Nona Esercitazione

Thread e memoria condivisa

Sincronizzazione tramite semafori

Agenda

Esercizio 1 – DA SVOLGERE

- Sincronizzazione avanzata produttore consumatore tramite semafori

Esercizio 2 – DA SVOLGERE

- Allocazione di una risorsa con priorità: algoritmo "Shortest Job First"

Esercizio 1 – sincronizzazione java con semafori

- Si realizzi un programma Java che simuli la gestione di una cucina di un ristorante.
- In particolare :
  - Due thread **AiutoCuoco** si incaricano di portare sul tavolo di preparazione (risorsa condivisa) gli ingredienti necessari (di due tipi diversi)
  - Un thread **Chef** sovrintende alla preparazione del piatto:
    - Attende che siano disponibili le quantità necessarie di ingredienti
    - Quando disponibili, le preleva dal tavolo e prepara il piatto

Esercizio 1 - Traccia (2/2)

- **Tavolo** ha **capacità limitata** (*diversa*) per ingrediente1 e ingrediente2 (risp. **M1** e **M2**)
- Ciascun ***AiutoCuoco*** porta periodicamente una unità di ingrediente
- Per preparare il piatto, **Chef** ha bisogno **di quantità prestabilite** per i 2 ingredienti
  - Q1 per ingrediente1
  - Q2 per ingrediente2

Esercizio 1 - Sincronizzazione

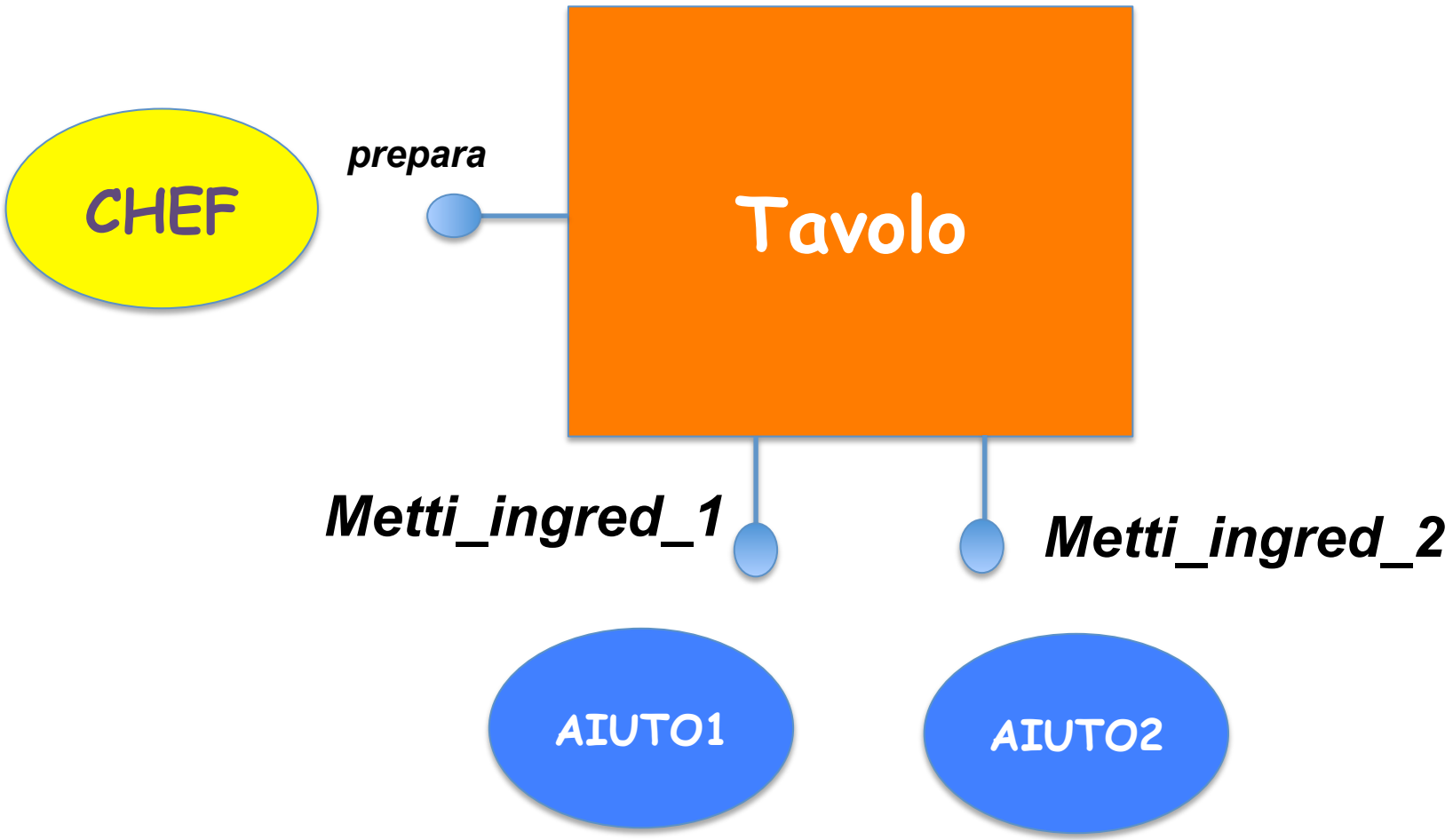
- **Tavolo** è una risorsa condivisa con *due buffer*, uno per ciascun ingrediente
  - dobbiamo **gestire della capacità**: *non è possibile ospitare altre unità di ingredienteX se il tavolo ha già saturato la capacità per X*
- Necessità di **mutua esclusione** tra thread che accedono al tavolo
- Thread **Chef** deve attendere che siano disponibili gli ingredienti
  - Soltanto allora comincia la preparazione del piatto (**sincronizzazione** tra thread)
  - Si assuma che **il tavolo sia inizialmente vuoto**

Esercizio 1 – Alcune note

- **Si utilizzino i semafori** (classe **Semaphore** di **java.util.concurrent**) per risolvere *tutti* i problemi di sincronizzazione tra thread
  - **Mutua esclusione** nell'accesso ad una risorsa condivisa
  - **Gestione della capacità** dei buffer ingredienti
  - **Ordinamento** di operazioni di thread diversi (lo Chef attende che ci siano abbastanza ingredienti prima di iniziare a cucinare)

- Per semplicità, si assuma che le operazioni degli **AiutoCuoco** e dello **Chef** si ripetano un numero *indefinito* di volte nel tempo.
  - I thread continuano ad operare fin quando la JVM che li esegue non viene interrotta dall'utente "a mano"

Impostazione



Classe Tavolo

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

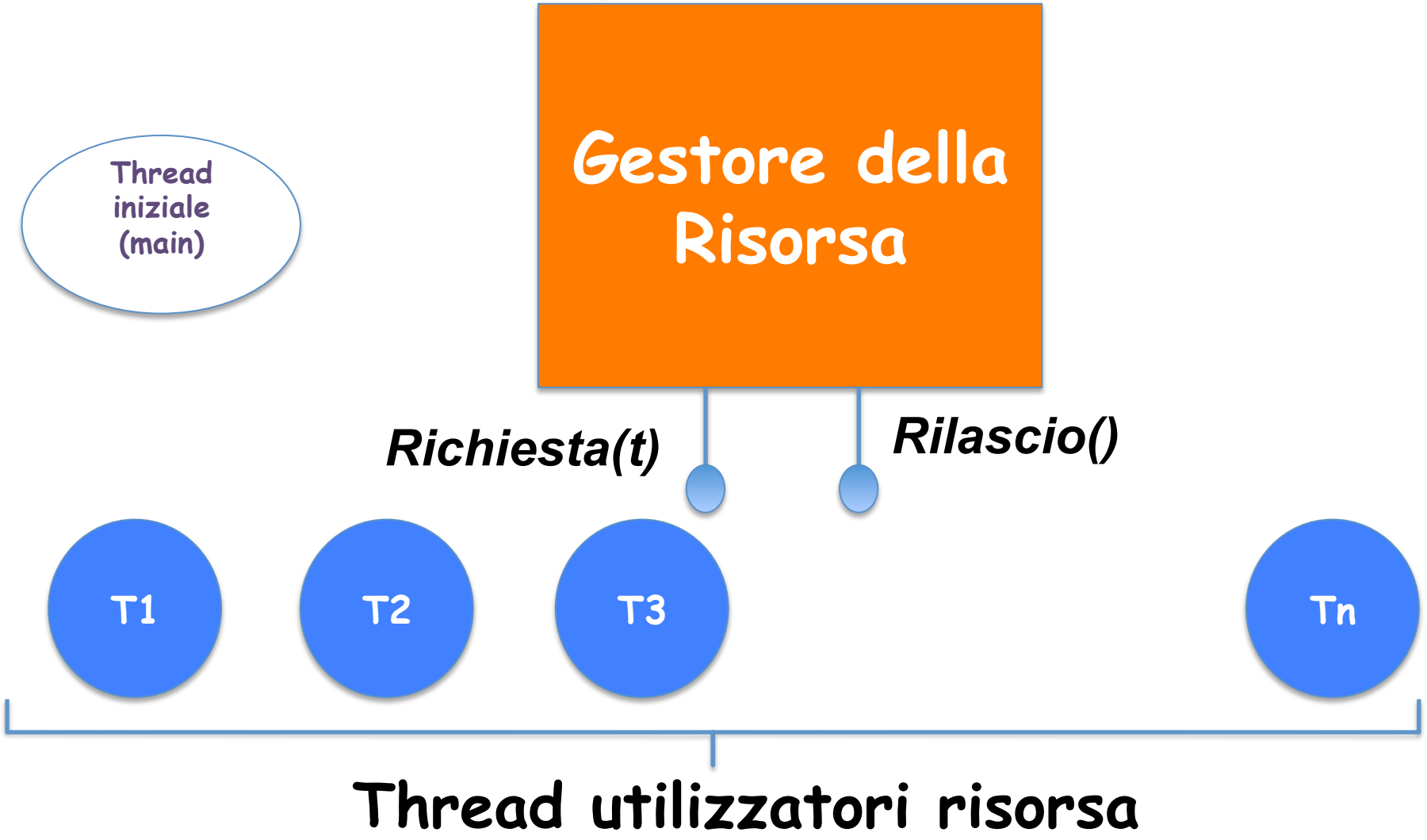
- Quanti semafori?
  - Mutua esclusione
  - Sospensione chef
  - Sospensione aiuto cuoco 1
  - Sospensione aiuto cuoco 2

Esercizio 2 - Traccia (1/2)

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

- **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)**: 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())

Suggerimenti: 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;
      }
```

...classe ThreadP

```
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);
      <utilizzo della risosa…>
      System.out.print("\n["+i+"]Thread:"+getName()
        +"e ho usato la CPU per "+tau+"ms...\n");
      g.rilascio();
      tau=r.nextInt(maxt); //calcolo nuovo CPU Burst
   }
 }catch(InterruptedException e){}
} //chiude run
}
```

Porzione di codice da eseguire in mutua esclusione. UN SOLO THREAD ALLA VOLTA!

Impostazione del gestore

Due cause di sospensione:

1. **Accessi alla risorsa mutamente esclusivi. Uno alla volta!** => all'inizio c'è 1 posto libero
   - => `Semaphore mutex = new Semaphore(1);`

   Se la risorsa potesse essere usata contemporaneamente da M processi (e.g.: una scatola che contiene M oggetti), allora:
   => all'inizio ci sono M posti liberi
   => `Semaphore mutex = new Semaphore(M);`

2. **C'è qualcuno più prioritario in coda**
   - abbiamo bisogno di un altro semaforo… che sia sempre rosso! => …`new Semaphore(0);`
   - vorremmo poter svegliare (`release()`) solo il processo più prioritario => creiamo un semaforo per ogni livello di priorità

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);
         sospesi[i]=0;
      }
   }                         // continua…
```

mi chiedo: "**Quanti sono i posti liberi prima che la acquire() debba sospendere il processo?**" Inizializ. a 0 perchè chiunque sia messo in coda deve essere subito sospeso. Se no non sarebbe una coda!