Processi e multitasking

Multitasking: caratteristica di un S.O. che permette l'esecuzione simultanea (o pseudosimultanea) di più processi

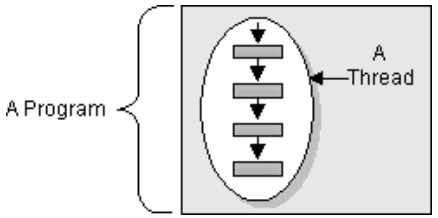
- ***Cooperative Multitasking****:* gestione affidata ai processi stessi, che mantengono il controllo del processore fino al rilascio spontaneo
- ***Preemptive Multitasking:*** gestione in time-slicing, completamente gestita dal S.O. Tipicamente: diversi algoritmi di scheduling per l'assegnamento dei tempi e delle priorità, round-robin tra processi con stessa priorità

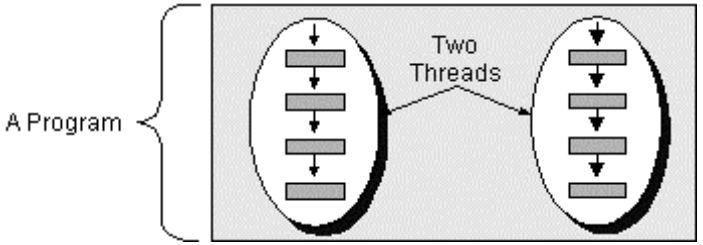
Contesto di un processo: insieme delle informazioni necessarie per ristabilire esattamente lo stato in cui si trova il sistema al momento in cui se ne interrompe l'esecuzione per passare ad un altro (stato dei registri del processore, memoria del processo, etc.)

L'avvicendamento dell'esecuzione comporta un ***context-switch***

Thread e Multithreading

Un ***thread*** (*lightweight process*) è un singolo flusso sequenziale di controllo all'interno di un processo



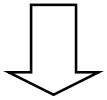


Multithreading: esecuzione contemporanea (o pseudocontemporanea) di diversi thread nell'ambito di uno stesso processo

- *Collaborative multithreading*
- *Preemptive multithreading*

Caratteristiche dei thread:

- eseguono all'interno del contesto di esecuzione di un unico processo
- non hanno uno spazio di indirizzamento riservato: tutti i thread appartenenti allo stesso processo condividono lo **stesso spazio di indirizzamento**
- hanno *execution stack* e *program counter* privati



+ context-switch molto meno oneroso

+ comunicazione più semplice

- più rilevanti e frequenti problemi di sincronizzazione

Modelli implementativi di multithreading:

- ***Molti a uno:*** I thread sono implementati a livello di applicazione, il loro scheduler non fa parte del S.O., che continua ad avere solo la visibilità del processo.
  + scheduling poco oneroso
  - perdita di parallelismo

  Es. Green-thread in Unix

- ***Uno a uno:*** I thread sono gestiti direttamente dal S.O. come entità primitive (thread nativi).
  + scheduling molto efficiente
  - alti tempi di creazione e di sincronizzazione

  Es. Windows NT

- ***Molti a molti*** *(modello a due livelli)**:*** Il sistema dispone di un pool di thread nativi (*worker*), ad ognuno dei quali viene assegnato di volta in volta un thread d'applicazione (*user thread*) da eseguire.
  + efficiente e poco oneroso creare e gestire un thread
  + molto flessibile
  - difficile definire la dimensione del pool di worker e le modalità di cooperazione tra i due scheduler

  Es. Solaris

Java Thread

Ogni esecuzione della macchina virtuale dà origine ad un processo, e tutto quello che viene mandato in esecuzione da una macchina virtuale dà origine a un thread

Le specifiche ufficiali della JVM stabiliscono che una VM gestisca i thread secondo uno scheduling preemptive (*fixed-priority scheduling*)… e basta!

Quindi:

non c'è garanzia che sia implementata una gestione in time-slicing

Inoltre:

c'è totale libertà sulle modalità di mappatura tra thread Java e thread del S.O.

Es. di modelli adottabili:

- ***Thread nativi**:* la JVM utilizza il supporto al multithreading fornito dal S.O.
  Es. Windows
- ***Green-thread**:* la JVM si fa interamente carico della gestione dei thread, ignorati dal S.O. che vede la JVM come un processo con un singolo thread
  Es. Unix

Programmazione concorrente in Java

Due modalità per implementare ***thread*** in Java:

1. come sottoclasse della classe **`Thread`**
2. come classe che implementa l'interfaccia **`Runnable`**

Modalità 1:
come sottoclasse della **classe `Thread`**

La classe **`Thread`** è una classe non astratta attraverso la quale si accede a tutte le principali funzionalità per la gestione dei thread.

Procedimento:

- sottoclassare la classe **`Thread`** ridefinendone il metodo **`run()`**
- creare un'istanza della sottoclasse tramite **`new`**
- mettere in esecuzione l'oggetto creato, un thread, chiamando il metodo **`start()`** che a sua volta richiama il metodo **`run()`**

Esempio di classe **`Simplethread`** ***sottoclasse*** di **`Thread`** (modalità 1):

```
public class SimpleThread extends Thread {

   public SimpleThread(String str) {
      super(str);
      }

      public void run() {
        for (int i = 0; i < 10; i++) {
           System.out.println(i + " " +
               getName());
           try {
               sleep((int)(Math.random()*
                  1000));
          } catch (InterruptedException e){}
           }
        System.out.println("DONE! " +
            getName());
              }


public class TwoThreadsTest {

   public static void main (String[] args) {
     new SimpleThread("Jamaica").start();
     new SimpleThread("Fiji").start();
     }
  }
```

Modalità 2:
come classe che implementa l'interfaccia **`Runnable`**

L'interfaccia **`Runnable`** contiene un solo metodo, identico a quello della classe **`Thread`**, che infatti la implementa. Consente a una classe non derivata da **`Thread`** di funzionare come tale, purchè venga agganciata a un oggetto thread

Procedimento:

- implementare il metodo **`run()`** nella classe
- creare un'istanza della classe tramite **`new`**
- creare un'istanza della classe **`Thread`** con un'altra **`new`**, passando al costruttore del thread un reference dell'oggetto runnable che si è creato
- invocare il metodo **`start()`** sul thread creato, producendo la chiamata al suo metodo **`run()`**

Esempio di classe **`EsempioRunnable`**
che **implementa l'interfaccia `Runnable`**
ed è **sottoclasse di `MiaClasse`** (modalità 2):

```
class EsempioRunnable extends MiaClasse
   implements Runnable {

   // non e' sottoclasse di Thread

   public void run() {
     for (int i=1; i<=10; i++)
        System.out.println(i + " " + i*i);
   }
}

public class Esempio {

   public static void main(String args[]){

      EsempioRunnable e =
           new EsempioRunnable();
      Thread t = new Thread (e);
      t.start();
   }
}
```

Classe `Thread` VS Interfaccia `Runnable`

Classe **`Thread`**:

+ modalità più **immediata e semplice**

– scarsa flessibilità derivante dalla necessità di ereditare dalla classe Thread, che impedisce di ereditare da altre classi…

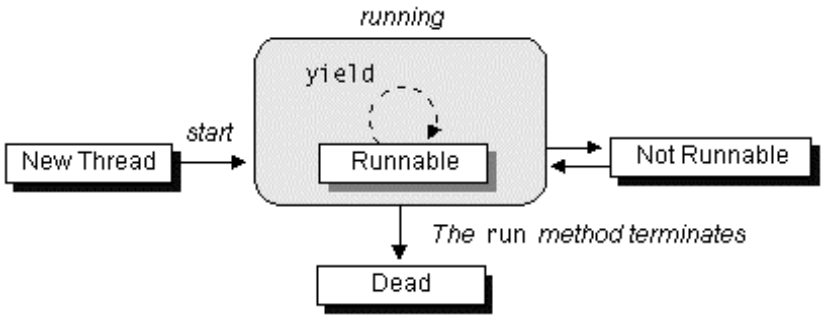
E se occorre definire thread che non siano **necessariamente** sottoclassi di Thread?

Interfaccia `Runnable`:

+ **maggiore flessibilità** derivante dal poter essere sottoclasse di qualsiasi altra classe, utile per ovviare all'impossibilità di avere ereditarietà multipla in Java

    - modalità un po' più macchinosa…

Il ciclo di vita di un thread



- **Creato**
  subito dopo l'istruzione **`new`**
  le variabili sono state allocate e inizializzate; il thread è in attesa di passare allo stato di eseguibile
- **Runnable**
  thread è in esecuzione, o in coda d'attesa per ottenere l'utilizzo della CPU
- **Not Runnable**
  il thread non può essere messo in esecuzione dallo scheduler. Entra in questo stato quando in **attesa** di un'operazione di I/O, o dopo l'invocazione dei metodi **`suspend()`**, **`wait()`**, **`sleep()`**
- **Dead**
  al termine "naturale" della sua esecuzione o dopo l'invocazione del suo metodo **`stop()`** da parte di un altro thread

Metodi per il controllo di thread

`start()` fa ***partire*** l'esecuzione di un thread. La macchina virtuale Java invoca il metodo `run()` del thread appena creato

`stop()` ***forza*** la ***terminazione*** dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente ***liberate*** (lock inclusi), come effetto della propagazione dell'eccezione `ThreadDeath`

`suspend()` ***blocca*** l'esecuzione di un thread in attesa di una successiva operazione di ***resume***. Non libera le risorse impegnate dal thread (possibilità di ***deadlock***)

`resume()` ***riprende*** l'esecuzione di un thread precedentemente ***sospeso***. Se il thread riattivato ha una priorità maggiore di quello correntemente in esecuzione, avrà subito accesso alla CPU, altrimenti andrà in coda d'attesa

`sleep(long time)` blocca per il ***tempo specificato*** in time l'esecuzione di un thread. Nessun *lock* in possesso del thread viene rilasciato.

`join()` ***blocca*** il thread chiamante in attesa della ***terminazione*** del thread di cui si invoca il metodo. Anche con ***timeout***

`yield()` ***sospende*** l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread, con la **stessa priorità**, in ***coda d'attesa***

I metodi precedenti interagiscono ***ovviamente*** con il ***gestore della sicurezza*** della macchina virtuale Java (`SecurityManager`, `checkAccess()`, `checkPermission()`)

Altri metodi fondamentali per le operazioni di ***sincronizzazione*** fra thread Java:
`wait()`, **`notify()`**, **`notifyAll()`**

(vedi lucido 18 e seguenti)

Il problema di `stop()` e `suspend()`

`stop()` e `suspend()` rappresentano azioni "brutali" sul ciclo di vita di un thread
=> rischio di determinare situazioni di blocco critico (***deadlock***)

Infatti:

- se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva***, tale risorsa rimane ***bloccata*** e non è utilizzabile da altri, perché il thread sospeso non ha avuto modo di rilasciare il *lock* su di essa
- se il ***thread interrotto*** stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera ***atomica***, l'interruzione può condurre ad uno ***stato inconsistente*** del sistema

➔ JDK 1.2, pur supportandoli ancora per ragioni di *back-compatibility*, ***sconsiglia*** l'utilizzo dei metodi `stop()`, `suspend()` e `resume()` (***metodi deprecated***)

Si consiglia invece di realizzare tutte le azioni di ***controllo*** e ***sincronizzazione*** fra thread tramite i metodi **`wait()`** e **`notify()`** su variabili condizione (astrazione di ***monitor***)

Priorità dei thread in Java

Scheduling: esecuzione di una molteplicità di thread su una singola CPU, in un qualche ordine

Macchina virtuale Java (JVM)
Fixed Priority Scheduling
algoritmo di scheduling molto semplice e deterministico

- JVM sceglie il thread in stato ***runnable*** con ***priorità*** più ***alta***

- Se più thread in attesa di eseguire hanno ***uguale priorità***, la scelta della JVM avviene con una modalità di tipo ***round-robin***.

La classe `Thread` fornisce i metodi:

- `setPriority(int num)`
- `getPriority()`

con valori di `num` compresi fra `MIN_PRIORITY` e `MAX_PRIORITY` (costanti definite anch'esse nella classe `Thread`)

Scheduling

Il ***thread*** messo ***in esecuzione*** dallo scheduler viene ***interrotto*** se e solo se:

- un thread con priorità ***più alta*** diventa ***runnable***;
- il metodo `run` ***termina l'esecuzione*** o il thread esegue un `yield`;
- il ***quanto*** di tempo assegnato si è ***esaurito*** (solo su sistemi che supportano *time-slicing*, come *Windows95/NT*)

```
public void run() {
   while (tick < 200000) {
         tick++;
         if ((tick % 50000) == 0)
            System.out.println("Thread #" +
              num + ", tick = " + tick);
   }
}
```

Time-Sliced System	***Non Time-Sliced System***
Thread #1, tick = 50000	Thread #0, tick = 50000
Thread #0, tick = 50000	Thread #0, tick = 100000
Thread #0, tick = 100000	Thread #0, tick = 150000
Thread #1, tick = 100000	Thread #0, tick = 200000
Thread #1, tick = 150000	Thread #1, tick = 50000
Thread #1, tick = 200000	Thread #1, tick = 100000
Thread #0, tick = 150000	Thread #1, tick = 150000
Thread #0, tick = 200000	Thread #1, tick = 200000

Sincronizzazione di thread

Differenti thread che fanno parte della stessa applicazione Java condividono lo ***stesso spazio*** di memoria

➔ è possibile che ***più thread*** accedano ***contemporaneamente*** allo stesso metodo o alla stessa sezione di codice di un oggetto

Servono meccanismi di sincronizzazione

JVM supporta la definizione di ***monitor*** per la sincronizzazione nell'accesso a risorse tramite la keyword **`synchronized`**, che può essere usata, con ***scope*** diversi del lock, su:

- singolo metodo (di istanza o di classe)
  ⇨ lock sull'oggetto o sulla classe
- blocco di istruzioni
  ⇨ lock su un oggetto specificato come parametro

In pratica:

- a ogni oggetto Java è automaticamente associato un ***lock***
- per accedere a un metodo o una sezione `synchronized`, un thread deve prima ***acquisire il lock*** dell'oggetto
- il ***lock*** è automaticamente ***rilasciato*** quando il thread esce dalla sezione `synchronized`, o se viene interrotto da un'eccezione
- un thread che non riesce ad acquisire un *lock* ***rimane sospeso*** sulla richiesta della risorsa fino a che il *lock* non è disponibile

NOTE:

- una variabile non può essere sottoposta a lock, quindi oltre a sincronizzare i metodi che la gestiscono bisogna impedirne l'accesso diretto
- ad ogni oggetto contenente metodi o blocchi `synchronized` viene assegnata ***una sola variabile condizione***

  ➔ due thread non possono accedere contemporaneamente a ***due sezioni `synchronized` diverse di uno stesso oggetto***

  Questo rende il modello Java ***meno espressivo*** di un *vero monitor*, che presuppone la possibilità di definire più sezioni critiche per uno stesso oggetto

Ogni oggetto Java (istanza di una sottoclasse qualsiasi della classe `Object`) fornisce i ***metodi di sincronizzazione***:

- **`wait()`**

  blocca l'esecuzione del thread invocante in attesa che un altro thread invochi i metodi `notify()` o `notifyAll()` per quell'oggetto. Il thread invocante deve essere in possesso del *lock* sull'oggetto; il suo blocco avviene dopo aver rilasciato il *lock*. Anche varianti con specifica di ***timeout***

- **`notify()`**

  risveglia un ***unico thread*** in attesa sul monitor dell'oggetto in questione. Se più thread sono in attesa, la scelta avviene in maniera ***arbitraria***, dipendente dall'implementazione della macchina virtuale Java. Il thread risvegliato compete con ogni altro thread, come di norma, per ottenere la risorsa protetta

- **`notifyAll()`**

  esattamente come `notify()`, ma risveglia ***tutti i thread*** in attesa per l'oggetto in questione. È necessario tutte le volte in cui ***più thread*** possono essere ***sospesi su differenti sezioni*** critiche dello stesso oggetto (***unica coda d'attesa***)

Esempio (*Produttori e Consumatori*):

```
public synchronized int get() {
   while (available == false)
      try {
         // attende un dato dai Produttori
         wait();
      } catch (InterruptedException e) {}
   }
   available = false;
   // notifica i produttori del consumo
   notifyAll();
…
}

public synchronized void put(int value) {
   while (available == true)
      try {
         // attende il consumo del dato
         wait();
      } catch (InterruptedException e) {}
   }
   …
   available = true;
   // notifica i consumatori della
   // produzione di un nuovo dato
   notifyAll();
}
```


Gruppi di thread

Obiettivo: raccogliere una ***molteplicità di thread*** all'interno di un ***solo gruppo*** per facilitare operazioni di ***gestione*** (ad es. sospendere/ interrompere/far ripartire l'esecuzione di un insieme di thread con un'unica invocazione)

JVM associa ***ogni thread*** ad ***un gruppo*** all'atto della ***creazione*** del thread
Questa associazione è ***permanente*** e non può essere modificata

```
ThreadGroup myThreadGroup =
    new ThreadGroup("Mio Gruppo");
Thread myThread =
    new Thread(myThreadGroup, "MioT");
```

I thread group sono organizzati secondo una **struttura gerarchica ad albero**

Scelta di ***default***:

stesso gruppo a cui appartiene il thread creatore (alla partenza di un'applicazione, la JVM crea un gruppo di thread chiamato ***"main"***)

Daemon Thread

I thread in Java possono essere di due tipi:

user thread e ***daemon thread***

Unica differenza: la virtual machine termina l'esecuzione di un daemon thread quando termina l'ultimo user thread

I daemon thread **svolgono servizi per gli user thread** e spesso restano in esecuzione per tutta la durata di una sessione della virtual machine

Es. garbage collector

Per default un thread assume lo stato del thread che lo crea.
E' possibile verificare lo stato con **`isDaemon()`** e modificarlo con **`setDaemon()`**, ma solo prima di mandarlo in esecuzione.