Reti e Laboratorio III
Modulo Laboratorio III
AA. 2025-2026

docente: Laura Ricci

laura.ricci@unipi.it

Correzione Assignment 2

Simulazione Ufficio Postale
attesa passiva
24/10/2025

SIMULAZIONE UFFICIO POSTALE

- simulare il flusso di clienti in un ufficio postale che ha 4 sportelli. Nell'ufficio esiste:
  - un'ampia sala d'attesa in cui ogni persona può entrare liberamente. Quando entra, ogni persona prende il numero dalla numeratrice e aspetta il proprio turno in questa sala.
  - una seconda sala, meno ampia, posta davanti agli sportelli, in cui si può entrare solo a gruppi di k persone

- una persona si mette quindi prima in coda nella prima sala, poi passa nella seconda sala.

- ogni persona impiega un tempo differente per la propria operazione allo sportello. Una volta terminata l'operazione, la persona esce dall'ufficio

SIMULAZIONE UFFICIO POSTALE

- Scrivere un programma in cui:
  - l'ufficio viene modellato come una classe JAVA, in cui viene attivato un `ThreadPool` di dimensione uguale al numero degli sportelli
  - la coda delle persone presenti nella sala d'attesa è gestita esplicitamente dal programma
  - la seconda coda (davanti agli sportelli) è quella gestita implicitamente dal `ThreadPool`
  - ogni persona viene modellata come un task, un task che deve essere assegnato ad uno dei thread associati agli sportelli
  - si preveda di far entrare tutte le persone nell'ufficio postale, all'inizio del programma

- Facoltativo: prevedere il caso di un flusso continuo di clienti e la possibilità che l'operatore chiuda lo sportello stesso dopo che in un certo intervallo di tempo non si presentano clienti al suo sportello.

SIMULAZIONE UFFICIO POSTALE

- la soluzione proposta prevede
  - entrata di tutti gli utenti nella prima sala, successivamente si inizia a farli passare nella seconda sala
  - attesa passiva: se la seconda stanza è piena (tutti gli sportelli occupati e massimo numero di persone in coda), l'utente (il task) viene inserito in una coda e viene risvegliato quando un thread diventa disponibile.

L'UFFICIO POSTALE

```
public class UfficioPC {
// Numero di clienti che entrano nell'ufficio.
public static final int numClienti = 15;
// Numero degli sportelli dell'ufficio.
public static final int numSportelli = 4;
// Dimensione della coda davanti agli sportelli.
public static final int dimCoda = 4;
// Coda della prima sala (senza vincoli di capacita').
public static BlockingQueue<Runnable> coda1 = new LinkedBlockingQueue<>();
// Coda della seconda sala (con capacita' fissa).
public static BlockingQueue<Runnable> coda2 = new ArrayBlockingQueue<>(dimCoda);
// Pool di thread personalizzato.
public static CustomPool pool = new CustomPool(numSportelli, coda2);
// Tempo massimo di attesa per la terminazione del pool.
public static final int poolTerminationDelay = 20000;
```

L'UFFICIO POSTALE

```
public static void main(String[] args) {
    // Avvio il thread produttore e il consumatore.
     Thread producer = new Thread(new Produttore(numClienti, coda1));
     producer.start();
     Thread consumer = new Thread(new Consumatore(coda1, pool));
     consumer.start();
     // Attendo la loro terminazione.
     try {producer.join(); consumer.join();}
     catch (InterruptedException e ) {System.err.println("Main: interruzione su join()");}
     // Avvio la terminazione del pool.
     pool.shutdown();
     try {
        if (!pool.awaitTermination(poolTerminationDelay, TimeUnit.MILLISECONDS))
           pool.shutdownNow();
        } catch (InterruptedException e) {pool.shutdownNow();} } }
```

IL TASK (L'UTENTE)

```
public class UfficioPC {

package UfficioPostale;


class MyTask implements Runnable {
     public final int id;
     public MyTask(int id) {this.id = id;}
     public void run() {
         System.out.printf("Hello from Task %d!\n", id);
      try {Thread.sleep(10000);}
      catch (InterruptedException e) {}
    }
  }
```

IL PRODUTTORE

```
package UfficioPostale;
import java.util.concurrent.BlockingQueue;
/**
 *    Il produttore inserisce nella prima coda `numElementi` task.
 *    Prima di terminare, inserisce anche un task "speciale" (con id=-1)
 *    in modo tale da far terminare anche il consumatore.
 */
class Produttore implements Runnable {
   public final int numElementi;
   private BlockingQueue<Runnable> coda;
   public Produttore(int numElementi, BlockingQueue<Runnable> coda) {
          this.numElementi = numElementi;
          this.coda = coda; }
   public void run() {
         for (int i = 0; i < numElementi; i++) {
                coda.add(new MyTask(i));}
         coda.add(new MyTask(-1));
     }}
```

IL CONSUMATORE

```
package UfficioPostale;

import java.util.concurrent.BlockingQueue;

/**
 *    Il consumatore preleva i task dalla prima coda e li invia al pool.
 *    Il task termina quando riceve l'"elemento speciale" da parte del produttore.
 */
class Consumatore implements Runnable {
     private BlockingQueue<Runnable> coda;
     private CustomPool pool;
     public Consumatore(BlockingQueue<Runnable> coda, CustomPool pool) {
        this.coda = coda;
       this.pool = pool;
     }


// prosegue slide successiva
```

IL CONSUMATORE

```
public void run() {while (true) {
    try {
       Runnable r = coda.take();
       // Controllo se ho ricevuto l'"elemento speciale"
       // che segnala la terminazione.
       if (((MyTask) r).id == -1) break;
       // Altrimenti lo invio al pool.
       pool.execute(r);
    } catch (InterruptedException e) {}
   }
  }}
```

IL CUSTOM POLL

```
package UfficioPostale;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.ThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

class CustomPool extends ThreadPoolExecutor {

    public CustomPool(int numThread, BlockingQueue<Runnable> coda) {

        // Chiamo il costruttore della superclasse.

        super(numThread, numThread, 0, TimeUnit.MILLISECONDS, coda, new
              ThreadPoolExecutor.AbortPolicy());

}


// prosegue slide successiva
```

IL CUSTOM POOL

```
@Override
public synchronized void execute(Runnable r) {
      BlockingQueue<Runnable> coda = this.getQueue();
      while (coda.remainingCapacity() == 0) {
            System.out.printf("Task %d must wait.\n", ((MyTask) r).id);
            try {this.wait();}
            catch (InterruptedException e) {}
        }
      super.execute(r);
}
@Override
protected synchronized void afterExecute(Runnable r, Throwable t) {
      super.afterExecute(r, t);
      this.notify();
}
}
```