Nona Esercitazione

Thread e memoria condivisa

Sincronizzazione tramite semafori

Semafori in Java

Dalla versione 5.0, è disponibile la classe Semaphore:

```
import java.util.concurrent.Semaphore;
```

tramite la quale si possono creare semafori, sui quali è possibile operare tramite i metodi:

- `acquire();` // implementazione di **p()**
- `release();` // implementazione di **v()**

Uso di oggetti Semaphore:

Inizializzazione ad un valore K dato:

```
Semaphore s=new Semaphore(k);
```

Operazioni: stessa semantica di p e v

```
s.acquire(); // esecuzione di p() su s
```

```
s.release(); // esecuzione di v() su s
```

Esempio sui Semafori

Allocazione di una risorsa con politica prioritaria (SJF)

Traccia (1/2)

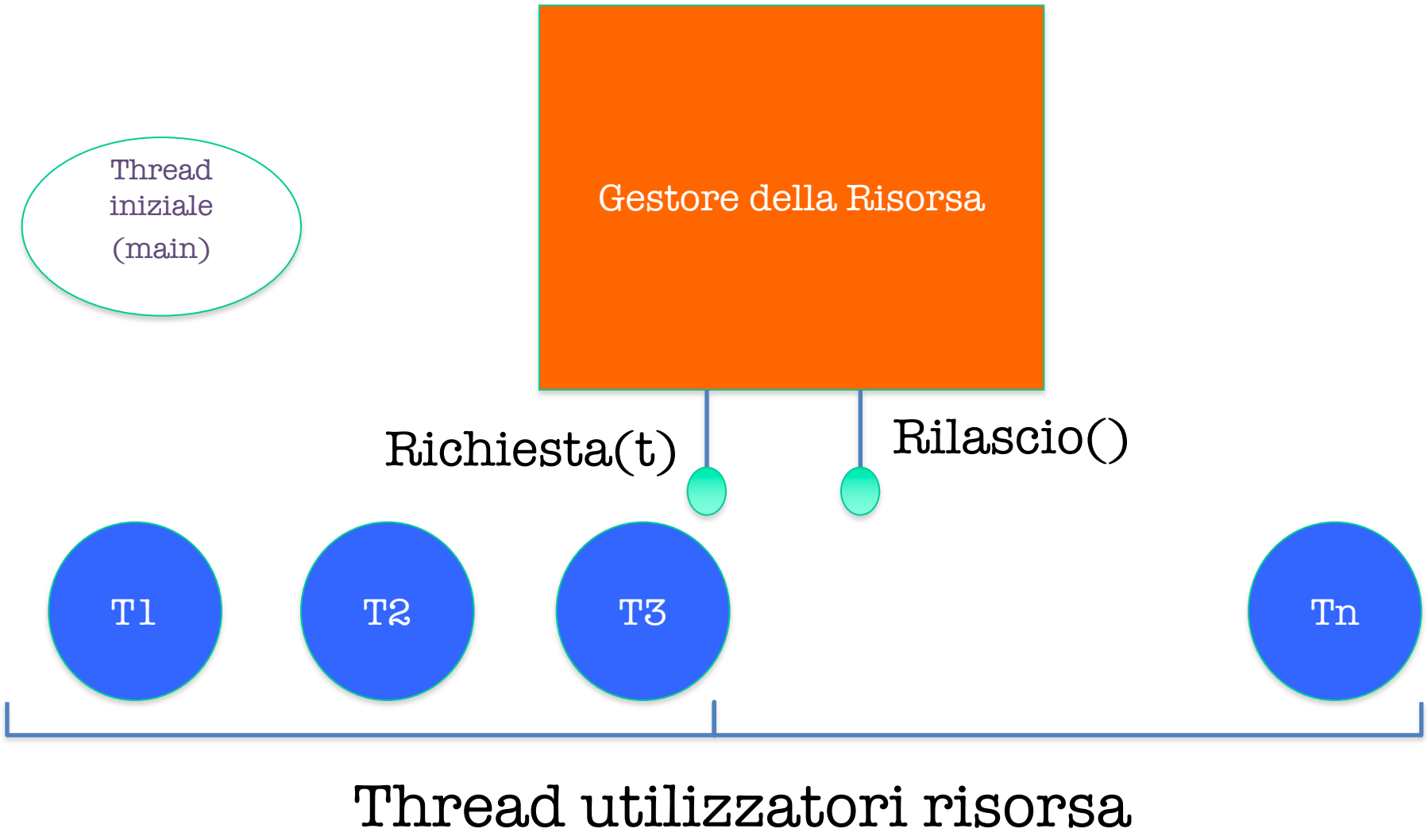
Si realizzi una applicazione Java che risolva il problema dell'allocazione di una risorsa secondo la politica "**Shortest Job First**", ovvero:

- **Una sola risorsa condivisa** da più thread

- Ogni thread utilizza la risorsa:
  - In modo **mutuamente esclusivo**
  - In modo **ciclico**
  - Ogni volta, **per una quantità di tempo arbitraria** (stabilita a run-time e dichiarata al momento della richiesta).

- **Politica di allocazione della risorsa**:
  - **SJF**: La precedenza va al thread che intende utilizzarla per il minor tempo.

Impostazione



Impostazione

- Quali classi ?
  - **ThreadP**: thread utilizzatori della risorsa; struttura ciclica e determinazione casuale del tempo di utilizzo
  - **Gestore**: mantiene lo stato della risorsa e implementa la politica di allocazione basata su priorità:
    - **Richiesta**(t) [t è il tempo di utilizzo]: **sospensiva** se
      - la risorsa è **occupata**,
      - oppure se c'è almeno un processo **più prioritario** (cioè che richiede un tempo minore di t) in attesa
    - **Rilascio**(): rilascio della risorsa ed eventuale risveglio del processo più prioritario in attesa (quello che richiede il minimo t tra tutti i sospesi).
  - **SJF**: classe di test (contiene il main())

Soluzione: classe ThreadP

```
import java.util.Random;

public class ThreadP extends Thread{
      Gestore g;
      Random r;
      int maxt;

      public  ThreadP(Gestore G, Random R, int
MaxT)
      {     this.r=R;
            this.g=G;
            this.maxt=MaxT;
      }
```

```
  public void run(){
   int i, tau;  long  t;
   try{
     this.sleep(r.nextInt(5)*1000);
     tau=r.nextInt(maxt);
     for(i=0; i<15; i++)    {
        g.richiesta(tau);
        this.sleep(tau);
        System.out.print("\n["+i+"]Thread:"+getName()
           +"e ho usato la CPU per "+tau+"ms...\n");
        g.rilascio();
        tau=r.nextInt(maxt);// calcolo nuovo tau
     }
   }catch(InterruptedException e){}
  } //chiude run
}
```

Uso della risorsa.

UN SOLO THREAD ALLA VOLTA!

Impostazione del gestore

Due cause di sospensione:

1. Accessi al Gestore della risorsa mutamente esclusivi: 1 alla volta! => definisco un semaforo di mutua esclusione

```
semaphore mutex = new Semaphore(1);
```

2. La risorsa è occupata, oppure c'è almeno un thread più prioritario in attesa:

   Quando la risorsa viene liberata deve essere svegliato il processo più prioritario => creiamo un semaforo per ogni livello di priorità:

```
semaphore []codaproc; //1 per ogni liv. Priorità
```

   Poichè ogni semaforo serve per sospendere processi la cui richiesta non può essere soddisfatta, ogni elemento di codaproc va inizializzato a 0 (semaforo "rosso").

   Necessità di individuare quanti siano i processi in attesa e la loro priorità:

```
int []sospesi;          //contatori thread sospesi
```

Classe Gestore

```
public class Gestore {
  int n;                  // massimo tempo di uso della risorsa
  boolean libero;
  Semaphore mutex;        //semaforo x la mutua esclusione
  Semaphore []codaproc; //1 coda per ogni liv. Prio (tau)
  int []sospesi;          //contatore thread sospesi

  public   Gestore(int MaxTime) {
    int i; this.n=MaxTime;
    mutex = new Semaphore(1);
    sospesi = new int[n];
    codaproc = new Semaphore[n];
    libero = true;
    for(i=0; i<n; i++) {
      codaproc[i]=new Semaphore(0);//semafori "condizione"
      sospesi[i]=0;
    }  }                   // continua…
```

...classe Gestore

```
/*richiesta per tau ms*/
public void richiesta(int tau){
   int i=0;
   try{
        mutex.acquire();
        while(piu_prio(tau)||libero==false){
              sospesi[tau]++;
              mutex.release();
              codaproc[tau].acquire();
              mutex.acquire();
              sospesi[tau]--;
        }
        libero = false;
        mutex.release();
        }catch(InterruptedException e){}
}
```

verifico che ci sia un processo più prioritario in attesa

...classe Gestore

```
// .. Continua

public void rilascio() {
      int da_svegliare, i;
      try{
            mutex.acquire();
            libero=true;
            da_svegliare = min_sosp();
            if (da_svegliare>=0)
                  codaproc[da_svegliare].release();
            mutex.release();
      }catch(InterruptedException e){}
}
```

Sveglio il processo più prioritario in attesa.

...classe Gestore (metodi utili)

```
private boolean piu_prio(int tau){
        int i=0;
        boolean risposta=false;
        for(i=0; i<tau; i++)
                if (sospesi[i]!=0)
                        return true;
        return risposta;
}

private int min_sosp(){
                int i=0, ris=-1;
                for(i=0; i<n; i++)
                        if (sospesi[i]!=0)
                                return i;
                return ris;
}
```

c'è qualcuno più prioritario del thread che userà la risorsa per tau secondi?

Chi è il processo più prioritario (con minor tau) sospeso in coda?

Soluzione: classe sjf

```
import java.util.*;
import java.util.Random;

public class sjf{
 public static void main(String args[]) {
      final int NT=10;//thread
      final int MAXT=500; // quanto di tempo massimo
      int i;
      Random r=new Random(System.currentTimeMillis());
      threadP []TP=new threadP[NT];
      gestore G=new gestore(MAXT);
      for (i=0; i<NT; i++)
            TP[i]=new threadP(G, r, MAXT);
      for (i=0;i<NT; i++)
            TP[i].start();
  }
}
```

Commento finale

- Nell'esempio abbiamo usato un semaforo per ogni livello di priorità; ogni semaforo è quindi usato per accodare processi con la medesima priorità ed è gestito in modo tale che ogni **p** (acquire()) risulti **sospensiva**.

→ semafori utilizzati in questo modo vengono detti «**semafori privati**» perchè ogni semaforo serve per sospendere i processi di un certo tipo (es: il semaforo codaproc[k] per i processi che hanno priorità=k); Pertanto si dice che il semaforo è «privato» per quella categoria di processi.

Schema tipico di uso dei semafori privati:

```
//ACQUISIZIONE RISORSA:

mutex.acquire();
while(<condizione di sospensione>){
      sospesi_k++;
      mutex.release();
      s_k.acquire();
      mutex.acquire();
      sospesi_k--;
}
<aggiornamento stato risorsa condivisa>
mutex.release();
```

```
//RIATTIVAZIONE Thread sospesi:

mutex.acquire();
...
<individuazione del tipo di processo k da
riattivare>
if ( <c'è almeno un processo nella coda di s_k > )
      s_k.release();
...
mutex.release();
```

Esercizio 1

Si consideri un importante museo di arte antica e contemporanea aperto al pubblico per le visite.

Al museo possono accedere due tipi di **utenti**:

1. **scolaresche**, ovvero gruppi di studenti, che accedono al museo gratuitamente, in virtù di una convenzione con il ministero della pubblica istruzione.
2. **gruppi generici**, che accedono al museo acquistando un biglietto per ogni componente del gruppo. Sia **PT** il prezzo del biglietto individuale.

Dato il valore delle opere esposte al museo, le **visite dei gruppi generici** possono avvenire soltanto con la **guida** di un addetto, messo a disposizione dall'amministrazione del museo. A questo proposito, si assuma che:

- il numero delle guide complessivamente presenti nel museo sia pari a **NA**,
- che ogni **guida** possa accompagnare **un gruppo alla volta**
- che tutte le **guide** siano **equivalenti**.

Le **scolaresche**, invece, **non necessitano di una guida** perchè in ogni scolaresca è presente un insegnante che svolge il ruolo di guida.

Si assuma inoltre che il museo non possa accogliere più di MAX utenti (numero di scolaresche e di gruppi generici) contemporaneamente.

Si progetti un'applicazione Java nella quale scolaresche e gruppi generici siano rappresentati da thread concorrenti, e il museo rappresenti una risorsa condivisa tra i thread.

Al termine di tutte le visite, il thread main deve stampare l'incasso totale (ovvero, la somma degli importi di tutti i biglietti venduti) e il numero totale di persone che hanno visitato il museo.

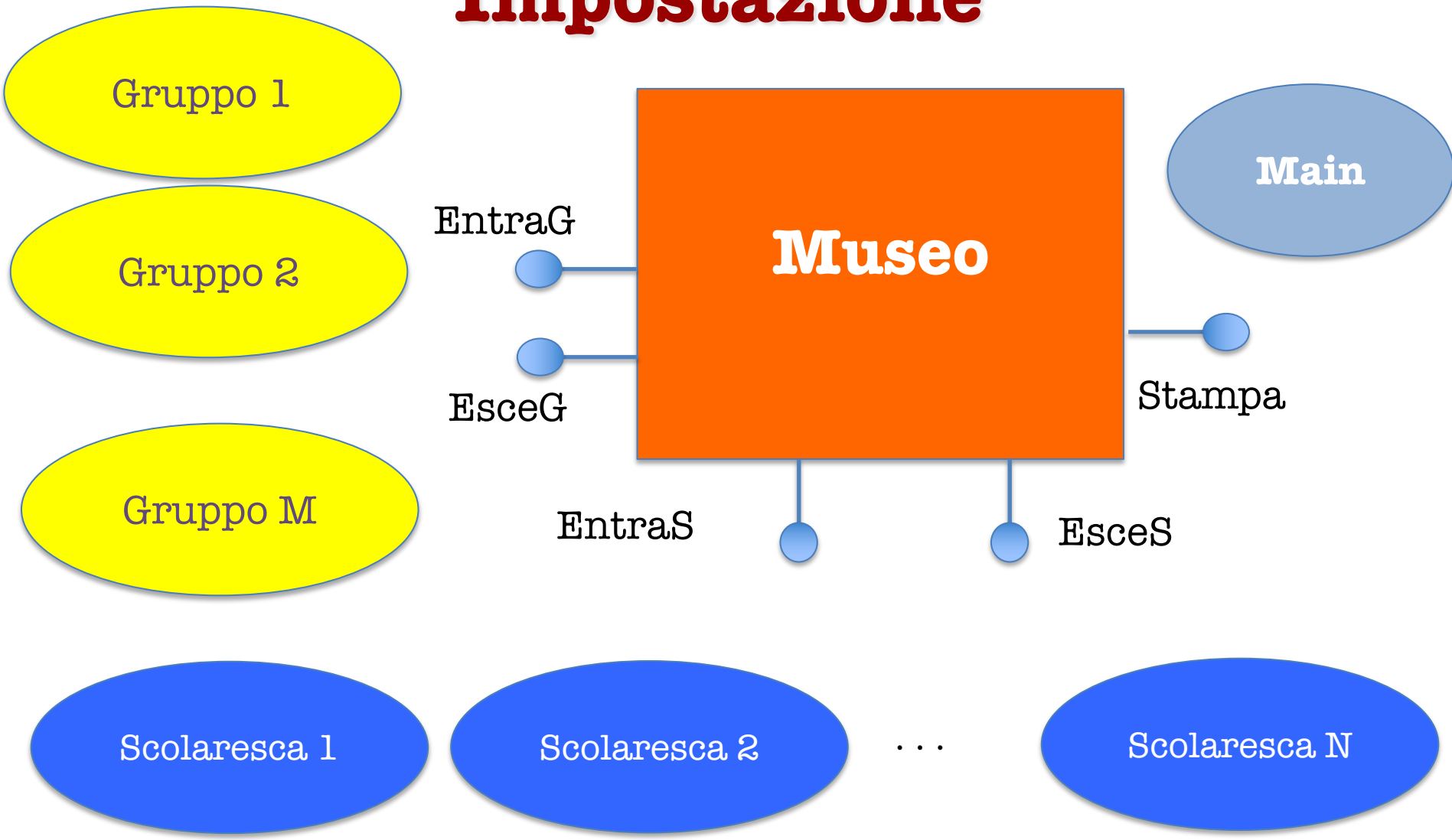
Impostazione – quali classi?

- **Scolaresca**: **thread** che rappresenta una scolaresca; è caratterizzato da una cardinalità, che ne esprime il numero di componenti.
- **Gruppo**: **thread** che rappresenta un gruppo generico; è caratterizzato da una cardinalità, che ne esprime il numero di componenti.

- **Museo**: risorsa condivisa; è caratterizzata dal numero di guide disponibili e dal numero di posti liberi; definisce i metodi:
  - **EntraS:** entrata nel museo di una scolaresca.
  - **EsceS:** uscita dal museo di una scolaresca.
  - **EntraG:** entrata nel museo di un gruppo.
  - **EsceG:** uscita dal museo di un gruppo.

- **Main:** definisce il metodo main, che crea le istanze di tutte le altre classi e attiva tutti i thread.

Impostazione



Classe Museo

- E' la risorsa condivisa tra i thread: definisce lo stato del museo e implementa la sincronizzazione tra thread.
- Quanti semafori?
  - **Mutua esclusione**: **SM** per rendere mutuamente esclusiva l'esecuzione di ogni metodo sulla risorsa Museo
  - **Sospensione Gruppi** in entrata: **SemG**
  - **Sospensione Scolaresche**: **SemS**

Suggerimento: gestire SemG e SemS come semafori **privati**.

Esercizio 2 – Il Museo con priorità sul tipo di utente

Estendere l'es.1 prevedendo che la gestione degli accessi venga esercitata in base a una politica con **priorità**.

In particolare, si assuma che, nell'entrata al museo, le **scolaresche abbiano la precedenza sui gruppi generici**.

Esercizio 3 – Priorità in base al tipo e numero di componenti

Estendere l'es.1 come segue.

Si assuma che la capacità del museo abbia una **capacità massima di visitatori fissata a MAXV**; ovvero il museo non può accogliere più di MAXV persone contemporaneamente (di conseguenza si elimini il vincolo sul massimo numero di gruppi/scolaresche).

Pertanto, ogni gruppo/scolaresca di numerosità N può entrare solo se ci sono almeno N posti liberi all'interno dal museo, altrimenti aspetta.

La gestione degli accessi dovrà essere realizzata in modo tale che:

- le **Scolaresche abbiano la precedenza sui Gruppi generici**;
- a parità di tipo di utente **i gruppi di minore numerosità vengano favoriti**.