

# **Reti e Laboratorio III**

## **Modulo Laboratorio III**

### **AA. 2025-2026**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

## **Correzione Assignment 2**

**Simulazione Ufficio Postale**

**17/10/2025**

# 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.

- la soluzione proposta è una soluzione di base, che può essere migliorata
- prevede
  - entrata di tutti gli utenti nella prima sala, successivamente si inizia a farli passare nella seconda sala
  - **attesa attiva**: se la seconda stanza è piena (tutti gli sportelli occupati e massimo numero di persone in coda) si ritesta attivamente lo stato della seconda sala, fino a che non si è liberato un posto
    - solo una prima soluzione, la soluzione migliore è non fare attesa attiva!
    - attesa della terminazione del threadpool per un intervallo di tempo fissato
- soluzione avanzata
  - eliminare attesa attiva (non banale)
  - prevedere di far entrare continuamente utenti nella seconda sala

```
import java.util.concurrent.*;  
  
public class Persona implements Runnable {  
  
    // Identificativo del cliente.  
    public final int id;  
  
    // Minimo intervallo di tempo per le operazioni del cliente.  
    public final long minDelay = 0;  
  
    // Massimo intervallo di tempo per le operazioni del cliente.  
    public final long maxDelay = 1000;  
  
    /**  
     *costruttore della classe Persona.  
     * @param id l'id del cliente  
     */  
  
    public Persona(int id) {  
        this.id = id;  
    }  
}
```

```
/**  
 * Metodo contenente la logica del cliente.  
 * Ogni cliente dell'ufficio genera un intervallo di tempo  
 * casuale e attende per tale numero di millisecondi prima  
 * di terminare.  
 */  
  
@Override  
public void run() {  
    System.out.printf("Cliente %d arrivato allo sportello.\n", id);  
    long delay = ThreadLocalRandom.current().nextLong(minDelay, maxDelay);  
    try {Thread.sleep(delay);}  
    catch (InterruptedException e) {  
        System.err.println("Interruzione su sleep.");  
        return;  
    }  
    System.out.printf("Cliente %d ha abbandonato l'ufficio.\n", id);  
}
```

- la classica classe per generare valori random è `java.util.Random`
- questa classe è thread-safe, ma non ha buone prestazioni in un ambiente multithreaded.
  - la classe utilizza un seed per generare numeri random
  - occorre realizzare una sezione critica per aggiornare questo seed
  - se il numero dei thread è alto, le prestazioni possono degradare notevolmente
- `java.util.concurrent.ThreadLocalRandom`
  - generatore di numeri casuali introdotto a partire da Java.7
  - un generatore “locale” per ogni thread attivato
  - diminuisce l’overhead dovuto agli accessi concorrenti

```
import java.util.concurrent.*;  
/**  
 * Versione che prevede - la chiusura degli sportelli in caso di inattività,  
 * - attesa attiva degli utenti che devono entrare nella seconda stanza  
 * @author Matteo Loporchio  
 */  
  
public class UfficioChiusura {  
    // Numero degli sportelli dell'ufficio.  
    public static final int numSportelli = 4;  
    // Dimensione della coda davanti agli sportelli.  
    public static final int dimCoda = 10;  
    // Numero di clienti da fare entrare nell'ufficio.  
    public static final int numClienti = 500;  
    // Tempo di attesa per ritentare di entrare nella seconda stanza se la coda  
    // sportelli e' piena.  
    public static final long queueDelay = 500;  
    // Tempo di attesa per la terminazione del pool.  
    public static final long terminationDelay = 5000;  
    // Tempo di inattività prima della chiusura di uno sportello.  
    public static final long closingDelay = 60000;
```

# L'UFFICIO

```
public static void main(String[] args) {  
    // Contatore delle persone servite.  
    int count = 0; System.out.println("Ufficio aperto!");  
    // Creo la coda (senza limiti di dimensione) per la prima sala.  
    BlockingQueue<Runnable> coda = new LinkedBlockingQueue<Runnable>();  
    // creo il pool di thread personalizzato usando AbortPolicy come politica di  
    // rifiuto (ovvero viene sollevata una RejectedExecutionException quando la  
    // coda del pool e' piena).  
    ThreadPoolExecutor pool = new ThreadPoolExecutor(  
        numSportelli,  
        numSportelli,  
        closingDelay,  
        TimeUnit.MILLISECONDS,  
        new ArrayBlockingQueue<Runnable>(dimCoda),  
        new ThreadPoolExecutor.AbortPolicy()  
    );  
    // Imposto la possibilita' di chiusura degli sportelli.  
    pool.allowCoreThreadTimeOut(true);
```



# ENTRARE NELLA PRIMA STANZA

```
// Metto in coda i clienti nella prima sala.  
  
for (int i = 0; i < numClienti; i++) coda.add(new Persona(i));
```



# ENTRARE NELLA SECONDA STANZA

```
while (!coda.isEmpty()) {  
    Persona p = (Persona) coda.peek();  
    try { pool.execute(p);  
        coda.poll();  
        count++; }  
    catch (RejectedExecutionException e) {  
        // se sono qui, significa che la coda davanti agli sportelli  
        // (ovvero la coda del pool) è piena.  
        // aspetto un certo intervallo di tempo affinche' si svuoti e poi  
        // ritento  
        System.out.printf("Coda sportelli piena. " +  
            "Il cliente con id=%d resta in attesa.\n", p.id);  
        try {  
            Thread.sleep(queueDelay); }  
        catch (InterruptedException x)  
        { System.err.println("Interruzione durante sleep."); } } }
```

# TERMINAZIONE

```
// a questo punto si chiude l'ufficio
// 1) si attende un certo intervallo di
// tempo affinche' tutti i thread possano terminare.
// 2) passato l'intervallo, l'esecuzione del pool
// viene interrotta immediatamente.

pool.shutdown();

try {

    if (!pool.awaitTermination(terminationDelay, TimeUnit.MILLISECONDS))
        pool.shutdownNow();

}

catch (InterruptedException e) {pool.shutdownNow();}

// Stampa di un messaggio di chiusura.

System.out.printf("Ufficio chiuso. Persone servite: %d\n", count);
}
```

- si blocca fino a che è verificata una delle seguenti condizioni
  - tutti i task hanno terminato la loro esecuzione
  - scade il time-out
  - il thread corrente viene interrotto
- restituisce
  - true se l'esecutore ha terminato tutti i task
  - false se il timeout è scattato prima della terminazione di tutti i task
- utile quando si vuole attendere la terminazione dei thread del pool, prima di proseguire con l'esecuzione del programma