Corso di Laurea in Informatica
I Anno Magistrale, Indirizzo Cloud Computing
Reti Geografiche: Struttura, Analisi e Prestazioni

Programmazione concorrente & Thread in Java (2)

Delfina Malandrino
dmalandrino@unisa.it
http://www.unisa.it/docenti/delfinamalandrino

1

Organizzazione della lezione

- 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

- 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

- 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

- 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

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

L'utilizzo di una macchina multiprocessore

- 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

- 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…

- . . . 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:
  - interferenza di thread
  - 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

Organizzazione della lezione

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

11

Un idioma per prevenire errori

- 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;
    }
}
```

12

Cosa comporta un metodo sincronizzato?

- 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)

13

Organizzazione della lezione

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

14

14

Lock intrinseci

- 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)

15

Synchronized statements

- 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

16

Sincronizzazione a grana fine

```
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

17

Sincronizzazione a grana fine

```
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

18

Sincronizzazione a grana fine

```
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

19

Sincronizzazione a grana fine

```
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

20

Sincronizzazione a grana fine

```
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!)

21

Organizzazione della lezione

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

22

Azioni atomiche

- 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

23

Un semplice esempio

```
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

24

Un semplice esempio

```
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

25

Un semplice esempio

```
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

26

Un semplice esempio

```
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

27

Un semplice esempio

```
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

28

Un semplice esempio

```
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

29

Un semplice esempio

```
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

30

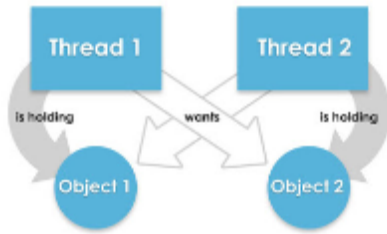
Organizzazione della lezione

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

31

Cosa è un deadlock?

- 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



32

Organizzazione della lezione

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

51

Starvation

- 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.

52

Livelock

- 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!

53

Corso di Laurea in Informatica
I Anno Magistrale, Indirizzo Cloud Computing
Reti Geografiche: Struttura, Analisi e Prestazioni

Programmazione concorrente & Thread in Java (2)

Delfina Malandrino
dmalandrino@unisa.it
http://www.unisa.it/docenti/delfinamalandrino

54