Esercizio Sincronizzazione Thread

Laboratorio di Programmazione di Rete A

Esercitazione di Laboratorio

17/10/2007

Esercizio

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

- i professori accedono in modo esclusivo a tutto il laboratorio, poiche' hanno necessita' di utilizzare tutti i computer per effettuare prove in rete.
- i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice i, poiche' su quel computer e' istallato un particolare software necessario per lo sviluppo della tesi.
- gli studenti richiedono l'uso esclusivo di un qualsiasi computer. I professori hanno priorita' su tutti nell'accesso al laboratorio, i tesisti hanno priorita' sugli studenti.

Esercizio

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.

Struttura delle Classi

- La classe astratta User rappresenta un utente generico. Implementa Runnable. E' estesa da 3 classi:
  - Student: gli studenti richiedono l'uso esclusivo di qualsiasi computer.
  - Graduate: i tesisti richiedono l'uso esclusivo di un determinato computer.
  - Professor: i professori accedono in modo esclusivo a tutto il laboratorio.
- La classe Tutor e' il monitor: sincronizza gli accessi al laboratorio.
- La classe Lab contiene il main: creazione e avvio del pool di thread.

La classe User

- La classe astratta User rappresenta un utente generico.
- Implementa Runnable: definisce un'attivita'.
- Contiene il riferimento all'oggetto monitor condiviso tra tutti gli utenti.
- Contiene un identificativo dell'utente, un contatore per il numero corrente e massimo di accessi al laboratorio.
- Dichiara due metodi astratti getComputer() e leaveComputer().
- Il metodo run e' implementato qui.

La Classe User: variabili d'istanza

```
abstract class User implements Runnable
{
   protected Tutor tutor;
   protected int userID;
   protected int totalAccesses;
   protected int currentAccesses;
   public static final int MIN_ACCESSES = 5;
   public static final int MAX_ACCESSES = 10;
   public static final int MAX_USE_TIME = 2;
   public static final int MAX_SLEEP_TIME = 5;
```

La Classe User: metodi

```
public User(Tutor tutor, int userID)
{
    this.totalAccesses = MIN_ACCESSES +
        (int)(Math.random() * (MAX_ACCESSES-MIN_ACCESSES));
    this.currentAccesses = 0;
    this.tutor = tutor;
    this.userID = userID;
}

public abstract void getComputer();

public abstract void leaveComputer();

public String toString()
{
    return this.getClass().getName() + " " + this.userID;
}
```

La Classe User: il metodo run

```
public void run()
{
   while(this.currentAccesses < this.totalAccesses)
   {
      //1: sleep
      try{Thread.sleep(1000 * (1 +
            (int)(Math.random() * MAX_SLEEP_TIME)));
      }catch(InterruptedException e)System.out.println(e);
      //2: get a computer
      this.getComputer();
      //3: use the computer
      try{Thread.sleep(1000 * (1 +
            (int)(Math.random() * MAX_USE_TIME)));
      }catch(InterruptedException e)System.out.println(e);
      this.currentAccesses++;
      //4: leave the computer
      this.leaveComputer();
   }}}
```

La Classe Student

```
class Student extends User
{
   private int myComputer;
   public Student(Tutor tutor, int userID)
   {
      super(tutor, userID);
   }

   public void getComputer()
   {
      this.myComputer = this.tutor.getComputer();
   }

   public void leaveComputer()
   {
      this.tutor.leaveComputer(this.myComputer);
   }
}
```

La Classe Graduate

```
class Graduate extends User
{
	private int reserved;
	public Graduate (Tutor tutor, int userID, int reserved)
	{
		super(tutor, userID);
		this.reserved = reserved;
	}

	public void getComputer()
	{
	   this.tutor.getReservedComputer(this.reserved);
	}
	public void leaveComputer()
	{
	   this.tutor.leaveReservedComputer(this.reserved);
	}
}
```

La Classe Professor

```
class Professor extends User
{
      public Professor(Tutor tutor, int userID)
      {
         super(tutor, userID);
      }

      public void getComputer()
      {
         this.tutor.getAllComputer();
      }

      public void leaveComputer()
      {
         this.tutor.leaveAllComputer();
      }
}
```

La classe Tutor: il monitor

- La classe Tutor sincronizza gli accessi al laboratorio.
- Variabili d'istanza:
  1. usedComputer: quanti computer sono occupati.
  2. professorWaiting: quanti professori sono in attesa del laboratorio.
  3. graduateWaiting[]: per ogni computer indica quanti tesisti sono in attesa di quel computer.
  4. computer[]: array di boolean per indicare se il computer i-esimo e' occupato.

Tutti i metodi della classe Tutor sono synchronized.

La Classe Tutor

```
class Tutor
{
   public static final int MAX_COMPUTER = 20;
   private int usedComputer = 0;
   private int professorWaiting = 0;
   private int graduateWaiting[] = new int[MAX_COMPUTER];
   private boolean computer[] = new boolean[MAX_COMPUTER];

   public Tutor()
   {
      for(int i = 0; i < MAX_COMPUTER; i++)
      {
         this.computer[i] = false;
         this.graduateWaiting[i] = 0;
      }
   }
```

La classe Tutor: i metodi invocati dagli studenti

Lo studente:

- invoca getComputer(): va in wait() se c'e' almeno un professore in coda o se tutti i computer liberi hanno almeno un tesista nella propria coda. Il metodo ritorna un intero che indica il computer libero.
- invoca leaveComputer(int i) quando ha finito di utilizzare il computer i. Questo metodo alla fine invoca notifyAll().
  - In tutor la notifyAll() e' utilizzata per indicare che un (o piu') computer e' stato lasciato libero: gli utenti in attesa vengono svegliati ma devono testare di nuovo le condizioni prima di procedere o ritornare in wait().

La Classe Tutor: i metodi invocati dagli studenti

```
public synchronized int getComputer()
{   while(true)
    {
        while(this.professorWaiting > 0 ||
                        (MAX_COMPUTER == this.usedComputer))
            waitComputer();
        for(int i = 0; i < MAX_COMPUTER; i++)
        {
            if(!this.computer[i] && this.graduateWaiting[i] == 0)
            {
                this.computer[i] = true;
                this.usedComputer++;
                return i;
            }
        }
        waitComputer();
    }
}
```

La Classe Tutor: i metodi invocati dagli studenti

```
public synchronized void leaveComputer(int i)
{
   this.computer[i] = false;
   this.usedComputer--;
   notifyAll();
}

//invocato da studenti, tesisti e professori
public synchronized void waitComputer()
{
   try
   {
      wait();
   }catch(InterruptedException e)
}
```

La classe Tutor: i metodi invocati dai tesisti

Il tesista:

- invoca getReservedComputer(int reserved) per richiedere il computer di indice reserved: va in wait() se c'e' almeno un professore in coda o il computer reserved e' occupato.
- invoca leaveReservedComputer(int reserved) quando ha finito per liberare il computer reserved. Questo metodo alla fine invoca notifyAll().

La Classe Tutor: i metodi invocati dai tesisti

```
public synchronized void getReservedComputer(int reserved)
{
   this.graduateWaiting[reserved]++;
   while(this.professorWaiting > 0 || this.computer[reserved])
      waitComputer();
   this.computer[reserved] = true;
   this.usedComputer++;
}

public synchronized void leaveReservedComputer(int reserved)
{
   this.computer[reserved] = false;
   this.usedComputer--;
   this.graduateWaiting[reserved]--;
   notifyAll();
}
```

La classe Tutor: i metodi invocati dai professori

Il professore:

- invoca getAllComputer(): va in wait() fino a quando il laboratorio non si libera.
- invoca leaveAllComputer() quando ha finito. Questo metodo alla fine invoca notifyAll().

La Classe Tutor: i metodi invocati dai professori

```
public synchronized void getAllComputer()
{
   this.professorWaiting++;
   while(this.usedComputer > 0)
      waitComputer();
   for(int i = 0; i < MAX_COMPUTER; i++)
      computer[i] = true;
   this.usedComputer = MAX_COMPUTER;
}
public synchronized void leaveAllComputer()
{
   for(int i = 0; i < MAX_COMPUTER; i++)
      computer[i] = false;
   this.usedComputer = 0;
   this.professorWaiting--;
   notifyAll();
}
}
```

La classe Lab

La classe Lab contiene il metodo main.

- Crea tre pool di thread.
- Crea l'oggetto monitor Tutor.
- Crea ed esegue tutti gli oggetti Student, Graduate e Professor, ai quali passa come riferimento l'oggetto monitor condiviso.
- Esegue la shutdown sui pool di thread.

La Classe Lab

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Lab
{
...
   public static void main(String[] args)
   {
      ...
      Tutor tutor = new Tutor();
      ExecutorService students =
         Executors.newFixedThreadPool(studentsNum);
      ExecutorService graduates =
         Executors.newFixedThreadPool(graduatesNum);
      ExecutorService professors =
         Executors.newFixedThreadPool(professorsNum);
```

La Classe Lab

```
        for(int i = 0; i < studentsNum; i++)
            students.execute(new Student(tutor, i));
        for(int i = 0; i < graduatesNum; i++)
            graduates.execute(new Graduate(tutor, i,
                              (int)(Math.random()*Tutor.MAX_COMPUTER)));
        for(int i = 0; i < professorsNum; i++)
            professors.execute(new Professor(tutor, i));
        students.shutdown();
        graduates.shutdown();
        professors.shutdown();
    }
}
```

Soluzioni Alternative: coda con priorita' e interrupt

- Mantenere nel monitor una coda con priorita'.
- Quando un computer si libera: interrupt() al primo delle coda.
- Problemi: un tesista puo' bloccare un utente su un computer non riservato.
  - Soluzione: se il tesista non ha bisogno del computer liberato, esegue una interrupt() al secondo utente con priorita' piu' alta, e cosi' via.

Soluzioni Alternative: synchronized statement o variabili di lock/condizione

- Invece di sincronizzare i metodi, si sincronizzano solo le sezioni critiche di codice.
  - Puo' portare a migliori prestazioni (maggiore concorrenza).
  - Gestione esplicita della unlock() (es. per evitare deadlock, all'interno di un finally).
- Variabili condizione: es. variabile che indica che il laboratorio e' stato liberato.
  - I professori eseguono la await() sulla variabile condizione, l'utente che verifica che il laboratorio e' libero esegue una signal() sulla stessa variabile.

Soluzioni Alternative: collezioni Java sincronizzate

- Non occorre una classe monitor (non si utilizzano metodi synchronized).
- La sincronizzazione e' gestita internamente dall'implementazione Java della specifica collezione.
- Es. BlockingQueue: implementazione thread-safe (lock interni). Paradigma consumatore/produttore:
  - Il produttore si blocca se la coda e' piena.
  - Il consumatore si blocca se la coda e' vuota.
- Per l'esercizio, ad es.: PriorityBlockingQueue<E> (gli elementi della coda devono implementare Comparable).