

ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

8: Programmazione concorrente

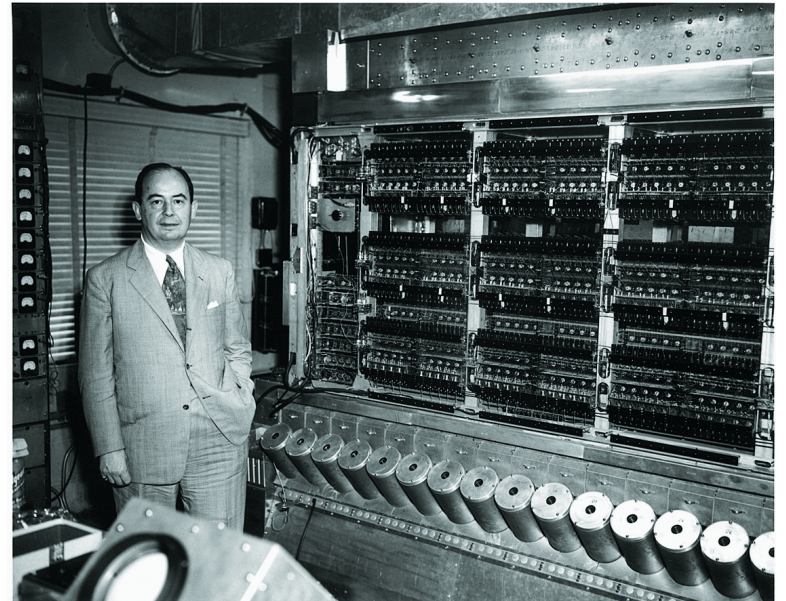
DEFINIZIONI

PROGRAMMAZIONE CONCORRENTE

Teoria e tecniche per la gestione di più processi sulla stessa macchina che operano contemporaneamente condividendo le risorse disponibili.

MOTIVAZIONI STORICHE DELLA PROGRAMMAZIONE CONCORRENTE

La macchina di Von Neumann è un modello utile nella ricerca teorica, ma che è presto è stato molto distanziato dalla realizzazione tecnica



La singola CPU è chiaramente un collo di bottiglia, che la tecnica ha presto individuato e cercato di rimuovere.

La macchina di Turing è un fondamentale risultato teorico, mentre la macchina di Von Neumann è importante dal punto di vista tecnologico, in quanto fornisce una possibile implementazione tecnicamente efficace.

Tuttavia, essa esegue una istruzione alla volta, e questo diventa molto rapidamente poco efficiente; emergono molto presto opportunità per raggiungere una maggiore efficienza al costo di complessità architetturale e allontanamento dalla teoria.

All'inizio degli anni sessanta, l'innovazione dei "channels" nei mainframe IBM permette di avere operazioni di I/O senza occupare la CPU: le periferiche diventano "intelligenti" e possono leggere i nastri o stampare risultati mentre la CPU fa altre operazioni.

Il modello economico spinge quindi ad affrontare il problema del coordinamento fra parti attive che lavorano contemporaneamente per ottenere una maggiore efficienza.

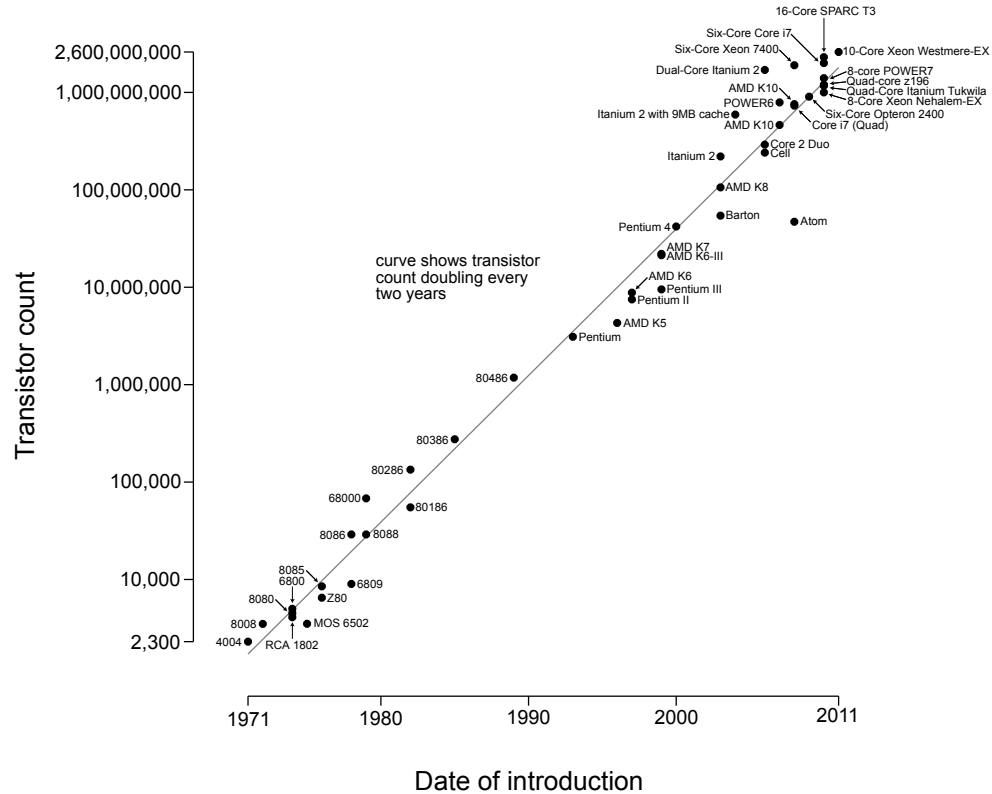
I Sistemi Operativi si trovano così nella necessità di gestire più attività contemporanee o in rapida successione.

Le risorse disponibili vanno distribuite fra queste attività ed utilizzate al meglio.

Per esempio: mentre un programma attende un caricamento da nastro alla memoria, un'altro può effettuare un calcolo che impegna la CPU.

Il risultato finale è che la CPU è efficace per un tempo maggiore, invece di dover attendere l'esecuzione delle operazioni di comunicazione.

Microprocessor Transistor Counts 1971-2011 & Moore's Law

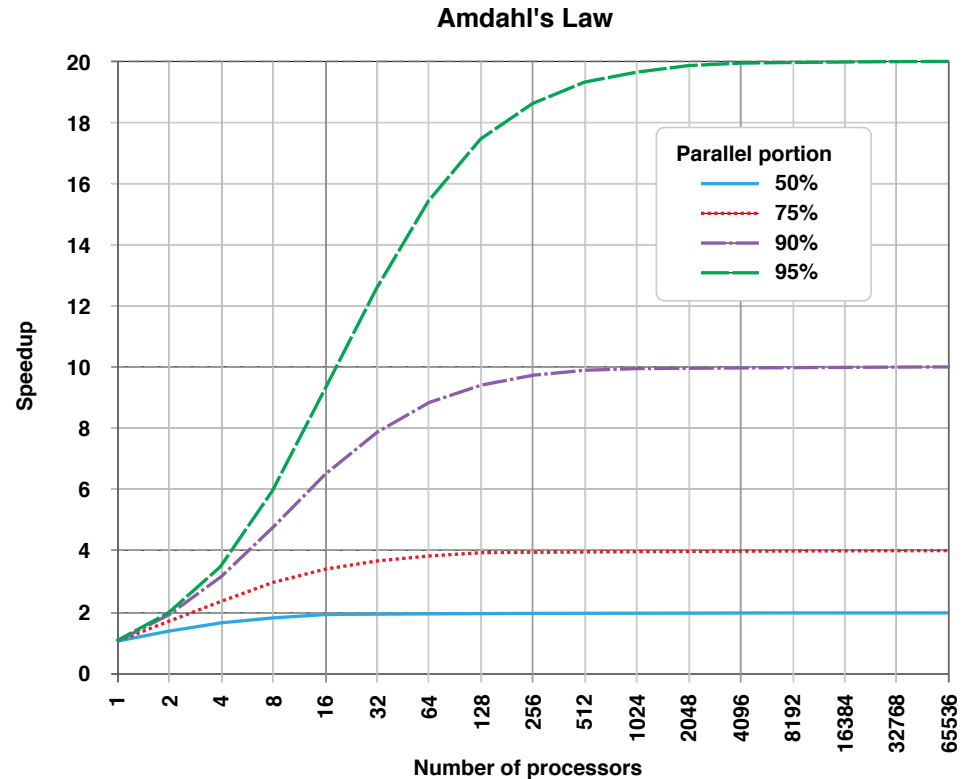


La legge di Moore
(1965) fornisce risorse
sempre crescenti (ma
sempre più
asimmetriche)

Questa legge ha, da qualche anno, raggiunto i limiti fisici del silicio (agli attuali 7nm ci si scontra con la compensazione degli effetti quantistici, oltre con le problematiche energetiche e termiche), e viene rincorsa attraverso la moltiplicazione dei core sullo stesso chip, l'ottimizzazione della concorrenza fra attività in aree diverse del chip, le tecniche di esecuzione predittiva, e così via.

Tutto questo però richiede sempre maggiore gestione della concorrenza e dell'accesso contemporaneo alle stesse risorse. Inoltre, alcune di queste tecniche (l'esecuzione predittiva e la gestione speculativa delle cache) si sono rivelate problematiche dal punto di vista della sicurezza (cfr. Spectre, Meltdown e tutti i lavori successivi)

La legge di Amdahl
(1967) individua i
limiti matematici
della possibile
efficienza che si può
ottenere dalla
parallelizzazione.



**Parallelism is using more resources to get
the answer faster**

Corollary: Only useful if it really does get
the answer faster

@BrianGoetz

Parallelizzare un'attività non è sempre possibile o semplice: il lavoro dedicato a rendere un'attività parallela dev'essere valutato in funzione di quale grado di parallelizzazione si può ottenere e quindi qual'è l'accelerazione che se ne ricava. Non sempre l'investimento può avere un ritorno positivo; molto dipende dal contesto del problema e dell'ambiente di esecuzione.

Nel sistema UNIX (e nei successivi) il principale concetto di gestione delle attività è il Processo.

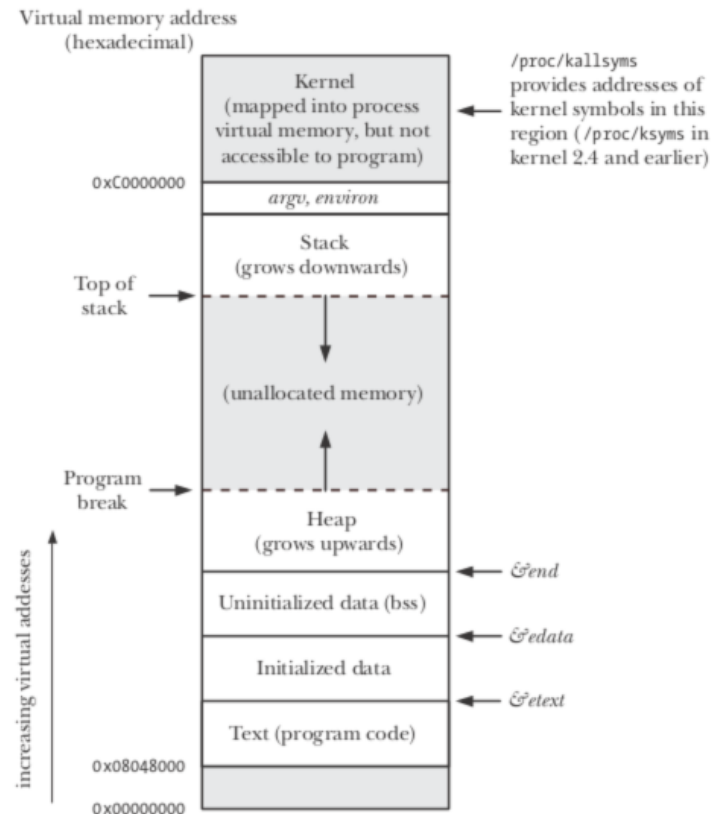


Figure 6-1: Typical memory layout of a process on Linux/x86-32

Un Processo descrive per il sistema operativo un programma in esecuzione e tutte le risorse che gli sono dedicate:

- memoria
- canali di I/O (file, pipe, socket)
- interrupt e segnali
- stato della CPU.

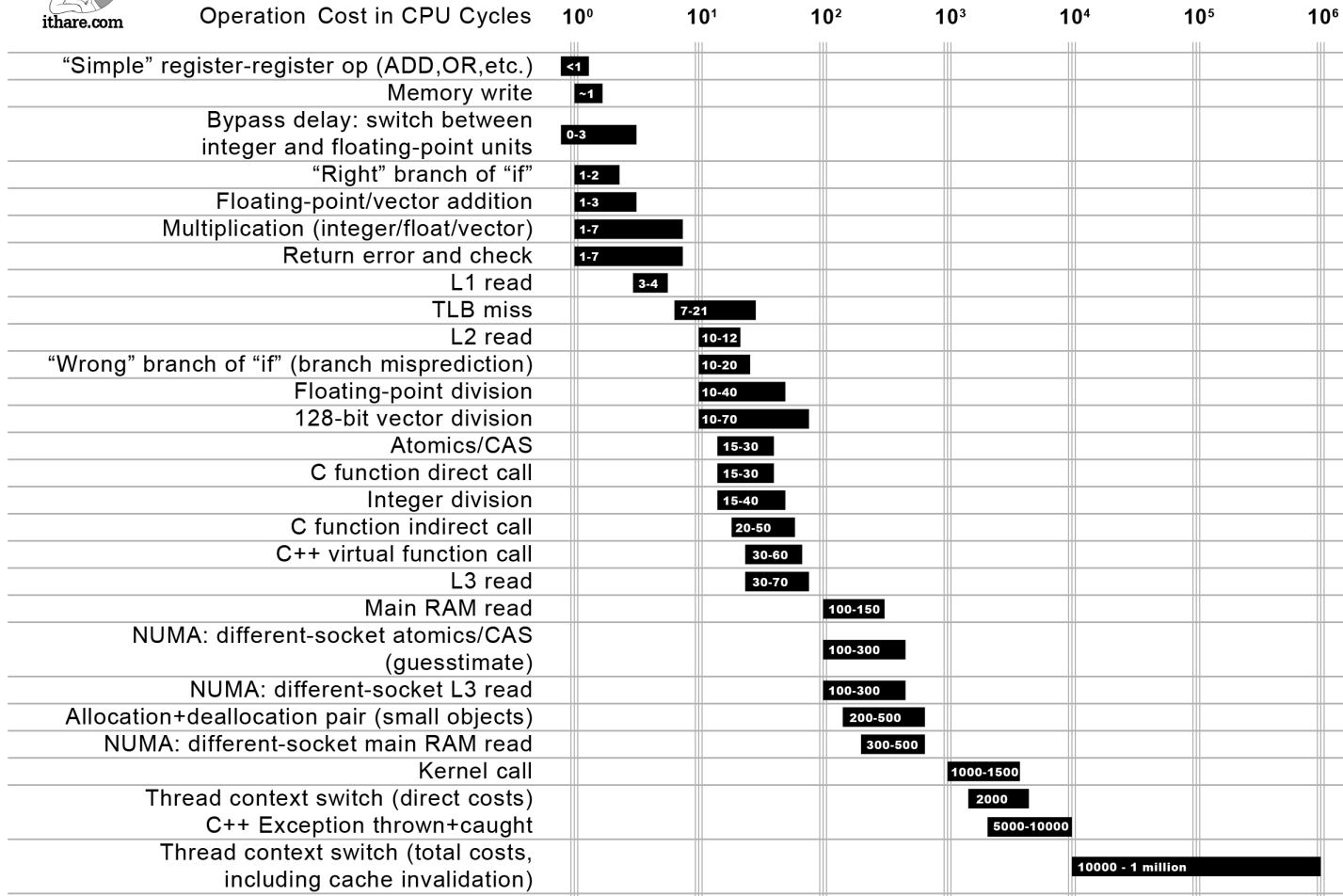
I processi normalmente **non** condividono risorse:
possono comunicare fra loro, ma solo interagendo
come entità separate.

L'obiettivo del sistema operativo è garantire l'utilizzo
efficace e paritario delle risorse, non favorirne l'uso
contemporaneo.

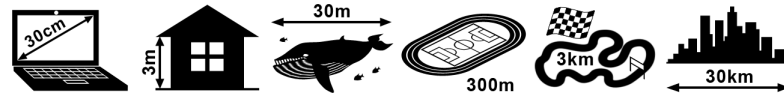
Creare, mettere da parte e portare in esecuzione un Processo sono operazioni relativamente costose:
il contesto di esecuzione deve essere salvato e messo da parte per poter essere recuperato quando è nuovamente il turno di utilizzare la CPU.



Not all CPU operations are created equal



Distance which light travels while the operation is performed



Per gestire più linee di esecuzione all'interno dello stesso processo è stato ideato il concetto di thread.

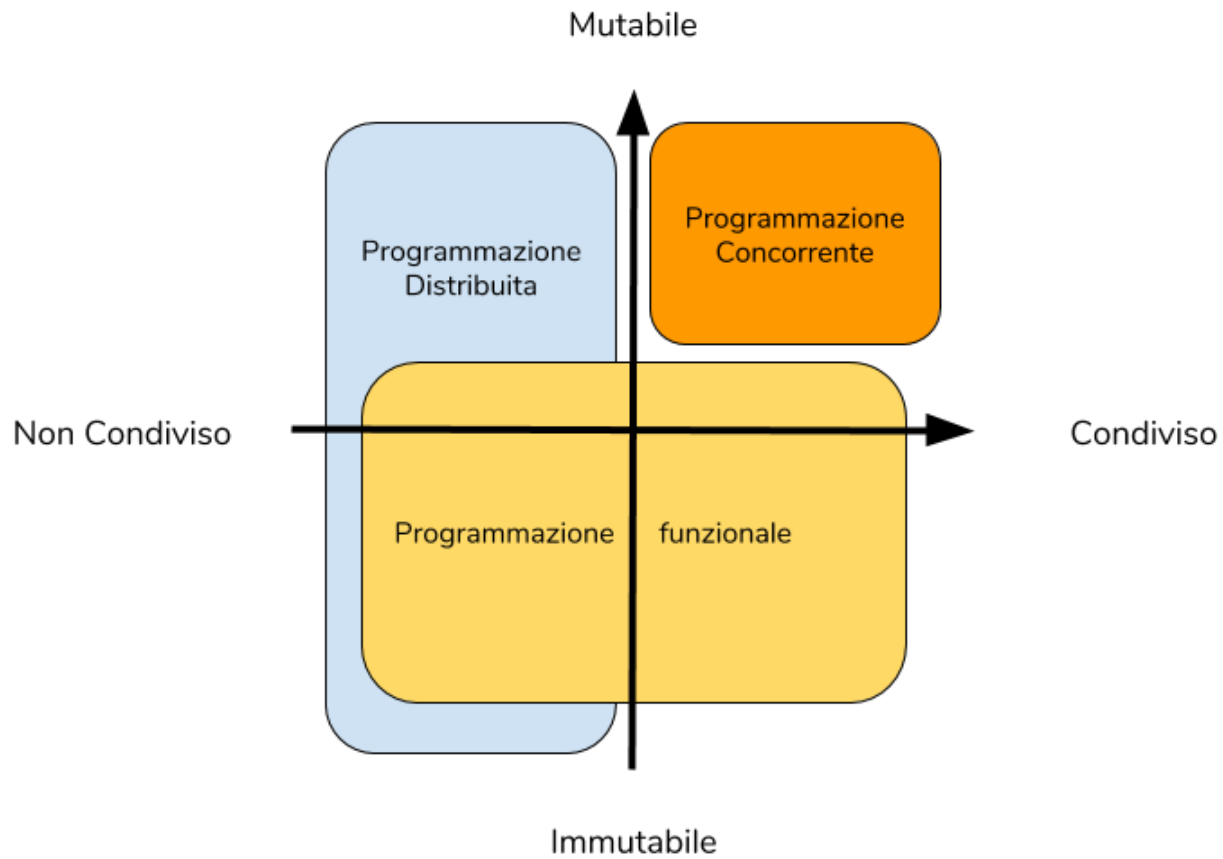


I **thread** condividono le risorse di uno stesso processo, rendendo più economico il costo di passaggio da un ramo di esecuzione all'altro.

Questo riporta però in carico all'applicazione il problema della gestione dell'accesso contemporaneo alle risorse, e della loro condivisione efficace fra i thread.

Per confrontare diversi paradigmi di programmazione,
può essere utile usare come assi di riferimento
l'approccio ai dati locali e la condivisione dello stato
del calcolo:

Dati	Stato
Mutabili	Condiviso
Immutabili	Non condiviso



La **programmazione distribuita** implica la comunicazione fra entità che non possono avere stato condiviso. Come questo stato venga gestito è **ininfluente**.

La **programmazione funzionale** tratta preferibilmente dati immutabili, con qualche concessione alla mutabilità per lo stretto necessario. Lo stato può essere distinto o (specie se immutabile) condiviso.

La **programmazione concorrente** si pone nel quadrante più difficile, dove lo stato è mutabile e condiviso, e quindi l'accesso e l'intervento su di esso va coordinato e gestito.

QUIZ

Dove si colloca la **programmazione ad oggetti** in questo diagramma?

PROBLEMI DELLA CONCORRENZA

Non determinismo

Un'esecuzione concorrente è inerentemente non deterministica.

Starvation

Un thread che non riceve abbastanza risorse non può fare il suo lavoro.

Race Conditions

Se più thread competono per le stesse risorse, il loro ordine di esecuzione può essere rilevante per il risultato.

Deadlock

Se due thread attendono ciascuno la risorsa che ha già preso l'altro, nessuno dei due può proseguire.

COFFMAN'S CONDITIONS

(da "System Deadlocks", ACM Computing Surveys
Giugno 1971)

- Mutual exclusion
- Hold and wait or resource holding
- No preemption
- Circular wait

Le condizioni di Coffman sono necessarie perché un
Deadlock *possa* avvenire.

Rimuovere anche una sola delle condizioni rende
impossibile entrare in un Deadlock.

Rimuovere la *mutua esclusione* può non essere fattibile
per certe risorse

Richiede algoritmi specifici detti *lock-free* o *wait-free*

Rimuovere *l'attesa* può portare a situazioni di starvation o attesa indefinita

Richiede un qualche sistema transazionale per ottenere più risorse contemporaneamente

Introdurre *la pre-emption* può essere estremamente costoso o impossibile

Oltre agli algoritmi lock- e wait-free una soluzione può essere l'uso di una forma di *optimistic concurrency control*.

Rimuovere *la circolarità* richiede imporre
un'ordinamento fra le risorse e la sequenza di
acquisizione

Non sempre è facile da individuare o creare (Dijkstra
propone un algoritmo).

TIPOLOGIE DI CONCORRENZA

Tipo	Strutture
Collaborativa	Co-Routines
Pre-Emptive	Processi, Threads
Real-Time	Processi, Threads
Event Driven/Async	Future, Events, Streams

COLLABORATIVA

I programmi devono esplicitamente cedere il controllo ad intervalli regolari.

E' un modello ancora rilevante in alcuni ambiti
(embedded, very high performance)

PRE-EMPTIVE

Il sistema operativo è in grado di interrompere l'esecuzione di un programma e sottrargli il controllo delle risorse per affidarle al programma seguente.

E' il modello più comune nei sistemi operativi moderni

REAL-TIME

Il sistema operativo garantisce prestazioni precise e prefissate nella suddivisione delle risorse fra i programmi.

E' molto complesso da implementare; solitamente è riservato ad applicazioni molto particolari.

EVENT DRIVEN/ASYNC

I programmi dichiarano le operazioni che vanno eseguite e lasciano all'ambiente di esecuzione la decisione di quando eseguirle e come assegnare le risposte.

Non è comune a livello di sistema operativo, ma sta diventando rapidamente popolare nell'organizzazione delle applicazioni.

JAVA THREADS

Nel linguaggio Java un Thread è rappresentato da una istanza dell'omonima classe.

```
/**  
 * Allocates a new Thread object.  
 *  
 * @param target the object whose run method  
 * is invoked when this thread is started.  
 * If null, this classes run method does nothing.  
 */  
public Thread(Runnable target)
```

Il principale metodo è `start ()`, che avvia un nuovo percorso di esecuzione (similmente ad una *fork*) che lavora all'interno della stessa JVM, condividendo lo stesso heap e quindi lo stesso stato complessivo.

```
/**  
 * Causes this thread to begin execution; the Java Virtual  
 * Machine calls the run method of this thread.  
 */  
void start()
```

Un metodo che useremo spesso negli esempi è `sleep()`, che mette in pausa il thread corrente per un determinato (approssimativamente) lasso di tempo.


```
/**  
 * Causes the currently executing thread to sleep  
 * (temporarily cease execution) for the specified  
 * number of milliseconds, subject to the precision  
 * and accuracy of system timers and schedulers.  
 *  
 */  
static void sleep(long millis)
```

```
/**
 * The Runnable interface should be implemented by any
 * class whose instances are intended to be executed
 * by a thread.
 */
@FunctionalInterface
public interface Runnable {

    /**
     * The general contract of the method run is that
     * it may take any action whatsoever.
     */
    void run();
}
```

Esempio: ThreadSupplier, fornitore di thread che aspettano del tempo

```
@Override
public Thread get() {
    return new Thread(() -> {
        String name = Thread.currentThread().getName();
        long time = waitTime.get();
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
}
```

`it.unipd.app2020.threads.SingleThread:`
lancia un singolo thread

```
public static void main(String[] args) {  
    Thread a = new ThreadSupplier().get();  
  
    out.println("Starting Single Thread");  
    a.start();  
    out.println("Done starting.");  
}
```

`it.unipd.app2020.threads.ManyThreads:`
lancia diversi thread in successione

```
public static void main(String[] args) {  
    var threads = Stream.generate(new ThreadSupplier());  
  
    out.println("Starting Threads");  
    threads.limit(10).forEach((Thread a) -> a.start());  
    out.println("Done starting.");  
}
```

TESTI UTILI:

5 things you didn't know about...

Java 10

<https://www.ibm.com/developerworks/java/library/j-5things17/index.html>

Multithreaded Java programming

<https://www.ibm.com/developerworks/java/library/j-5things15/index.html>

5 things you didn't know about...

`java.util.concurrent`

<https://www.ibm.com/developerworks/java/library/j-5things4/index.html>

<https://www.ibm.com/developerworks/java/library/j-5things5/index.html>

Introducing Junit 5

<https://www.ibm.com/developerworks/java/library/j-introducing-junit5-part1-jupiter-api/index.html>

<https://www.ibm.com/developerworks/java/library/j-introducing-junit5-part2-vintage-jupiter-extension-model/index.html>


Java 8 Idioms

<https://www.ibm.com/developerworks/java/library/j-java8idioms1/index.html>

LINK INTERESSANTI

Per unire l'utile al dilettevole:

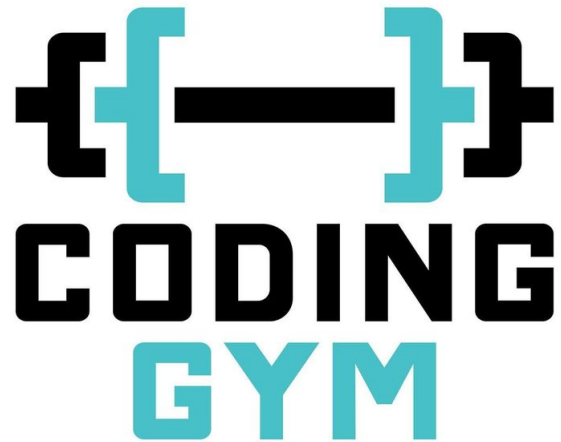
Advent of Code

The logo for Advent of Code, featuring the text "Advent of Code" and "{:year 2019}" in a green, monospace-style font on a dark blue background.

Advent of Code
{:year 2019}

<https://adventofcode.com/>

Coding Gym



<https://coding-gym.org>

