Sincronizzazione

Thursday, 16 March 2023 08:50

Quando iniziamo a fare concorrenza, si siamo più veloci però nuove termonologie entrano:

- Cooperazione, serve che nel codice ci siano iterazioni prevedibili/desiderate
  Una coordinazione che avviene attraverso scambio di informazioni che può accadere:
  Questa sincronizzazione può accadere:
  - Diretta
  - Esplicita
- Competizione, dove i nostri agenti, siccome condividono le risorse, competono per averle
  E qui c'è bisogno di una sincronizzazione, che può essere:
  - Indiretta
  - Implicita
- Interferenze, sono i bug della programmazione concorrente che accadono per via di una competizione con poca sincronizzazione e sincronizzazione.

Parola chiave sincronizzazione, e qui ci sono 2 meccanismi implementati dalla JVM:

- Memoria condivisa
  - Mutua esclusione dove, in certe aree di codice solo poche persone possono entrare
  - Sincronizzazione su condizione, dove si sospende un thread fino ad una condizione
- Scambio di messaggi
  - Primitive send/receive
  - Si potrebbe sospendere l'esecuzione di un thread fino ad un evento

Tipologie sincronizzazione:

- Join, noi aspettiamo un thread che raggiunga la fine
  In questo caso noi entriamo in stato di wait
- Barriere
  Ogni thread esegue una operazione e, finita quella operazione, aspetta che gli altri abbiano finito. Questo può avvenire attraverso un contatore che, ogni volta un thread finisce viene decrementato, e quando arriva a 0 tutto viene rilasciato

Se però non vengono implementate, i seguenti problemi occorrono:

- Race condiction
  Questo avviene quando, tanti thread leggono e scrivono oppure scrivono in uno stessa area di memoria condivisa senza controllare che possano farlo.
  Questo causa che, certi risultati potrebbero venire sovrascritti, ex, 5 thread leggono il numero e lo incrementano di 1.
  Il primo thread legge, i=i+1, scrive, però il secondo thread legge anche lui fa i=i+5
  Mettiamo caso i=1
  Il risultato dovrebbe essere i=1 -> i=i+1=1+1=2 -> i=i+5=2+5=7
  Però la realtà è che succede: i=1 -> i=1+1=2 -> i = 1+5=6
  Questo perché il secondo thread legge in memoria prima che il primo possa aver scritto
  Questo avviene siccome le variabili non sono atomiche, cioè più thread possono accederci contemporaneamente.

  Questa area di codice dove tutti e due scriviamo viene chiamata area critica.
  Per sistemarlo:

  1) lock dove ci permette accesso esclusivo dentro quell'area

```
l.lock();
try {
    // doing some action
  try {
    counter++;
    Thread.sleep(1000);
  } catch (InterruptedException e) {
  }
} finally {
  // Release the permit.
  l.unlock();
}
```

  2) Semafori
     Praticamente, noi abbiamo un contatore che inizia ad N
     Quando qualcuno vuole fare accesso ad un aria di memoria, fa semaforo.acquire()
     Che decrementa il contatore se è maggiore di 0
     Se è uguale a 0, aspetta che viene incrementato.
     Quando poi si fa .release(), noi incrementiamo il contatore

```
try {
  sem.acquire();
  // doing some action
  Thread.sleep(1000);
} catch (InterruptedException exc) {}
// Release the permit.
sem.release();
```

3) Syncronized

E' possibile rendere una funzione/classe syncronized, aka noi praticamente mettiamo

Un gigante lock per tutta la funzione

```
public synchronized void deposito(
float cifra){
    saldo += cifra;
}

public synchronized void prelievo(
float cifra){
    if(saldo > cifra){
        saldo -= cifra;
        return cifra;
    }
    float prelevati = saldo;
    saldo = 0.0f;
}
```

Nota: non si può usare syncronized né su costruttore e né sui campi

Deposito e prelievo **condividono** il locker

Questo syncronized si potrebbe comportare come se fosse un locker

```
synchronized(this){
   while(writers>0){
      try {
         wait();
      } catch(InterruptedException e){}
   }
   readers++;
}
// Do the reading...
try{
   Thread.sleep(50);
} catch(InterruptedException e){}
int contSnapshot = content;
synchronized(this){
   readers--;
   if(readers==0) notifyAll();
}
return contSnapshot;
```

In questo caso, noi dobbiamo usare un oggetto che verrà usato come riferimento per bloccare/sbloccare

Usando this usiamo come oggetto la classe stessa, però è possibile creare un Object as;

Se vogliamo un qualcosa di globale possiamo creare una classe statica

```
class StaticSharedVariable {
  // due modi per ottenere lock a livello di
classe

  private static int shared;
  public int read() {
    synchronized(StaticSharedVariable.class) {
```

```
    synchronized(staticSharedVariable.class) {
      return shared;
    }

    public synchronized static void write(int i) {
      shared=i;
    }
}
```

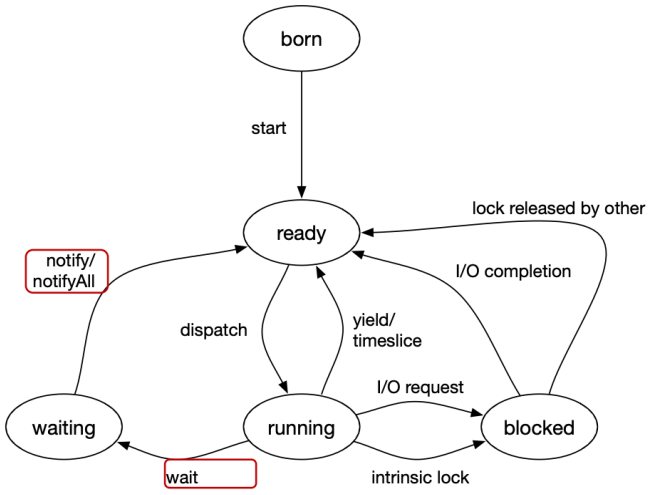
Possiamo estendere una classe e fare un override mettendoci il syncronized

```
public synchronized boolean assegnaPosti(String cliente, int numPosti) {
    System.out.println("Assegnatore derivato");
    return super.assegnaPosti(cliente,numPosti);
  }
```

4) Monitor

Il nostro ultimo (spero) costrutto per la sincronizzazione; monitor.
Praticamente, qui dentro tutte le funzioni e i campi sono syncronized
Problema: usa tanto buffer, e l'invio/ricezione dei messaggi è asincrona
Detto questo, le politich:

- Se il buffer è pieno, aspetti e non inserisci nulla
- Se è buoto, non cancelli
- Necessarie delle primitive per attendere che un dato viene messo -> wait
- E notificare gli altri che un dato è stato messo -> notify
  Ogni volta che c'è una modifica, facciamo un notify
  Nuovo diagramma dei thread:



Si comprende che avremo bisogno di 2 liste:

1) Thread bloccati per l'accesso
2) Thread bloccati nell'attesa di un notify/simile

TODO