

Università di Ferrara
Laurea Triennale in Informatica
A.A. 2021-2022
Sistemi Operativi e Laboratorio

6. Introduzione ai Thread Java e POSIX

Prof. Carlo Giannelli

`http://www.unife.it/scienze/informatica/insegnamenti
/sistemi-operativi-laboratorio`
`http://docente.unife.it/carlo.giannelli`
`https://ds.unife.it/people/carlo.giannelli/`

Thread e Multithreading

- Per risolvere i problemi di efficienza del modello a processi pesanti (modello ad ambiente locale) è possibile far ricorso al **modello ad ambiente globale**, a processi leggeri (o thread).
- **Thread**: singolo flusso sequenziale di esecuzione all'interno di un processo.
- **Multithreading**: esecuzione concorrente (in parallelo o interleaved) di diversi thread nel contesto di un unico processo.

Thread

Un thread è un ***singolo flusso sequenziale*** di controllo all'interno di un processo

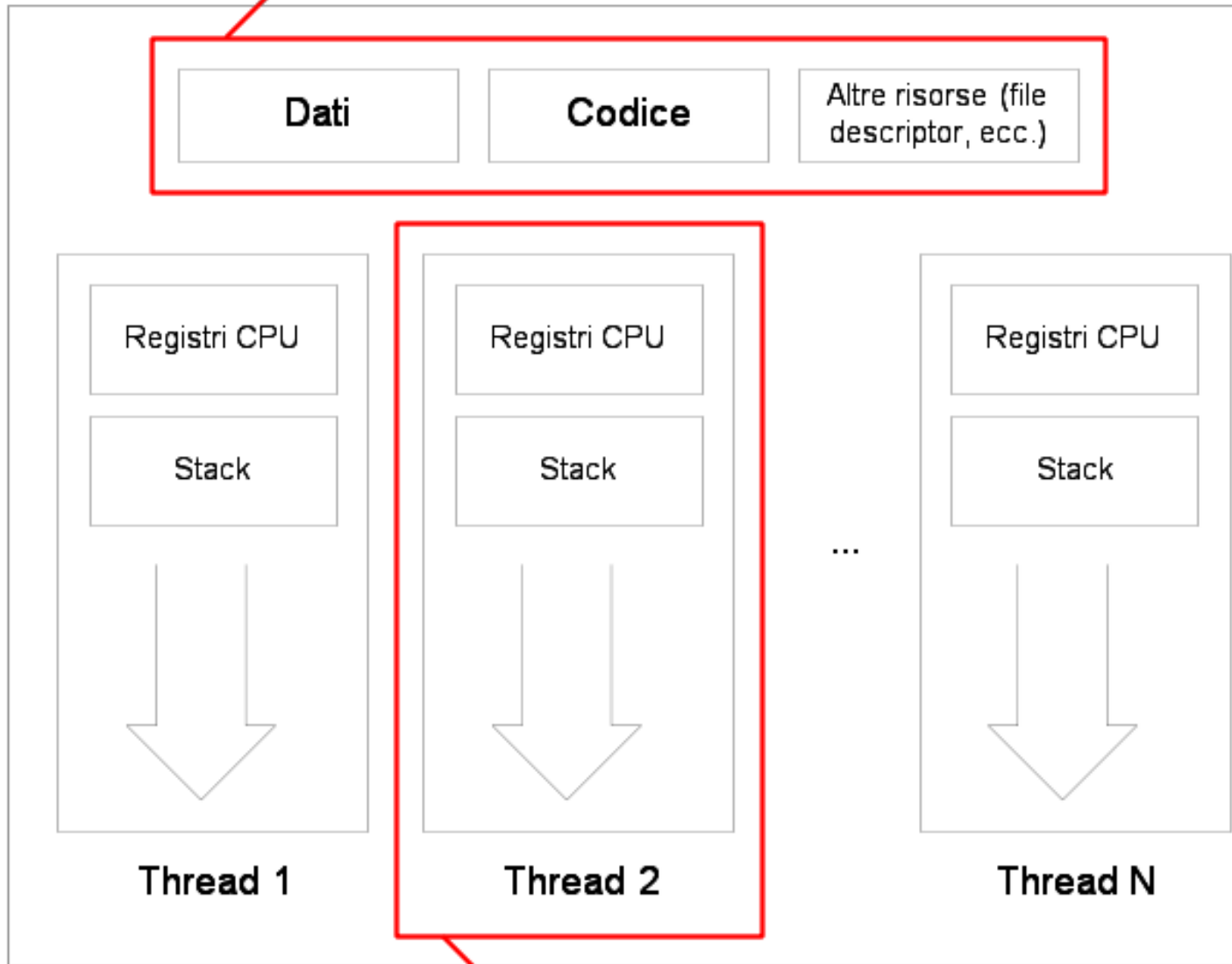
Un thread (o processo leggero) è un'unità di esecuzione che ***condivide codice e dati*** con altri thread ad esso associati

Un ***thread***

- ***NON*** ha spazio di memoria riservato per dati e heap: tutti i thread appartenenti allo stesso processo condividono lo ***stesso spazio di indirizzamento***
- ha ***stack*** e ***program counter privati***

Processi a thread multipli (multithreaded)

Spazio di indirizzamento e risorse
sono condivisi tra i vari thread



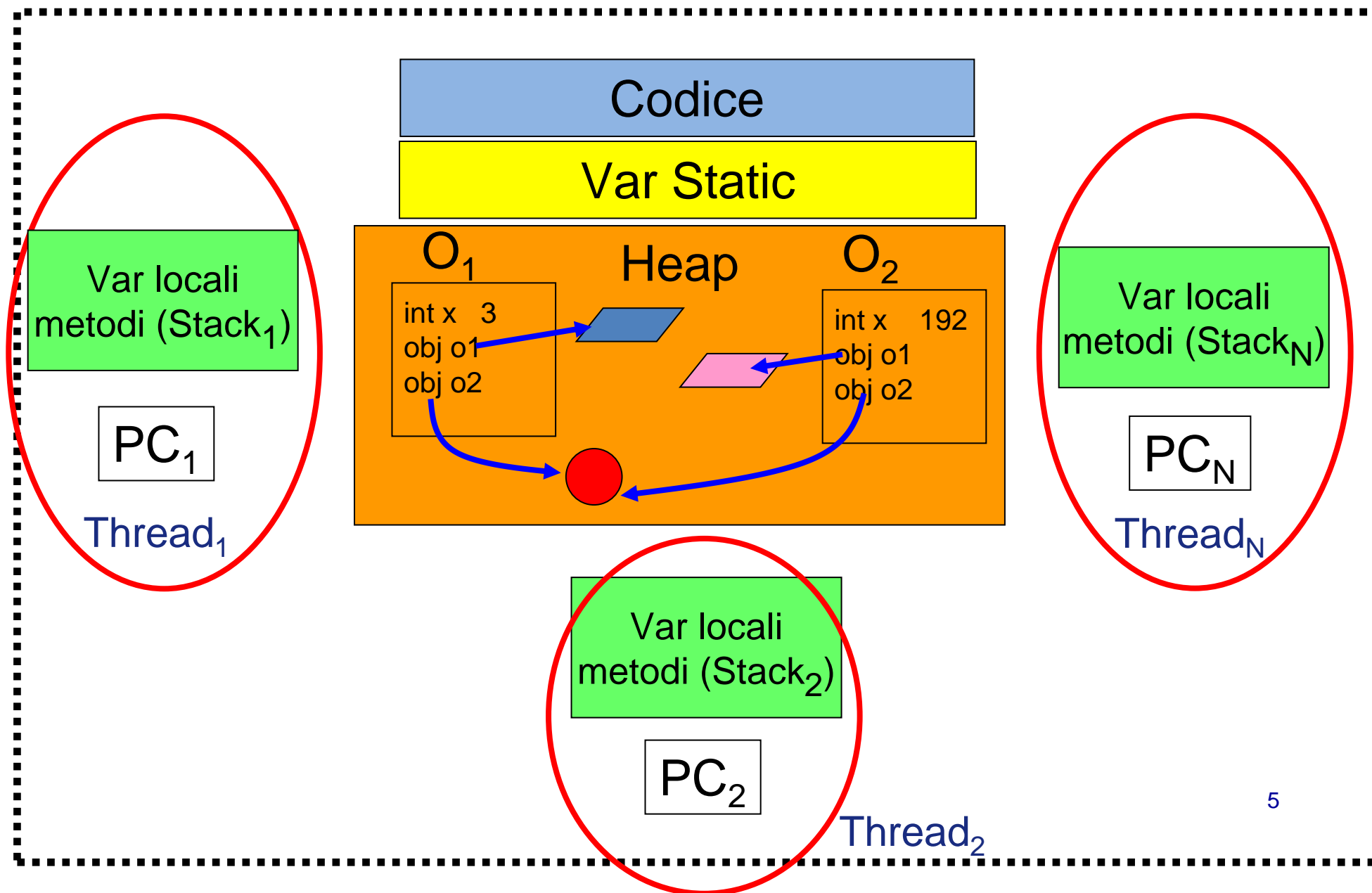
L'esecuzione dei vari thread è
indipendente e concorrente

Separazione tra:

- unità di esecuzione (thread)
- unità di allocazione risorse (processi)

Processo e Thread

Processo



Modello ad ambiente globale

Caratteristiche del modello computazionale multithreaded (**modello ad ambiente globale**):

- I thread ***non*** hanno uno spazio di indirizzamento riservato: tutti i thread di un processo condividono lo stesso spazio di indirizzamento → possibilità di definire dati thread-local, sia in Java che in POSIX
- I thread hanno **execution stack** e **program counter privati**
- La comunicazione fra thread può avvenire *direttamente*, tramite la condivisione di aree di memoria → necessità di **meccanismi di sincronizzazione**

Nel modello ad ambiente globale il termine "**processo**" non identifica più un singolo flusso di esecuzione di un programma, ma invece ***il contesto di esecuzione di più thread***, con tutti i dati che possono essere condivisi da questi ultimi.

Vantaggi e svantaggi del Multithreading

- + Context switch e creazione **più leggeri**
- + **Minore** occupazione di risorse
- + Comunicazione tra thread **più efficiente**
- + I thread permettono di sfruttare al massimo le architetture hardware con CPU multiple (**prestazioni**)
- **Problemi di sincronizzazione** più rilevanti e frequenti
- **Memoria virtuale condivisa limitata** alla memoria riservata dal SO per un singolo processo
- **Maggiore difficoltà** nello sviluppo e nel debugging!!!

In presenza di applicazioni che richiedono la condivisione di informazioni si può ottenere un notevole **incremento di efficienza** e un conseguente **miglioramento delle prestazioni**.

Multithreading: esempio di utilizzo

Applicazioni web:

Client-side: i browser web usano i thread per creare più connessioni simultanee verso server diversi, per scaricare diverse risorse in parallelo da uno stesso server, e per gestire il processo di visualizzazione di una pagina web. Tale concorrenza permette di ridurre il tempo di risposta alle richieste dell'utente e quindi aumentare la qualità dell'esperienza percepita.

Server-side: i server web usano i thread per gestire le richieste dei client. I thread possono *condividere in modo efficiente* sia la cache in cui vengono memorizzate le risorse più utilizzate recentemente che le informazioni contestuali.

Altri esempi:

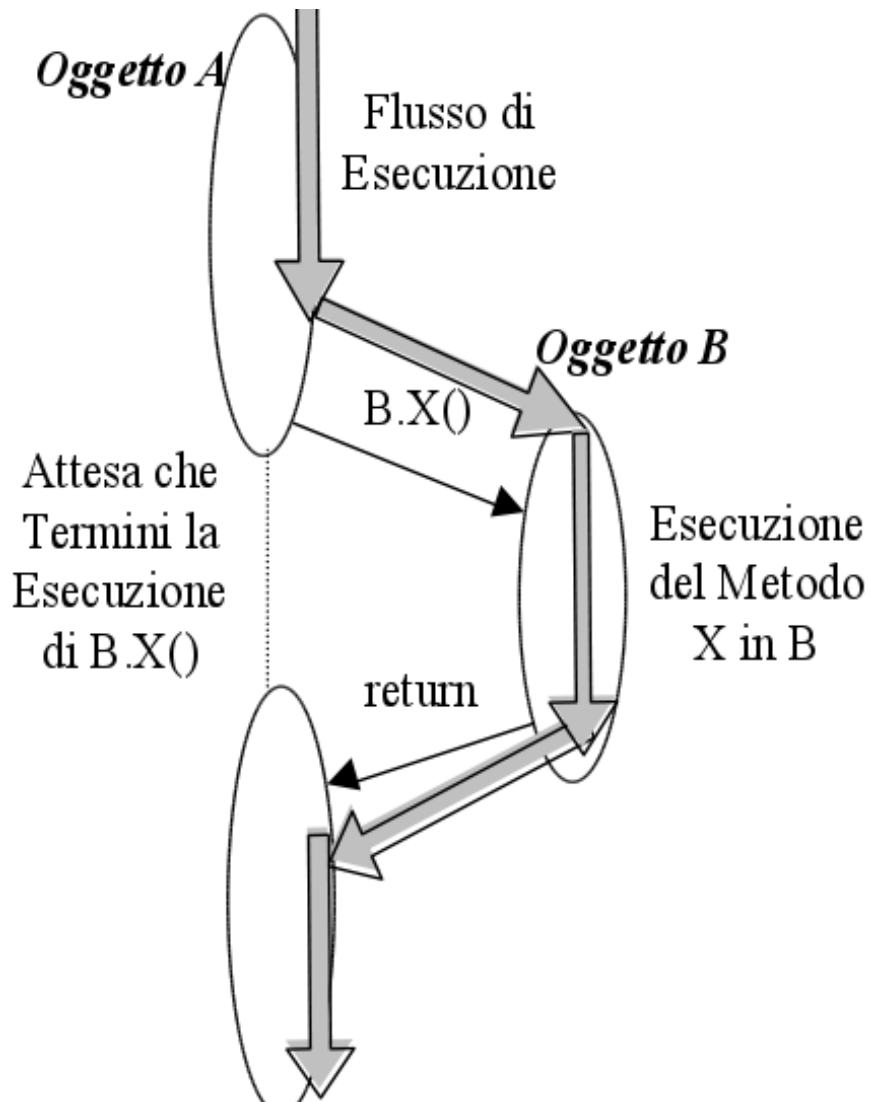
Implementazione di algoritmi parallelizzabili, non-blocking I/O, timer multipli, task indipendenti, responsive User Interface (UI), ecc...

Multithreading in Java – 1/4

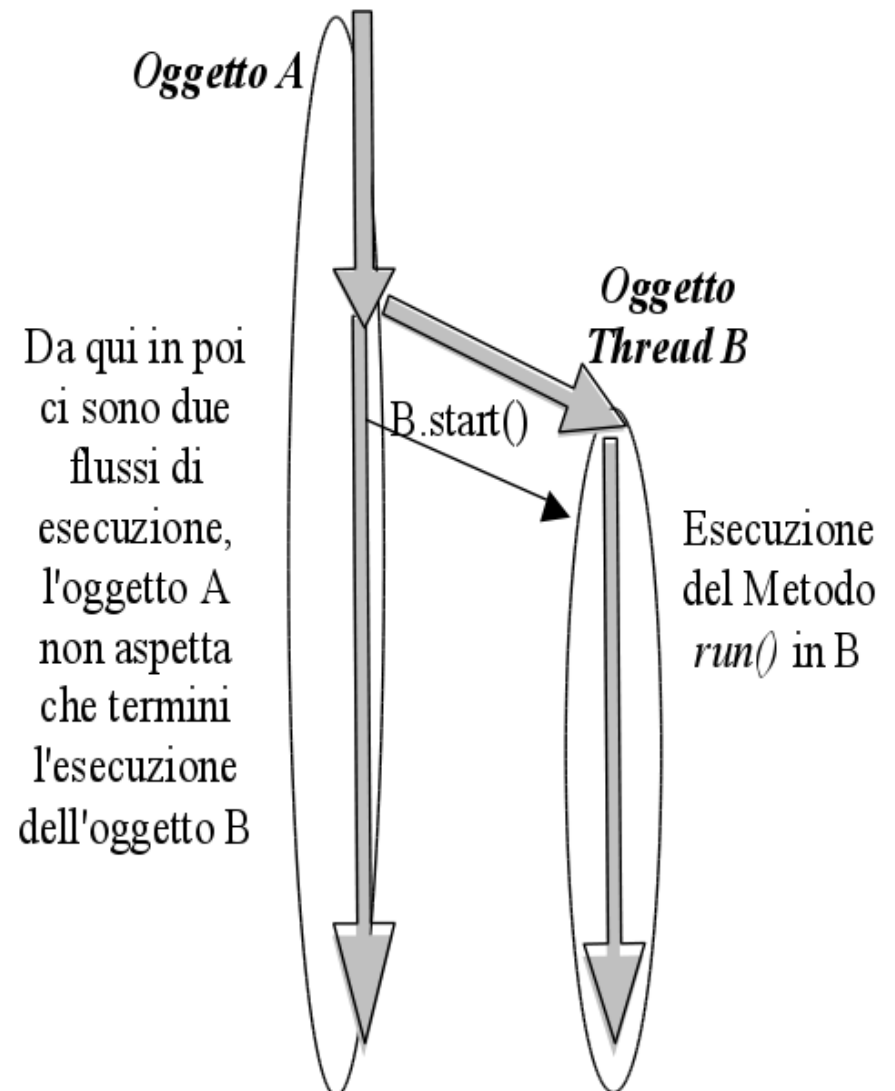
- Il linguaggio Java supporta nativamente il multithreading.
- **Ogni esecuzione della JVM dà origine a un unico processo**, e tutto quello che viene mandato in esecuzione dalla macchina virtuale dà origine a un thread.
- Un thread è un oggetto particolare al quale si richiede un servizio (chiamato `start()`) corrispondente al lancio di un'attività, di un thread, ma che **non si aspetta che il servizio termini**: esso procede in concorrenza a chi lo ha lanciato.

Multithreading in Java – 2/4

Normale Richiesta di Servizio



Richiesta di Servizio start() a un Thread



Multithreading in Java – 3/4

A livello di linguaggio, i thread sono rappresentati da istanze della classe ***Thread***.

Per creare un nuovo thread ci sono due metodi:

1. **Istanziare Thread** passando come parametro un oggetto ottenuto implementando **l'interfaccia Runnable**
2. **Estendere direttamente la classe Thread**

In entrambi i casi il programmatore deve implementare **una classe che definisca il metodo run()**, contenente il codice del thread da mandare in esecuzione.

Multithreading in Java – 4/4

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

L'interfaccia **Runnable** definisce il solo metodo **run()**, identico a quello della classe **Thread** (che infatti implementa l'interfaccia **Runnable**). L'implementazione della interfaccia **Runnable** consente alle istanze di una classe non derivata da **Thread** di essere eseguite come un thread (purché venga agganciata a un oggetto di tipo **Thread**).

Un programma Java termina quando termina l'ultimo dei thread in esecuzione (contrariamente a quanto accade in POSIX threads e nei thread Windows).

Multithreading in Java - Metodo 1

Implementazione interfaccia Runnable

Procedimento:

1. Definire una classe che implementi l'interfaccia **Runnable**, quindi definendone il metodo **run()**.
2. Creare un'istanza di tale classe.
3. Creare un'istanza della classe **Thread**, passando al costruttore un reference all'oggetto *Runnable* creato precedentemente.
4. Invocare il metodo **start()** sull'oggetto *Thread* appena creato. Ciò produrrà l'esecuzione, in un thread separato, del metodo **run()** dell'istanza di *Runnable* passata come argomento al costruttore di *Thread*.

Metodo 1: Esempio 1

```
class EsempioRunnable extends MiaClasse
    implements Runnable {
    public void run() {
        for (int i=1; i<=10; i++)
            System.out.println(i + " " + i*i);
    }
} // fine class MiaClasse

public class Esempio {
    public static void main(String args[]) {
        EsempioRunnable e = new EsempioRunnable();
        Thread t = new Thread (e);
        t.start();
    }
} // fine classe Esempio
```

Multithreading in Java - Metodo 2

Sottoclasse Thread

Procedimento:

1. definire una sottoclasse della classe **Thread**, facendo un opportuno override del metodo **run()**
2. creare un'istanza di tale sottoclasse
3. invocare il metodo **start()** su tale istanza, la quale, a sua volta, richiamerà il metodo **run()** in un thread separato

Metodo 2: Esempio 1/2

```
public class SimpleThread extends Thread{  
  
    public SimpleThread(String str){  
        super(str);  
    }  
  
    public static void main(String[] args){  
        SimpleThread st1 = new  
            SimpleThread("Thread 1");  
        st1.start();  
    }  
}
```


Metodo 2: Esempio 2/2

```
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());  
}  
  
} // fine classe SimpleThread
```

Multithreading in Java - Confronto tra i due metodi

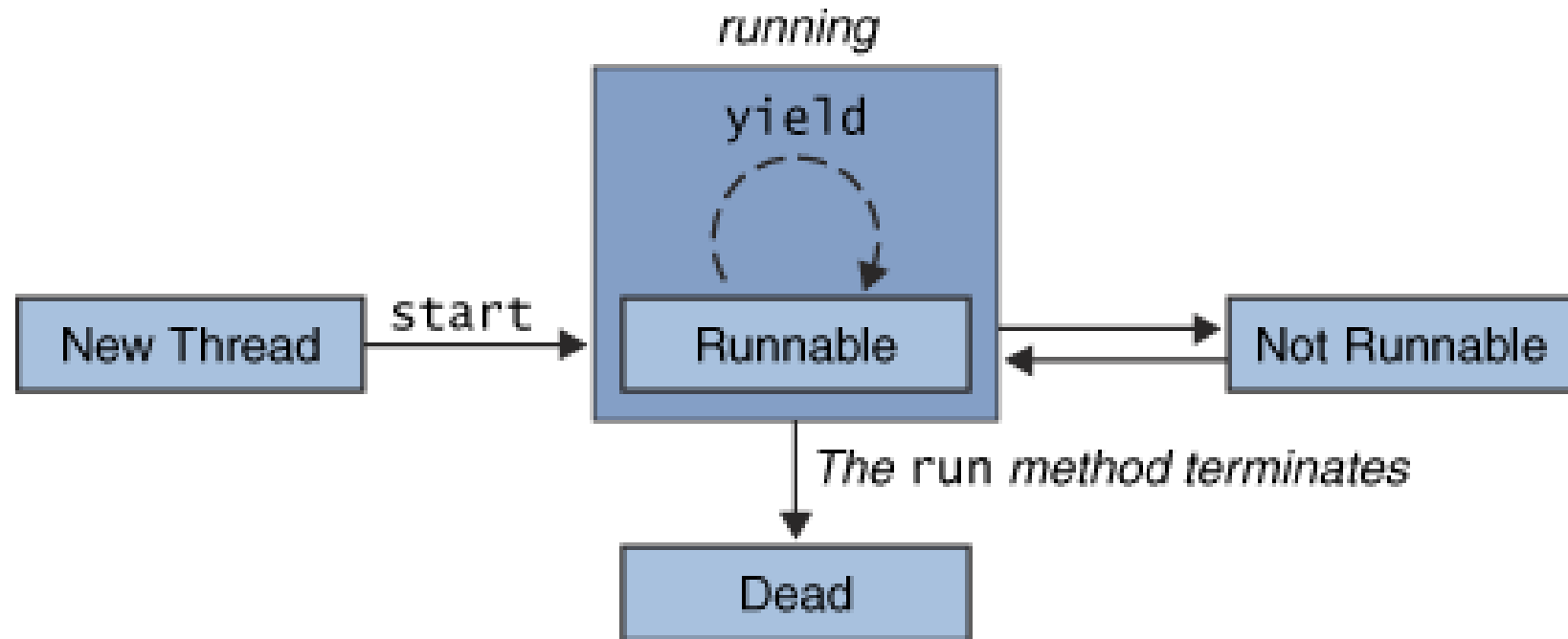
Metodo 1:

- + *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*.

Metodo 2:

- + Modalità *più immediata e semplice*.
- Scarsa flessibilità derivante dalla necessità di ereditare dalla classe Thread, che *impedisce di ereditare da altre classi*.

Java - Ciclo di vita di un thread – 1/2



Durante il suo ciclo di vita, un thread può essere in uno dei seguenti stati:

- New Thread (Creato)
- Runnable
- Not Runnable (Bloccato)
- Dead

Java - Ciclo di vita di un thread – 2/2

- **New Thread (Creato):** (subito dopo l'istruzione new) le variabili sono state allocate e inizializzate. Il thread è in attesa di passare allo stato Runnable, transizione che verrà effettuata in seguito a una chiamata al metodo start.
- **Runnable:** il thread è in esecuzione, oppure pronto per l'esecuzione e in coda d'attesa per ottenere l'utilizzo della CPU.
- **Not Runnable (Bloccato):** il thread non può essere messo in esecuzione dallo scheduler della JVM. I thread entrano in questo stato quando sono in attesa di un'operazione di I/O o sospesi da una primitiva di sincronizzazione come sleep() o wait().
- **Dead:** il thread ha terminato la sua esecuzione in seguito alla terminazione del flusso di istruzioni del metodo run() o alla chiamata del metodo stop() da parte di un altro thread.

Java – Metodi per il controllo di un thread – 1/2

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 (neanche i lock)** 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.

Java – Metodi per il controllo di un thread – 2/2

sleep(long t) blocca per un **tempo specificato** (t) 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 in coda d'attesa.

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

`stop()` e `suspend()` rappresentano azioni “brutali” sul ciclo di vita di un thread → rischio di determinare situazioni di deadlock o di inconsistenze:

- 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 a uno **stato inconsistente** del Sistema

Da JDK 1.4 l'utilizzo dei metodi `stop()`, `suspend()` e `resume()` sono supportati solo per *back-compatibility*, ma il loro utilizzo è **sconsigliato** (**deprecated**)

Si consiglia invece di realizzare tutte le azioni di **sincronizzazione** fra thread tramite i meccanismi di sincronizzazione forniti dal linguaggio

Java – How to Stop a Thread (Sbagliato!)

```
private Thread blinker;
```

```
public void start() {  
    blinker = new Thread(this);  
    blinker.start();  
}
```

```
public void stop() {  
    blinker.stop();    /* ~~~UNSAFE!!!~~~ */  
}
```

```
public void run() {  
    while (true) {  
        try {Thread.sleep(interval);}  
        catch (InterruptedException e) {}  
        repaint();  
    }  
}
```


Java – How to Stop a Thread (Corretto!) 1/2

```
/* Attenzione all'uso di volatile */  
private volatile Thread blinker;
```

```
public void start() {  
    blinker = new Thread(this);  
    blinker.start();  
}
```

```
public void stop() {  
    blinker = null;  
}
```

volatile variable → the value of the variable will **never be cached** thread-locally: all reads and writes will go straight to "main memory"; access to the variable acts as though it is **enclosed in a synchronized block**, synchronized on itself.

Java – How to Stop a Thread (Corretto!) 2/2

```
public boolean isDone() {  
    return blinker == null;  
}  
  
public void run() {  
    while ( !isDone() ) {  
        try {  
            Thread.sleep(interval);  
        }  
        catch (InterruptedException e) { }  
        repaint();  
    }  
}
```

Java - Scheduling dei thread – 1/3

La macchina virtuale Java ha un algoritmo di scheduling basato su livelli di priorità statici fissati a priori (**Fixed Priority Scheduling**):

- il thread da mandare in esecuzione viene scelto fra quelli nello stato *runnable*, tipicamente quello con *priorità più alta*
- il livello di priorità non viene mai cambiato dalla JVM (*statico*)
- le specifiche ufficiali della JVM non stabiliscono come venga scelto il thread da mandare in esecuzione tra quelli disponibili (es. caso di più thread con lo stesso livello di priorità). La politica dipende dalla **specifica implementazione della JVM** (FIFO, Round-Robin, etc...)

Di conseguenza, non c'è nessuna garanzia che sia implementata una gestione in *time-slicing* (*Round-Robin*)

Java - Scheduling dei thread – 2/3

Il thread messo in esecuzione dallo scheduler viene interrotto soltanto se si verifica uno dei seguenti eventi:

- Il thread **termina la sua esecuzione** oppure chiama un metodo che lo fa **uscire dallo stato runnable** (ad esempio, *Yield()*)
- Un thread con **priorità più alta** diventa disponibile per l'esecuzione (ovverosia entra nello stato runnable)
- Il thread **termina il proprio quanto di tempo** (solo nel caso di *scheduling Round-Robin*)
- Il processore riceve un **Interrupt Hardware**

Java - Scheduling dei thread – 3/3

Esempio di codice eseguito su macchine diverse: due Thread eseguiti concorrentemente

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

Round-robin System	FIFO 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

Java - Priorità di Scheduling – 1/2

Per la gestione delle priorità, la classe `Thread` fornisce i metodi:

`setPriority (int num)`

`getPriority()`

- La priorità di un thread è un numero compreso tra `Thread.MIN_PRIORITY` (1) e `Thread.MAX_PRIORITY` (10).
- Il default è `Thread.NORM_PRIORITY` (5).
- La convenzione standard per l'assegnazione di priorità ad un thread è:

Range	Use
-----	-----
10	Crisis management
7-9	Interactive, event-driven
4-6	I/O
2-3	Background computation
1	Run only if nothing else can

Java - Priorità di Scheduling – 2/2

- **Non si devono progettare applicazioni Java assumendo che il thread a priorità più elevata sarà sempre quello in esecuzione, se nello stato runnable.**
- Il meccanismo di assegnazione di priorità diverse ai vari thread è stato pensato solo per consentire agli sviluppatori di dare *suggerimenti* alla JVM per migliorare l'efficienza di un programma.
- Alla fine, è sempre la JVM ad avere l'ultima parola nello scheduling. Diverse JVM possono avere comportamenti diversi.
- L'assegnazione di diverse priorità ai thread non è un sostituto delle primitive di sincronizzazione.

Interferenza fra thread – 1/5

Si consideri il seguente problema di **accesso concorrente** alla medesima risorsa. Due thread devono effettuare 30 versamenti di 100€ sullo stesso conto corrente.

```
import java.lang.*;

public class ProblemaConcorrenza{

    public static void main(String argv[]){
        BankAccount ba = new BankAccount(0);
        Employee e1 = new Employee("Mason", ba);
        Employee e2 = new Employee("Steinberg", ba);
        e1.start();
        e2.start();
    } // fine main

}
```


Interferenza fra thread – 2/5

```
class Employee extends Thread {
    private String _name; private BankAccount _conto;
    private static final int NUM_OF_DEPOSITS = 30;
    private static final int AMOUNT_PER_DEPOSIT = 100;

    Employee(String name, BankAccount conto) {
        _name = name; _conto = conto; }

    public void run() {
        try{
            for (int i = 1; i <= NUM_OF_DEPOSITS; ++i) {
                Thread.sleep(500);
                _conto.increase(AMOUNT_PER_DEPOSIT);
            }
            System.out.println ("Impiegato " + _name + " ha
                                versato un totale di " + NUM_OF_DEPOSITS *
                                AMOUNT_PER_DEPOSIT);
        } catch (InterruptedException e) {}
    } // fine run
} // fine classe Employee
```

Interferenza fra thread – 3/5

```
class BankAccount {  
  
    BankAccount(int initialValue) {  
        _value = initialValue;  
    }  
  
    public void increase(int amount) {  
        int temp = _value;  
        Simulate.HWinterrupt();  
        _value = temp + amount;  
        System.out.println("Nuovo saldo: " + _value);  
    }  
  
    private int _value;  
  
} // fine classe BankAccount
```

Interferenza fra thread – 4/5

```
class Simulate {  
  
    public static void HWinterrupt() {  
        if (Math.random() < 0.5) {  
            try {  
                Thread.sleep(200);  
            }  
            catch (InterruptedException e) {};  
        }  
    }  
  
}
```

Il metodo **HWinterrupt** della classe **Simulate** simula (tramite una chiamata al metodo statico **sleep()** della classe **Thread**) l'arrivo di un **interrupt hardware** che, nel 50% dei casi, deschedula il thread corrente dalla CPU (forzando una sleep di 200ms).

Interferenza fra thread – 5/5

Risultato dell'esecuzione:

```
$ java ProblemaConcorrenza
```

```
[...]
```

```
Nuovo saldo: 4900
```

```
Impiegato Mason ha versato un totale di 3000
```

```
Impiegato Steinberg ha versato un totale di 3000
```

A fronte di un versamento totale di 6000€ ci ritroviamo con un conto di 4900€!!!! Perché?

C'è un problema di **interferenza!** (chiamato anche “**situazione di corsa**”, o “**race condition**”)

Sincronizzazione di thread - 1/2

- Differenti thread condividono lo stesso spazio di memoria (heap)
 - è possibile che **più thread accedano contemporaneamente a uno stesso oggetto**, invocando un metodo che modifica lo stato dell'oggetto
 - stato finale dell'oggetto sarà funzione **dell'ordine con cui i thread accedono ai dati**
- Servono meccanismi di sincronizzazione

Sincronizzazione fra thread - 2/2

- In Java, la sincronizzazione è implementata a livello di linguaggio (modificatore **synchronized**) e attraverso primitive di sincronizzazione di basso (*wait()*, *notify()*, *notifyAll()*) e di alto livello (**cuncurrency utilities**)
- Anche lo standard POSIX definisce funzioni e costrutti per la costruzione e la manipolazione di mutex e di altri meccanismi di sincronizzazione

Accesso esclusivo

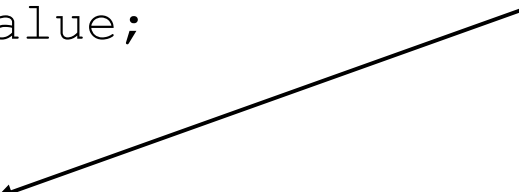
- Per evitare che thread diversi interferiscano durante l'accesso ad oggetti condivisi si possono imporre **accessi esclusivi in modo molto facile** in Java
- JVM supporta la definizione di lock sui singoli oggetti tramite la keyword `synchronized`
- `Synchronized` può essere definita:
 - **su metodo**
 - su singolo blocco di codice (non lo vedremo)

BankAccount – Interferenza risolta

```
class BankAccount {
```

```
    BankAccount(int initialValue) {  
        _value = initialValue;  
    }
```

Il modificatore **synchronized** permette l'accesso in **mutua esclusione al blocco** modificato



```
    public synchronized void increase(int amount) {  
        int temp = _value;  
        Simulate.HWinterrupt();  
        _value = temp + amount;  
        System.out.println("Nuovo saldo: " + _value);  
    }
```

```
    private int _value;  
}
```


Synchronized

In pratica:

- a ogni oggetto Java è automaticamente associato un unico lock
- quando un thread vuole accedere ad un metodo/blocco synchronized, si deve acquisire **il lock dell'oggetto** (impedendo così l'accesso ad ogni altro thread)
- **lock viene automaticamente rilasciato** quando il thread esce dal metodo/blocco synchronized (o se viene interrotto da un'eccezione)
- thread che non riesce ad acquisire un lock **rimane sospeso sulla richiesta** della risorsa fino a che il lock non è disponibile

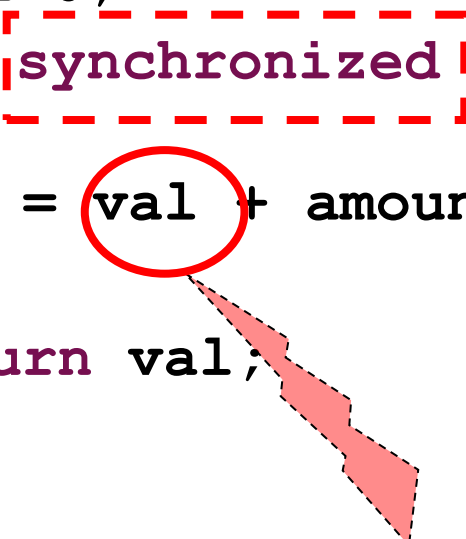
Synchronized

- **Ad ogni oggetto viene assegnato un solo lock** (lock a livello di oggetto, non di classe né di metodo in Java)
 - due thread non possono accedere contemporaneamente a due metodi/blocchi synchronized diversi di uno stesso oggetto
- Tuttavia altri thread sono **liberi di accedere a metodi/blocchi non synchronized** associati allo stesso oggetto

Esempio uso di synchronized

```
public class MyThread extends Thread{  
    private SharedObject obj;  
    ...  
    public void run(){ obj.incr(100); }  
}
```

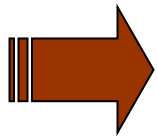
```
public class SharedObject {  
    int val=0;  
    public synchronized int incr (int amount) {  
        val = val + amount;  
        return val;  
    }  
}
```

A red dashed rectangle encloses the 'synchronized' keyword and the 'incr' method in the SharedObject class. A red circle highlights the 'val' variable in the assignment statement 'val = val + amount;'. A red arrow points from the 'val' in the assignment to the 'return val;' statement below it.

Sincronizzazione

Esistono situazioni in cui **synchronized non è sufficiente per impedire accessi concorrenti**

Supponiamo che un metodo synchronized sia l'unico modo per variare lo stato di un oggetto



Che cosa accade se il thread che ha acquisito il lock si blocca all'interno del metodo stesso in attesa di un cambiamento dello stato?

Soluzione: uso di wait()

Soluzione tramite uso di wait()

Thread che invoca wait()

- si blocca in attesa che un altro thread invochi notify() o notifyAll() per quell'oggetto
- deve essere in possesso del lock sull'oggetto
- **al momento della invocazione rilascia il lock**

notifyAll()

- **notify()** - il thread che la invoca
 - risveglia uno dei thread in attesa, scelto **arbitrariamente**
- **notifyAll()** - il thread che la invoca
 - **risveglia tutti i thread in attesa**: essi competeranno per l'accesso all'oggetto

notifyAll() è preferibile (può essere necessaria) se più thread possono essere in attesa

Alcune regole empiriche

1. Se due o più thread possono modificare lo stato di un oggetto, è necessario ***dichiarare synchronized i metodi di accesso a tale stato***
2. Se deve attendere la ***variazione dello stato di un oggetto***, thread deve invocare **wait()**
3. Ogni volta che un metodo attua una ***variazione dello stato di un oggetto***, esso deve invocare **notifyAll()**
4. È necessario verificare che ad ogni chiamata a **wait()** corrisponda una chiamata a **notifyAll()**

Esempio: più produttori - più consumatori 1/2

```
// var available condivisa tra thread stessa JVM
boolean available = true;

public synchronized int get() {
    while (available == false)
        try {
            wait(); // attende un dato dai Produttori
        } catch (InterruptedException e) {}
    }

    /* qui codice accesso a buffer condiviso... */
    available = false;
    notifyAll(); // notifica i produttori del consumo
}
```


Esempio: più produttori - più consumatori 2/2

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

Thread safety – 1/2

- Qualsiasi programma in cui più thread accedono a una stessa risorsa (con almeno un'operazione di scrittura), senza un'adeguata sincronizzazione, è **bacato**. Questo tipo di bug, secondo il quale l'output di una porzione di codice dipende dall'ordine in cui i thread accedono le risorse condivise, è tipicamente molto difficile da individuare e correggere, in quanto potrebbe presentarsi solo in particolari condizioni.
- L'*unico modo* di risolvere il problema della thread safety è quello di utilizzare opportunamente le **primitive di sincronizzazione**.

Thread safety – 2/2

- Una porzione di codice è ***thread-safe*** se si comporta correttamente quando viene usata da più thread, *indipendentemente dal loro ordine di esecuzione, senza alcuna sincronizzazione o coordinazione da parte del codice chiamante.*
- È *molto* più facile scrivere codice thread-safe, pianificando un adeguato uso delle primitive di sincronizzazione al momento del progetto, che modificarlo successivamente per renderlo tale.

Java – Daemon Thread

- I thread in Java possono essere di due tipi: **user thread** e **daemon thread**.
- L'unica differenza tra le due tipologie di thread sta nel fatto che la virtual machine di Java 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 (ad esempio, il *garbage collector*).
- Di default, un thread assume lo stato del thread che lo crea. È possibile verificare e modificare lo stato di un thread con i metodi `isDaemon()` e `setDaemon()`, ma solo prima di mandarlo in esecuzione.

Multithreading in POSIX threads – 1/3

Per realizzare programmi multithreaded in Unix, si deve fare riferimento alla API standard POSIX threads (o Pthreads):

- Va incluso il file di header specifico:
 - `#include <pthread.h>`
- Primitiva di sistema per la creazione (*spawning*) di un nuovo thread:
 - `int pthread_create (pthread_t *t,
 const pthread_attr_t *attr,
 void *(*start_func) (void*),
 void *arg);`
- Primitiva di sistema per effettuare il *join* con un thread:
 - `int pthread_join (pthread_t t, void **retval);`

Multithreading in POSIX threads – 2/3

- Primitiva di sistema per lo spawning di un nuovo thread:
 - `int pthread_create (pthread_t *t,
 const pthread_attr_t *attr,
 void *(*start_func) (void*),
 void *arg);`
- Restituisce il valore 0 in caso di successo, o un valore positivo in caso di errore
- Il thread *t* viene creato con gli attributi specificati in *attr* ed esegue la funzione *start_func*, con argomento *arg*
- Attributi specificano diversi comportamenti: *detached mode*, stack size, scheduling priority, ecc.

Multithreading in POSIX threads – 3/3

- Primitiva di sistema per effettuare il join con un thread:
 - `int pthread_join (pthread_t t, void **retval);`
- Restituisce il valore 0 in caso di successo, o un valore positivo in caso di errore
- Il thread che lancia la chiamata di join si mette in attesa per la terminazione del thread *t*
- Al ritorno dalla chiamata, *retval* conterrà il valore di ritorno della funzione eseguita dal thread *t*
 - *retval* può essere NULL
- Se un thread viene creato in modalità non-detached (default in POSIX), allora la join è necessaria per evitare di lasciare zombie, e quindi un *memory leak*.

Thread detached in POSIX

- Anche Pthreads supporta la creazione di thread di tipo “demone”, che però prendono il nome di ***thread detached***. La differenza fra le due tipologie di thread sta nella *gestione della memoria* e nella *possibilità di effettuare il join* con un altro thread.
- Un **thread non-detached** permette ad altri thread di sincronizzarsi con la sua terminazione, tramite la primitiva *pthread_join()*. È *necessario effettuare la join* per liberare la memoria associata ai thread non-detached che abbiano finito la loro esecuzione (evitare di lasciare *zombie*).
- Un **thread detached** *non è joinable*, ma *il sistema effettua automaticamente il release della memoria* ad esso associata quando questo termina la sua esecuzione (non c'è il problema di thread zombie).

Esempio con POSIX threads – 1/3

Vediamo un semplice esempio di codice:

```
#include <pthread.h>
#include <stdio.h>

void *count_to_five (void *arg) {
    int i;
    for (i = 0; i < 5; ++i) {
        printf("Thr %d: %d\n", pthread_self(), i);
    }

    return NULL;
}
```

Esempio con POSIX threads – 2/3

```
int main (void){
    pthread_t t1, t2;
    void *ret1, *ret2;

    /* Lancia thread figli */
    pthread_create(&t1, NULL, count_to_five, NULL);
    pthread_create(&t2, NULL, count_to_five, NULL);

    /* Attende terminazione thread figli */
    pthread_join(t1, &ret1);
    pthread_join(t2, &ret2);

    return 0;
}
```

Esempio con POSIX threads – 3/3

- Compilazione e linking da terminale:
 - gcc nomefile.c -o threads -lpthread
- Notare che va specificato il linking alla libreria pthread
- Output su macchina Fedora Linux 18, kernel 3.10.13:

Esecuzione 1

- Thr 1717503744: 0
- Thr 1717503744: 1
- Thr 1717503744: 2
- Thr 1717503744: 3
- Thr 1717503744: 4
- Thr 1709111040: 0
- Thr 1709111040: 1
- Thr 1709111040: 2
- Thr 1709111040: 3
- Thr 1709111040: 4

Esecuzione 2

- Thr 2121590528: 0
- Thr 2121590528: 1
- Thr 2113197824: 0
- Thr 2113197824: 1
- Thr 2113197824: 2
- Thr 2113197824: 3
- Thr 2113197824: 4
- Thr 2121590528: 2
- Thr 2121590528: 3
- Thr 2121590528: 4