Lezione n.3
LPR A - Informatica
Threads:
Race Conditions, Locks
6/10/2008
Laura Ricci

TERMINAZIONE: THREAD DEMONI

- Thread Demone: fornisce un servizio, generalmente in background, fintanto che il programma è in esecuzione, ma non è considerato parte fondamentale di un programma
- Esempio: thread temporizzatori che scandiscono il tempo per conto di altri threads
- Quando tutti i thread non demoni hanno completato la lori esecuzione, il programma termina, anche se ci sono thread demoni in esecuzione
- Se ci sono thread non demoni in esecuzione, il programma non termina
  - Esempio: i thread attivati nel thread pool rimangono attivi anche se non esistono task da eseguire
- Si dichiara un thread demone invocando il metodo setdaemon(true), prima di avviare il thread
- Se un thread è un demone, allora anche tutti i threads da lui creti lo sono

TERMINAZIONE: THREAD DEMONI

```
public class simpledaemon extends Thread {
    public simpledaemon (  ) {
    setDaemon(true);
    start( );   }
    public void run(  ) {
            while(true) {
            try {
                sleep(100);
             } catch (InterruptedException e) {
                                throw new RuntimeException(e);}
            System.out.println("mi sono svegliato"+this);   }   }
```

TERMINAZIONE: THREAD DEMONI

```
public static void main(String[ ] args) {
    for(int i = 0; i < 5; i++)
        new simpledaemon( );
    } }
```

- il main crea 5 threads demoni
- ogni thread 'si addormenta' e si risveglia per un certo numero di volte, poi quando il main termina (non daemon thread), anche i threads terminano)
- Se pongo setDaemon(false), il programma non termina, i thread continuano ad 'addormentarsi' e risvegliarsi'

GRUPPI DI THREADS

- Alcuni programmi contengono un alto numero di threads
- Può essere utile classificare i threads in base alla loro funzionalità
- Esempio: un browser include molti threads il cui compito è scaricare le immagini contenute in una pagina web
- Se l'utente preme il pulsante stop( ), è comodo utilizzare un metodo per interrompere tutti i threads simultaneamente
- Gruppi di threads : consentono di classificare threads in base alle loro funzionalità in modo che si possa poi lavorare su tutti threads di un gruppo simultaneamente
- Esempio:

```
String groupName = ..... ;
ThreadGroup g = new  ThreadGroup(groupName);
Thread t =  new Thread(g, threadName);
.....
g.interrupt(   );
```

CONDIVISIONE DI RISORSE TRA THREADS

- Più threads attivati da uno stesso programma possono condividere un insieme di oggetti. Gli oggetti possono essere passati al costruttore del thread
- Esempio:

```
public class oggettocondiviso { ……..  }
public class condivisione extends Thread {
        oggettocondiviso ref;
        public condivisione (oggettocondiviso oc)  {ref=oc;};
        public void run (){   };
public static void main(String args[ ])
{   oggettocondiviso oc = new oggettocondiviso();
new condivisione(oc).start();
new condivisione(oc).start();  }
}
```

ACCESSO A RISORSE CONDIVISE

- L'interazione incontrollata dei threads sull'oggetto condiviso può produrre risultati non corretti
- Consideriamo il seguente esempio:
  - Definiamo una classe EvenValue che implementa un generatore di numeri pari.
  - Ogni oggetto istanza della classe ha un valore uguale ad un numero pari
  - Se il valore del numero è x, il metodo next( ), definito in EvenValue assegna il valore x+2
  - Si attivano un insieme di threads che condividono un oggetto di tipo EvenValue e che invocano concorrentemente il metodo next( )
  - Non si possono fare ipotesi sulla strategia di schedulazione dei threads

ACCESSO A RISORSE CONDIVISE

```
public  interface ValueGenerator {
                public int next( );     }


public class EvenValue implements ValueGenerator {
      private int currentEvenValue = 0;
      public int next( ) {
      ++currentEvenValue;
      Thread.yield ( );
      ++currentEvenValue;
      return currentEvenValue;}; }
```

Thread.*yield* ( ): "suggerisce" allo schedulatore di sospendere l'esecuzione del thread che ha invocato la yield( ) e di cedere la CPU ad altri threads

ACCESSO A RISORSE CONDIVISE

```
import java.util.concurrent.*;
public class threadtester implements Runnable {
      private Generator g;
      public threadtester(Generator g)  {this.g = g;}
      public void run( )
            {for (int i=0; i<5; i++)
                  {int val= g.next();
                  if (val %2 !=0) {System.out.println(Thread.currentThread( )+"errore"+val);}
                        else System.out.println(Thread.currentThread( )+"ok"+val); }}
public static void test(ValueGenerator g, int count)
      {ExecutorService exec= Executors.newCachedThreadPool();
      for (int i=0; i<count; i++)
            {exec.execute(new threadtester(g));}; exec.shutdown( ); }}
```

ACCESSO A RISORSE CONDIVISE

```
public class AccessTest {
public static void main(String args[ ])
      { EvenValue eg=new EvenValue( );
            threadtester.test(eg,2);} }
```

OUTPUT: Thread[pool-1-thread-1,5,main]ok2

Thread[pool-1-thread-2,5,main]ok4

Thread[pool-1-thread-1,5,main]errore7

Thread[pool-1-thread-2,5,main]errore9

Thread[pool-1-thread-1,5,main]ok10

Thread[pool-1-thread-2,5,main]errore13

Thread[pool-1-thread-1,5,main]ok14

Thread[pool-1-thread-1,5,main]errore17

Thread[pool-1-thread-2,5,main]ok18

Thread[pool-1-thread-2,5,main]ok20

RACE CONDITIONS: MOTIVAZIONI

- Perchè si è verificato l'errore?
- Supponiamo che il valore corrente di currentEvenValue sia 0.
- Il primo thread esegue il primo assegnamento

  ++currentEvenValue

  e viene quindi deschedulato, in seguito alla yield( ),currentEvenValue assume valore 1
- A questo punto si attiva il secondo thread, che esegue lo stesso assegnamento e viene a sua volta deschedulato, currentEvenValue assume valore 2
- Viene riattivato il primo thread, che esegue il secondo incremento, il valore assume valore 3
- ERRORE! : Il valore restituito dal metodo next( ) è 3.

RACE CONDITIONS: MOTIVAZIONI

- Nel nostro caso la race condition è dovuta alla possibilità che un thread invochi il metodo next( ) e venga deschedulato prima di avere completato l'esecuzione del metodo
- In questo modo la risorsa viene lasciata in uno stato inconsistente

  (un solo incremento per currentEvenValue)
- Classe Thread Safe: l'esecuzione concorrente dei metodi definiti nella classe non provoca comportamenti scorretti
- EvenValue non è una thread safe
- Per renderla thread safe occorre garantire che le istruzioni contenute all'interno del metodo EvenValue vengano eseguite in modo atomico o indivisibile o in mutua esclusione

RACE CONDITIONS: MOTIVAZIONI

- Race Condition: si può verificare anche nella esecuzione di una singola istruzione di assegnamento
- Consideriamo di nuovo l'istruzione che genera il successivo numero pari

  ++currentEvenValue;
- L'istruzione può essere elaborata come segue
  1) il valore di currentEvenValue viene caricato in un registro del processore
  2) si somma 1
  3) si memorizza il risultato in currentEvenValue
- Un thread T potrebbe eseguire i passi 1), 2) e poi venire deschedulato,
- Viene quindi schedulato un secondo thread Q che aggiorna che aggiorna currentValue correttamente
- T esegue il passo 3),considerando il vecchio valore di currentEvenValue e distruggendo così l'aggiornamento di Q

RACE CONDITIONS: ESEMPI

Un altro esempio di una classe non thread safe

```
public class LazyInitRace {
        private Expensive Object instance=null;
        public ExpensiveObject getInstance( ){
            if (instance == null)
                    instance = new ExpensiveObject();
             return instance:
        }}
```

- Lazy Initialization =
  - alloca un oggetto solo se non esiste già un'istanza di quell'oggetto
  - deve assicurare che l'oggetto venga inizializzato una sola volta

RACE CONDITIONS: ESEMPI

Un altro esempio di una classe non thread safe

```
public class LazyInitRace {
        private ExpensiveObject instance=null;
        public ExpensiveObject getInstance(){
        if (instance == null)
              instance = new ExpensiveObject();
        return instance:
        }  }
```

- Il precedente programma non è corretto perchè contiene una race condition
- Il thread A esegue getInstance, trova (instance==**null**), poi viene deschedulato. Il thread B esegue getInstance e trova a sua volta (instance== **null**). I due thread restituiscono due diverse istanze di ExpensiveObject
- L'applicazione può richiedere di allocare una sola istanza di ExpensiveObject

JAVA: MECCANISMI DI LOCK

- Occorre definire un insieme di meccanismi per garantire che un metodo (es: next) venga eseguito in mutua esclusione quando invocato sullo stesso oggetto
- Se un thread T esegue un metodo , nessun altro thread può eseguire lo stesso metodo sullo stesso oggetto fino a che T ha terminato l'esecuzione di quel metodo
- JAVA offre un meccanismo implicito di locking (intrinsic locks) che consente di assicurare la atomicità di porzioni di codice eseguite in modo concorrente sullo stesso oggetto

JAVA: MECCANISMO DELLE LOCK

- Sincronizzazione di un blocco di codice

```
synchronized (obj)
    { // blocco di codice che accede o modifica l'oggetto
    }
```

- L'oggetto obj può essere quello su cui è stato invocato il metodo che contiene il codice (this) oppure un altro oggetto
- Il thread che esegue il blocco sincronizzato deve acquisire la lock su l'oggetto obj
- La lock viene rilasciata nel momento in cui il thread termina l'esecuzione del blocco (es: return, throw, esecuzione dell'ultima istruzione del blocco)
- Iniziando l'esecuzione del blocco sincronizzato si acquisice implicitamente la lock( ), uscendo dal blocco si rilascia implicitamente la lock stessa

JAVA: RENDERE ATOMICO IL METODO NEXT

```
public class EvenValue implements ValueGenerator{
    private int currentEvenValue = 0;
    public int next(  ){
            synchronized(this){
                ++currentEvenValue;
                Thread.yield ( );
                ++currentEvenValue;
                return currentEvenValue;}};}
```

- Questa modifica consente di evitare le race conditions
- ATTENZIONE: in generale non è consigliabile l'inserimento di una istruzione che blocca il thread che la invoca (sleep( ), yield( ),....) all'interno di un blocco sincronizzato
- Infatti il thread che si blocca non rilascia la lock ed impedisce ad altri threads di invocare il metodo next( ) sullo stesso oggetto

JAVA 1.5: LOCKS ESPLICITE

- A partire da JAVA 1.5 è possibile definire ed utilizzare oggetti di tipo lock( )

```
import java.util.concurrent.locks.*;
class X {
    private final ReentrantLock mylock = new ReentrantLock( );

    // .
    .public void m(  ) {
    mylock.lock(   );  // block until condition holds
    try {
            // ... method body

        } finally {lock.unlock( )   } } }
```

JAVA 1.5: LOCK ESPLICITE

```
import java.util.concurrent.locks.*;
public class EvenGenerator implements ValueGenerator {
    private int currentEvenValue = 0;
    ReentrantLock evenlock=new ReentrantLock( );
    public int next( ) {
        try {
            evenlock.lock( );
            ++currentEvenValue;
            Thread.yield ( );
            ++currentEvenValue;
            return currentEvenValue;
        } finally {evenlock.unlock( );}};}
```

JAVA: MECCANISMO DELLE LOCK

- Locks esplicite: si definisce un oggetto di tipo ReentrantLock( )
  - Quando un thread invoca il metodo lock( ) su un oggetto di tipo ReentrantLock( ), il thread rimane bloccato se qualche altro thread ha già acquisito la lock( ) sullo stesso oggetto
  - Quando un thread invoca la unlock( ) su un oggetto di tipo ReentrantLock, uno dei thread eventualmente bloccati su quell'oggetto viene risvegliato

- Lock Implicite su metodi
  - Definite associando al metodo la parola chiave synchronized
  - Equivale a sincronizzare tutto il blocco di codice che corrisponde al corpo del metodo
  - L'oggetto su cui si acquisisce la lock è quello su cui viene invocato il metodo

I METODI SYNCHRONIZED

- La parola chiave synchronized nella intestazione di un metodo ha l'effetto di serializzare gli accessi al metodo

```
public synchronized int EvenValue (int val)
```

- Se un thread sta eseguendo il metodo next( ), nessun altro thread eseguirà lo stesso codice sullo stesso oggetto finchè il primo thread non termina l'esecuzione del metodo
- Implementazione:
  - Supponiamo che il metodo M synchronized appartenga alla classe *C*
  - ad ogni oggetto *O* istanza di *C* viene associata una lock *L(O)*
  - quando un thread T invoca M su O, T tenta di acquisire L(O), prima di iniziare l'esecuzione di M. Se T non acquisisce L(O), si sospende

I METODI SYNCHRONIZED

- Se rendiamo synchronized il metodo next(), l'output che otteniamo sarà

```
Thread[pool-1-thread-1,5,main]ok2
Thread[pool-1-thread-2,5,main]ok4
Thread[pool-1-thread-1,5,main]ok6
Thread[pool-1-thread-2,5,main]ok8
Thread[pool-1-thread-1,5,main]ok10
Thread[pool-1-thread-2,5,main]ok12
Thread[pool-1-thread-1,5,main]ok14
Thread[pool-1-thread-2,5,main]ok16
Thread[pool-1-thread-1,5,main]ok18
Thread[pool-1-thread-2,5,main]ok20
```

I METODI SYNCHRONIZED

- Importante: la lock( ) è associata all'istanza di un oggetto, non al metodo o alla classe (a meno di metodi statici che vedremo in seguito)
- Diversi metodi sincronizzati invocati sull'istanza dello stesso oggetto competono per la stessa lock( ), quindi risultano mutuamente esclusivi
- Metodi sincronizzati che operano su istanze diverse dello stesso oggetto possono essere eseguiti in modo concorrente
- All'interno della stessa classe possono comparire contemporaneamente metodi sincronizzati e non (anche se raramente)
  - I metodi non sincronizzati possono essere eseguiti in modo concorrente
  - In ogni istante, su un certo oggetto, possono essere eseguiti concorrentemente più metodi non sincronizzati e solo uno dei metodi sincronizzati della classe

I METODI SYNCHRONIZED

L'esempio seguente istanzia due istanze diverse dell' oggetto EvenValue( ) e le passa a due thread distinti. Si considera la versione non sincronizzata del metodo next( )

```
public class EvenValue implements Generator{
private int currentEvenValue = 0;
      public int next( ){ ++currentEvenValue; ++currentEvenValue;
                return currentEvenValue; }; }
public class synchrotest {
          public static void main(String args[  ])
          {EvenValue eg1=new EvenValue();
          EvenValue eg2=new EvenValue();
          threadtester1.test(eg1,eg2);}}
```

I METODI SYNCHRONIZED

```
import java.util.concurrent.*;
import java.util.Random;
public class threadtester1 implements Runnable{
private Generator g;
public threadtester1 (Generator g)  {this.g = g;}
public void run( )
      {for (int i=0; i<5; i++)
            {int val= g.next();
                  if (val %2 =0)
                        {System.out.println(Thread.currentThread()+"errore"+val);}
                  else System.out.println(Thread.currentThread()+"ok"+val);
                  int x = (int)Math.random() * 1000;
            try {Thread.sleep(x);} catch (Exception e) {   };  }}
```

I METODI SYNCHRONIZED

```
public static void test(Generator g1, Generator g2)
    {ExecutorService exec= Executors.newCachedThreadPool();
    exec.execute(new tester(g1));
    exec.execute(new tester(g2)); } }
```

OUTPUT: il risultato è corretto anche se next() non è sincronizzato

```
Thread[pool-1-thread-1,5,main]ok2
Thread[pool-1-thread-2,5,main]ok2
Thread[pool-1-thread-2,5,main]ok4
Thread[pool-1-thread-2,5,main]ok6
Thread[pool-1-thread-2,5,main]ok8
Thread[pool-1-thread-2,5,main]ok10
Thread[pool-1-thread-1,5,main]ok4
Thread[pool-1-thread-1,5,main]ok6
Thread[pool-1-thread-1,5,main]ok8
Thread[pool-1-thread-1,5,main]ok10
```

LOCK RIENTRANTI

- Le locks intrinseche di JAVA sono rientranti, ovvero la lock( ) su un oggetto O viene associata al thread che accede ad O.

- se un thread tenta di acquisire una lock che già possiede, la sua richiesta ha successo

- Ovvero....un thread può invocare metodo sincronizzato S su un oggetto O e all'interno di S vi può essere l'invocazione ad un altro metodo sincronizzato su O e così via

- Il meccanismo delle lock rientranti favorisce la prevenzione di situazioni di deadlock

LOCK RIENTRANTI

- Implementazione delle lock rientranti
  - Ad ogni lock viene associato un contatore ed un identificatore di thread
  - Quando un thread T acquisisce una lock( ), la JVM alloca una struttura che contiene l'identificatore T e un contatore, inizializzato a 0
  - Ad ogni successiva richiesta della stessa lock( ), il contatore viene incrementato mentre viene decrementato quando il metodo termina
- La lock( ) viene rilasciata quando il valore del contatore diventa 0

REENTRANT LOCK: UN ESEMPIO

```
public class ReentrantExample {
    public synchronized void doSomething( ) {
                        ..............} }

public class ReentrantExtended extends ReentrantExample{
    public synchronized void doSomething( ){
        System.out.println(toString( )+":chiamata a doSomething");
        super.doSomething();
        } }
```

- La chiamata super.doSomething( ) si bloccherebbe se la lock( ) non fosse rientrante ed il programma risulterebbe bloccato

MUTUA ESCLUSIONE: RIASSUNTO

- Interazione implicita tra diversi threads: i thread accedono a risorse condivise.
- Per mantenere consistente l'oggetto condiviso occorre garantire la mutua esclusione su di esso.
- La mutua esclusione viene garantita associando una lock( ) ad ogni oggetto
- I metodi synchronized garantiscono che un thread per volta possa eseguire un metodo sull'istanza di un oggetto e quindi garantiscono la mutua esclusione sull'oggetto

ESERCIZIO: RACE CONDITIONS

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