedere alla stessa risorsa simultaneamente (deadlock e livelock)

Cosa abbiamo visto nella lezione precedente

10

Avendo visto gli errori…

11

- Che strumenti abbiamo a disposizione?
- Il linguaggio Java ci offre strumenti efficaci a prevenire gli errori di:
  - interferenza di thread
  - inconsistenza di memoria
- Questi strumenti mostrano un trade-off tra:
  - facilità di uso: quanto si deve comprendere l'uso dei dati condivisi da parte del programma specifico
  - efficienza: quanto gli strumenti rendono colli di bottiglia sequenziali i programmi concorrenti
- Non si riesce ad ottenere entrambi gli obiettivi e sono necessari compromessi

11

Organizzazione della lezione

12

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread:
  - i problemi Deadlock
  - Altri problemi
- Conclusioni

12

Un idioma per prevenire errori

13

- I metodi sincronizzati (synchronized) sono un costrutto del linguaggio Java, che permette di risolvere semplicemente gli errori di concorrenza
  - al costo di inefficienza
- Per rendere un metodo sincronizzato, basta aggiungere **synchronized** alla sua dichiarazione:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

13

Cosa comporta un metodo sincronizzato?

14

- Non è possibile che due esecuzioni dello stesso metodo sullo stesso oggetto siano interfogliate
- Quando un thread esegue un metodo sincronizzato per un oggetto, gli altri thread che invocano metodi sincronizzati dello stesso oggetto sono sospesi fino a quando il primo thread non ha finito
- Quando un thread esce da un metodo sincronizzato, allora si stabilisce una relazione **happens-before** con tutte le successive invocazioni dello stesso metodo sullo stesso oggetto
  - **i cambi allo stato, effettuati dal thread appena uscito sono visibili a tutti i thread**
- I costruttori non possono essere sincronizzati (solo il thread che crea dovrebbe avere accesso all'oggetto in costruzione)

14

Leaking (Escape) del riferimento

18

- Quando si costruisce un oggetto che sarà condiviso, non si deve far "scappare" il riferimento prima che questo si riferisca ad un oggetto completamente costruito
- Ad esempio, se esiste una lista che mantiene tutte le istanze di un oggetto di una classe
- Se nel costruttore si inserisce instances.add(this)
- Ma altri thread possono usare instances e accedere all'oggetto prima che sia completamente costruito

15

Organizzazione della lezione

16

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread:
  - i problemi Deadlock
  - Altri problemi
- Conclusioni

16

Lock intrinseci

17

- Un lock intrinseco (o monitor lock) è una entità associata ad ogni oggetto
- Un lock intrinseco garantisce sia accesso esclusivo sia accesso consistente (relazione happens-before)
- Un thread deve
  - acquisire il lock di un oggetto
  - rilasciarlo quando ha terminato
- Quando il lock che possedeva viene rilasciato, viene stabilita la relazione happens-before
- Quando un thread esegue un metodo sincronizzato di un oggetto ne acquisisce il lock, e lo rilascia al termine (anche se c'è una eccezione)

17

Synchronized statements

21

- Specificando di quale oggetto si usa il lock:

```
    public void addName(String name)
    {  synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- In questa maniera, si sincronizzano gli accessi solo durante la modifica, ma poi si provvede in maniera concorrente all'inserimento in lista

18

Sincronizzazione a grana fine

22

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        //molto codice
        synchronized(lock1) {
              c1++;
        }
        //molto codice
    }

    public void inc2() {
        //molto codice
        synchronized(lock2) {
              c2++;
        }
        //molto codice
    }
}
```

Due variabili

19

Sincronizzazione a grana fine

23

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        //molto codice
        synchronized(lock1) {
              c1++;
        }
        //molto codice
    }

    public void inc2() {
        //molto codice
        synchronized(lock2) {
              c2++;
        }
        //molto codice
    }
}
```

Due variabili

Dichiarazione di due lock

20

Sincronizzazione a grana fine

24

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        //molto codice
        synchronized(lock1) {
              c1++;
        }
        //molto codice
    }

    public void inc2() {
        //molto codice
        synchronized(lock2) {
              c2++;
        }
        //molto codice
    }
}
```

Due variabili

Dichiarazione di due lock

Accesso a `c1` con il `lock1`

21

Sincronizzazione a grana fine

25

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        //molto codice
        synchronized(lock1) {
              c1++;
        }
        //molto codice
    }

    public void inc2() {
        //molto codice
        synchronized(lock2) {
              c2++;
        }
        //molto codice
    }
}
```

Due variabili

Dichiarazione di due lock

Accesso a `c1` con il `lock1`

Accesso a `c2` con il `lock2`

22

Sincronizzazione a grana fine

26

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        //molto codice
        synchronized(lock1) {
              c1++;
        }
        //molto codice
    }

    public void inc2() {
        //molto codice
        synchronized(lock2) {
              c2++;
        }
        //molto codice
    }
}
```

Due variabili

Dichiarazione di due lock

Accesso a `c1` con il `lock1`

Accesso a `c2` con il `lock2`

Con `synchronized` sul metodo si sequenzializza tutto (Amdahl!)

23

Sincronizzazione a grana fine

27

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        //molto codice
        synchronized(lock1) {
              c1++;
        }
        //molto codice
    }

    public void inc2() {
        //molto codice
        synchronized(lock2) {
              c2++;
        }
        //molto codice
    }
}
```

Due variabili

Dichiarazione di due lock

Accesso a `c1` con il `lock1`

Accesso a `c2` con il `lock2`

Con `synchronized` sul metodo si sequenzializza tutto (Amdahl!)

Con `synchronized` su `this` non sarebbero indipendenti!

24

Sincronizzazione di metodi statici

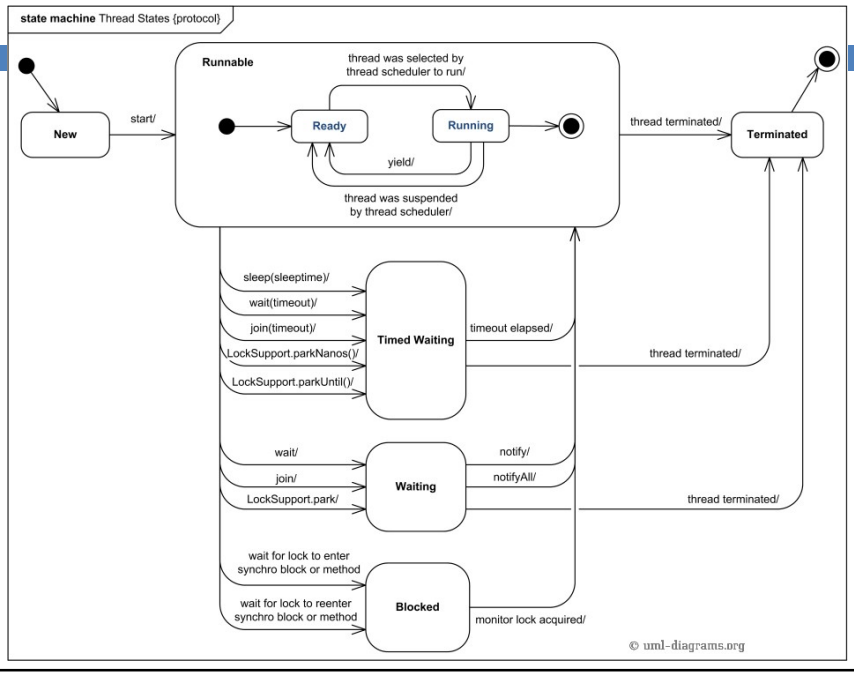
- I metodi statici si riferiscono alla classe
- Un metodo statico synchronized previene l'esecuzione interfogliata di tutti gli altri metodi statici
- In pratica, si acquisisce il lock dell'oggetto **`ClassName.class`**

```
public static void foo() {
    synchronized(ClassName.class) {
        //Body
    }
}
```

Attenzione!

Metodi sincronizzati statici garantiscono accesso in mutua esclusione a metodi sincronizzati statici, mentre metodi sincronizzati di istanza garantiscono accesso in mutua esclusione ai metodi sincronizzati di quella istanza

25



26

Organizzazione della lezione

27

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread:
  - i problemi Deadlock
  - Altri problemi
- Conclusioni

27

Azioni atomiche

28

- Azioni che non sono interrompibili e si completano (del tutto) oppure per niente
- Si possono specificare azioni atomiche in Java per:
  - read e write su variabili di riferimento e su tipi primitivi (a parte long e double)
  - read e write su tutte le variabili volatile
- Write a variabili volatile stabiliscono una relazione happens-before con le letture successive
- Tipi di dato definiti in java.util.concurrent.atomic

28

Un semplice esempio

29

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Package

29

Un semplice esempio

30

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
}
```

Package

Classe

30

Un semplice esempio

31

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Package

Classe

Variabile istanza

31

Un semplice esempio

32

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Package Classe

Variabile istanza

Metodo non sincronizzato

32

Un semplice esempio

33

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Package Classe

Variabile istanza

Metodo non sincronizzato

Uso di metodi atomici

33

Un semplice esempio

34

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Package Classe

Variabile istanza

Metodo non sincronizzato

Uso di metodi atomici

Uso di metodi atomici

34

Un semplice esempio

35

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Package Classe

Variabile istanza

Metodo non sincronizzato

Uso di metodi atomici

Uso di metodi atomici

Lettura

35

Organizzazione della lezione

36

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread: i problemi
  - Deadlock
  - Altri problemi
- Conclusioni

36

Organizzazione della lezione

37

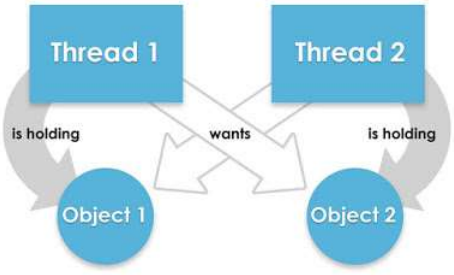
- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread: i problemi
  - Deadlock
  - Altri problemi
- Conclusioni

37

Cosa è un deadlock?

38

- Quando due thread sono bloccati, ognuno in attesa dell'altro
- Ad esempio
  - Il thread 1 ha il lock di una risorsa X (Object1) e cerca di ottenere il lock di Y (Object 2) . . .
  - . . . mentre un thread Thread 2 ha il lock della risorsa Y e cerca di ottenere il lock di X
- In questa maniera, il nostro programma concorrente si blocca e non c'è maniera di sbloccarlo



38

ALPHONSE E GASTON

(COMICS DI INIZIO '900)

39

- Alfonso e Gastone sono due amici molto cortesi
- Rigida regola di cortesia: quando ti inchini ad un amico, devi rimanere inchinato finché il tuo amico non ha una possibilità di restituire l'inchino
- Ma se i metodi bow (inchino) vengono invocati assieme…
- Entrambi i thread si bloccheranno invocando bowBack
  - Nessuno dei due blocchi avrà fine, perchè ogni thread aspetta l'altro

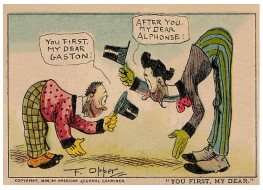
39

Gli inchini di ALPHONSE E GASTON -1

40

```
public class Deadlock {
 static class Friend {
 private final String name;
 public Friend(String name) {
         this.name = name;
  }
  public String getName() {
      return this.name;
  }

  public synchronized void bow(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed to me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
  }

  public synchronized void bowBack(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed back to me!%n",
          this.name, bower.getName());
  }
 }

 //...
```

Classe interna



40

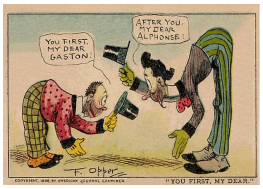
Gli inchini di ALPHONSE E GASTON -1

41

```
public class Deadlock {
 static class Friend {
 private final String name;
 public Friend(String name) {
        this.name = name;
  }
  public String getName() {
      return this.name;
  }

  public synchronized void bow(Friend bower)
      {  System.out.format("%s:%s"
          +"has bowed to me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
  }

  public synchronized void bowBack(Friend bower)
      {  System.out.format("%s:%s"
          +"has bowed back to me!%n",
          this.name, bower.getName());
  }
 }

 //...
```

Classe interna

Campo



41

Gli inchini di ALPHONSE E GASTON -1

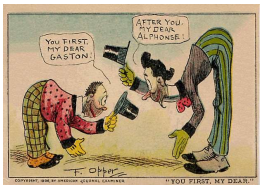
42

```
public class Deadlock {
 static class Friend {
 private final String name;
 public Friend(String name) {
        this.name = name;
  }
  public String getName() {
      return this.name;
  }

  public synchronized void bow(Friend bower)
      {  System.out.format("%s:%s"
          +"has bowed to me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
  }

  public synchronized void bowBack(Friend bower)
      {  System.out.format("%s:%s"
          +"has bowed back to me!%n",
          this.name, bower.getName());
  }
 }

 //...
//...
```

Classe interna

Campo

Costruttore



42

Gli inchini di ALPHONSE E GASTON -1

43

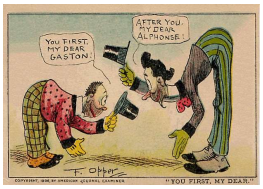
```
public class Deadlock {
 static class Friend {
 private final String name;
 public Friend(String name) {
        this.name = name;
  }
  public String getName() {
      return this.name;
  }

  public synchronized void bow(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed to me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
  }

  public synchronized void bowBack(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed back to me!%n",
          this.name, bower.getName());
  }
 }

 //...
```

Classe interna

Campo

Costruttore

"Inchino" …



43

Gli inchini di ALPHONSE E GASTON -1

44

```
public class Deadlock {
 static class Friend {
 private final String name;
 public Friend(String name) {
        this.name = name;
  }
  public String getName() {
      return this.name;
  }

  public synchronized void bow(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed to me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
  }

  public synchronized void bowBack(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed back to me!%n",
          this.name, bower.getName());
  }
 }

 //...
```

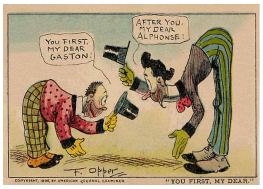
Classe interna

Campo

Costruttore

"Inchino" …

…con risposta



44

Gli inchini di ALPHONSE E GASTON -1

45

```
public class Deadlock {
 static class Friend {
 private final String name;
 public Friend(String name) {
         this.name = name;
  }
  public String getName() {
      return this.name;
  }

  public synchronized void bow(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed to me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
  }

  public synchronized void bowBack(Friend bower) {
      System.out.format("%s:%s"
          +"has bowed back to me!%n",
          this.name, bower.getName());
  }
 }

 //...
```

Classe interna

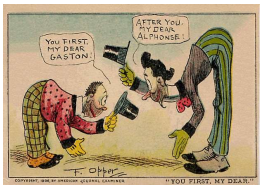
Campo

Costruttore

"Inchino" …

…con risposta

La risposta



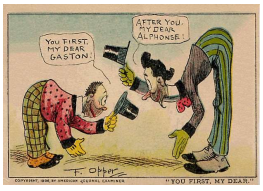
45

Gli inchini di ALPHONSE E GASTON -1

46

```
 //...

 public static void main(String[] args) {

   final Friend alphonse = new Friend("Alphonse");
   final Friend gaston = new Friend("Gaston");

   new Thread(new Runnable() {
       public void run() { alphonse.bow(gaston); }
   }).start();

   new Thread(new Runnable() {
       public void run() { gaston.bow(alphonse); }
   }).start();

  }//end main

  }//end class
```

Si creano Alphonse e Gaston



46

Gli inchini di ALPHONSE E GASTON -1

47

```
//...

public static void main(String[] args) {

  final Friend alphonse = new Friend("Alphonse");
  final Friend gaston = new Friend("Gaston");

  new Thread(new Runnable() {
      public void run() { alphonse.bow(gaston); }
  }).start();

  new Thread(new Runnable() {
      public void run() { gaston.bow(alphonse); }
  }).start();

 }//endmain

  }//endclass
```

Si creano Alphonse e Gaston

Classe anonima di tipo `Runnable` passata al costruttore di `Thread`

47

Gli inchini di ALPHONSE E GASTON -1

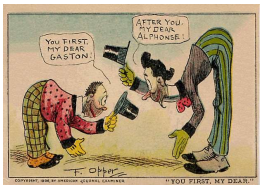
48

```
//...

public static void main(String[] args) {

  final Friend alphonse = new Friend("Alphonse");
  final Friend gaston = new Friend("Gaston");

  new Thread(new Runnable() {
      public void run() { alphonse.bow(gaston); }
  }).start();

  new Thread(new Runnable() {
      public void run() { gaston.bow(alphonse); }
  }).start();

 }//endmain

  }//endclass
```

Si creano Alphonse e Gaston

Classe anonima di tipo `Runnable` passata al costruttore di `Thread`

con il metodo da eseguire …



48

Gli inchini di ALPHONSE E GASTON -1

49

```
//...

public static void main(String[] args) {

  final Friend alphonse = new Friend("Alphonse");
  final Friend gaston = new Friend("Gaston");

  new Thread(new Runnable() {
      public void run() { alphonse.bow(gaston); }
  }).start();

  new Thread(new Runnable() {
      public void run() { gaston.bow(alphonse); }
  }).start();

 }//endmain

 }//endclass
```

L'output:

Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed to me!
Nessuno bows back!

Si creano Alphonse e Gaston

Classe anonima di tipo `Runnable` passata al costruttore di `Thread`

con il metodo da eseguire …

…e si lancia

49

Gli inchini di ALPHONSE E GASTON -1

50

Alphonse's thread

```
A: alphonse.bow(gaston) - acquires alphonse's lock
A: gaston.bowBack(alphonse) - acquires gaston's lock
A: both methods return, thus releasing both locks
```

Gaston's thread

```
G: gaston.bow(alphonse) - acquires gaston's lock
G: alphonse.bowBack(gaston) - acquires alphonse's lock
G: both methods return, thus releasing both locks
```

50

Gli inchini di ALPHONSE E GASTON -1

51

Alphonse's thread

```
A: alphonse.bow(gaston) - acquires alphonse's lock
A: gaston.bowBack(alphonse) - acquires gaston's lock
A: both methods return, thus releasing both locks
```

Gaston's thread

```
G: gaston.bow(alphonse) - acquires gaston's lock
G: alphonse.bowBack(gaston) - acquires alphonse's lock
G: both methods return, thus releasing both locks
```

Possibile deadlock

```
A: alphonse.bow(gaston) - acquires alphonse's lock
G: gaston.bow(alphonse) - acquires gaston's lock
G: attempts to call alphonse.bowBack(gaston), but blocks waiting on alphonse's lock
A: attempts to call gaston.bowBack(alphonse), but blocks waiting on gaston's lock to
```

51

Perché il deadlock?

52

```
static class Friend {
 //...
 public void bow(Friend bower)
  {  synchronized(this) {
      System.out.format("%s:%s has bowed to
          me!%n",
        this.name, bower.getName());
       bower.bowBack(this);
  }
 }
 public void bowBack(Friend bower)
  {  synchronized(this) {
  System.out.format("%s:%s has bowed back to
         me!%n",
          this.name, bower.getName());
  }
 }
}
```

Riscriviamo i metodi `synchronized` in questa maniera

52

Perché il deadlock?

53

```
static class Friend {
 //...
 public void bow(Friend bower)
  {  synchronized(this) {
      System.out.format("%s:%s has bowed to
           me!%n",
        this.name, bower.getName());
       bower.bowBack(this);
  }
 }
 public void bowBack(Friend bower)
  {  synchronized(this) {
  System.out.format("%s:%s has bowed back to
          me!%n",
           this.name, bower.getName());
  }
 }
}
```

Riscriviamo i metodi `synchronized` in questa maniera

Lock esplicito sul lock dell'oggetto

53

Perché il deadlock?

54

```
static class Friend {
 //...
 public void bow(Friend bower)
  {  synchronized(this) {
      System.out.format("%s:%s has bowed to
           me!%n",
        this.name, bower.getName());
       bower.bowBack(this);
  }
 }
 public void bowBack(Friend bower)
  {  synchronized(this) {
  System.out.format("%s:%s has bowed back to
          me!%n",
           this.name, bower.getName());
  }
 }
}
```

Riscriviamo i metodi `synchronized` in questa maniera

Lock esplicito sul lock dell'oggetto

…idem

54

Perché il deadlock?

55

```
static class Friend {
 //...
 public void bow(Friend bower)
  {  synchronized(this) {
      System.out.format("%s:%s has bowed to
           me!%n",
        this.name, bower.getName());
       bower.bowBack(this);
  }
 }
 public void bowBack(Friend bower)
  {  synchronized(this) {
   System.out.format("%s:%s has bowed back to
          me!%n",
           this.name, bower.getName());
  }
 }
}
```

Riscriviamo i metodi `synchronized` in questa maniera

Lock esplicito sul lock dell'oggetto

…idem

A questo punto Alphonse acquisisce il suo lock e cerca di acquisire quello di Gaston

55

Perché il deadlock?

56

```
static class Friend {
 //...
 public void bow(Friend bower) {
  synchronized(this) {
      System.out.format("%s:%shasbowedto
           me!%n",
        this.name, bower.getName());
       bower.bowBack(this);
  }
 }
 public void bowBack(Friend bower) {
  synchronized(this) {
    System.out.format("%s:%s has bowed back to
           me!%n",
             this.name, bower.getName());
  }
 }
}
```

Domanda

Funziona?

Riscriviamo i metodi `synchronized` in questa maniera

Lock esplicito sul lock dell'oggetto

…idem

A questo punto Alphonse acquisisce il suo lock e cerca di acquisire quello di Gaston

. . . che fa lo stesso: prima il suo e poi quello di Alphonse

56

Organizzazione della lezione

57

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread: i problemi
  - Deadlock
  - Altri problemi
- Conclusioni

57

Starvation

58

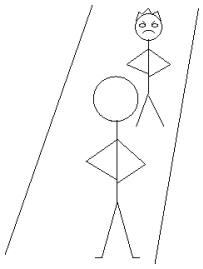
- Quando un thread non riesce a acquisire accesso ad una risorsa condivisa . . .
- . . . in maniera da non riuscire a fare progresso
  - risorsa è indisponibile per thread "ingordi"
- Esempio: un metodo sincronizzato che impiega molto tempo
  - se invocato spesso, altri thread possono essere prevenuti dall'accesso
- Arbitrarietà dello scheduler
  - Attenzione: priorità dei thread nella JVM dipendente dal mapping effettuato sui thread del S.O.!
  - priorità 3 e 4 in JVM possono essere mappate su stessa priorità del S.O.

58

Livelock

59

- Un thread A può reagire ad azioni di un altro thread B. . .
- . . . che reagisce con una risposta verso A
- I due thread non sono bloccati (non è un deadlock!) ma sono occupati a rispondere alle azioni dell'altro
- Anche se sono in esecuzione, non c'è progresso!
- Un esempio: due persone che si incontrano in un corridoio stretto, sullo stesso lato
  - attitudine belligerante: aspettare che l'altro si sposti
  - attitudine garbata: spostarsi di lato
- 2 belligeranti: deadlock!
- 2 garbati: livelock!



59

Conclusioni

60

- La legge di Amdahl
- Sincronizzazione di thread: gli strumenti
  - Metodi sincronizzati
  - Lock intrinseci
  - Accesso atomico
- Sincronizzazione di thread:
  - i problemi Deadlock
  - Altri problemi
- Conclusioni

Nelle prossime lezioni:

Alcuni esempi sui thread

60