Undicesima Esercitazione

Accesso a risorse condivise
tramite monitor in Java

Andrea Reale
andrea.reale@unibo.it

Agenda

- **Esercizio 1**
  - Sincronizzazione di Thread con risorse multiple
- **Esercizio 2**
  - Accesso a risorsa condivisa con sincronizzazioni complesse tramite Monitor

Esercizio 1

Accesso competitivo a risorse condivise e sincronizzazione con risorse multiple

Esercizio 1 - Le sedie musicali

Si vuole realizzare un programma Java che simuli una partita al gioco delle ***"sedie musicali"*** (http://it.wikipedia.org/wiki/Sedie_musicali).

- Il gioco consiste in **N partecipanti** + 1 game master. Gli N partecipanti si contendono N-1 sedie.
- Si procede a fasi ad eliminazioni successive.
  - All'inizio di ogni fase i giocatori sono in piedi.
  - Il game master fa partire la musica che rimane accesa per un numero casuale di secondi (da 1 a 5) .
  - Quando la musica viene spenta, i giocatori si contendono le N-1 sedie disponibili
  - Quello che non riesce a sedersi viene eliminato e non partecipa alla fase successiva.

Esercizio 1 - Traccia (2/2)

- All'inizio di ogni fase, il game master rimuove una sedia da quelle disponibili, così che ci sia sempre una sedia in meno rispetto al numero dei giocatori.
- Il gioco finisce quando resta un solo giocatore.

Si modellino i **Giocatori** ed il **GameMaster** come thread Java:

- I thread di esecuzione corrispondenti ai giocatori **competeranno** per accedere ad una *risorsa scarsa* come quella delle **Sedie**
- I thread giocatore dovranno coordinare le loro azioni rispetto allo stato della risorsa **Musica**.
- Il thread corrispondente al **GameMaster** dovrà andare a modificare opportunamente la risorsa **Musica** e la risorsa **Sedie**, a seconda dello stato del gioco e dei giocatori.

Note alla soluzione

- Si controlli l'accesso alle risorse **Musica** e **Sedie** facendo uso del meccanismo dei **Monitor**.

- Si presti attenzione ai seguenti dettagli di sincronizzazione
  - Può il **GameMaster** far partire/stoppare la musica "a piacimento" oppure deve rispettare anch'egli dei vincoli di sincronizzazione con gli altri thread? Quali? (Non dimenticarsi che anche la musica è una risorsa condivisa)
  - Qual'è il momento più opportuno per rimuovere la sedia durante una nuova fase di gioco?

Variante Opzionale

- Si relizzi una *versione grafica* del gioco, in cui il programma offra un'interfaccia che mostri visivamente le sedie e la loro occupazione.
- Si **associ il ruolo del game master all'utente del programma,** che tramite interfaccia grafica è opportunamente abilitato ad accendere/spegnere la musica.

Classe Giocatore

```
public class Giocatore implements Runnable {
   private Sedie chairs;
   private Musica music;
   private boolean standing;

   public Giocatore(Sedie chairs, Musica music) {
      this.chairs = chairs;
      this.music = music;
      this.standing = true;
   }
   public void run() {
      while(true) {
         music.waitForMusicStart();
         if (!standing){
            chairs.releaseChair();
            standing = true;
         }
         music.waitForMusicStop();
         standing = ! chairs.tryToSeat();
         if (standing)
            break; // Esci dal gioco
      }
      return;
   }
}
```

Classe GameMaster (1/2)

```
public class GameMaster implements Runnable {
   private final Sedie gm;
   private final Musica mm;
   private final Random randomGen;

   public GameMaster(Sedie cm, Musica mm) {
      this.gm = cm;
      this.mm = mm;
      this.randomGen = new Random();
   }
   ... // CONTINUA
```

Classe GameMaster (2/2)

```
   public void run() {
      while (gm.getNumChairs() > 0) {
         try {
            mm.toggleMusic();
            / Sleep casuale tra uno e cinque secondi
            int sleepTime = this.randomGen.nextInt(4000) + 1000;
            Thread.sleep(sleepTime);
            gm.waitAllFree();
            gm.removeChair();
            mm.toggleMusic();
            gm.waitAllTaken();
         } catch (InterruptedException e) {
            e.printStackTrace();
            continue;
         } // end of try-catch
}  // end of while
   }  // end of run
} // end of class
```

Risorsa Sedie (1/4)

```
public class Sedie {
   private int freeChairs;
   private int numChairs;
   private final Lock chairsLock = new ReentrantLock();
   private final Condition chairsFree;
   private final Condition chairsTaken;

   public Sedie(int initialPlayers) {
      freeChairs = initialPlayers;
      numChairs = initialPlayers;

      chairsFree = chairsLock.newCondition();
      chairsTaken = chairsLock.newCondition();
   }

   public int getNumChairs() {
      try {
         chairsLock.lock();
         return numChairs;        } finally {
         chairsLock.unlock();
      }
   }
   // CONTINUA
```

Risorsa Sedie (2/4)

```
   public boolean tryToSeat() {
      try {
         chairsLock.lock();
         if (freeChairs > 0) {
            freeChairs--;
            if (freeChairs == 0)
               chairsTaken.signalAll();
            return true;
         } else {
            return false;
         } // end of if-else
      } finally {
         chairsLock.unlock();
      }
   }
   public void releaseChair() {
      try {
         chairsLock.lock();
         freeChairs++;
         if (freeChairs == numChairs)
            chairsFree.signalAll();
      } finally {
         chairsLock.unlock();
      }
   }
   // CONTINUA...
```

Risorsa Sedie (3/4)

```
public void removeChair() {        try {
      chairsLock.lock();    numChairs--;
      if (freeChairs > numChairs)
         freeChairs = numChairs;
   } finally {
      chairsLock.unlock();
   }
}
// CONTINUA ...
```

Risorsa Sedie (4/4)

```
public void waitAllFree() throws InterruptedException {        try {    chairsLock.lock();
      while (freeChairs != numChairs) {        chairsFree.await();
      }
   } finally {
      chairsLock.unlock();
   }
}

public void waitAllTaken() throws InterruptedException {
   try {
      chairsLock.lock();
      while (freeChairs != 0) {
         chairsTaken.await();
      }
   } finally {
      chairsLock.unlock();
   }
}
} // end of class
```

Risorsa Musica (1/2)

```
public class Musica {     private final Lock musicLock = new ReentrantLock();

   // Perché mi basta una sola condition per entrambe le attese?
   private final Condition musicChange = musicLock.newCondition();
   private boolean musicPlaying = false;
   public boolean toggleMusic() {
      try {
         musicLock.lock();
         musicPlaying = !musicPlaying;
         musicChange.signalAll();
         return musicPlaying;
      } finally {
         musicLock.unlock();
      }
   }

   // CONTINUA ...
```

Risorsa Musica (2/2)

```
   public void waitForMusicStart() {
      try {
         musicLock.lock();
         while (!musicPlaying)
            musicChange.await();
      } catch (InterruptedException e){
         // Gestione interruzione imprevista
      } finally {
          musicLock.unlock();
      }
   }
   public void waitForMusicStop() {
      try {     musicLock.lock();
         while (musicPlaying)
            musicChange.await();
      } catch (InterruptedException e) {     // Gestione interruzione imprevista
      } finally {
         musicLock.unlock();
      }
   }
}
```

Classe Main

```
public class ChairGameApp {
   private final static int NUM_PLAYERS = 10;

   public static void main(String[] args) {
      Musica mm = new Musica();
      Sedie cm = new Sedie(NUM_PLAYERS);
      Thread[] players = new Thread[10];
      Thread master = new Thread(new GameMaster(cm, mm));
      for (int i=0; i<NUM_PLAYERS; i++)
         players[i] = new Thread(new Giocatore(cm, mm));

      for (Thread t : players)
         t.start();

      master.start();
   }
}
```

Esercizio 2

Gestione di risorse condivise con condizioni d'accesso complesse tramite Monitor

Esercizio 2 - Traccia (1/3)

In un parco di divertimenti c'e` l'attrazione "***Il castello stregato***"

- Il gioco e` costituito da un castello che puo` essere visitato da **adulti** e **bambini**.
- Ogni visitatore **entra** nel castello, svolge la visita in modo libero per un arco di tempo di durata arbitraria (*casuale*), ed infine **esce**.
- Durante la visita, i clienti assistono ad un insieme di effetti speciali (***programma***)
  - In ogni istante di tempo nel castello si svolge un solo programma.

Esercizio 2 - Traccia (2/3)

I programmi previsti sono due:

- **Horror**, il cui contenuto **non** e` adatto a *visitatori bambini*
- **Magic**, il cui contenuto **e` adatto** sia a *visitatori adulti* che *bambini*

L'accesso al castello e` regolato dalle seguenti **politiche**:

- I visitatori accedono uno alla volta al castello;
- La capacita` massima del castello e` di **MAX** visitatori
- Un **visitatore bambino** non puo` accedere al castello se il programma corrente e` **"horror"**; in tal caso il bambino *aspetta* che il programma diventi "**magic"**
- Un **visitatore adulto** puo` assistere ad entrambi i programmi; ogni visitatore adulto e` *associato* al tipo di programma al quale e` interessato.

Esercizio 2 - Traccia (3/3)

Si realizzi un'applicazione Java che, utilizzando l'astrazione di *monitor*, implementi una *politica di accesso* e gestione del castello. In particolare, **oltre ai vincoli gia` elencati, si dovranno rispettare le seguenti regole di accesso**:

- I visitatori bambini devono avere la precedenza sugli adulti
- Gli adulti che richiedono il programma magic hanno la precedenza su quelli che richiedono il programma horror
- **Il programma del castello viene automaticamente cambiato**, solo una volta che **l'ultimo visitatore** interessato al programma corrente **e` uscito**, e se ci sono persone interessate all'altro programma in coda. (*I vincoli di precedenza vanno comunque considerati*)

L'applicazione Java dovra` instanziare un numero casuale di visitatori che, eseguendo tramite **thread separati e concorrenti**, dovranno accedere al castello stregato.

Note alla soluzione

- Come devono essere gestite le code di attesa per l'ingresso al castello?
  - **Quante code**? Di che tipo?

- Gestione del **cambio di programma**
  - Svuotamento opportuno (**rispettando le priorita`**) delle code che attendevano il nuovo programma