SISTEMI OPERATIVI

(MODULO DI INFORMATICA II)

Sincronizzazione in Java

(Java **object lock** e segnali **wait-notify-notifyAll**)

Prof. Luca Gherardi

Prof.ssa Patrizia Scandurra (anni precedenti)

Università degli Studi di Bergamo

a.a. 2012-13

La gestione dei thread in Java

- Creazione e avvio
- Terminazione e cancellazione dei thread
- **La gestione dei segnali**
- **Sincronizzazione**
  - **Java object lock e segnali wait-notify-notifyAll**
- Dati specifici dei thread
- Schedulazione
- Gruppi di thread

Interazioni tra thread

- Due tipi:
  - ***cooperativo***, per lo scambio di info
  - ***competitivo***, per l'accesso ad oggetti comuni
- Per entrambi i tipi di interazione sono necessari due diversi meccanismi di sincronizzazione:
  1. **Sincronizzazione indiretta** o di **mutua esclusione**
  2. **Sincronizzazione diretta**
- Supportati dalla JVM ed introdotti a "livello di linguaggio":
  1. Meccanismo Java degli **"object lock"**
  2. Meccanismo Java **"wait-notify"**(gestione dei segnali)

Sincronizzazione indiretta (o di mutua esclusione)

- **Meccanismo di "object lock"**: meccanismo interno alla JVM che **associa un lock ad ogni oggetto da condividere**

- Applicabile:
  - a livello di **blocchi di codice**: **blocchi sincronizzati** eseguiti in modo mutuamente esclusivo rispetto ad altre esecuzioni dello stesso blocco o di altri blocchi sincronizzati sullo stesso oggetto

  - a livello di **metodi: metodi sincronizzati** il cui corpo è un unico "blocco sincronizzato" mediante il lock dell'oggetto su cui il metodo viene invocato

Esempio di riferimento

- Consideriamo di condividere oggetti della classe:

```
class OggettoCondiviso {
    static int iQuanti = 0;
    int conta;

    OggettoCondiviso(int conta) {
        this.conta = conta;
        iQuanti++;
    }

    void decrem(int dec) {
        conta-=dec;
    }
}
```

La keyword **static** assicura la presenza di una sola istanza

Sincronizzazione indiretta – blocco sincronizzato (1)

```
class MyRunnable implements Runnable {
    int iNum;
    OggettoCondiviso so;

    MyRunnable(int iNum, OggettoCondiviso so) {
        this.iNum = iNum;
        this.so = so;
    }

    public void run() {

        synchronized(so) {
            so.conta = so.conta + 7;
        }
    }
}
```

Sincronizzazione indiretta sull'oggetto so

Il lock è ottenuto sull'oggetto so

Sincronizzazione indiretta – blocco sincronizzato (2)

```
class MyRunnable implements Runnable {
    int iNum;
    OggettoCondiviso so;

    MyRunnable(int iNum, OggettoCondiviso so) {
        this.iNum = iNum;
        this.so = so;
    }

    public void run() {

        synchronized(OggettoCondiviso.class) {
            OggettoCondiviso.iQuanti++;
        }
    }
}
```

Sincronizzazione **indiretta** sulla classe OggettoCondiviso per campi static

Sincronizzazione indiretta – metodi sincronizzati (1)

```
class OggettoCondiviso {
    static int iQuanti = 0;
    int conta;

    OggettoCondiviso(int conta) {
        this.conta = conta;
        iQuanti++;
    }

    synchronized void decrem(int dec) {
        conta-=dec;
    }
}
```

Sincronizzazione **indiretta** sul metodo decrem: l'oggetto su cui avviene la sincronizzazione e' quello su cui viene invocato il metodo decrem

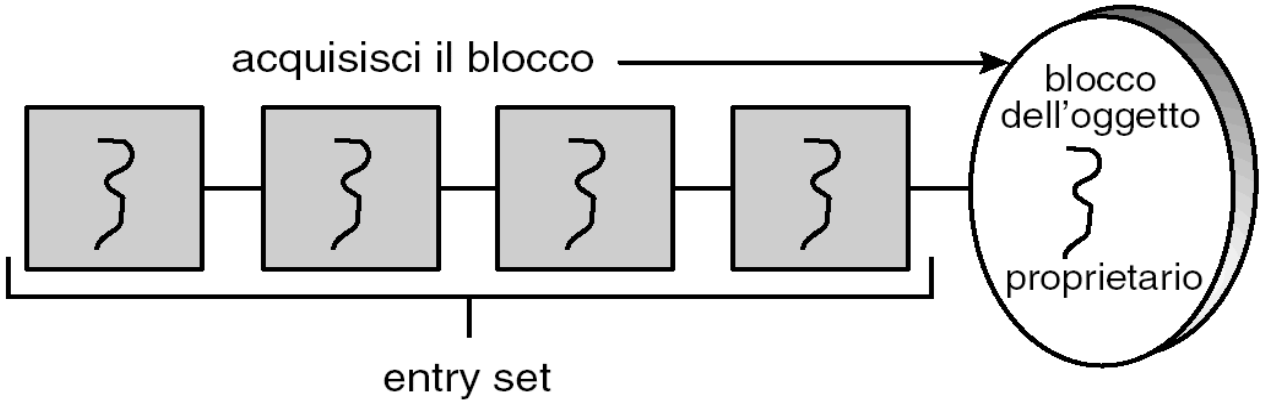
Sincronizzazione indiretta – metodi sincronizzati (2)

```
class OggettoCondiviso {
    static int iQuanti = 0;
    int conta;

    OggettoCondiviso(int conta) {
        this.conta = conta;
        iQuanti++;
    }

    static synchronized void decrQuanti() {
        iQuanti--;
    }
}
```

Sincronizzazione indiretta sul metodo static decrQuanti

Affermazioni sulla sincronizzazione indiretta (1)

- Ogni oggetto ha associato a sè un singolo **lock**
- Quando **un metodo/blocco** è **dichiarato *synchronized***, la sua esecuzione **richiede il possesso del lock sull'oggetto interessato**
  - Se quest'ultimo è già posseduto da un altro thread, il thread che chiama il metodo synchronized si blocca e viene messo nell'**entry set** per il lock di quell'oggetto
- Il lock sull'oggetto viene rilasciato quando il thread che lo detiene esce dal metodo/blocco

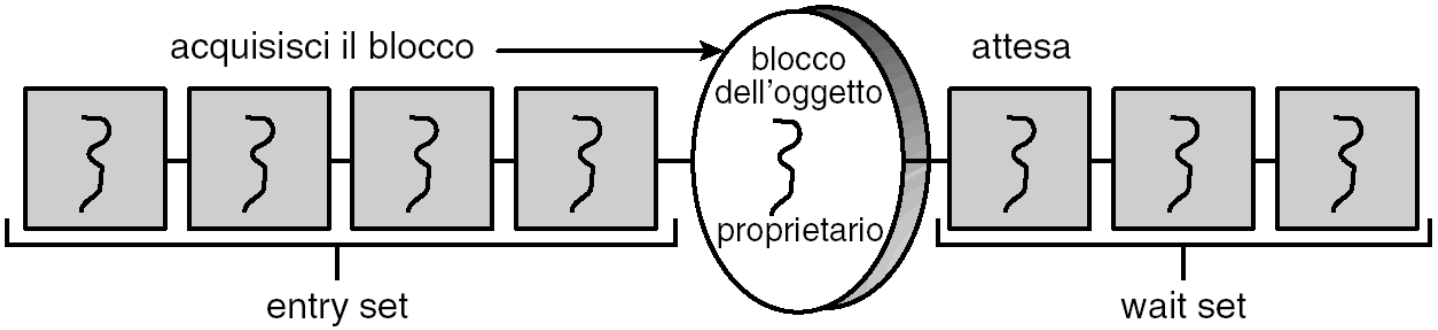


Affermazioni sulla sincronizzazione indiretta (2)

- Un metodo sincronizzato può invocare un altro metodo sincronizzato sullo stesso oggetto senza bloccarsi
  - al fine di evitare condizioni di blocco critico (deadlock)
- Due diversi metodi, uno sincronizzato ed uno no, possono essere eseguiti concorrentemente sullo stesso oggetto

Sincronizzazione diretta

- **Meccanismo di "wait-notify"**: oltre al lock e alla coda entry set, la JVM associa implicitamente **ad ogni oggetto** anche **una coda di attesa dei thread** detta ***wait set***
- I thread entrano ed escono da questa coda utilizzando rispettivamente i due metodi: **wait()** e **notify()**
  - invocabili solo all'interno di un blocco o metodo sincronizzato (cioè quando il thread detiene il lock dell'oggetto)



Sincronizzazione diretta – wait()

```
class MyRunnable implements Runnable {
    int iNum;
    OggettoCondiviso so;

    public void run() {

        try {
            synchronized(so) {
                while (so.conta < 1)
                    so.wait();

                so.conta-=2;
            }
        }
        catch (InterruptedException e) {
            System.out.println("uscito dalla wait per interruzione");
        }
    }
}
```

Sospende il thread fino a che un altro thread non esegue una notify sullo stesso oggetto

Un altro thread ha lanciato un interrupt() a questo thread

Sincronizzazione diretta – notify()

```
class MyRunnable implements Runnable {
    int iNum;
    OggettoCondiviso so;

    public void run() {
        synchronized(so) {
            so.increm(2);
            so.notify();
        }
    }
}
```

risveglia un thread che aveva eseguito una wait sullo stesso oggetto

notifyAll() risveglia tutti i threads che avevano eseguito una wait sullo stesso oggetto

Il metodo wait()

- Quando un thread chiama il metodo `wait()`, accadono i seguenti fatti:
  1. il thread rilascia il lock dell'oggetto
  2. l'esecuzione del thread viene sospesa
  3. il thread viene inserito nel wait set dell'oggetto

Il metodo notify()

Quando un thread chiama il metodo `notify()`

- solitamente alla fine del blocco/metodo sincronizzato

questa chiamata:

1. seleziona arbitrariamente un thread *T* dal wait set
   - se il wait set è vuoto, non viene eseguita nessuna azione
2. sposta *T* dal wait set all'entry set
3. cambia lo stato di *T* da bloccato ad eseguibile
4. Ora *T* è selezionabile per competere per il lock con gli altri thread

Java threads – Producer-consumer

- La soluzione al problema del buffer limitato ricorre ai meccanismi di sincronizzazione di Java wait() e notify()

Soluzione tramite "memoria condivisa"

<esempio **boundedbuffer** >

Notifiche multiple

- `notify()` sveglia un **singolo thread <u>casualmente</u>** tra quelli in attesa nel wait set e lo pone nell'entry set
  - questo può non essere il thread scelto dallo sviluppatore
  - Java non permette allo sviluppatore di specificare il thread che deve essere scelto!

- `notifyAll()` **sveglia TUTTI i thread dal wait set** e li mette nell'entry set.
  - Questo permette ai thread di decidere chi fra di loro debba continuare l'esecuzione
  - `notifyAll()` è una strategia più sicura, appropriata per situazioni in cui più thread possono essere nel wait set di oggetto

Metodi “deprecati”

- stop()
- suspend()
- resume()
- destroy()

Differenza tra Thread.sleep() e wait()

- `Thread.sleep()` pone il thread in esecuzione nello stato "Non runnable" per un certo quantitativo di tempo
  - Il thread non rilascia gli eventuali lock che aveva acquisito
  - Se ad esempio il thread è all'interno di un metodo synchronized nessun altro thread potrà accedere quel metodo
- `object.wait()` pone il thread in esecuzione nello stato "non runnable" cosi come `sleep()`, ma con una differenza
  - La chiamata del metodo `wait()` rilascia il lock acquisito all'ingresso del metodo synchronized