Processi e multitasking

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

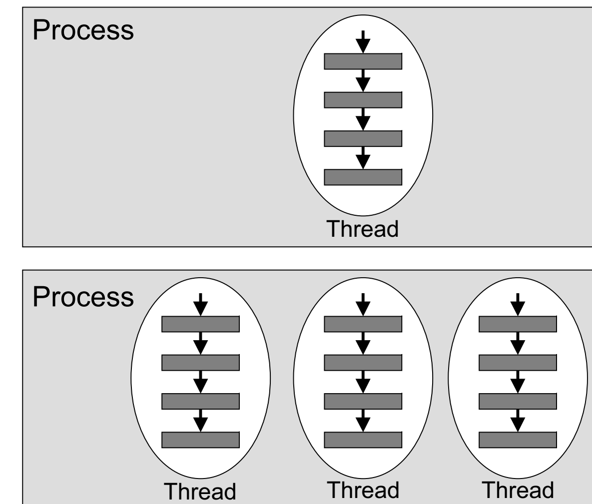
- ***Cooperative Multitasking****:* gestione affidata ai processi stessi, che mantengono il controllo del processore fino al rilascio spontaneo
- ***Preemptive Multitasking:*** gestione in time-slicing, completamente gestita dal S.O. Tipicamente: diversi algoritmi di scheduling per l'assegnamento dei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

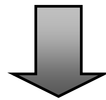
- *Collaborative multithreading*
- *Preemptive multithreading*

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**



Rispetto ai processi pesanti:

- ☺ context-switch meno oneroso
- ☺ comunicazione più semplice
- ☹ problemi di sincronizzazione più rilevanti e frequenti

Modelli d'implementazione multithreading

- ***Molti a uno:*** I thread sono implementati **a livello applicativo**, il loro scheduler esegue in spazio utente e 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 sincronizzazione

  Es. Windows NT

- ***Molti a molti*** *(modello a due livelli)**:*** Il S.O. 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

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)

Per esempio, in Java la 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

In generale i **daemon** sono **processi** che eseguono un **ciclo infinito** di attesa di richieste ed esecuzione delle stesse.

Java riconosce l'importanza dei thread daemon e introduce due tipi di thread:

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. I

Es. garbage collector

Multithreading in Java

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 mappaggio tra thread Java e thread del S.O.

Es. di politiche di implementazione:

- ***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

N.B.: i programmi devono essere corretti indipendentemente dalla politica adottata

Livello di linguaggio

Java fornisce due modalità per implementare ***thread***:

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

Processi come sottoclassi 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:

1. definire una sottoclasse della classe **`Thread`** ridefinendone il metodo **`run()`**
2. creare un'istanza della sottoclasse tramite **`new`**
3. mettere in esecuzione l'oggetto creato, un thread, chiamando il metodo **`start()`** che a sua volta richiama il metodo **`run()`**

Esempio di processo **`EsempioThread`** ***realizzato come sottoclasse*** di **`Thread`**:

```
public class EsempioThread extends Thread {

   public EsempioThread(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 EsempioThread("Jamaica").start();
     new EsempioThread("Fiji").start();
     }
  }
```

Processi come classi che implementano 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:

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

Esempio di processo **`EsempioRunnable`** ***realizzato come classe che implementa l'interfaccia*** **`Runnable`** ed è sottoclasse di **`MiaClasse`**:

```
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`

<u>Classe **`Thread`**</u>:

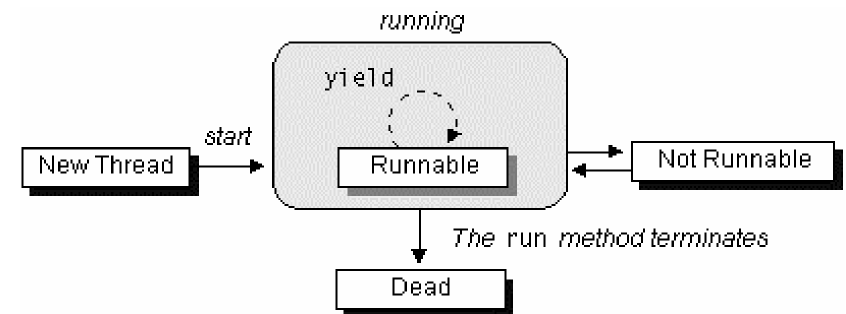
- 🙂 modalità più **immediata e semplice**
- ☹ scarsa flessibilità derivante dalla necessità di ereditare dalla classe Thread, che impedisce di ereditare da altre classi…

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

<u>**Interfaccia `Runnable`**</u>:

- 🙂 **maggiore flessibilità** derivante dal poter essere sottoclasse di qualsiasi altra classe, utile per ovviare all'impossibilità di avere ereditarietà multipla in Java
- ☹ modalità leggermente più macchinosa…

Il ciclo di vita di un thread



- **Created (New Thread)**
  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. La memoria è ancora allocata ma non più accessibile (-> garbage collector).

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

`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 su 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()`)

In realtà per la **gestione della sincronizzazione** in Java sono ***sufficienti*** i seguenti metodi che vedremo nel dettaglio fra poco:

`wait()`, **`notify()`**, **`notifyAll()`**

Metodi deprecati per il controllo di thread

Prime versioni della JVM:
altri metodi per la gestione diretta dei thread;
Da tempo però
alcuni metodi deprecati in quanto non safe.

`stop()` ***DEPRECATED, NON USARE!!!***

forza la ***terminazione*** dell'esecuzione di un thread e il rilascio di tutti gli oggetti acquisiti. Nel caso in cui lo stato di tali oggetti sia lasciato in stato inconsistente → **possibilità di *deadlock***!!!

`suspend()` ***DEPRECATED, NON USARE!!!***

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()` ***DEPRECATED, NON USARE!!!***

riprende l'esecuzione di un thread precedentemente ***sospeso*** con la primitiva suspend, anche questo metodo è deprecated.

Per ulteriori informazioni sui metodi deprecated, si veda:
http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html

Sincronizzazione di thread: metodi e sezioni critiche

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 quindi **meccanismi di sincronizzazione**

JVM supporta la sincronizzazione nell'accesso a risorse tramite la keyword

`synchronized`

➔ uso di **lock** per ottenere la **mutua esclusione**

Synchronized può essere usata con ***scope*** diversi su:

- **singolo metodo**
  lock sull'oggetto su cui viene invocato il metodo
- **singolo metodo di classe**
  lock sulla classe sulla quale viene invocato il metodo
- **blocco di istruzioni** (sezione critica)
  lock su un oggetto specificato come parametro (se non ne viene specificato nessuno si assume quello su cui viene invocato il metodo all'interno del quale si trova la sezione critica)

Sincronizzazione e variabili condizione (lock)

- a ogni oggetto Java è **automaticamente** associato un ***lock*** (N.B.: UNO solo)
- per accedere a un metodo o a una sezione `synchronized`, un thread deve prima ***acquisire il lock*** dell'oggetto di cui si invoca il metodo o su cui è sincronizzata la sezione critica
- il ***lock*** è automaticamente ***rilasciato*** quando il thread esce dal metodo o 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

Esempi di uso di synchronized: Accesso alla classe ContoCorrente

```
Class CCGiovane{

   private float ammontare;
   private static float tassoInteresse;
   private String nome;
   private int idCC
   …
```

synchronized su **metodo**:

```
 public synchronized float
       versaDenaro(float versamento){
   ammontare = ammontare + versamento;
   return ammontare;
 }
```

synchronized su **metodo di classe**:

```
public synchronized static float
          aggiorna_tasso(float newTasso){
  tassoInteresse = newTasso;
  return newTasso;
 }
```

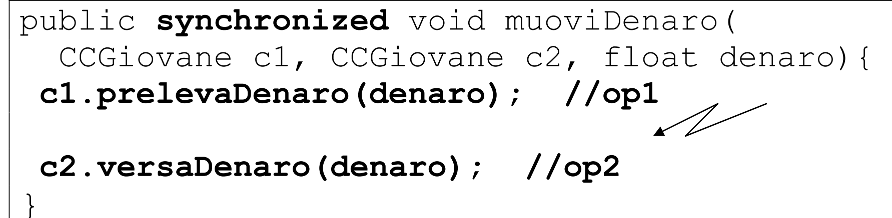
In questo modo si gestisce l’accesso alla variabile di classe tassoInteresse, che fissa il tasso di interesse applicato a tutti i CC di tipo Giovane, in modo corretto, cioè **mutuamente esclusivo**, da parte di thread diversi che accedano in modo concorrente ai metodi di classe.

Uso di synchronized e transazioni: Accesso alla classe Banca

```
Class Banca{

   private CCGiovane[] contiCorrenti;
   private int numCC=0;
   private String messaggioPubblicitario;
   …
```

synchronized su **metodo**:



```
public synchronized void muoviDenaro(
  CCGiovane c1, CCGiovane c2, float denaro){
 c1.prelevaDenaro(denaro);  //op1

 c2.versaDenaro(denaro);  //op2
}
```

Cosa accadrebbe se la variabile di oggetto **contiCorrenti** fosse visibile al di fuori dell’oggetto Banca, cioè **non fosse dichiarata private**? Ciò potrebbe creare problemi per la **consistenza dell’oggetto Banca**?

Si pensi ad esempio al caso in cui un thread, senza utilizzare i metodi offerti dall’oggetto Banca, **acceda direttamente a contiCorrenti durante l’operazione di movimento di denaro** fra due conti correnti dello stesso utente (cioè fra l’esecuzione di op1 e op2).

synchronized su **sezione critica**:

```
…

public void stampa_contiCorrenti(){
 synchronized(this) {
    CCGiovane[] copia = copiaCC(contiCorrenti);
 }
 stampa(copia);
}
```

In questo caso per **non bloccare tutte le operazioni synchronized** sui CC, dal momento che l'operazione di stampa potrebbe durare molto, viene fatta una copia temporanea dei CC e viene mandata in stampa tale copia.

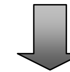
synchronized su **metodi di lettura**:

```
public synchronized void leggiCCGiovani(){
 for(int i=0 ; i<numCC; i++){
    System.out.println(contiCorrenti[i]);
 }
}
```

Si noti che **anche i metodi di lettura**, su oggetti che possono essere acceduti in modo concorrente **vanno sincronizzati**. Perchè? Cosa potrebbe accadere altrimenti?

NOTA BENE:

- I ***metodi synchronized*** e le ***sezioni synchronized(this)*** sono gestiti usando obbligatoriamente ***un'unica variabile condizione*** associata all'oggetto



più thread NON possono accedere contemporanea-mente a metodi synchronized diversi oppure a sezioni synchronized(this) diverse di uno stesso oggetto.

ATTENZIONE!
Il **lock non vale** sui **metodi non sincronizzati,** quindi metodi sincronizzati e non possono eseguire parallelamente:

```
public String leggiMessaggioPubblicitario(){
   return messaggioPubblicitario;
 }
```

`muoviDenaro` e **`leggiMessaggioPubblicitario`** possono eseguire parallelamente…

Sincronizzazione di thread: wait e notify

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 di uso di wait() e notify(): Produttore e Consumatore

```
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();
}
```

Dopo notifyAll quanti thread vengono messi in esecuzione?