Esercitazione di Sistemi Distribuiti e Pervasivi

Gestione della concorrenza nelle applicazioni di rete

Gabriele Civitarese
gabriele.civitarese@unimi.it

EveryWare Lab
Università degli Studi di Milano
-
Docente: Claudio Bettini

Slide realizzate a partire da versioni precedenti di Letizia Bertolaja, Sergio Mascetti, Dario Freni e Claudio Bettini

Copyright

Alcune slide di questo corso sono in parte personalizzate da quelle fornite dall'autore del testo di riferimento e distribuite dalla casa editrice (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007). Tutte le altre slide sono invece di proprietá del docente. Tutte le slide sono soggette a diritto d'autore e quindi non possono essere ri-distribuite senza consenso. Lo stesso vale per eventuali registrazioni o videoregistrazioni delle lezioni.

Some slides for this course are partly adapted from the ones distributed by the publisher of the reference book for this course (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007). All the other slides are from the teacher of this course. All the material is subject to copyright and cannot be redistributed without consent of the copyright holder. The same holds for audio and video-recordings of the classes of this course.

Programmazione concorrente

Cos'è un programma *concorrente*?

- Un insieme di istruzioni che *potrebbero* essere eseguite simultaneamente
- Da non confondere con un sistema distribuito, che prevede diversi processi che lavorano in parallelo comunicando con un protocollo
  - Un programma concorrente potrebbe persino girare su un singolo processore (pseudo-parallelismo)
- Noi avremo a che fare con programmi Java con più thread che lavorano in contemporanea
- Non-determinismo sull'ordine con cui le istruzioni vengono eseguite
  - Dipende dallo scheduling dei thread!

Problematiche

- I thread condividono lo spazio di memoria del processo a cui appartengono
- Scambio di informazioni molto efficiente ma soggetto ad alcuni problemi
  - *Thread Interference*
  - *Memory Inconsistency*
- La sincronizzazione serve ad evitare questi errori

Esempio thread interference

```
class Counter {
  private int c = 0;

  public void increment() {
    int newValue = c + 1;
    c = newValue;
  }

  public void decrement() {
    int newValue = c - 1;
    c = newValue;
  }

  public int value() {
    return c;
  }

}
```

- Due thread A e B eseguono contemporaneamente operazioni su un oggetto `Counter`
- A invoca `increment()` e B invoca `decrement()`
- **L'ordine in cui vengono effettuate le operazioni non è deterministico**
- Cosa può succedere?

Esempio thread interference (2)

```
class Counter {
  private int c = 0;

  public void increment() {
    int newValue = c + 1;
    c = newValue;
  }

  public void decrement() {
    int newValue = c - 1;
    c = newValue;
  }

  public int value() {
    return c;
  }

}
```

- c ha valore 0
- Possibile ordine di esecuzione:

1. *A*: `newValue = c+1 = 1;`
2. *B*: `newValue = c-1 = -1;`
3. *A*: `c = newValue = 1;`
4. *B*: `c = newValue = -1;`

- Il risultato dell'operazione di A è andato perso (come se non avesse eseguito).

Esempio memory inconsistency

```
class Store {
  private int c = 10;
  //return true if can sell
  public boolean sell() {
    if(c>0){
      c--;
      return true;
    }
    return false;
  }
}
```

- dopo un po' di chiamate, c ha valore 1
- Ordine di esecuzione:

1. *A*: `if (c>0)`
2. *B*: `if (c>0)`
3. *A*: `c--; return true;`
4. *B*: `c--; return true;`

- Abbiamo venduto due volte l'ultimo oggetto rimasto!

Accesso sincronizzato

- Ogni oggetto ha un *intrinsic lock* associato, detto anche *monitor*
- Se un metodo viene dichiarato `synchronized`, prima dell'esecuzione del metodo viene ottenuto l'intrinsic lock.
- L'uso di metodi `synchronized` garantisce che solo un thread per volta può accedere all'oggetto.
- Esempio di dichiarazione:
  - `public synchronized int methodName(int param)`

Chiamata di un metodo sincronizzato

Quando un thread esegue un metodo sincronizzato su un oggetto `obj`:

- Il valore dell'intrinsic lock associato ad `obj` viene controllato:
- Se "disponibile":
  - Valore dell'intrinsic lock impostato a "non disponibile"
  - Metodo eseguito
  - Quando il metodo termina, intrinsic lock impostato a "disponibile"
- Se "non disponibile":
  - Attesa fino a quando l'intrinsic lock non diventa disponibile.

Esempio di sincronizzazione

- Consideriamo una classe `MyCollection` che mantenga un insieme (collezione) di oggetti
- Un thread vuole ordinare l'insieme
- Un altro thread vuole ottenere l'elemento più piccolo dell'insieme.
- Perché è richiesta una sincronizzazione?
- Come è possibile effettuarla?

Esempio di sincronizzazione (2)

```
class MyCollection {
  synchronized void sortItems() {
    //Sorting implementation
  }
  synchronized Object getSmallest() {
    //Complicated code to get the minimum
  }
}
```

- Nota: gli attributi non possono essere dichiarati `synchronized`
- L'accesso alle strutture dati che necessitano sincronizzazione non può avvenire direttamente ma solo tramite metodi

Correzione classe Counter

```
class Counter {
  private int c = 0;

  public synchronized void increment(){
    int newValue = c + 1;
    c = newValue;
  }

  public synchronized void decrement(){
    int newValue = c - 1;
    c = newValue;
  }

  public synchronized int value(){
    return c;
  }
}
```

- *A* invoca `increment()` e *B* invoca `decrement()`
- Unico possibile ordine di esecuzione:

1. *A*: `newValue = c+1 = 1;`
2. *A*: `c = newValue = 1;`
3. *B*: `newValue = c-1 = 0;`
4. *B*: `c = newValue = 0;`

synchronized statement

- Un altro modo di scrivere codice sincronizzato sono i `synchronized` statement
  - Es:

```
synchronized(objVar) {
  codice da eseguire;
}
```

- Il parametro da specificare è l'oggetto contenente l'intrinsic lock che si vuole utilizzare
- `objVar` tipicamente è anche l'oggetto sul quale vogliamo garantire atomicità
- Per sincronizzarsi su tipi primitivi (`int`, `float`, ...) si creano oggetti "dummy" usati solo per il lock

synchronized statement (2)

I seguenti codici si comportano alla stessa maniera. Perché?

```
public synchronized void methodA(){
  codice da eseguire;
}
```

```
public void methodB() {
  synchronized(this) {
    codice da eseguire;
  }
}
```

synchronized statement (3)

Perché dovrei usare un synchronized statement?

- Per evitare "eccessi di sincronizzazione"
- Per ottenere una sincronizzazione piú fine

Supponiamo di avere due variabili $c1$ e $c2$, campi privati della stessa classe. $c1$ e $c2$ **non** sono in alcun modo relati.

synchronized statement (3)

FineGrainedSynchronization.java

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

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

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

synchronized e metodi statici

- Anche un metodo statico può essere definito `synchronized`

```
class Foo{
  synchronized static void foo(){
    //some boring stuff
  }
}
```

- Intuitivamente, il lock acquisito non è più legato all'istanza, ma alla classe. Equivalente a:

```
class Foo{
  static void foo(){
    synchronized(Foo.class){
      //some boring stuff
    }
}
```

Deadlock

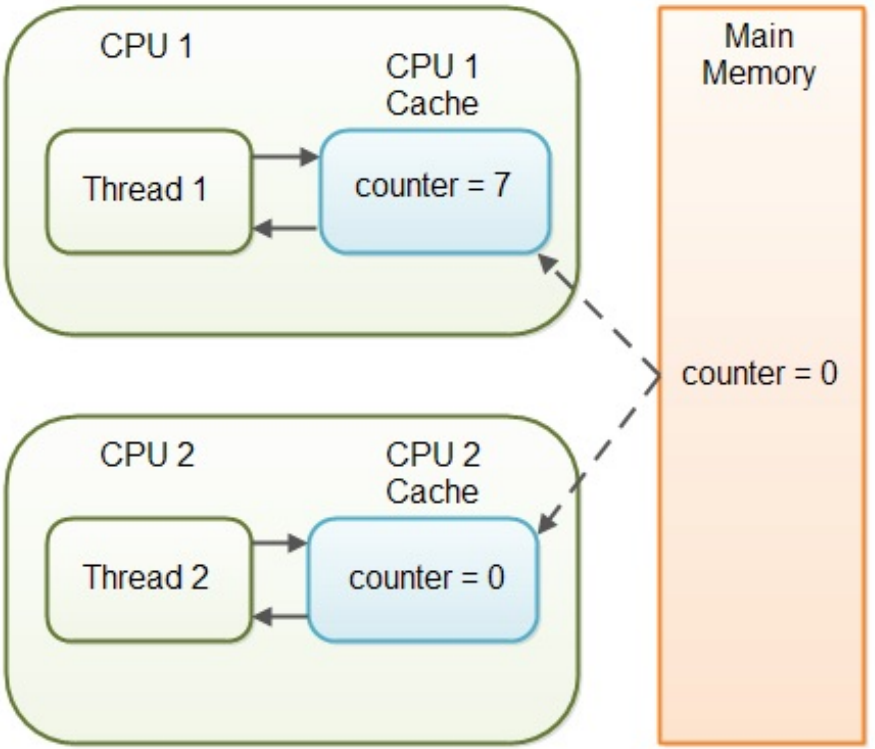
```
public class Model {
  private View myView;
  public synchronized void updateModel(Object someArg) {
    doSomething(someArg);
    myView.somethingChanged();
  }
  public synchronized Object getSomething() {
    return someMethod();
  }
}
public class View {
  private Model underlyingModel;
  public synchronized void somethingChanged() {
    doSomething();
  }
  public synchronized void updateView() {
    Object o = myModel.getSomething();
  }
}
```

Cosa può succedere se *updateModel()* e *updateView()* vengono chiamati in concorrenza da due thread?

Osservazioni

- L'intrinsic lock associato ad un oggetto `obj` viene utilizzato da tutti i metodi `synchronized` e dai `synchronized` statement che specificano come parametro `obj`
- **Non** è garantito che l'ordine di esecuzione dei thread in attesa su un lock sia uguale all'ordine in cui i thread hanno richiesto il lock
- L'uso di `synchronized` garantisce due proprietà:
  - **Mutua esclusione**: solo un thread alla volta può avere un particolare lock
  - **Visbilità**: i cambiamenti effettuati ai dati condivisi prima del rilascio del lock devono essere resi visibili ai thread che acquisiranno il lock successivamente
- **Miraccomando**: quando più thread scrivono e leggono gli stessi dati, sincronizzate sempre!

Approfondendo la visibilità



- Per ottimizzare, un thread può copiare variabili dalla memoria centrale in una cache della CPU
- In un ambiente multi-core, ogni thread può copiare le variabili in una cache diversa
- Senza sincronizzazione, non si hanno garanzie sulla freschezza del dato

Volatile

La keyword *volatile*

- Keyword da assegnare a *variabili* (e.g. `volatile int x;`)
- Versione "light" di `synchronized`
- Garantisce la **visibilità**, ma non la **mutua esclusione**
- I thread vedono automaticamente il valore più fresco di variabili `volatile`
  - Atomicità garantita solo per lettura e scrittura
- Si decide di usarlo solo per semplicità e scalabilità, ma va gestito con molta attenzione
- Su una variabile `volatile` si scrivono solo valori indipendenti da un qualsiasi altro stato del programma, inclusa la variabile stessa
  - `x++` su una variabile `volatile` non è thread-safe! Perchè?

volatile: Esempio

```
class Worker {
  volatile boolean shutdownRequested;

  public void shutdown() { shutdownRequested = true; }

  public void doWork() {
    while (!shutdownRequested) {
    // do some random stuff
    }
  }
}
```

- Un thread esegue in loop `doWork()`, un altro si occupa di interrompere il lavoro con `shutdown()`
- `volatile` assicura che i due thread abbiano la stessa vista, semplificando rispetto all'uso di `synchronized`

Esercizio

- Servizio di prenotazione biglietti per il teatro.
- Semplificazioni: un solo spettacolo, una sola fascia di prezzi.
- Da realizzarsi con un server concorrente.
- Problema: non vendere più biglietti di quelli che sono disponibili.
- Si crei, oltre alle classi già illustrate per la comunicazione multithread, una classe "Prenotazioni" con un metodo senza parametri che controlla se ci sono posti e:
  - In caso affermativo restituisce il numero del posto prenotato;
  - In caso negativo restituisce zero.
- Testare con l'uso di `Thread.sleep()` che i problemi di sincronizzazione siano effettivamente risolti

Segnalazione

Metodi utilizzati per coordinare l'esecuzione di più thread

- `wait()`
- `notify()`
- `join()`
- `sleep()`
- `...`

The Java Tutorials: Concurrency

Thread.join()

- Il metodo `join()` si usa per attendere la fine di un thread
  - Il thread che richiama il metodo `join()` si ferma fino a che non termina il thread associato all'oggetto su cui viene invocato
- Può essere invocato con un parametro opzionale che rappresenta il timeout (in millisecondi)
  - In questo caso, anche se il thread non termina prima del timeout, il thread richiamante riprende dall'istruzione successiva
- Lo stato di un thread può essere controllato con il metodo `isAlive()`
  - Restituisce `true` se il thread è attivo, `false` se il thread è terminato

Thread.sleep()

- Il metodo `sleep()` interrompe temporaneamente l'esecuzione del thread corrente.
  - A cosa serve?
    - Effettuare azioni periodiche
    - Consentire l'esecuzione di altri thread (ad es. per debugging)
  - È un metodo statico, si richiama usualmente direttamente dalla classe `Thread`
    - Parametro di tempo espresso in millisecondi
    - Es.: `Thread.sleep(10000) //attende 10 secondi`

`Object.wait()` e `Object.notify()`

- Metodi della classe `Object` (non di `Thread`!), per cui applicabili ad ogni oggetto
- Ad ogni oggetto è associata una lista di thread in attesa
  - Un thread che richiama una `wait()` su un determinato oggetto resta sospeso fin quando un altro thread non lo risveglia inviando una `notify()` sullo stesso oggetto
- Utilizzati per coordinare due o più thread

Osservazioni su `wait()` e `notify()`

- Più thread possono essere in attesa sullo stesso oggetto
- La chiamata `notify()` risveglia **un** solo thread
  - Attenzione: non è garantito che l'ordine con cui i thread vengono risvegliati sia lo stesso ordine con cui i thread sono entrati in attesa!
- `notifyAll()` risveglia **tutti** i thread in attesa

Utilizzo di `wait()` e `notify()`

- Creiamo una classe per consentire a due thread di aspettarsi a vicenda prima di proseguire l'esecuzione
- Perché il metodo `meetUp()` è dichiarato `synchronized`?
- Come potrebbe essere modificato questo esempio per essere applicato a più di due thread?

```
public class CheckPoint {
  boolean hereFirst = true;
  synchronized void meetUp () {
    if (hereFirst) {
      hereFirst = false;
      wait();
    }
    else {
      notify();
      hereFirst = true;
    }
  }
}
```

Rapporto tra `wait()` e intrinsic lock

- Per quanto detto fino ad ora, l'esempio precedente NON dovrebbe funzionare: perché?
  - Primo thread arriva, acquisisce il lock e poi va in wait;
  - Il secondo thread arriva e non può prendere il lock sull'oggetto;
  - Deadlock!

Rapporto tra `wait()` e intrinsic lock (2)

- Perché, invece, funziona?
  - `wait()` rilascia l'intrinsic lock associato all'oggetto su cui viene invocata;
    - il lock deve essere acquisito prima della chiamata `wait()`, altrimenti viene sollevata una eccezione
  - Quindi il secondo thread può eseguire il metodo...
    - e risvegliare il primo thread.
  - Quando il primo thread viene risvegliato, deve riacquisire i lock prima di poter eseguire:
    - Quindi deve aspettare che il secondo thread esca da metodo

Busy waiting

Classico esempio di busy waiting

```
...
while (true) {
  if (eventHappened){
    doSomething();
  }

  Thread.sleep(SOME_TIME_IN_MILLISECONDS);
}
...
```

Perchè è il male?

- La CPU è impegnata inutilmente, togliendo spazio ad altri thread per computazione utile
- Sì, anche se usiamo le *sleep*
- La soluzione è usare sistemi di segnalazione (*wait()* e *notify()*)
- Inseritelo nel vostro progetto solo se il vostro obiettivo è non passare l'esame

Busy waiting (2)

Come evitiamo il busy waiting?

```
...
while (!eventHappened) {
  wait();
  if(eventHappened)
    doSomething();
 }
...
```

- Invece di continuare ad iterare aspettando un cambiamento, il thread si mette in `wait()`
- Un altro thread si occuperà di usare `notify()` per notificarlo dell'avvenuto evento, svegliandolo

...e nei synchronized statement?

wait()/notify() e synchronized statement

```
...
synchronized(obj){
        ...
        obj.wait();
        ...

}

...

synchronized(obj){
        ...
        obj.notify();
        ...

}
```

- I metodi wait e notify sono sempre legati all'oggetto di sincronizzazione
- Pensate sempre in termini di *intrinsic lock*!

Esercizio

- Un veterinario ha una sala d'attesa che può contenere solo cani e gatti
- Un gatto non può entrare nella sala se sono già presenti un cane o un gatto
- Un cane non può entrare nella sala se è già presente un gatto
- Non ci possono essere più di 4 cani in tutto
- Gli animali restano all'interno della stanza per un periodo di tempo randomico
- Gli animali che non possono entrare attendono finchè diventa possibile
- Risolvere il problema con `synchronized`, `wait` e `notify`:
  - implementando un metodo *enterRoom* e un metodo *exitRoom*
  - generando randomicamente animali (thread) che chiamano questi metodi

La librerira `java.util.concurrent`

`java.util.concurrent`

- Java mette a disposizione librerie per facilitare la programmazione concorrente
- Si evita di reinventare la ruota con `syncrhonized`, `wait` e `notify`
- Mette a disposizione svariati strumenti:
  - Primitive di sincronizzazione ad alto livello (semafori, lock, ...)
  - Strutture dati che già supportano la concorrenza
  - Tipi di dato atomici (e.g. `AtomicInteger`)

La librerira `java.util.concurrent` (2)

Tutto molto bello!

- Tutto molto bello, la libreria facilita il lavoro permettendoci di fare cose complesse
- Ma... sarà **vietato** usarla nel progetto.
- Perchè?
  - Il docente è cattivo...
  - ...o più semplicemente lo scopo del corso è quello di capire più a "basso livello" come funziona la concorrenza in Java

Produttore - consumatore

- *Pattern* di comunicazione multi-processo
- Il produttore è l'entità (thread, nel nostro caso) che invia i dati
- Il consumatore è l'entità (thread, nel nostro caso) che riceve i dati
- I dati vengono trasmessi attraverso un *buffer* di memoria condiviso, tipicamente una coda
- È possibile considerare più produttori e/o più consumatori

Problema

Bisogna evitare che un consumatore debba controllare costantemente il buffer per controllare se ci sono dati

Esempio di implementazione della coda

```
public class Queue {
  public ArrayList<String> buffer = new ArrayList<String>();

  public synchronized void put(String message) {
    buffer.add(message);
    notify();
  }
  public synchronized String take() {
    String message = null;
    while(buffer.size() == 0) {
      try { wait(); }
      catch (InterruptedException e) { e.printStackTrace(); }
    }
    if(buffer.size() > 0) {
      message = buffer.get(0);
      buffer.remove(0);
    }
    return message;
  }
}
```

Esercizio

- Semplice servizio di *chat* via socket
- Ogni utente scrive i messaggi da inviare via tastiera
- La comunicazione è asincrona (ogni utente puó inviare messaggi indipendentemente dagli altri)
- I thread devono comunicare attraverso un buffer condiviso

Varianti

- Connessione diretta fra due utenti
- Chat-room: un server riceve connessioni da più utenti e inoltra i messaggi a tutti i partecipanti