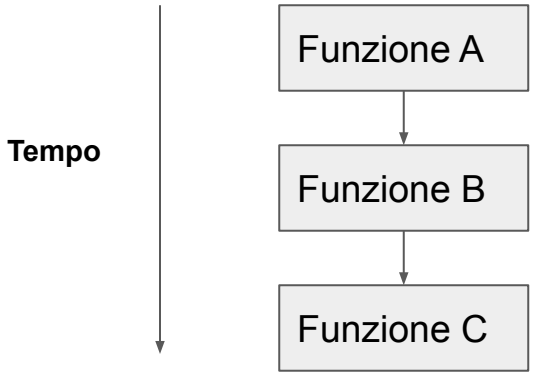
Programmazione Concorrente e multithread

Emanuele Ing. Benatti

Motivazioni - Programmazione Sequenziale

- Fino ad ora, si è applicato il paradigma di programmazione **sequenziale**
- **ogni programma ha un singolo flusso di esecuzione,**
- ogni programma è in grado di eseguire una funzione per volta
- problemi “complessi” nella programmazione sequenziale seguono comunque l’unico flusso di esecuzione.



Motivazioni - Programmazione concorrente

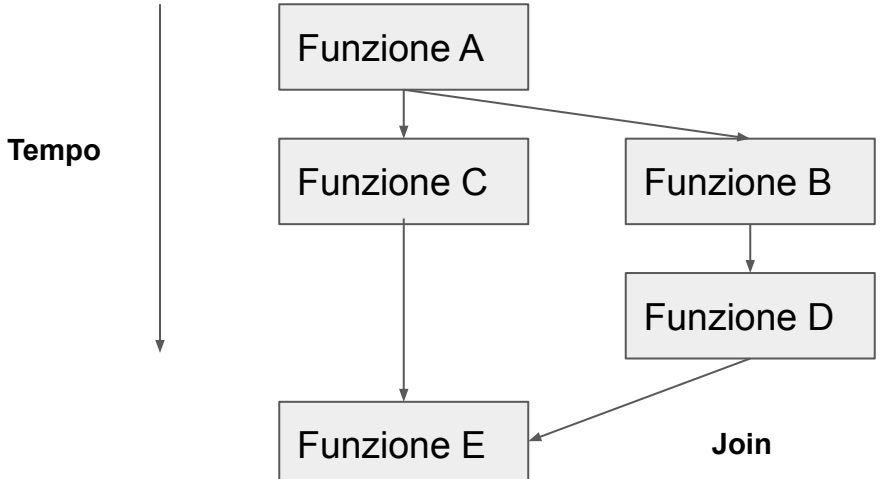
- Negli ultimi anni, spesso gli applicativi software sono diventati “più complessi”
- **l’hardware presente sui dispositivi ha visto un aumento dei core presenti a livello di CPU e dei thread all’interno delle singole CPU.**
- alcuni problemi si descrivono meglio in maniera dividendo un compito in sotto compiti da svolgere in parallelo e in maniera indipendente.

Esempio: un programma browser elabora parallelamente scheda rispetto alle altre gestendo il suo ciclo di vita.

Programmazione concorrente

Programmazione Concorrente: programmazione in grado di eseguire più attività in parallelo sfruttando le caratteristiche hardware della macchina.

Un programma parallelo può avviare nuovi **processi**, o nuovi **thread,** realizzando una contemporaneità reale o simulata dell'esecuzione.



Programmazione concorrente

Un programma (o applicazione) si dice concorrente se è in grado di eseguire più attività (task) parallelamente ("simultaneamente" o "contemporaneamente")

L'unità di esecuzione avviate possono essere nuovi processi (**programma concorrente multi-processo**) o nuovi thread (**programma concorrente multi-thread**)

La contemporaneità può essere:

- Reale nel caso di più esecuzione su CPU o core
- Simulata alternando ciclicamente l'esecuzione blocchi di istruzioni dei diversi rami,

Programmazione concorrente

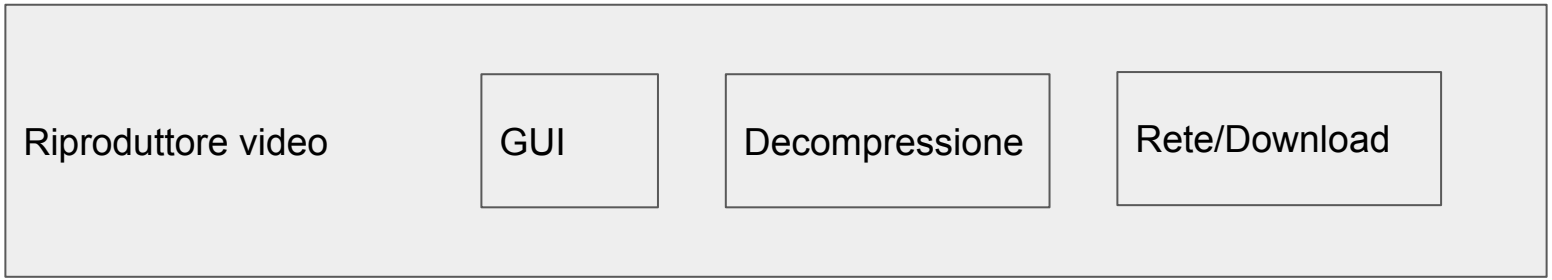
Un programma concorrente ci sono dei rami punti dove l'esecuzione del programma si divide in due flussi di esecuzione. (invocazione di un processo o thread)
Questi flussi di esecuzione possono riunirsi tramite un'operazione di attesa detta **join.**

Questa operazione che due flussi di esecuzione possano riunirsi prevede che ci sia una sorta di implicita sincronia tra i vari flussi di esecuzione e che, comunque, le operazioni fatte dai vari rami portano a dei risultati coerenti con quello che fa il programma concorrente.

Esempi di programmi concorrenti

- **Tutti i programmi a finestre sono multitasking , compreso il S.O.**

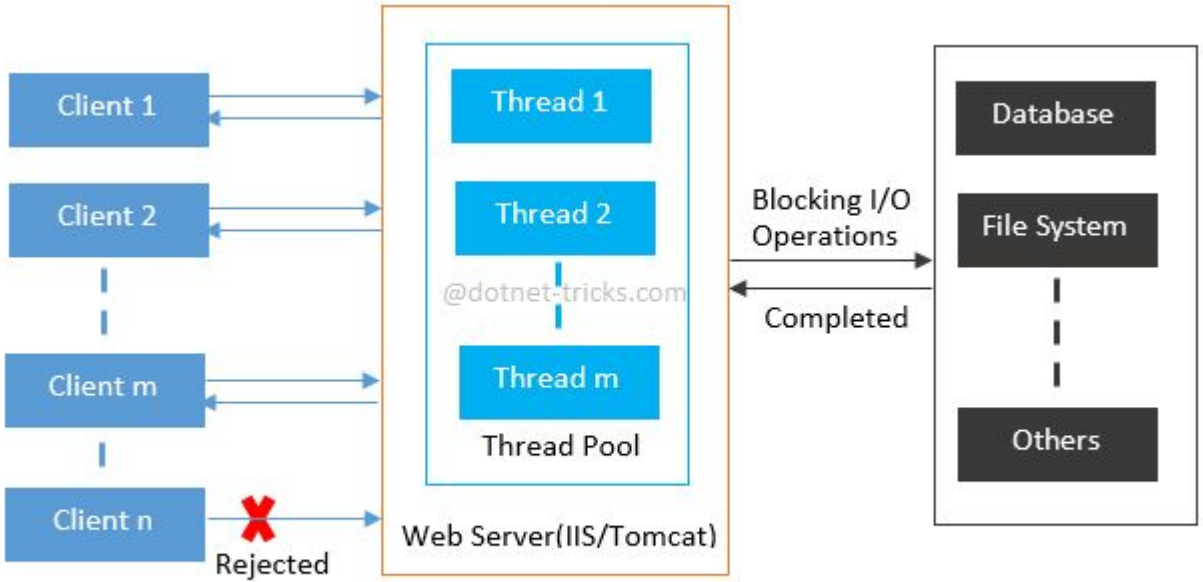
- **Tutte le applicazioni “Complesse” sono multitasking**



- **I server web sono multi tasking**
- **Le app android sono multi tasking**

Esempi di programmi concorrenti - Web Server

-



Multi-Threaded Web Server request processing

Vantaggi programmazione concorrente

- Sfruttare maggiormente le caratteristiche hardware della macchina, **riducendo i tempi di esecuzione** e sfruttando i tempi di I/O
- Possibilità di Pre elaborare informazioni mentre l'utente effettua altre operazioni,
- Specializzare i moduli del software (classi) in modo che sia più facile individuare e gestire le varie parti del SW.

Svantaggi programmazione concorrente

- Programmazione nettamente più complicata rispetto ad un programma sequenziale,
- Necessità di sincronia tra i vari blocchi e “pezzi” del codice,
- Gestione più complessa dei dati condivisi,
- Problemi (eventuali) di Deadlock o corsa critica.

N.B. Ancora oggi molti programmi commerciali e non sfruttano i vantaggi della programmazione concorrente.

Programmazione concorrente: Processi e Thread

Da TDP (3a) e Sistemi e Reti: in informatica ci si basa su due concetti di base:

- Processi
- Thread

Programmazione concorrente: Processi

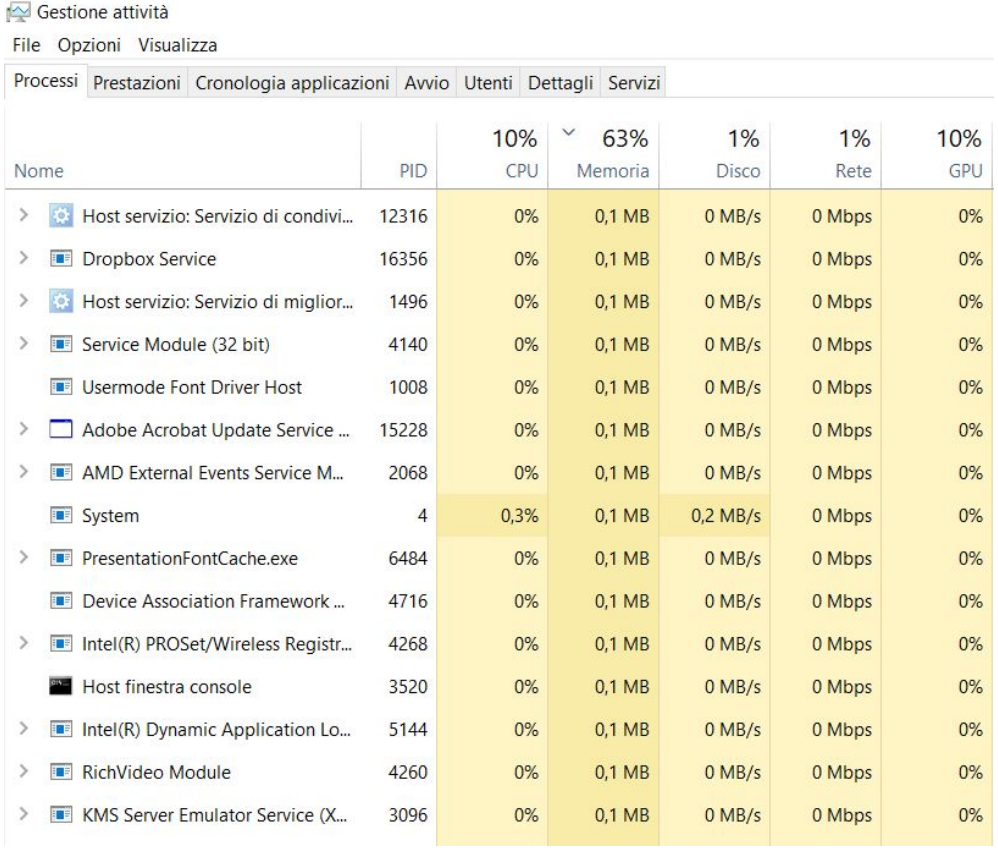
Un **processo** è definito come un **programma in esecuzione** dotato di tutte le risorse necessarie all'esecuzione.

Un processo ha:

- Un PID
- una priorità,
- un utente / sistema che lo ha avviato,
- risorse hardware usate dal processo,
- uno stato del processo (ready, run, wait),
- lo stato dell'ultima esecuzione eseguita,
- il PCB

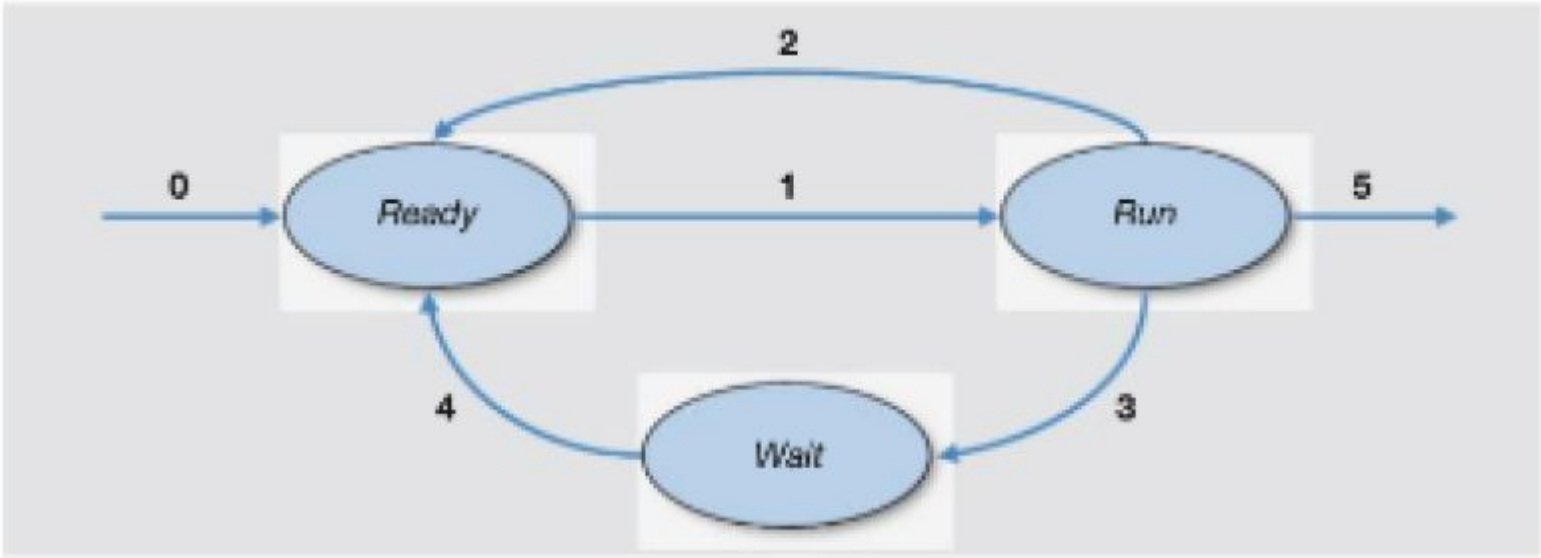
In definitiva: un processo è autonomo e indipendente dagli altri.

Programmazione concorrente: Processi



Nome	PID	10% CPU	63% Memoria	1% Disco	1% Rete	10% GPU
Host servizio: Servizio di condivi...	12316	0%	0,1 MB	0 MB/s	0 Mbps	0%
Dropbox Service	16356	0%	0,1 MB	0 MB/s	0 Mbps	0%
Host servizio: Servizio di miglior...	1496	0%	0,1 MB	0 MB/s	0 Mbps	0%
Service Module (32 bit)	4140	0%	0,1 MB	0 MB/s	0 Mbps	0%
Usermode Font Driver Host	1008	0%	0,1 MB	0 MB/s	0 Mbps	0%
Adobe Acrobat Update Service ...	15228	0%	0,1 MB	0 MB/s	0 Mbps	0%
AMD External Events Service M...	2068	0%	0,1 MB	0 MB/s	0 Mbps	0%
System	4	0,3%	0,1 MB	0,2 MB/s	0 Mbps	0%
PresentationFontCache.exe	6484	0%	0,1 MB	0 MB/s	0 Mbps	0%
Device Association Framework ...	4716	0%	0,1 MB	0 MB/s	0 Mbps	0%
Intel(R) PROSet/Wireless Registr...	4268	0%	0,1 MB	0 MB/s	0 Mbps	0%
Host finestra console	3520	0%	0,1 MB	0 MB/s	0 Mbps	0%
Intel(R) Dynamic Application Lo...	5144	0%	0,1 MB	0 MB/s	0 Mbps	0%
RichVideo Module	4260	0%	0,1 MB	0 MB/s	0 Mbps	0%
KMS Server Emulator Service (X...	3096	0%	0,1 MB	0 MB/s	0 Mbps	0%

Programmazione concorrente: Stati di un Processo



Programmazione concorrente: Processi e Multi tasking

Il PCB ha all'interno tutti i dati necessari a memorizzare lo stato di un processo.

Lo scheduling del Sistema operativo gestisce la sequenza di esecuzione dei processi, cioè l'ordine in cui un processo viene eseguito rispetto all'altro secondo alcuni criteri:

- priorità dei processi,
- tempo di esecuzione del singolo processo,
- eventuali interrupt.

Il cambio tra l'esecuzione di un processo e un altro è detta context switching e prevede la "preparazione dell'ambiente" per l'esecuzione del processo successivo.

Programmazione concorrente: Processi e Multi tasking

Il sistema operativo gestisce lo scorrere dei vari processi come essi vengano effettivamente eseguiti contemporaneamente.

Il fatto di avere sistemi multi core comunque fa fare questo comportamento.

Il S.O. può riservare uno o più core per altri scopi (ad esempio gestire interrupt)

Programmazione concorrente: Time Slice Round Robin

L’algoritmo di scheduling più usato e banale prevede che ogni processo abbia un tempo (ad esempio 20 ms) per eseguire le sue operazioni e, qualora non abbia terminato, esso venga messo in coda a favore di un altro processo.

Esempio A = 40; B = 20; C=40 (ms)

Il context switching ha un tempo diverso da zero ma molto minore dello slice.

Programmazione concorrente: Time Slice Round Robin

Il tempo del “time slice” è fisso e determinato dal S.O. e la CPU in quel tempo viene assegnata ad uso esclusivo, mentre altri processi possono farne richiesta mettendosi in coda (round robin).

Al termine del tempo, il S.O. deve ripristinare l’ambiente di lavoro del processo in modo e impostarlo per il processo successivo in modo che esso possa eseguire i suoi task in modo trasparente da quello che è successo prima.

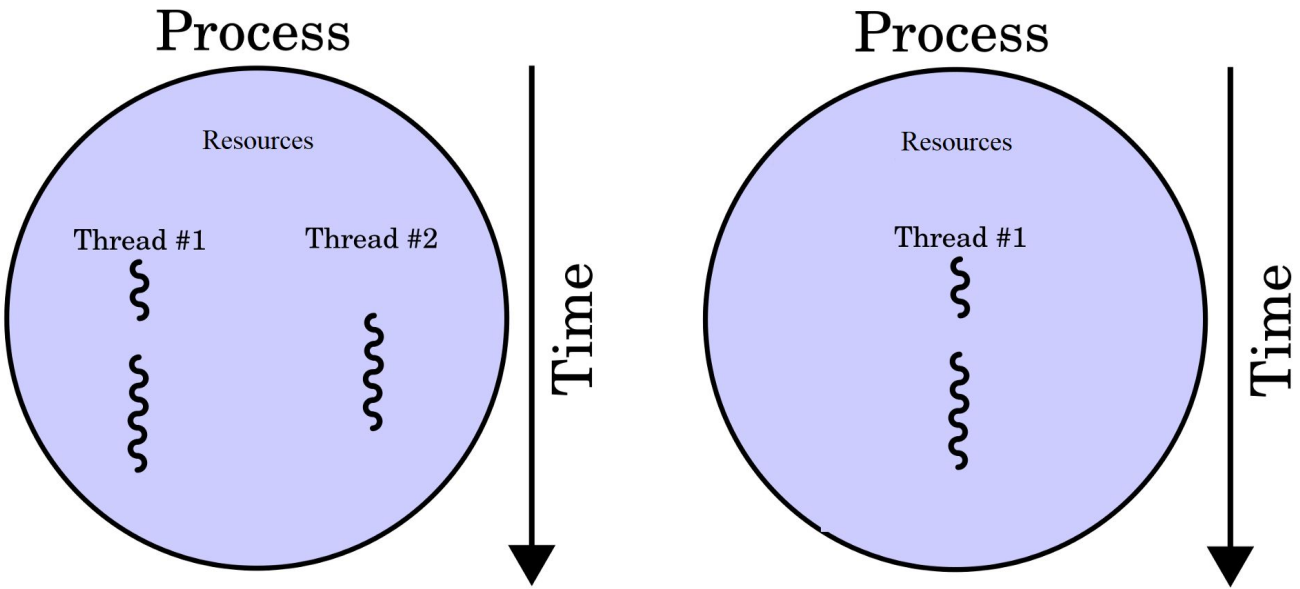
Osservazione: se il context switching impiegasse tanto tempo effettivo per effettuare le operazioni dei processi sarebbe irrilevante.,

Programmazione concorrente: Thread

All’interno di un processo un thread è un “flusso di esecuzione” che:
- condivide (usa) le risorse (file, dispositivi) e la memoria del processo,
- esegue una parte del problema,

Viene detto processo “leggero” in quanto:
- il context switching tra thread è più leggero,
- un processo può avere tanti thread all’interno di esso,
- occupa meno spazio in quanto usa le risorse condivise del processo stesso.

Programmazione concorrente: Thread



Thread a livello utente e a livello Kernel

Mentre i processi sono gestiti a livello di S.O. I thread esistono due tipi di Thread.

A livello Kernel :

- gestiti dal S.O. e riconosciuti da esso,
- di più complessa implementazione software,
- più lenti,
- dipendenti dal S.O. stesso

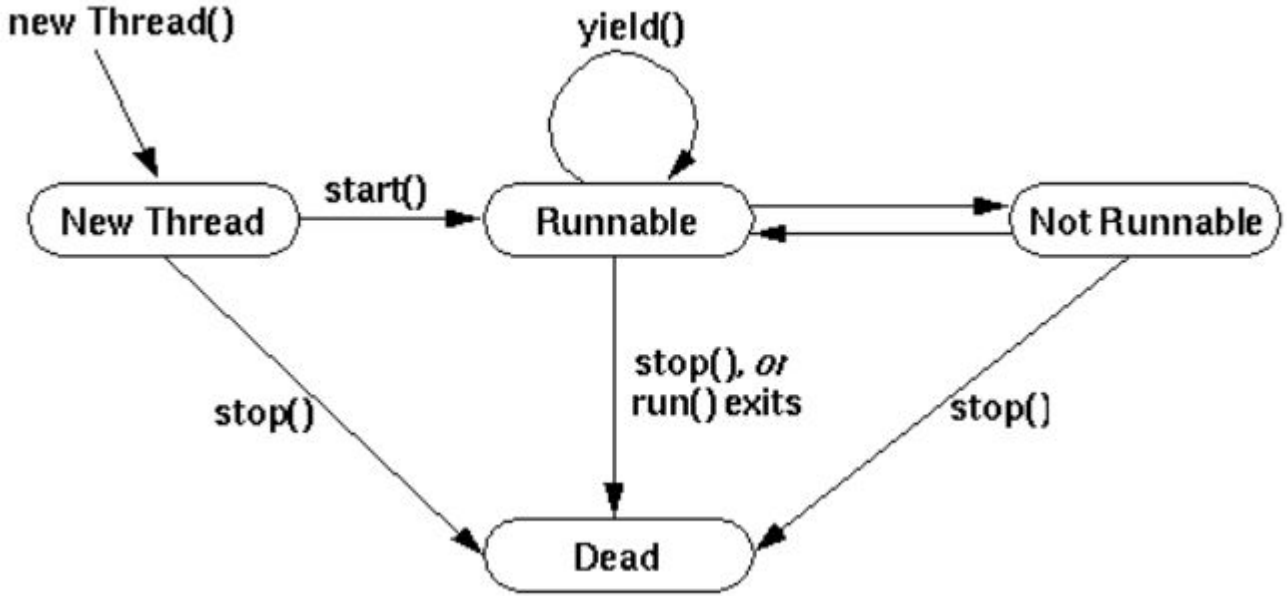
Esempi: Windows Kernel, Solaris

Thread a livello utente

I Thread di livello utente invece permettono come detto di eseguire diversi flussi di esecuzione. Tramite chiamate API alle librerie del linguaggio di programmazione che si sta utilizzando.
- Sono di facile implementazione,
- Riferiscono ad un solo processo, un thread di base non può comunicare con un processo diverso da quello dove viene creato.
- Sono trasparenti a livello di S.O.

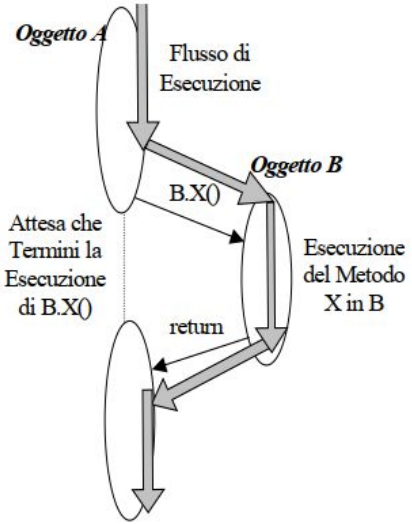
Esempio: Java Thread, C Thread, Posix

Thread a livello utente - Ciclo di vita (2)

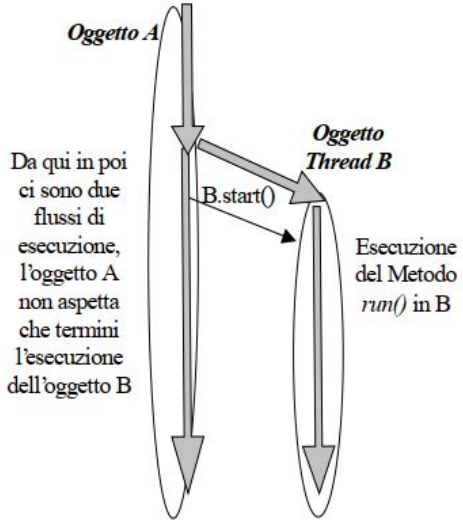


Thread a livello utente - Ciclo di vita

Normale Richiesta di Servizio



Richiesta di Servizio start() a un Thread



Thread in Java

Java supporta i Thread a livello utente tramite:

- L'interfaccia **Runnable**,
- La classe **Thread**,
- La classe **Semaphore**
- **La parola synchronized**

Thread in Java - Hello world

```
public class FirstThread extends Thread {
    public FirstThread () {     }
   public void run() {
      System.out.println(“Hello World”);
   }
}

Nel main
 FirstThread test = new FirstThread ();
 test.start(); //avvio il thread
```

Thread in Java - Hello world

```
public class FirstThread extends Thread {
    public FirstThread () {     }
   public void run() {
      System.out.println("Hello World");
   }
}

Nel main
 FirstThread test = new FirstThread ();
 test.start(); //avvio il thread
```

Start avvia il thread, esegue il costruttore e avvia il metodo run() che è il corpo del thread. Le istruzioni dopo test.start() sono eseguite immediatamente

Thread in Java - Hello world - v2

```
public class FirstRunnable implements Runnable {
   public NamedRunnable() {
   }
   public void run() {  // The run method prints a message to
standard output.
      System.out.println("hello world”);
   }
}
//nel main
FirstRunnable test= new FirstRunnable();
Thread hello= new Thread(test);
hello.start();
```

Un secondo metodo è quello di implementare l’interfaccia runnable come in questo esempio

Thread in Java - Hello world - v2

Vantaggio di questa soluzione: possibilità di sfruttare l’ereditarietà multipla di interfacce usando i Thread.
Cosa che non è possibile fare con extends visto che Java non supporta l’ereditarietà multipla!

Thread in Java - metodi Join() isAlive ed eccezioni

- Ottima cosa nell'aprire un Thread gestire l'eccezione `InterruptedException : eccezione di interruzione`
- `Il metodo join() della classe Thread permette di attendere il termine (se non già avvenuto) di un Thread`
  `es. test.join() ; test.join(500);`
- `Il metodo isAlive()  della classe Thread permette di verificare se un Thread è attivo o no tramite il suo oggetto`

Thread in Java - metodi Join() isAlive ed eccezioni

- il metodo join(500);
  Sospende il thread dove è stato invocato fino a che il thread in attesa termina oppure se non passati 500ms.
- il metodo join();
  Potenzialmente resta in attesa infinita.

Esempio: thread stampa numeri positivi e negativi

- Scrivi un programma Java che avvi due Thread. Il primo stampa i primi 100 numeri positivi e il secondo stampa i primi 100 numeri negativi, in entrambi i casi partendo da zero. (implementabile sia come classe estesa che come interfaccia runnable)

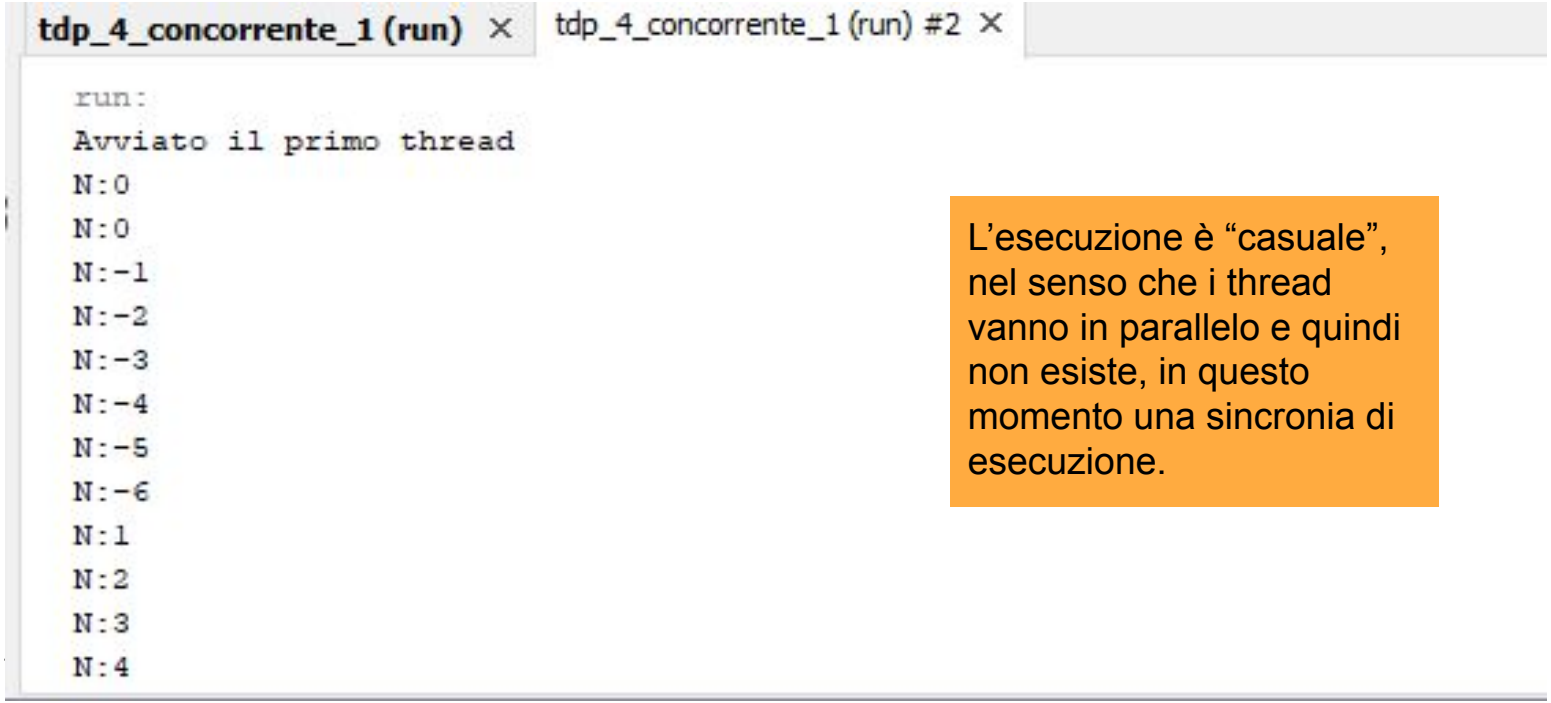
Esempio: thread stampa numeri positivi e negativi

```
public static void main(String[] args) {
        try{

        System.out.println("Avviato il primo thread");
        ThreadPositivi t1 = new ThreadPositivi ();
        ThreadNegativi t2 = new ThreadNegativi ();
        t1.start(); //avvio il thread
        t2.start(); //avvio il thread
        t1.join();
        System.out.println("Terminato il primo thread");
        t2.join();
        System.out.println("Terminato il secondo thread");

    }catch(InterruptedException e)
    {
        System.out.println("Errore nel Thread");
    }
}
```

Esempio: thread stampa numeri positivi e negativi

```
*/
public class ThreadPositivi extends Thread {
    public void run() {
     for( int i=0; i<100; i++ ) {
         System.out.println("N:" + i);
     }
 }
}
```

Esempio: thread stampa numeri positivi e negativi - Esecuzione



L'esecuzione è "casuale", nel senso che i thread vanno in parallelo e quindi non esiste, in questo momento una sincronia di esecuzione.

Metodi Sleep e SetName

In un thread è possibile sospendere l'esecuzione del thread stesso tramite il metodo **Sleep**, ed assegnare un nome al Thread per poterlo distinguere (oppure usare un codice identificativo generato automaticamente).

Esempio

```
    public void run()
{
        Thread.sleep(200) //si interrompe per 200 ms,e genera una
                          //eccezione InterruptedException se fermata
}
```

E' inoltre possibile associare un nome in fase di creazione di un Thread con il metodo SetName

Esempio / Esercizio

Scrivi un programma Java che avvia tre nuovi thread: il primo stampa 100 volte il messaggio "Ciao", il secondo "A" e il terzo "Tutti".

Il thread principale deve assegnare un nome ("T1", "T2", "T3") ad ogni thread

Ogni thread deve effettuare una pausa di un secondo tra un messaggio e il successivo e, oltre al messaggio, deve stampare il proprio nome

Risolvi il problema prima usando una sola classe che estende Thread e poi usando un Runnable

Costruttori di Thread

E’ inoltre possibile creare dei costruttori con parametri per passare valori al thread in fase di creazione.

es. `Thread t1=new Thread(int [] v);`

Sincronia e asincronia tra thread

La parte logicamente più difficile è quella di lavorare con thread diversi se essi necessitano di una forma di sincronia implicita, ed esplicita con altri thread.

Cosa succede se un thread scatena una eccezione, e questa non viene gestita?

Metodi per gestire la sincronia

```
t.stop() ;  //interrompe immediatamente il thread t, metodo deprecato
t.interrupt(); //scatena la richiesta di interruzione per interrompere il thread t
bool isInterrupted(); // restituisce true se il thread t è interrotto, false se è attivo
bool interrupted();   // restituisce true se è in atto una richiesta di interruzione
```

Esempio

Scrivi un programma che avvia un thread T e lo sospenda quando viene premuto un **tasto da tastiera nel programma principale.**

Esempio (codice)

```
1   package esercizio;
2
3   import java.util.Scanner;
4   public class Main {
5
6      public static void main(String[] args) throws InterruptedException
7      {
8          System.out.println("Digita un carattere");
9          Thread t=new myThread();
10         t.setName("T:contatore");
11         t.start();
12         System.out.println(Thread.currentThread().getName());
13         System.out.println("in attesa del carattere");
14         Scanner S=new Scanner(System.in);
15         S.next();
16         t.interrupt();
17
18
19     }
```

Il Thread principale avvia il thread secondario T e si pone in attesa del carattere, disinteressandosi di cosa fa T.

Esempio (codice)

```
package esercizio;

class myThread extends Thread {
    private int contatore=0;
    public myThread() {

    }
    @Override
    public void run() {

     System.out.println(Thread.currentThread().getName());

     while(isInterrupted()==false)
     {
         //System.out.println("Non sono interrotto e ho eseguito il "+this.contatore+" controllo\n");
         this.contatore++;
     }
     System.out.println("Esco dal Thread per interruzione");

    }
}
```

Il Thread secondario si pone in attesa della pressione di un tasto da tastiera nel thread principale (main), e ogni ciclo controlla se è arrivato il momento di interrompersi.

I metodi wait(), notify()

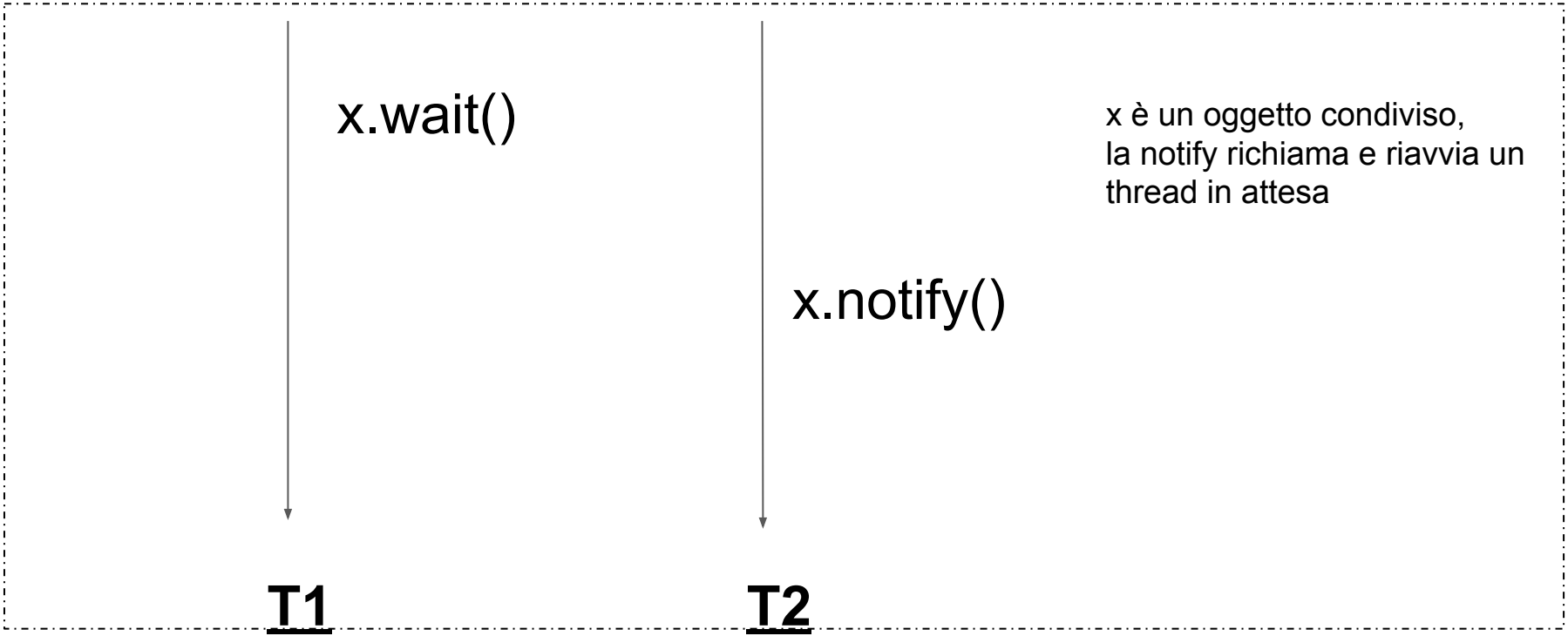
Dato un Oggetto x della classe Object la classe Object ha i seguenti metodi.

`void wait();` // sospende l'esecuzione del thread t fintantochè un altro thread (o il programma principale) invochi un notify() oppure un notifyAll() sull'oggetto x.

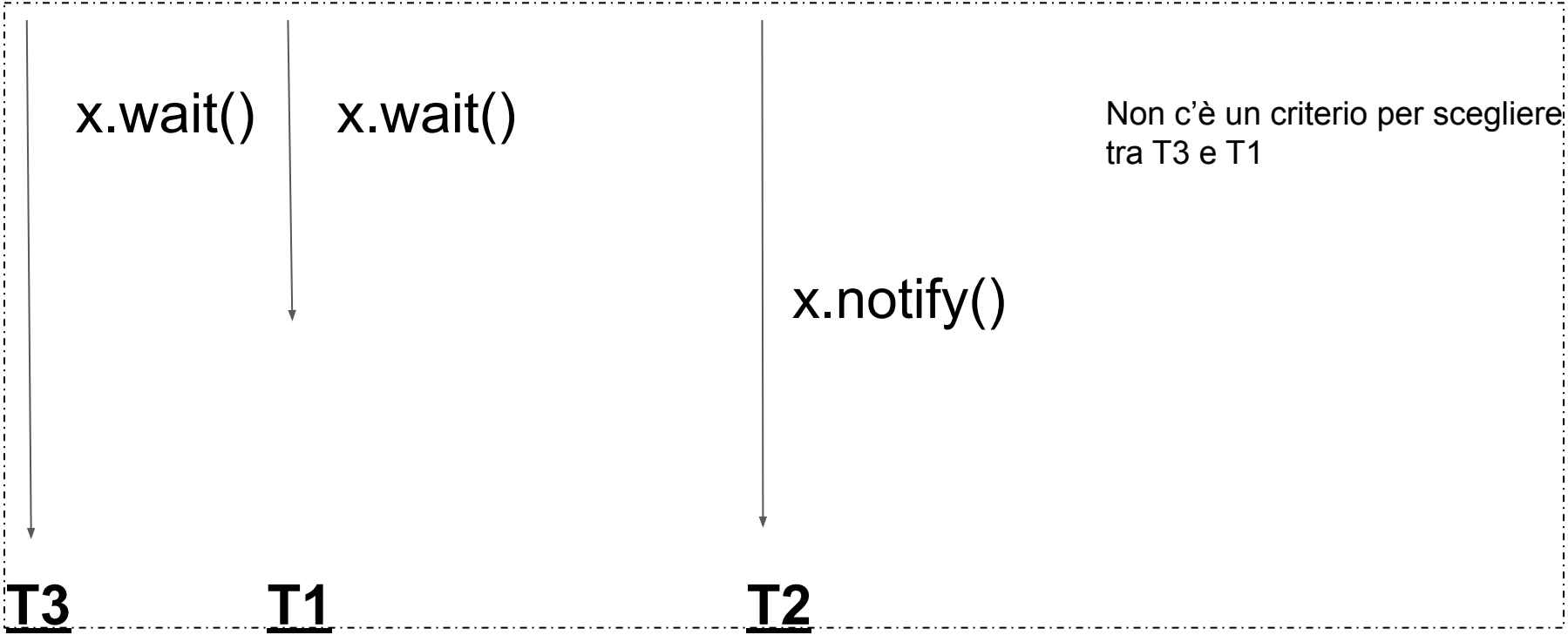
`void notify();` // sblocca la sospensione e fa ripartire l'esecuzione di un qualsiasi thread in attesa dell'oggetto x che si erano messi in wait.

Attenzione: ogni oggetto che invoca la wait si pote in attesa (bloccante) fintantoche qualcuno non invocherà la notify per riattivarlo. Anche più thread possono essere in coda di attesa dell'oggetto x.

I metodi wait(), notify()



I metodi wait(), notify()



I metodi notifyAll() e setPriority()

Dato un Oggetto x della classe Object la classe Object ha i seguenti metodi.

`void notifyAll();` // sblocca la sospensione e fa ripartire l'esecuzione tutti i thread in attesa dell'oggetto x che si erano messi in wait.

void setPriority(int i); // permette di dare una priorità nell'argoritmo di scheduling (tra 1 e 10).

Vedi le costanti **`Thread.NORM_PRIORITY Thread.MAX_PRIORITY e Thread.MIN_PRIORITY`**

Attenzione alle risorse condivise.

Esercizi

Scrivi un programma che conta quanti numeri primi ci sono tra

1 a 100.000 , distribuendo il calcolo su due thread, calcola il tempo di esecuzione con e senza Thread.

V2: rifai l’esercizio con mezzo milione di numeri e con 1 milione, distribuendo il calcolo su un thread ogni 100.000 numeri interi.