Corso sul linguaggio Java

Modulo JAVA9

B3.1–Mutua esclusione

Prerequisiti

- Programmazione concorrente
- Sezione critica
- Mutua esclusione
- **lock**() e **unlock**()

Introduzione

Si considerino le seguenti situazioni reali:

- le operazioni svolte da due *thread* su una risorsa condivisa non sono eseguite in modo atomico (ad esempio, due *thread* che eseguono operazioni su un dato condiviso, possono essere interrotti da un *context switch* prima che abbiano terminato la "transazione", ossia le operazioni in corso).
- i *thread* non sono del tutto indipendenti (ad esempio, *thread1* per procedere attende il risultato di un calcolo prodotto da *thread2*)

Poiché queste situazioni possono portare a risultati imprevedibili, occorre studiare un meccanismo che consenta ai *thread* di essere eseguiti in modo **sincronizzato**.

Modelli di concorrenza

Affrontiamo il problema della concorrenza svolgendo le nostre considerazioni secondo i due classici modelli:

- **mutua esclusione** (in questa Unità)
- **produttore/consumatore** (nella successiva Unità)

Mutua esclusione

Il problema della **mutua esclusione** si verifica quando due o più *thread* concorrono per l'utilizzo di una medesima risorsa, ad esempio la scheda audio o una stampante.

Prendiamo in esame l'esempio di prenotazione di posti, svolta da parte di clienti di un servizio (aereo, teatro, ecc).

Sono presenti *thread* di richiesta, ad arrivo casuale, ciascuno per un dato numero di posti.

L'assegnazione deve tenere conto delle molteplici richieste a fronte di un numero limitato di posti.

Esempio: l'Assegnatore

L'applicazione si compone quindi della classe *Assegnatore* seguente:

Assegnatore	
- **int** tot_posti	// variabile globale
+ **boolean** assegna_posti (**String** cliente, **int** num_posti)	// verifica ed eventuale // assegnazione effettiva // dei posti
+ **int** totale_posti()	// get per *tot_posti*

Esempio: l'*Assegnatore*

ATTIVITA': implementare in Java la classe *Assegnatore*

```
class Assegnatore
{ private int tot_posti = 20;
  public boolean assegna_posti (String cliente, int num_posti)
  {  System.out.println("*** Richiesta di " + num_posti + " da " + cliente);
     try { if (tot_posti >= num_posti)
            {System.out.println("*** Assegna "+ num_posti + " a " + cliente);
             Thread.sleep((int) ( Math.random() * 5000 ));
              tot_posti -= num_posti;
              return true;     }
           }
      catch (InterruptedException e) { }
      return false;
  }
  int totale_posti() {return tot_posti;}
}
```

Questa classe, se è possibile, assegna effettivamente ad un dato cliente il numero di posti richiesto.

Esempio: il *Richiedente*

Nell'applicazione sarà presente anche la classe *Richiedente*.

Richiedente **extends Thread**	
-- **int** num_posti	// variabile globale
-- **int** sleepTime	// tempo casuale
-- Assegnatore assegnatore	// per assegnare i posti
+ **void** Richiedente (**String** nome, **int** num_posti, Assegnatore assegnatore)	// costruttore classe
+ **void** run()	// verifica se l'assegnatore è in grado di assegnare i posti

Esempio: il *Richiedente*

ATTIVITA': implementare in Java la classe *Richiedente*.

```
class Richiedente extends Thread
{   private int num_posti;
    int sleepTime;
    private Assegnatore assegnatore;
    public Richiedente(String nome, int num_posti, Assegnatore assegnatore)
    {       super(nome);
            sleepTime = (int) ( Math.random() * 5000 );
            this.num_posti = num_posti;
            this.assegnatore = assegnatore;
    }
    public void run()
    {       // vedi seguito      }
}
```

Questa classe indica i vari processi *client* che richiedono l'assegnazione di posti all'Assegnatore che agisce da server.

Il tempo di ritardo casuale indica l'arrivo asincrono delle richieste dei *client*.

Esempio: il *Richiedente*

```
public void run()
{   System.out.println("-" + getName() + ": richiede " + num_posti + "...");
    try { Thread.sleep(sleepTime); }
    catch (InterruptedException e) {}
    if (assegnatore.assegna_posti(getName(), num_posti))
      System.out.println("-" + getName() + ": ottenuti " + num_posti + "...");
    else
      System.out.println("-" + getName() + ": posti non disponibili");
    }
```

Questo metodo è l'effettivo codice eseguito in parallelo dai vari processi *client*.

L'applicazione *AssegnaPosti*

```
public class AssegnaPosti
{   public static void main(String args[]) throws InterruptedException
    {  Assegnatore assegnatore = new Assegnatore();
       Richiedente client1 = new Richiedente("cliente1", 3, assegnatore);
       Richiedente client2 = new Richiedente("cliente2", 10, assegnatore);
       Richiedente client3 = new Richiedente("cliente3", 5, assegnatore);
       Richiedente client4 = new Richiedente("cliente4", 3, assegnatore);
       client1.start(); client2.start(); client3.start(); client4.start();
       client1.join(); client2.join(); client3.join(); client4.join();
       System.out.println("Numero di posti ancora disponibili: " +
    }
}
```

Il **main**() della classe *AssegnaPosti* lancia i vari *thread*, ma ….. si otterranno risultati imprevedibili che, nella realtà vanno evitati.

Sincronizzazione

Per evitare l'imprevedibilità dei risultati è necessario che un solo thread alla volta debba eseguire il metodo *assegna posti()* della classe *Assegnatore*.

Se un *thread* lo sta già eseguendo, gli altri *thread* che cercano di eseguirlo dovranno aspettare. Si tratta di un esempio di **sezione critica**.

In altre parole, più esecuzioni concorrenti di quel metodo devono in realtà avvenire in modo sequenziale ossia essere **sincronizzate.**

Clausola **synchronized**

In Java, il problema della sincronizzazione è risolto in modo estremamente elegante, grazie al fatto che ogni oggetto (istanza di **Object**) ha associato un **mutual exclusion lock** (in altre parole un **mutex**) che impedisce che le sezioni critiche di codice su esso possano essere interrotte.

Non si può accedere direttamente a questo **lock**, che però può essere gestito automaticamente quando si dichiara un metodo o un blocco di codice con la clausola **synchronized**.

E se ci sono più thread in attesa di eseguire un metodo synchronized?

Clausola **synchronized**

La JVM gestisce una coda per ogni oggetto che contiene metodi dichiarati **synchronized**.

Se un *thread t2* chiama un metodo **synchronized** mentre lo sta già eseguendo un altro *thread t1*, esso viene inserito in coda. Quando *t1* termina l'esecuzione del metodo viene, prelevato il prossimo *thread* dalla coda, in questo caso *t2* e così via.

In questo modo abbiamo garantito la mutua esclusione, ovvero un solo *thread* alla volta eseguirà la sezione critica. .

Clausola **synchronized**

Quando un metodo è **synchronized** lo si può istanziare su un oggetto solo se si è acquisito il **lock** su tale oggetto.

Quindi i metodi **synchronized** hanno accesso esclusivo ai dati incapsulati nell'oggetto (se a tali dati si accede solo con metodi **synchronized**) e consentono la **mutua esclusione**.

I metodi non **synchronized** non richiedono l'accesso al **lock** e quindi si possono richiamare in qualsiasi momento.

Vediamo come, con una semplice modifica del metodo *assegna_posti*() della classe *Assegnatore*, può essere elegantemente risolto il problema.

Clausola **synchronized**

- Metodi synchronized

Il metodo *assegna_posti(…)* diventa **synchronized** e quindi consente la mutua esclusione

```
class Assegnatore
{  private int tot_posti = 20;
   public synchronized boolean assegna_posti (String cliente, int num_posti)
  { System.out.println("--Richiesta di " + num_posti + " da " + cliente);
    try { if (tot_posti >= num_posti)
          { System.out.println("---Assegna " + num_posti + " a " + cliente);
            Thread.sleep((int) ( Math.random() * 5000 ));
            tot_posti -= num_posti;
            return true;     }
        }
        catch (InterruptedException e) { }
        return false;
    }
    int totale_posti() { return tot_posti; }
}
```

Clausola **synchronized**

- Metodi synchronized

La clausola **synchronized** applicata ad un metodo consente la **mutua esclusione** attraverso il seguente meccanismo:

- un *thread* deve eseguire un metodo **synchronized** su un oggetto: resta in blocco finchè non riesce ad ottenere il **lock** sull'oggetto
- quando ottiene il **lock** può eseguire il metodo (e tutti gli altri metodi **synchronized**), mentre i *thread* concorrenti vengono bloccati;
- gli altri *thread* rimarranno bloccati finchè il **lock** non viene rilasciato
- quando il *thread* esce dal metodo **synchronized**, rilascia automaticamente il **lock**
- a quel punto gli altri *thread* proveranno ad acquisire il **lock** e solo uno ci riuscirà, mentre gli altri torneranno in attesa.

Clausola **synchronized**

- Blocchi synchronized

```
class Assegnatore
{   private int tot_posti = 20;
    public boolean assegna_posti(String cliente, int num_posti)
    {  System.out.println("Richiesta di " + num_posti + " da " + cliente);
       synchronized (this)
       {  if (tot_posti >= num_posti)
          {  System.out.println("Assegnazione " + num_posti + " a " + cliente);
             tot_posti -= num_posti;
             return true;
          }
       }
       return false;
    }
    int totale_posti() {return tot_posti;}
}
```

Blocco di codice **synchronized**

Clausola **synchronized**

- Blocchi synchronized

La clausola **synchronized** applicata ad un blocco di codice di un oggetto implementa il concetto di sezione critica sul blocco.

A volte è conveniente rispetto al dichiarare **synchronized** un intero metodo perché si riduce la parte di codice da serializzare.

Clausola **synchronized**

- Blocchi synchronized

```
public void run()
{   System.out.println("-" + getName() + ": richiede " + num_posti + "...");
    synchronized (assegnatore);
    if (assegnatore.assegna_posti(getName(), num_posti))
      System.out.println("-" + getName() + ": ottenuti " + num_posti + "...");
    else
      System.out.println("-" + getName() + ": posti non disponibili");
}
```

Si può anche sincronizzare solo il codice interno al metodo **run**() del processo *client Richiedente*.

Argomenti

- Modelli di concorrenza
- Mutua esclusione
- Esempio: l'*Assegnatore*
- Esempio: il *Richiedente*
- L'applicazione *AssegnaPosti*
- Sincronizzazione
- Clausola **synchronized**
  - Metodi **synchronized**
  - Blocchi **synchronized**

Altre fonti di informazione

- Camagni, Nicolassy–Java Interfacce grafiche e programmazione concorrente, ed. Hoepli