



**Lezione n.3**  
**LPR A - INFORMATICA**  
**Threads: Mutua Esclusione,**  
**Sincronizzazione esplicita**  
**3/3/2008**  
**Laura Ricci**

# 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 un oggetto `EvenGenerator`. L'oggetto rappresenta un generatore di numeri pari
  - L'oggetto ha valore iniziale 0.
  - Supponiamo che l'oggetto abbia valore  $x$ . Il metodo `next()` definito da `EvenGenerator` consente di ottenere  $x+2$
  - Si attivano un insieme di threads che condividono un oggetto di tipo `EvenGenerator` e che invocano *concorrentemente* il metodo `next()`
  - Non si possono fare ipotesi sulla strategia di schedulazione dei threads: ogni thread può essere deschedulato in ogni istante, durante la sua esecuzione

# ACCESSO A RISORSE CONDIVISE

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

```
public class EvenGenerator implements Generator {  
    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 mandare in esecuzione 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(Generator g, int count)
        {ExecutorService exec= Executors.newCachedThreadPool();
        for (int i=0; i<count; i++)
            {exec.execute(new threadtester(g));} }}
```

# ACCESSO A RISORSE CONDIVISE

```
public class AccessTest {  
    public static void main(String args[ ]) {  
        { EvenGenerator eg=new EvenGenerator();  
            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

# ACCESSO A RISORSE CONDIVISE

- L'accesso **incontrollato** alla risorsa condivisa EvenGenerator ha prodotto un comportamento scorretto del programma
- **Race Condition**: Si verifica quando il risultato di una computazione dipende da un particolare interleaving dei threads a tempo di esecuzione
- Nel nostro caso la race condition è dovuta alla possibilità che un thread invochi il metodo **next()** e venga descheduled prima di avere completato l'esecuzione del metodo
- In questo modo la risorsa viene lasciata in uno stato inconsistente (un solo incremento per currentEvenValue )
- Il metodo Evengenerator contiene una sequenza di operazioni che devono essere eseguite in modo **atomico** o **indivisibile** o in **mutua esclusione**

# 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 =
  - pospone l'inizializzazione di un oggetto fino a che non è realmente necessario
  - 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 descheduled. 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 istanze di `ExpensiveObject`

# JAVA: MECCANISMI DI LOCK

- Occorre definire un insieme di meccanismi per garantire che un metodo (next, getInsyance) 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)
- L'unico modo per acquisire una lock intrinseca su un oggetto è quello di eseguire un blocco sincronizzato, l'unico modo di rilasciare una lock è quella di uscire da un blocco sincronizzato

# JAVA: MECCANISMO DELLE LOCK

- Sincronizzazione di metodi:
  - 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 seguenti frammenti di codice sono equivalenti

```
public synchronized void set(int x)
{this.x = x}
```

```
public void set(int x)
synchronized(this)
{this.x=x }
```

# I METODI SYNCHRONIZED

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

`public synchronized int EvenGenerator (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 `istanza di C` (oggetto `O`) viene associata una `lock L`
  - quando un thread `T` invoca `M` su `O`, `T` tenta di acquisire `L`, prima di iniziare l'esecuzione di `M`. Se `T` non acquisisce `L`, 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
```

# 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

# METODI SYNCHRONIZED

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

```
public class EvenGenerator implements Generator{
    private int currentEvenValue = 0;

    public int next( ){ ++currentEvenValue; ++currentEvenValue;
                        return currentEvenValue; }; }

public class synchrotest {
    public static void main(String args[ ])
    {EvenGenerator eg1=new EvenGenerator();
    EvenGenerator eg2=new EvenGenerator();
    threadtester1.test(eg1,eg2);}}
```



# 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) { }; }}

}
```

# 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
```

# REENTRANT LOCKS

- 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

# REENTRANT LOCKS

- 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**

# THREADS COOPERANTI: IL MONITOR

- L'**interazione esplicita** tra threads avviene in un linguaggio ad oggetti come JAVA mediante l'utilizzo di **oggetti condivisi**
- **Esempio:** produttore/consumatore il **produttore P** produce un nuovo valore e lo comunica ad un thread **consumatore C**
- Il valore prodotto viene incapsulato in un **oggetto condiviso** da P e da C, ad esempio una **coda** che memorizza i messaggi scambiati tra P e C
- La mutua esclusione sull'oggetto condiviso è garantita dall'uso di metodi **synchronized**, ma...non è sufficiente garantire **sincronizzazioni esplicite**
- E' necessario introdurre costrutti per **sospendere** un thread T quando una condizione C non è verificata e per **riattivare** T quando diventa vera
- Esempio: il produttore si sospende se il buffer è pieno, si riattiva quando
  - c'è una posizione libera

# THREAD COOPERANTI: IL MONITOR

- Monitor= Classe di oggetti utilizzabili da un insieme di threads
- Come ogni classe, il monitor contiene un insieme di campi ed un insieme di metodi
- La mutua esclusione può essere garantita dalla definizione di metodi synchronized. Un solo thread per volta si "all'interno del monitor"
- E' necessario inoltre
  - definire un insieme di **condizioni sullo stato** dell'oggetto condiviso
  - implementare meccanismi di sospensione/riattivazione dei threads sulla base del valore di queste condizioni
  - Implementazioni possibili:
    - definizione di **variabili di condizione**
    - metodi per **la sospensione** su queste variabili
    - definizione di **code** associate alle variabili in cui memorizzare i threads sospesi



# THREADS COOPERANTI: IL MONITOR

## JAVA

- non supporta variabili di condizione
- assegna al programmatore il compito di gestire le condizioni mediante variabili del programma
- definisce meccanismi che consentono ad un thread
  - di sospendersi `wait( )` in attesa che sia verificata una condizione
  - di segnalare `notify( )` , `notifyall ( )` ad un altro/ad altri threads sospesi che una certa condizione è verificata
- implementa per ogni oggetto condiviso  $O$  una coda in cui vengono memorizzati tutti i **threads sospesi** in attesa del verificarsi di una condizione sullo stato di  $O$ .
- Per ogni oggetto implementa due code:
  - una per i threads in attesa di acquisire la `lock( )`
  - per i threads in attesa del verificarsi di una condizione

# THREADS COOPERANTI: IL MONITOR

- Produttore/Consumatore: due thread si scambiano dati attraverso un oggetto condiviso **buffer**
- Ogni thread deve acquisire la `lock()` sull'oggetto `buffer`, prima di inserire/prelevare elementi
- Una volta acquisita la `lock()`
  - il consumatore controlla se ci sono elementi nel `buffer`, ed eventualmente si sospende se il `buffer` è vuoto, altrimenti inserisce un elemento nel `buffer` ed eventualmente risveglia il consumatore.
  - il produttore controlla se c'è almeno una posizione libera nel `buffer`, in caso contrario si sospende. Quando inserisce un elemento dal `buffer`, controlla se vi sono eventuali consumatori in attesa

# THREADS COOPERANTI: IL MONITOR

## MONITOR



Coda dei threads in attesa della lock

Coda dei threads in attesa del verificarsi di una condizione

# I METODI WAIT/NOTIFY

Metodi invocati sull'oggetto condiviso (se non compare il riferimento all'oggetto, l'oggetto implicito riferito è `this`)

- `void wait()` sospende il thread in attesa che sia verificata una condizione
- `void wait(long timeout)` sospende per al massimo timeout millisecondi
- `void notify()` notifica `ad un thread in attesa` il verificarsi di una certa condizione su
- `void notifyall()` notifica `a tutti i threads in attesa` il verificarsi di una condizione

tutti questi metodi

- fanno parte della classe `Object` (tutte le classi ereditano da `Object`,...)
- per invocare questi metodi occorre aver acquisito la lock (`synchronized`) sull'oggetto  
⇒ devono essere invocati all'interno di un metodo o di un blocco sincronizzato

# UN ESEMPIO: L'AUTOCARROZZERIA

- In un'autocarrozzeria si devono verniciare alcune auto.
- Per ogni auto si devono alternare un certo numero di fasi di **ceratura** con fasi di **lucidatura**
- Esistono due operai che sono specializzati, rispettivamente, nel processo di ceratura ed in quello di lucidatura
- Si deve simulare il comportamento dall'autocarrozzeria in JAVA
  - Si attivino due thread che simulino il comportamento dei due operai
  - Si definisce l'oggetto condiviso **Auto**, tramite cui interagiscono i due thread, che definisca i metodi per la corretta sincronizzazione tra i due thread
  - Il programma Autocarrozzeria deve creare un oggetto Auto, attivare i due thread, passando a ciascuno di essi l'oggetto condiviso, lasciarli lavorare per un certo numero di secondi, quindi interromperli

# UN ESEMPIO: L'AUTOCARROZZERIA

```
public class Car {  
    private boolean dacerare = true;  
  
    public synchronized void cerata ()  
    {dacerare = false;  
    notify();  
    }  
  
    public synchronized void possocerare( ) throws InterruptedException{  
        while (dacerare == false)  
            try { wait();} catch (InterruptedException e){ }  
    }  
}
```

# UN ESEMPIO: L'AUTOCARROZZERIA

```
public synchronized void lucidata()
```

```
{ dacerare = true;
```

```
  notify();
```

```
}
```

```
public synchronized void possolucidare () throws InterruptedException{
```

```
  { while (dacerare==true)
```

```
    try { wait();} catch (InterruptedException e) {throw e;};
```

```
  } }
```

```
}
```

# UN ESEMPIO:L' AUTOCARROZZERIA

```
import java.util.concurrent.*;

public class Ceratura implements Runnable{

    private Car car;boolean vai=true;

    public Ceratura (Car c){ car = c;}

    public void run(){

        while (vai ) {try {

            System.out.println ("Ceratura!!");

            TimeUnit.MILLISECONDS.sleep(200);

            car.cerata();

            car.possocerare();

        } catch (InterruptedException e)

            {System.out.println("Fine del Thread Ceratura"); vai=false;}}}}
```



# UN ESEMPIO: L'AUTOCARROZZERIA

```
import java.util.concurrent.*;

public class Lucidatura implements Runnable{

    private Car car;boolean vai=true;

    public Lucidatura (Car c) {car=c;};

    public void run( ) {

        while (vai) {

            try {car.possolucidare();

                System.out.println("Lucidatura!!!");

                TimeUnit.MILLISECONDS.sleep(200);

                car.lucidata(); }

            catch (InterruptedException e) {System.out.println ("Fine del

                Thread Lucidatura");vai=false;} }

        } }

    }
```

# UN ESEMPIO:L' AUTOCARROZZERIA

```
import java.util.concurrent.*;

public class Autocarrozzeria {

    public static void main (String args[ ]) throws Exception
    {Car car = new Car();
    ExecutorService exec= Executors.newFixedThreadPool(2);
    exec.execute(new Ceratura(car));
    exec.execute(new Lucidatura(car));
    TimeUnit.SECONDS.sleep(3);
    exec.shutdownNow(); } }
```

# L' AUTOCARROZZERIA: OUTPUT DEL PROGRAMMA

Ceratura!!

Lucidatura!!!

Ceratura!!

Lucidatura!!!

Ceratura!!

Lucidatura!!!

Ceratura!!

Lucidatura!!!

Ceratura!!

Lucidatura!!!

Fine del Thread Lucidatura

Fine del Thread Ceratura

# WAIT E NOTIFY

wait

- rilascia la lock( ) sull'oggetto prima di sospendere il thread
- in seguito ad una notifica, può riacquisire la lock()

notify( )

- risveglia uno dei thread nella coda di attesa di quell'oggetto

notifyAll( )

- risveglia tutti i threads in attesa
- i thread risvegliati competono per l'acquisizione della lock( )
- i thread verranno eseguiti uno alla volta, quando riusciranno a riacquisire la lock( ) sull'oggetto

In tutti i casi la lock() è quella dell'oggetto acceduto con il metodo synchronized

# WAIT E NOTIFY

- Il metodo `wait` permette di attendere un cambiamento su una condizione "fuori dal monitor", in modo passivo
- Evita il controllo ripetuto di una condizione (**polling**)
- A differenza di `sleep()` e di `yield()` rilascia la lock sull'oggetto
- L'invocazione di un metodo `wait`, `notify()`, `notifyall()` fuori da un metodo `synchronized` solleva l'eccezione `IllegalMonitorException()`
- Infatti prima di invocare questi metodi occorre aver acquisito la lock su un oggetto condiviso
- **Nota Bene:** nell'esempio avrei potuto usare un `if` nei metodi `possolucidare()` e `possocerare()`. In generale è necessario **ricontrollare la condizione**, dopo che si è stati svegliati da una `wait()`

# CONFRONTO TRA NOTIFY E NOTIFYALL

Differenze tra `notify( )` e `notifyall( )`

- `notify( )` riattiva uno dei tasks associati alla coda associata all'oggetto su cui si invoca la `notify( )`.
- E' errato dire, genericamente: la `notifyAll` riattiva tutti i tasks in attesa.
- `notifyAll( )` riattiva tutti i tasks in attesa su un oggetto, l'oggetto è quello su cui è invocata la `notifyall( )` (this nel caso in cui l'oggetto non è esplicitamente riferito).

# CONFRONTO NOTIFY E NOTIFYALL

```
public class sincronizza {  
  
    // definizione dell'oggetto condiviso; per semplicità la omettiamo  
    synchronized void attendi( ) throws InterruptedException {  
    try { //testa una condizione sull'oggetto; per semplicità la omettiamo  
        wait( );  
        System.out.println(Thread.currentThread()+" ");  
    } catch(InterruptedException e){throw e;};}  
  
    synchronized void risveglia( )    {notify();    }  
    synchronized void risvegliatutti( )    {notifyAll();}  
  
}
```

# CONFRONTO TRA NOTIFY E NOTIFYALL

```
public class ThreadBlocco implements Runnable {
    sincronizza mys;boolean vai=true;
    public ThreadBlocco (sincronizza s)
        {mys=s;}
    public void run( ){
        while (vai) {
            try{
                mys.attendi( );
            }catch (InterruptedException e){System.out.println("sono interrotto"+
                Thread.currentThread( ));
            }
        }
    }
}
```



# CONFRONTO TRA NOTIFY E NOTIFYALL

```
import java.util.*;
import java.util.concurrent.*;
public class provanotify {
public static void main (String args[ ]) throws Exception
{
    sincronizza mys1 = new sincronizza();
    sincronizza mys2 = new sincronizza();
    ExecutorService exec = Executors.newCachedThreadPool();
    for (int i=0; i<4; i++)
        exec.execute(new ThreadBlocco(mys1));
    exec.execute(new ThreadBlocco(mys2));
}
```

# CONFRONTO TRA NOTIFY E NOTIFYALL

```
boolean turno=true;
for (int i=0; i<5; i++)
{ if (turno){
    System.out.println("notify");
    mys1.risveglia();
    turno=false;
    Thread.sleep(500);}
else { System.out.println("notifyall");
    mys1.risvegliatutti();
    turno=true;
    Thread.sleep(500);} }
System.out.println("ora risveglio l'ultimo thread");
mys2.risvegliatutti();  exec.shutdownNow( ); } }
```

# CONFRONTO TRA NOTIFY E NOTIFYALL

## notify

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

## notifyall

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

Thread[pool-1-thread-3,5,main]

Thread[pool-1-thread-4,5,main]

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

## notify

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

## notifyall

Thread[pool-1-thread-4,5,main]

Thread[pool-1-thread-3,5,main]

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

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

# CONFRONTO TRA NOTIFY E NOTIFYALL

ora risveglio l'ultimo task

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

sono stato interrotto Thread[pool-1-thread-5,5,main]

sono stato interrotto Thread[pool-1-thread-4,5,main]

sono stato interrotto Thread[pool-1-thread-3,5,main]

sono stato interrotto Thread[pool-1-thread-2,5,main]

sono stato interrotto Thread[pool-1-thread-1,5,main]

# ESERCIZIO

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 installato 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

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.