PROGRAMMAZIONE CONCORRENTE E DISTRIBUITA

A.A. 2018/2019

Laurea triennale in Informatica

CHI SONO IO

Michele Mauro

CTO - Warda s.r.l

@michelemauro

@scalagoon

REPERIBILITÀ

michele.mauro@unipd.it

subject: [pcd]

MATERIALE

https://bitbucket.org/albadmin/pcd1819-mod2/

Nel repository trovate:

- queste slide in formato originale e PDF
- il sorgente degli esempi presentati
- il build.gradle per eseguire il codice
- alcuni paper citati nelle slide

ARGOMENTI: CONCORRENZA

Threads

Locking

Concurrent Data Structures

Parallel Streams

Laboratorio concorrenza

Altre astrazioni

ARGOMENTI: DISTRIBUZIONE (2/2)

Distribuzione.

Sockets.

Serializzazione.

Laboratorio Distribuzione

Stato distribuito

DEFINIZIONI

PROGRAMMAZIONE CONCORRENTE

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

Speaker notes

L'enfasi è sul fatto che i processi condividono le stesse risorse, sullo stesso hardware. Quando si parla di programmazione concorrente, la gestione delle risorse comuni è l'argomento principale, e non può essere nascosto da un'astrazione. Le tecniche di programmazione concorrente sono volte a costruire un modello con cui maneggiare e gestire l'accesso alle risorse in comune.

PROGRAMMAZIONE DISTRIBUITA

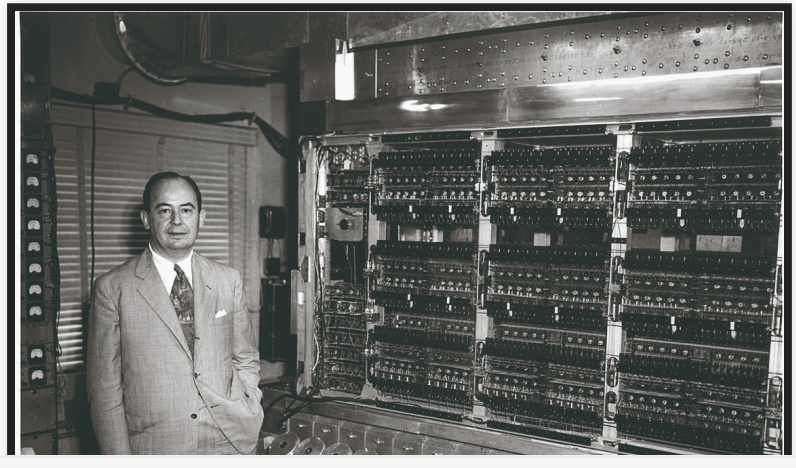
Teoria e tecniche per la gestione di più processi su macchine diverse che operano in modo coordinato allo svolgimento di un unico compito.

Speaker notes

Qui l'enfasi è sulla separazione fra i processi su macchine diverse. Quindi, le tecniche di programmazione concorrente si occupano di comunicazione, trasmissione di messaggi o dello stato di un processo verso un'altra macchina, di collaborazione attraverso questo scambio di messaggi e informazioni. Le tecniche forniscono un modello della comunicazione fra processi su macchine differenti, con tutte le problematiche che questa comunicazione può comportare.

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



Speaker notes

Immagine da https://www.ias.edu/ideas/2012/george-dyson-ecp

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

Speaker notes

Meno tempi di attesa di I/O significa maggiore sfruttamento della CPU ed in definitiva prestazioni migliori a parità di tempo; in un regime di noleggio questo è un incentivo economico diretto.

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.

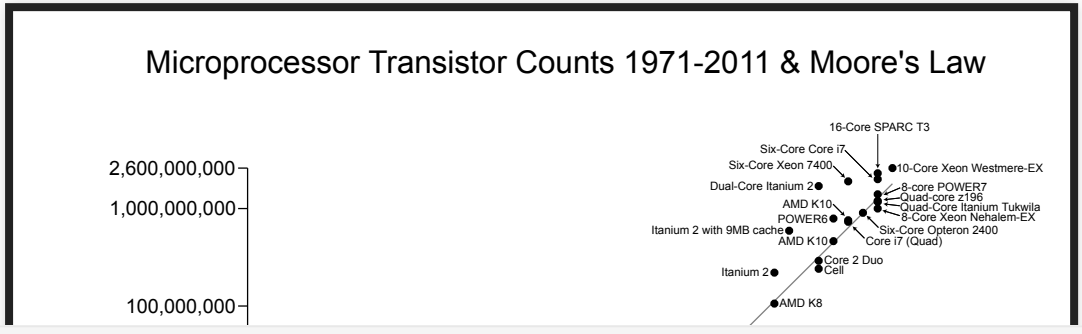
Le risorse disponibili vanno distribuite fra queste

Speaker notes

La concorrenza diventa immediatamente una funzione di alto livello in carico al sistema operativo, vista anche la necessità di lavorare a strettissimo contatto con la gestione delle risorse.

Per esempio: mentre un programma attende un caricamento da nastro alla memoria, un'altro può effettuare un calcolo che impegna la CPU. Il sistema operativo mantiene la contabilità (in modo a volte letterale: il costo era orario, come le risorse cloud di oggi; del resto si trattava in modo analogo di "ferro" non di proprietà dell'utente). Il risultato finale è che la CPU è occupata per maggior tempo, invece di dover attendere l'esecuzione delle operazioni di comunicazione.

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



Speaker notes

"Il numero di transistor per chip raddoppia ogni anno"

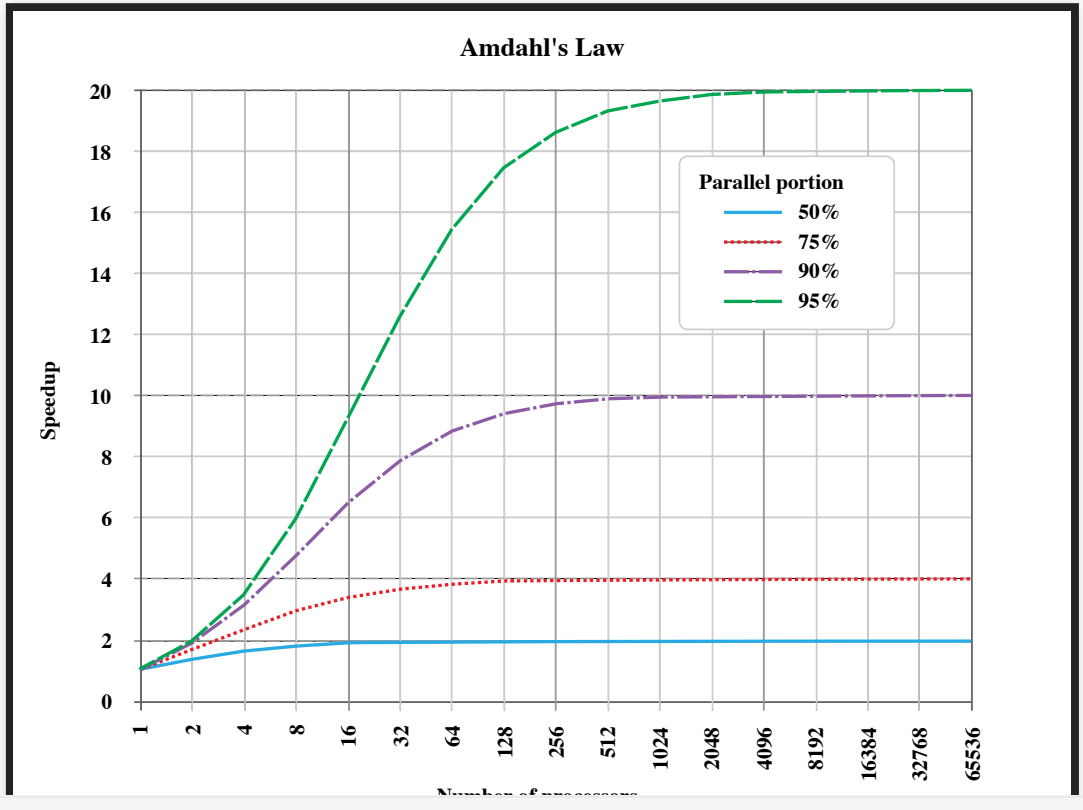
Immagine By Wgsimon - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15193542

Immagine da https://en.wikipedia.org/wiki/Moore%27s_law#/media/File:Transistor_Count_and_Moore%27sLaw-_2011.svg

Trovate il paper originale di Moore in papers/l1.

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 Amhdal (1967) individua i limiti matematici della possibile efficenza che si può ottenere dalla parallelizzazione.



Speaker notes

"Lo speedup dipende dalla parte parallelizzabile del programma da eseguire" Immagine By Daniels220 at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6678551

Un riassunto dell'intervento di Amdhal in cui viene enunciata la legge è presente in papers/l1.

Parallelism is using more resources to get the answer faster

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

@BrianGoetz

Speaker notes

https://qconlondon.com/london-2017/system/files/presentation-slides/concurrenttoparallel.pdf

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.

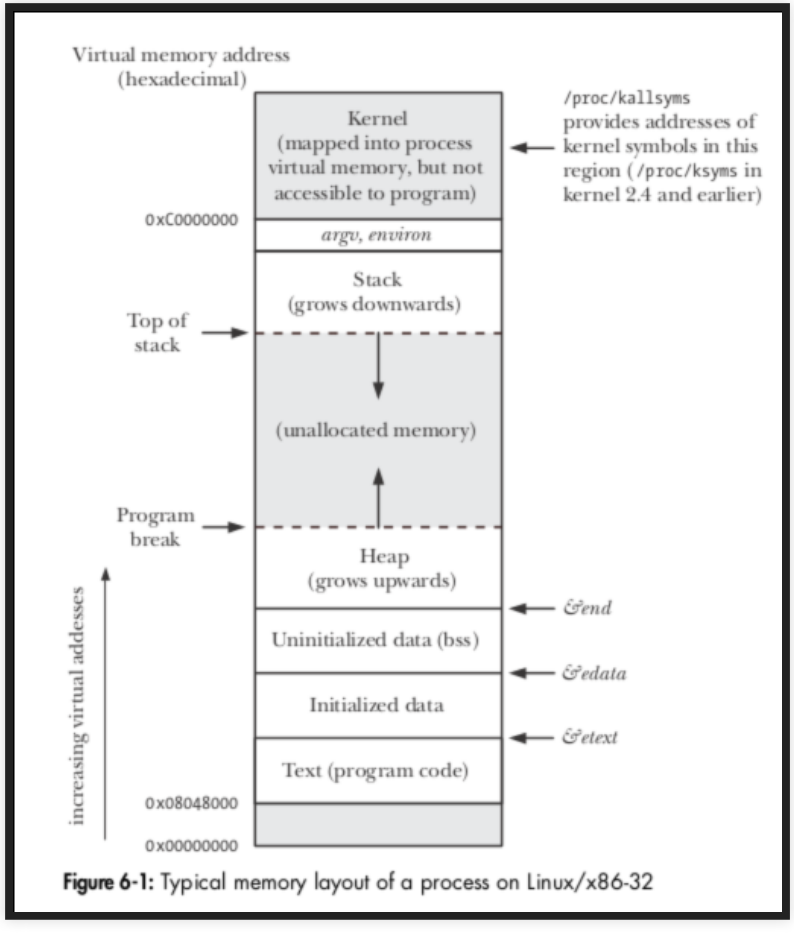


Figure 6-1: Typical memory layout of a process on Linux/x86-32

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

Speaker notes

Immagine da http://www.programering.com/a/MzNwUjMwATM.html

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

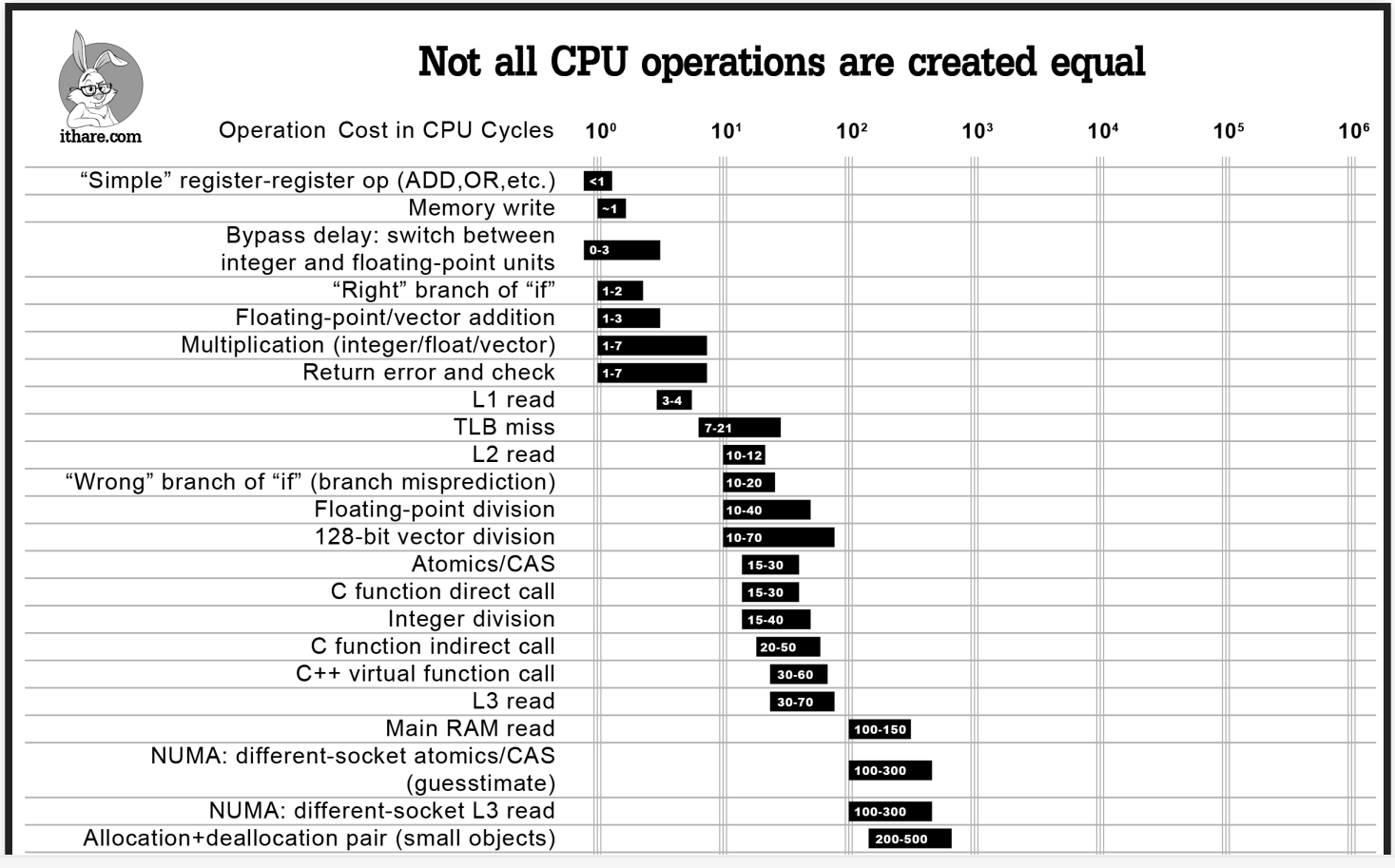
- memoria
- canali di I/O (file, pipe, socket)
- interrupt e segnali

Speaker notes

il sistema operativo maneggia la suddivisione delle risorse fra le varie attività in corso attraverso la metafora del processo. Diventa necessario il supporto di una gerarchia di operazioni: alcune possibili solo dal sistema operativo, altre disponibili anche ai processi; solo in questo modo si può garantire la corretta collaborazione fra le varie attività ed impedire che una si appropri di tutte le risorse a scapito delle altre (cfr. Problema di 5 filosofi).

Creare o passare da un Processo ad un altro è un'operazione relativamente costosa:

il contesto di esecuzione deve essere salvato e messo da parte per poter essere recuperato quando è nuovamente il turno di esecuzione.

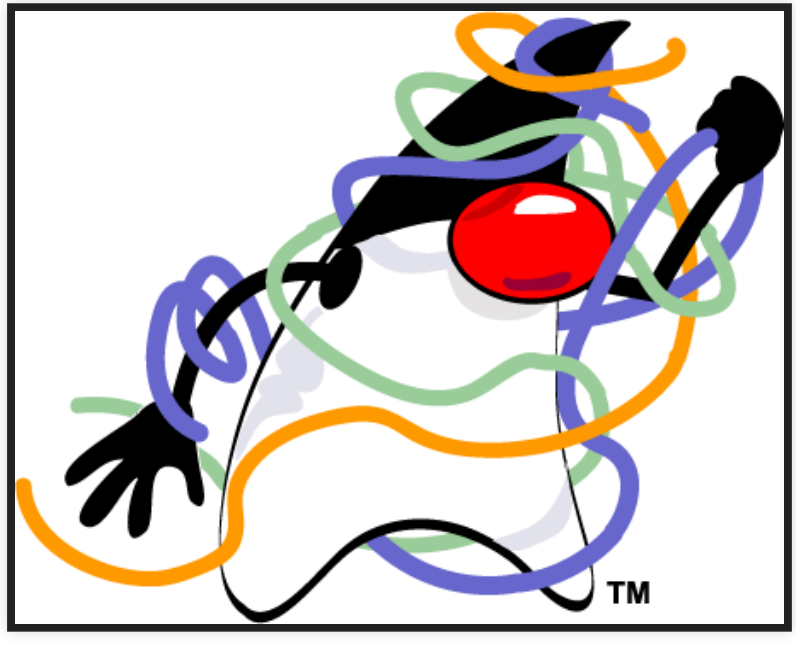


Speaker notes

da http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Notate che la scala del grafico è logaritmica: ogni colonna corrisponde ad un ordine di grandezza. Anche le operazioni di cambio di contesto fra thread sono comunque costose, quelle di cambio di contesto fra processi possono essere ancora più costose perché possono includere spostamenti di ampie porzioni della RAM, molteplici chiamate al kernel e altro.

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



Speaker notes

Immagine da http://crunchify.com/java-simple-thread-example/

Il thread è un processo all'interno di un processo.

I thread di 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.

Speaker notes

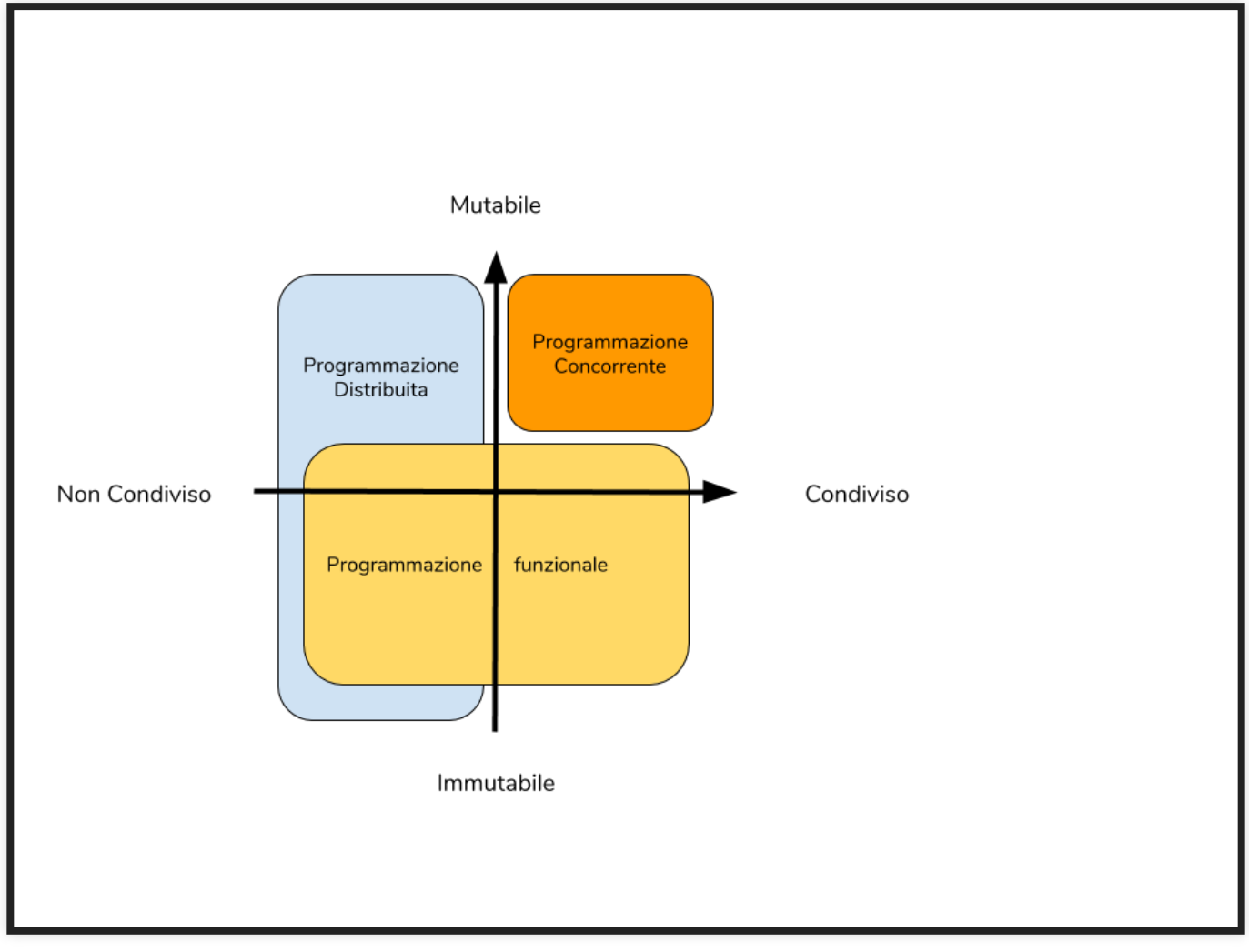
Ci troviamo nuovamente sul problema dei 5 filosofi. Non solo, ora è esplicitamente responsabilità dell'applicazione gestire le risorse fra i suoi diversi thread ed evitare conflitti.

Per collocarci rispetto ad altri paradigmi, usiamo come assi di riferimento l'approccio ai dati locali e la condivisione dello stato del calcolo:

Dati	Stato
Mutabili	Condiviso
Immutabili	Non condiviso

Speaker notes

Intendiamo come "approccio ai dati" l'uso di variabili locali mutabili o immutabili: come vedremo, questa caratteristica è importante perché rimuove alcune problematiche, impedendo però comportamenti che potrebbero essere interessanti. La condivisione o meno dello stato invece è il semplice fatto di accedere contemporaneamente, da parte di più linee di esecuzione, agli stessi dati.


Mutabile
Programmazione Distribuita
Programmazione Concorrente
Non Condiviso
Condiviso
Programmazione funzionale
Immutabile

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.

Speaker notes

Abbiamo consegnato ad altri il controllo della sequenza di esecuzione, e questa sequenza dipende da eventi esterni e contingenti (carico istantaneo, azioni dell'utente, segnali dall'esterno).

Starvation

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

Speaker notes

Se il thread fa parte di una sequenza di attività, le attività successive che dipendono dal suo lavoro rimarranno non svolte o saranno in ritardo

Race Conditions

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

Speaker notes

E' inoltre assai difficile da verificare tale ordine di esecuzione, perché può dipendere dagli stessi fattori indicati in precedenza, oltre che altri: situazione contingente, scelte del sistema operativo, configurazione del compilatore, ecc. In generale, l'ispezione del codice sorgente è poco utile, perché la relazione fra cosa c'è scritto nel codice sorgente e cosa viene effettivamente eseguito è molto labile. Per esempio, l'ordine di esecuzione non è garantito, a meno di (costose) richieste esplicite.

Deadlock

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

Speaker notes

E' definita anche una condizione "dinamica", detta "Livelock", in cui due thread si scambiano il possesso reciproco di due risorse, senza mai riuscire ad averle entrambe per proseguire: dall'esterno, si può osservare attività (i due thread si scambiano continuamente il possesso delle risorse) ma in realtà non viene svolto nessun lavoro.

Esiste un risultato teorico che indica quando può avvenire un deadlock.

COFFMAN'S CONDITIONS

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

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

Speaker notes

Tradotto dal punto di vista delle risorse, questo significa che: la risorsa non deve essere condivisibile; il processo deve cercare risorse usate da altri; non ci dev'essere modo di sottrarre una risorsa ad un processo che l'ha ottenuta; la catena delle attese fra i processi è circolare, cioè P1 attende una risorsa che ha P2, che attende una risorsa che ha P3, che attende anche lui una risorsa che ha P1. Sfortunatamente, sono condizioni molto comuni in quanto semplificano la realizzazione dell'ambiente di esecuzione.

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

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

Speaker notes

La necessità è completa: devono valere tutte. Purtroppo, lavorare per escluderle è costoso e a volte poco praticabile.

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

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

Speaker notes

alcune tecniche di programmazione sono molto interessanti per introdurre alcune tipologie di mutua esclusione; per esempio alcune tecniche di programmazione funzionale come la modellazione degli effetti collaterali tramite monadi.

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

richiede un qualche sistema transazionale per ottenere più risorse contemporaneamente

Speaker notes

Programmare tutte le possibili casistiche di attesa e prenotazione di multiple risorse, e le varie modalità di fallimento, può diventare più complesso del compito che si sta cercando di parallelizzare, e molto più difficile da dimostrare corretto.

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*.

Speaker notes

Il costo computazionale e di comunicazione per realizzare un sistema transazionale di questo tipo lo rende economico non a livello di sistema operativo, ma a livello applicativo specializzato: un esempio classico sono i database relazionali, dove vari tipi di controllo della concorrenza permettono di scegliere con continuità fra performance e correttezza.

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).

Speaker notes

Ma questo significa anche che il sistema ed i threads devono essere coscienti gli uni degli altri, e delle rispettive caratteristiche: questo non sempre è possibile a priori, e può essere molto complesso da risolvere nel caso generale.

TIPOLOGIE DI CONCORRENZA

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

Speaker notes

una possibile classificazione delle tipologie di concorrenza si può ottenere incrociando le strutture a disposizione del programmatore per realizzare le applicazioni concorrenti, ed il grado di collaborazione che il sistema operativo esige in cambio.

COLLABORATIVA

I programmi devono esplicitamente cedere il controllo ad intervalli regolari.

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

Speaker notes

Le coroutines hanno una applicabilità ed una vita indipendente dalla pura concorrenza: in alcune implementazioni consentono, per esempio, di ottenere sistemi di runtime privi di stack, e quindi con consumo di memoria fisso.

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

Speaker notes

ogni risorsa o quasi può essere sottratta al controllo di un processo senza che questo se ne accorga o possa farci nulla.

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.

Speaker notes

Per es. strumenti di misura o controllo industriale, aereonautico o aereospaziale, per es. sistema operativo "Luminary099" per il modulo lunare della missione Apollo 11 (https://github.com/chrislgarry/Apollo-11).

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

Speaker notes

per es. applicazioni scritte secondo il Reactive Manifesto; applicazioni per smartphone; piattaforme di data streaming o fast data.

JAVA THREADS

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

```java
/**
 * 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.

Speaker notes

vale a dire che chiamando quel metodo il percorso di esecuzione non è più univoco: da un lato, il metodo ritorna ed il programma chiamante prosegue, dall'altro il metodo chiamato viene eseguito contemporaneamente in una nuova linea di esecuzione.

```
/**
 * 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)
```

```java
/**
 * 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();
    }
```

Speaker notes

Punti notevoli:

- la lambda che viene riconosciuta come implementazione di `Runnable`
- l'uso di metodi statici di `Thread` per controllare il comportamento del thread corrente
- l'uso di una lambda come strategia (nel senso del pattern Strategy) di generazione dei tempi di attesa.

`pcd2019.threads.SingleThread`:

lancia un singolo thread

```
public static void main(String[] args) {
  Thread a = new ThreadSupplier().get();

  System.out.println("Starting Single Thread");
  a.start();
  System.out.println("Done starting.");
}
```

Speaker notes

Punti di attenzione:

- il programma non termina dopo la conclusione di main(), ma attende che il thread completi la sua esecuzione.

`pcd2019.threads.ManyThreads`:

lancia diversi thread in successione

```
public static void main(String[] args) {
  Stream<Thread> threads = Stream.generate(new ThreadSupplie

  System.out.println("Starting Threads");
  threads.limit(10).forEach((Thread a) -> a.start());
  System.out.println("Done starting.");
}
```

Speaker notes

ecco il motivo per implementare il `ThreadSupplier` in questo modo: possiamo usarlo come generatore di uno Stream, ottenere tutti i `Thread` che ci servono e trattarli in successione.

TESTI UTILI:

Java 11 Javadocs

https://docs.oracle.com/en/java/javase/11/docs/api/jav
summary.html

5 thinks 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 thinks 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

PUBBLICITÀ

Per unire l'utile al dilettevole:

Advent of Code

https://adventofcode.com/

Devoxx e Voxxed Days

https://beta.devoxx.com/

https://beta.voxxeddays.com/#/

Speaker notes

Il Devoxx è la conferenza principale, in Belgio ; si tratta di un'evento da migliaia di partecipanti, con centinaia di interventi. Ha alcune date collaterali che puntano alle stesse dimensioni. I Voxxed Days sono eventi molto più ridotti, solitamente di una sola giornata, più focalizzati ed economici, che di solito non superano i 200-300 partecipanti e la ventina di interventi. Il prossimo anno ci sarà il primo a Milano, ma l'edizione di Lugano è molto facile da raggiungere, e un'altra sede è Vienna. La qualità è molto alta (si tratta comunque di eventi specialistici) ma sono molto utili per cominciare a conoscere le comunità internazionali.

Google Veneto DevFest

https://gdg-venezia.github.io/devfest-veneto-18/

Speaker notes

Nel Veneziano c'è un gruppo ufficiale di sviluppatori che fanno riferimento all'ecosistema Google, ed anche quest'anno organizzano un evento presso Fabrica a Villorba di Treviso. Si tratta di un evento molto informale e legato al territorio, in cui si possono fare interessanti incontri. E' comunque un'ottima scusa per visitare un luogo davvero particolare. Se non siete già iscritti, prendete nota per il prossimo anno.