Esercitazione n.3
LPR-B-08
Thread synchronization

12/10/2008

Vincenzo Gervasi

Esercizio 1

Si scriva un programma Java che dimostri che si possono verificare delle *race conditions* anche con una singola istruzione di incremento di una variabile.

- Scrivere una classe **Counter** che offre un metodo **next**() che incrementa una variabile locale, e un metodo **getCount**() che ne restituisce il valore.
- Scrivere un task **TaskCounter** che implementa **Callable** e che riceve nel costruttore un **Counter** e un intero **n**. Il task invoca la **next**() del **Counter** un numero casuale di volte compreso tra **n/2** e **n**, e restituisce il numero casuale calcolato.
- Il main crea un **Counter** e un pool di threads, in cui esegue **M** copie di **TaskCounter** passando a ognuna di esse il **Counter** e un valore **N**; quindi stampa la somma dei valori restituiti dagli **M** threads, e il valore finale del contatore ottenuto con **getCount**(): se questi due valori sono diversi c'è stata una race condition. **M** e **N** devono essere forniti dall'utente.

Esercizio 2

- Si consideri il metodo **next**() della classe **Counter** dell'Esercizio 1. Modificarlo in modo da renderne l'esecuzione non interrompibile, e rieseguire il programma verificando che non si verificano più race conditions. Fare questo nei tre modi visti:
  - usando un comando **synchronized**
  - usando un **lock esplicito**
  - dichiarando **synchronized** il metodo **next**()

Esercizio 3

Simulare il comportamento di una banca che gestisce un certo numero di conti correnti. In particolare interessa simulare lo spostamento di denaro tra due conti correnti.

Ad ogni conto è associato un thread T che implementa un metodo che consente di trasferire una quantità casuale di denaro tra il conto servito da T ed un altro conto il cui identificatore è generato casualmente.

- sviluppare una versione non thread safe del programma in modo da evidenziare un comportamento scorretto del programma
- definire 3 versioni thread safe del programma che utilizzino, rispettivamente
  - Lock esplicite
  - Blocchi sincronizzati
  - Metodi sincronizzati

Esercizio 4

- La classe Buffer ha una politica Last In First Out (LIFO), quindi non preserva l'ordine. Scrivere la classe CircularBuffer che estende Buffer e realizza una politica FIFO, gestendo l'array in modo circolare.
- Definire le interfacce generiche Producer<E>, Consumer<E> e Buffer<E>, che definiscono un sistema produttore/consumatore per un generico tipo di dati E.
- Implementare le interfacce in modo che il produttore produca una sequenza di stringhe, leggendole da un file passato come parametro al task, e il consumatore scriva le stringhe che prende dal buffer in un altro file.
- Nel main, creare e attivare un produttore e due o più consumatori. Verificare che la concatenazione dei file generati dai consumatori sia uguale, a meno dell'ordine delle righe, al file letto dal produttore.

Esercizio 5

Il laboratorio di Informatica del Polo Marzotto è utilizzato da tre tipi di utenti, studenti, tesisti e professori ed ogni utente deve fare una richiesta al tutor per accedere al laboratorio. I computers del laboratorio sono numerati da 1 a 20. Le richieste di accesso sono diverse a seconda del tipo dell'utente:

a) i professori accedono in modo esclusivo a tutto il laboratorio, poichè hanno necessità di utilizzare tutti i computers per effettuare prove in rete.

b) i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice i, poichè su quel computer è istallato un particolare software necessario per lo sviluppo della tesi.

c) gli studenti richiedono l'uso esclusivo di un qualsiasi computer.

I professori hanno priorità su tutti nell'accesso al laboratorio, i tesisti hanno priorità sugli studenti.

(prosegue nella pagina successiva)

Esercizio 5 (continua)

Scrivere un programma JAVA che simuli il comportamento degli utenti e del tutor. Il programma riceve in ingresso il numero di studenti, tesisti e professori che utilizzano il laboratorio ed attiva un thread per ogni utente. Ogni utente accede k volte al laboratorio, con k generato casualmente. Simulare l'intervallo di tempo che intercorre tra un accesso ed il successivo e l'intervallo di permanenza in laboratorio mediante il metodo sleep. Il tutor deve coordinare gli accessi al laboratorio. Il programma deve terminare quando tutti gli utenti hanno completato i loro accessi al laboratorio.