Lezione n.11
LPR-B-09
Strutture dati e concorrenza
Situazioni di deadlock
9/12/2009
Andrea Corradini

COLLEZIONI E COLLEZIONI SINCRONIZZATE

- **JAVA Collections**: strutture dati predefinite incluse nel package java.util

  a partire da JAVA 1.2
- Alcuni esempi: HashTable, Vectors, List, ArrayList, Set, ...
- Synchronized Collections: Definiscono strutture dati thread safe, cioè garantiscono che lo stato della struttura risulti corretto anche nel caso in cui la struttura venga acceduta in modo concorrente da più threads
- Thread Safety: la struttura dati viene incapsulata e ogni metodo pubblico viene sincronizzato

Tabelle hash: la classe HashTable

- Teoria di tabelle hash vista a **Algoritmica**
- La classe **java.util.HashTable<K,V>** ne fornisce un'implementazione
- E' una classe generica: **K** è il tipo delle chiavi, **V** quello dei valori.
- Chiavi e valori possono essere oggetti Java qualunque
- Metodi principali (sincronizzati!):
  - **V get(Object key)**
  - **V put(K key, V value)**
  - **Enumeration<K> keys()**
  - **Collection<V> values()**
  - **V remove(Object key)**
- Semplice da usare, dopo averla provata!!!
- Efficiente rispetto a liste (Vector, ArrayList)

BASTANO METODI SINCRONIZZATI?

- Thread Safety: la struttura dati viene incapsulata e ogni metodo pubblico viene sincronizzato
- Se l'operazione che il client (il thread che utilizza la collezione) è composta da una serie di operazioni elementari, il client deve spesso implementare ulteriori sincronizzazioni.
- Si possono definire blocchi di codice sincronizzati che incapsulano più operazioni elementari effettuate su una collezione sincronizzata

Esempio: Necessità di ulteriore sincronizzazione

```
import java.util.*;
public class ThreadRemover extends Thread {
        Vector v;
        public ThreadRemover(Vector v) {this.v = v;}
        public void run( ) {
            int lastIndex = v.size(   ) - 1;
                Object o = v.remove(lastIndex); }
}
```

- ThreadRemover elimina da un Vector l'ultimo elemento
- Le operazioni **size()** e **remove()** sono **synchronized** in **Vector**, ma non sono sufficienti a garantire l'assenza di ***race conditions***:

Esempio: Necessità di ulteriore sincronizzazione

```
import java.util.*;
public class ThreadRemover extends Thread {
        Vector v;
        public ThreadRemover(Vector v) {this.v = v;}
        public void run( ) { synchronized(v) {
            int lastIndex = v.size(   ) - 1;
                Object o = v.remove(lastIndex); } }
}
```

- ThreadRemover elimina da un Vector l'ultimo elemento
- Le operazioni **size()** e **remove()** sono **synchronized** in **Vector**, ma non sono sufficienti a garantire l'assenza di ***race conditions***:

  devono essere eseguite in modo atomico.

CODE

- L'interfaccia **Queue<E>** definisce i metodi di una coda FIFO, che può essere di capacità limitata o illimitata
- Metodi per inserimento, rimozione e lettura del primo:

	Throws exception	Returns special value
Insert	boolean add(e)	boolean offer(e)
Remove	E remove()	E poll()
Examine	E element()	E peek()

- **add** e **offer** non restituiscono mai **false** se la capacità è illimitata
- Molto utile anche l'interfaccia **java.util.Deque<E>** (letto **deck**, per "double ended queue") che ha metodi per inserire/rimuovere/esaminare il primo o l'ultimo elemento. Può essere usata per code e pile.

CODE BLOCCANTI

- Code bloccanti: introdotte in JAVA 5 come supporto per il paradigmi computazionali di tipo produttore/consumatore
- Sono code sincronizzate. Offrono metodi aggiuntivi che bloccano invece di fallire, eventualmente con timeout. Comportamento nuovo:
  - inserimento: aggiunge un elemento in fondo alla coda, se la coda non è piena, altrimenti blocca il thread che ha invocato l'operazione.
  - rimozione: elimina il primo elemento della coda, se questo esiste, altrimenti blocca il thread che ha invocato l'operazione
- L'interfaccia **java.util.concurrent.BlockingQueue** definisce questo tipo di code
- Ci sono varie classi che la implementano

CODE BLOCCANTI

- L'interfaccia **java.util.concurrent.BlockingQueue** prevede i metodi:

	Throws exc.	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

CODE BLOCCANTI: TIPI DEFINITI

Classi che implementano l'interfaccia BlockingQueue

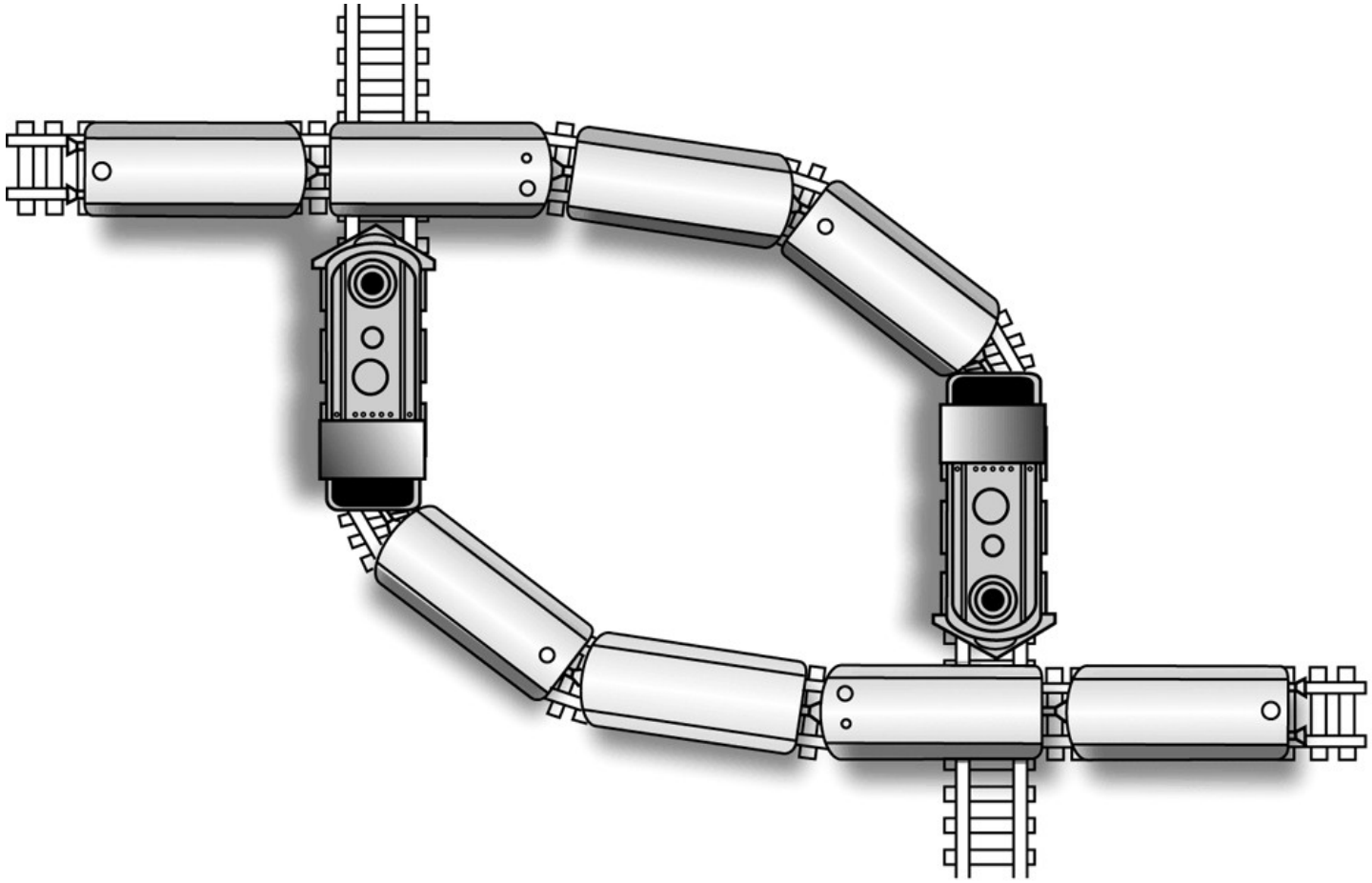
- LinkedBlockingQueue: non si definisce un limite superiore alla capacità della coda
- ArrayBlockingQueue: definisce un numero fisso di posizioni della coda
- SynchronousQueue: non è una vera e propria coda, in quanto non ha capacità di memorizzazione. Mantiene solo le code per la gestione dei threads che aspettano in attesa di produrre/consumare elementi. Risparmia il tempo necessario per la bufferizzazione degli elementi prodotti
- PriorityBlockingQueue: implementa una coda con priorità

Deadlock (stallo)

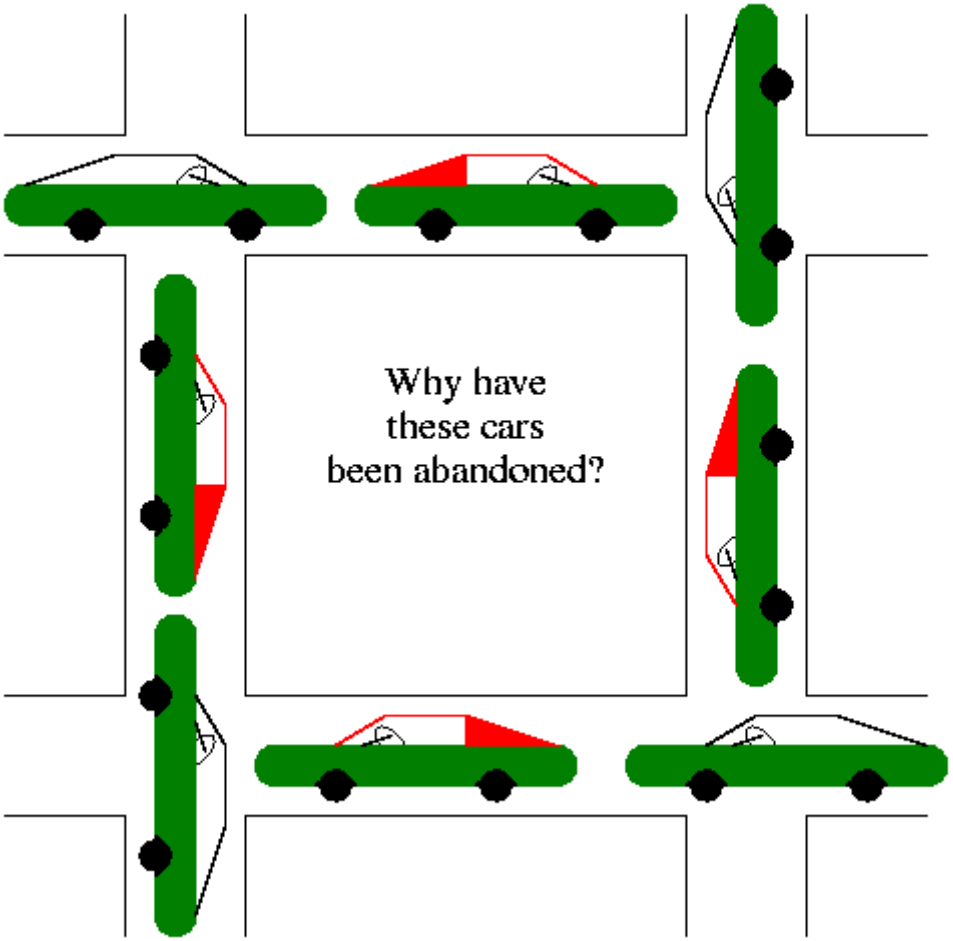
- Il deadlock si verifica quando due processi sono in attesa l'uno dell'altro
- Es: uno accede alla stampante, ma è in attesa del lettore CD, l'altro accede al CD e aspetta la stampante
- **Condizioni necessarie** perché si verifichi un deadlock
  - Le risorse per cui si compete non sono condivisibili (ognuna può essere allocata ad un solo processo alla volta)
  - Le risorse sono richieste "un po' alla volta"
  - Una risorsa allocata non può essere liberata con la forza
-
- Se una di queste condizioni è falsa, non c'è deadlock.

Deadlock: un esempio

Quali sono le risorse? Come si può sbloccare?



Deadlock: un altro esempio




BANCO REAL
11455

Esempio

- Quale condizione viene meno con le seguenti proposte per evitare il deadlock su un ponte a una corsia?
  - Non lasciare entrare una macchina sul ponte finché non è vuoto;
  - Se due macchine si incontrano, una delle due deve fare retromarcia;
  - Aggiungere una nuova corsia al ponte.