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

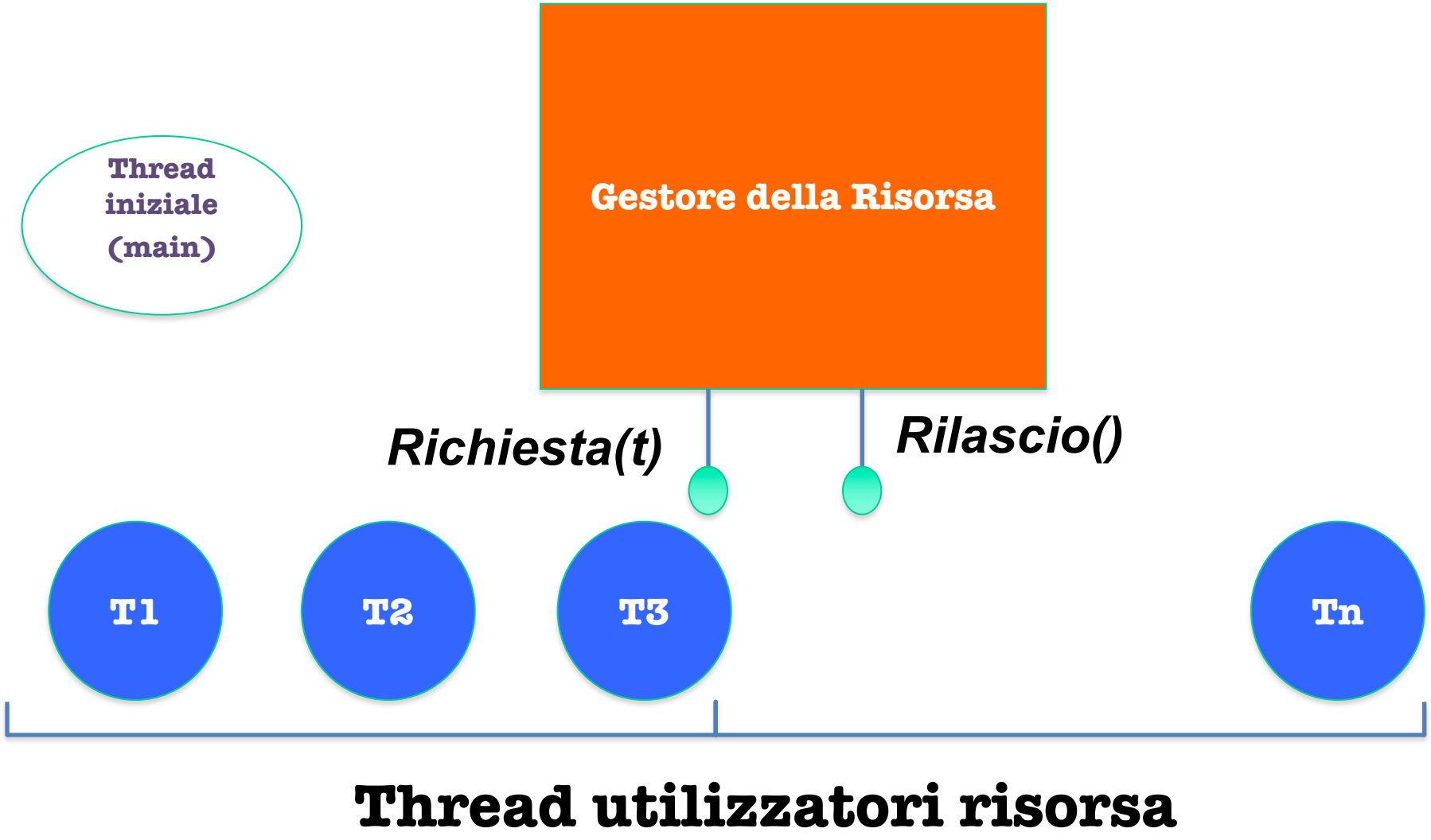
Allocazione di una risorsa con politica prioritaria (SJF)

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
```

   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. Priorità
(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){}
}
```

...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();
  }
}
```

Esercizio 1 – la fabbrica di auto

Si consideri lo stabilimento di produzione di una casa automobilistica.

Si consideri, in particolare, la fase di **montaggio delle ruote**, che prevede l'installazione di 4 cerchi e 4 pneumatici per ogni auto.

La fornitura di cerchi e pneumatici viene eseguita da 2 nastri trasportatori dedicati (1 per i cerchi e 1 per gli pneumatici), che consegnano tali elementi - uno alla volta - a un unico **deposito**, dal quale verranno successivamente prelevati da un sistema robotico (Robot) dedicato al montaggio.

Il deposito è caratterizzato da capacità limitate pari a :

- **MaxP**: numero massimo di pneumatici che possono essere contemporaneamente stoccati nel deposito;

- **MaxC**: numero massimo di cerchi che possono essere contemporaneamente stoccati nel deposito.

Il **Robot** ha un funzionamento ciclico (1 ciclo per ogni montaggio): ad ogni ripetizione, una volta disponibili i 4 cerchi e i 4 pneumatici necessari per un'auto, li preleva dal deposito e procede al montaggio.

Si realizzi un programma Java che realizzi la fase di montaggio ruote nel quale i due nastri trasportatori e il Robot siano rappresentati da thread concorrenti.

Impostazione

Impostazione – quali classi?

- **ThreadNC, ThreadNP**: nastri trasportatori che accedono ciclicamente al deposito per depositare un oggetto alla volta.
- **ThreadRobot**: robot per il montaggio ruote; accede al deposito per prelevare 4 cerchi e 4 pneumatici alla volta.
- **Gestore_Deposito**: mantiene lo stato della risorsa e implementa la politica di allocazione basata su priorità. Deve implementare i metodi:
  - **Preleva ()**: sospensivo se
    - non ci sono i 4 cerchi e i 4 pneumatici necessari per un nuovo montaggio.
  - **AggiungiP ()**: sospensivo se non c'è spazio per un nuovo pneumatico.
  - **AggiungiC ()**: sospensivo se non c'è spazio per un nuovo cerchione.

  Ogni metodo deve essere eseguito sul gestore in mutua esclusione.
- **Fabbrica**: definisce il metodo main, che crea le istanze di tutte le altre classi.

Classe Gestore_Deposito

- Rappresenta il deposito: ne gestisce lo stato e implementa la sincronizzazione tra thread.

- Quanti semafori?
  - Mutua esclusione
  - Sospensione Robot
  - Sospensione NastroC
  - Sospensione NastroP

Classe Deposito

Modello produttore-consumatore con varianti:

- 2 produttori
- 2 prodotti diversi e 2 capacità diverse
- il produttore produce unità, ma il consumatore consuma 4 cerchi e 4 pneumatici alla volta

Inizializzazione semafori:

- Mutua esclusione => `new Semaphore(1)`
- Sospensione Robot (consumatore): quando si sospende? com'è il deposito all'inizio? => v.i.0
- Sospensione NastroC(produttore) quando la capacità MAXC di cerchi è stata saturata. All'inizio il deposito è vuoto, quindi inizialmente quanto vale il semaforo?
- Sospensione NastroP(produttore):...

Esercizio 2 – la fabbrica di auto con priorità

Si estenda il problema dell'esercizio 1, considerando, invece di un solo robot, **due robot** dedicati ciascuno al montaggio delle ruote delle auto prodotte. In particolare:

- **robotA**: monta le ruote delle auto destinate al mercato estero;
- **robotB**: monta le ruote delle auto dedicate a mercato italiano.

(si assuma che ruote e cerchi usati da entrambi i robot siano uguali)

Si realizzi un programma Java per la gestione del deposito e la sincronizzazione dei thread, che, in aggiunta ai vincoli dati, nel prelievo dal deposito **dia la priorità al RobotA**.

Esercizio 2
Impostazione – quali classi?

- **ThreadNC, ThreadNP**: nastri trasportatori che accedono ciclicamente al deposito per depositare un oggetto alla volta.
- **ThreadRobotA, ThreadRobotB**: robot per il montaggio ruote.
- **Gestore_Deposito**: mantiene lo stato della risorsa e implementa la politica di allocazione basata su priorità. Deve implementare i metodi:
  - **PrelievoRuoteA()**: sospensiva se
    - non ci sono i 4 cerchi e i 4 pneumatici necessari per un nuovo montaggio.
  - **PrelievoRuoteB()**: sospensiva se
    - non ci sono i 4 cerchi e i 4 pneumatici necessari per un nuovo montaggio.
    - **c'è il RobotA in attesa.**
  - **AggiungiP()**: sospensivo se non c'è spazio per un nuovo pneumatico. Se l'aggiunta crea le condizioni per poter risvegliare un robot, ne viene risvegliato uno, tenendo conto della priorità.
  - **AggiungiC()**: sospensivo se non c'è spazio per un nuovo cerchione. Se l'aggiunta crea le condizioni per poter risvegliare un robot, ne viene risvegliato uno, tenendo conto della priorità.
- **Fabbrica**: definisce il metodo main:

Impostazione del gestore

Quali semafori:

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

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

2. **Sospensione dei thread:**
   - ThreadNC -> semaphore SNC
   - ThreadNP -> semaphore SNP
   - RobotA: -> semaphore SRA
   - RobotB -> semaphore SRN

   Come li gestisco? Suggerimento: Semafori condizione (v.SJF) -> v.i.O

   Necessità di individuare se vi siano processi in attesa (soprattutto per RobotA)

```
int sospesiRA;        //contatore thread sospesi su SRA
...
```