UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI **MATEMATICA E INFORMATICA**

Interfacce Grafiche e Programmazione ad Eventi

Carmine Dodaro

Anno Accademico 2019/2020

Programmazione concorrente

Cos'è

La programmazione concorrente indica la possibilità di scrivere programmi che possono eseguire più operazioni in parallelo.

Perché usare la programmazione concorrente?

Usare la programmazione concorrente garantisce l'esecuzione di compiti potenzialmente pesanti in background in modo che l'utente non debba attendere l'esito dell'operazione prima di eseguirne altre.

Processi e thread

Cosa sono?

Un processo è un ambiente di esecuzione in cui gira il programma che l'ha creato. Ogni processo:

- afferisce ad un singolo programma
- è composto da un insieme di thread che condividono la stessa memoria
- ha sempre un thread di esecuzione rappresentato da sé stesso
- non vede le strutture dati degli altri processi

Un thread è un'unità di eleborazione in cui il processo può essere suddiviso. Ogni thread:

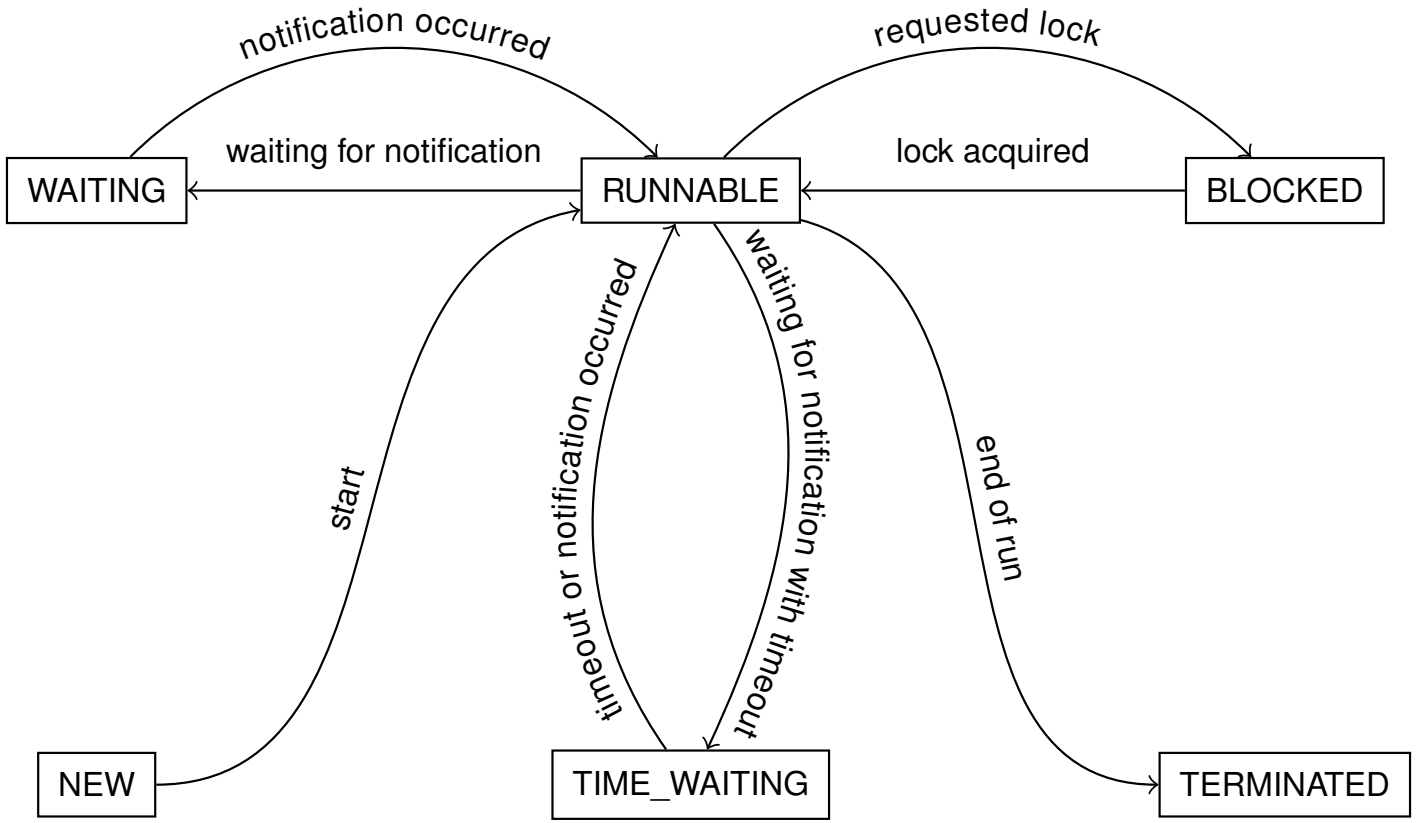
- ce ne sono più di uno per processo
- condividono le stesse risorse e strutture dati

Stati di un thread in Java

Un thread in Java può essere nei seguenti stati:

- NEW: un thread che non è ancora partito
- RUNNABLE: un thread che è eseguito
- BLOCKED: un thread che è bloccato per un lock
- WAITING: un thread che aspetta (per un tempo indefinito) che un altro thread faccia alcune azioni
- TIMED_WAITING: un thread che aspetta (per un tempo fissato, chiamato waiting time) che un altro thread faccia alcune azioni
- TERMINATED: un thread che ha finito il suo ciclo di vita

Interazione tra gli stati



I thread in java

La classe Thread

In Java i thread sono rappresentati da oggetti della classe Thread. La classe Thread mette a disposizione un metodo run in cui inserire il codice per effettuare le operazioni specifiche del thread.

```
public class ClassThread1 extends Thread {
  @Override
  public void run() {
    super.run();
    while(true) {
      System.out.println("Hello world!");
    }
  }
}

... main:

ClassThread1 t1 = new ClassThread1();
t1.start();
//Le operazioni scritte qui sono eseguite nonostante il codice
    di run contenga un while(true)!
```

La classe Thread

La classe Thread permette anche di interrompere l'esecuzione del thread per un certo periodo di tempo, quindi il thread passa da uno stato di running ad uno di waiting. Il metodo da utilizzare è sleep, che riceve come parametro un intero rappresentante il numero di millisecondi di pausa.

```
@Override
public void run() {
  super.run();
  while(true) {
    try {
      System.out.println("Hello world!");
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      return;
    }
  }
}
```

L'interfaccia Runnable

Un'alternativa all'uso della classe Thread è l'uso dell'interfaccia Runnable.

```
public class ClassThread2 implements Runnable {
  @Override
  public void run() {
    while(true) {
      try {
        System.out.println("Hello world!");
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        return;
      }
    }
  }
}

... main:

ClassThread2 t2 = new ClassThread2();
Thread t = new Thread(t2);
t.start();
```

Sincronizzazione

Gestire la sincronizzazione

In un programma con più thread è opportuno gestire la sincronizzazione tra tutti i thread in modo da garantire che non ci siano accessi simultanei alla stessa risorsa.

La sincronizzazione è attuata attraverso operazioni di lock, block e release, attraverso cui si garantisce che un solo thread alla volta possa avere accesso a determinati dati in un preciso momento (mutua esclusione).

In Java la sincronizzazione tra i vari thread è gestita attraverso un meccanismo di controllo chiamato monitor lock. Ad ogni oggetto e ad ogni classe è associato un monitor che serve a proteggere le loro variabili. Nell'ambito del programma possiamo decidere quali blocchi di codice devono essere protetti, cioè porzioni di codice che devono essere eseguite in modo esclusivo e sincronizzato. da un solo thread per volta.

Sincronizzazione

Sincronizzazione nel codice

```
public synchronized void syncMethod() {
  //here sync
}
```

Ogni thread che proverà ad usare il metodoSincronizzato ha la garanzia che nessun altro thread lo potrà eseguire nello stesso momento (lock). I thread che proveranno ad usare quel metodo andranno in uno stato di blocco (block). Quando l'esecuzione del metodo termina per il thread che ha richiesto il lock, allora il metodo è reso disponibile anche gli altri thread (release).
Si può anche sincronizzare solo una porzione del metodo:

```
public void partialSyncMethod()
  //here no sync
  synchronized(this) {
    //here sync
  }
  //here no sync
}
```

Condizioni di blocco

Blocchi

Un programma concorrente può bloccarsi a causa di tre differenti cause:

- Deadlock: due thread sono bloccati perché aspettano tra di loro che ciascuno liberi una risorsa.
- Starvation: un thread ottiene un lock su una risorsa e compie delle operazioni che non consentono mai di rilasciarlo.
- Livelock: è simile in linea di principio al deadlock, solo che in questo caso i due thread non sono bloccati, ma compiono continuamente delle operazioni, l'uno rispetto all'altro, che non permettono di terminare la propria esecuzione.

Alcuni metodi utili sui thread

Alcuni metodi

- wait(); permette di mandare in uno stato di wait un thread per un tempo indefinito.
- notify(); notifica ad un thread su cui era stato chiamato il metodo wait() che può riprendere le operazioni.
- notifyAll(); notifica a tutti i thread su cui era stato chiamato il metodo wait() che possono riprendere le operazioni.
- join(); pone in attesa un thread fino a che non cessa di esistere.
- yield(); il thread cede spontaneamente il proprio tempo ad altri thread.
- setPriority(); riceve come parametro un intero
  - Thread.MAX_PRIORITY
  - Thread.NORM_PRIORITY
  - Thread.MIN_PRIORITY