



ALMA MATER STUDIORUM A.D. 1088

**UNIVERSITÀ DI BOLOGNA**

Scuola

**Ingegneria e Architettura**

Corso di studio (laurea magistrale)

**Ingegneria e Scienze informatiche**

Sede didattica

**Cesena**

A.A.

**2017/2018**

Insegnamento

**Programmazione Concorrente e Distribuita**

**corso integrato con Sviluppo di Sistemi Software**

Titolare

**Prof. Alessandro Ricci**

# Sommario

- Concurrent Programming - Intro (module 1.1) .....2
  - Terminologia .....2
  - Paradigmi concorrenti .....3
  - Computer paralleli .....3
  - Flynn’s taxonomy .....3
  - Concurrent programming: performance .....4
  - Concurrent programming: progettazione e astrazione .....5
  - Definizioni .....5
  - Sulle spalle dei giganti: le origini della programmazione concorrente .....7
  - Linguaggi e macchine “Concorrenti” .....8
  - Bibliografia .....14
- Modeling concurrent program execution (module 1.2) .....15
  - Realizzazione di modelli per programmi concorrenti .....15
  - Proprietà relative alla correttezza: safety, liveness, fairness .....20
  - Il problema della sezione critica .....21
  - Correttezza: verifica, test, validazione .....27
  - Metodi formali per la verifica .....28
  - Tecniche di verifica: model-checking .....30
  - Tecniche di verifica: verifica induttiva delle invarianti .....31
- Process Interaction and Coordination (module 1.3) .....31
  - Meccanismi di base e astrazioni per la coordinazione .....31
- Visual formalisms (module 1.4) .....47
  - Reti di Petri .....47
  - Diagrammi di stato (TO DO) .....56
  - Diagrammi di attività (TO DO) .....56
- Multithreaded Programming in Java: Introduction (module lab-1.1) .....57
  - Meccanismi .....57
  - Definire i thread .....57
  - Sincronizzazione .....59
  - Attività di laboratorio .....59
  - First Look at Concurrent Programs .....60
  - First Look at Synchronization: join .....60
  - First utilities .....60
  - Sincronizzazione implicita .....62

---

Sincronizzazione esplicita .....	62
Monitoraggio dei thread.....	62
Disciplina per la programmazione .....	62
Thread Safety & Liveness (module lab-1.2) .....	63
Deadlock .....	64
Starvation.....	69
Poor responsiveness .....	69
Livelock .....	69
Library Mechanisms for Thread Coordination in Java (module lab-1.3) .....	70
Collezioni synchronized .....	70
Collezioni concurrent.....	70
Aggregated operation and Lambda .....	71
Synchronizers.....	71
GUI Frameworks and Concurrency (module lab-1.4) .....	73
Esempio di un semplice Executor per la GUI .....	73
Implementing Monitors in Java (module lab-1.5) .....	76
Primo approccio (meccanismi di basso livello).....	76
Secondo approccio (meccanismi di alto livello) .....	80
From Threads to Tasks (module-2.1).....	86
Task-oriented analysis .....	86
Task-oriented design & programming.....	86
Asynchronous Programming (module-2.2) .....	88
Event-driven programming.....	88
Promise-Async Await (module-2.3) .....	92
Reactive Programming (module-2.4).....	101
Manifesto sistemi reattivi.....	101
Lifting .....	102
Composizione di event streams e behaviours .....	102
Reactive extensions (Rx) .....	102
Asynchronous Programming: challenges .....	109
Java Executors (module-lab-2.1) .....	109
Esecuzione sequenziale dei task.....	109
Politiche più flessibili: l’executor framework .....	110
Executors shutdown .....	115
Algoritmi map-reduce, il pattern Fork-Join .....	117
Bibliografia .....	118

Message Passing Models (module 3.1) .....119

    Primitive di base per la comunicazione .....119

    Comunicazione sincrona vs comunicazione asincrona.....119

    Schemi di comunicazione .....119

    Comunicazione utilizzando condizioni “guardia” .....122

    Interazione peer-to-peer .....124

    I filosofi a cena .....126

    Synchronous Message Passing .....127

    Rendez-vous.....128

    Remote Procedure Call (RPC) .....130

    Implementazioni esistenti per message passing basato to canali.....131

    Middleware orientato ai messaggi (MOM) .....131

ATTORI .....132

module-1.1 - concurrent programming - intro.pdf	10/11/2018
module-1.2 - modeling concurrent program execution.pdf	11/11/2018
module-1.3 - process interaction and coordination.pdf	12/11/2018
module-1.4 - visual formalisms.pdf	13/11/2018
module-2.1 - from threads to tasks.pdf	14/11/2018
module-2.2 - asynchronous programming - intro.pdf	15/11/2018
module-2.3 - promises - async await.pdf	16/11/2018
module-2.4 - reactive programming.pdf	17/11/2018
module-3.1 - message passing models.pdf	18/11/2018
module-3.2 - actors.pdf	19/11/2018
module-4.1 - distributed computing - introduction.pdf	20/11/2018
module-4.2 - distributed computing - some algorithms.pdf	21/11/2018
module-5.1 - elements of reactive distributed systems design.pdf	22/11/2018
module-5.2 - reactive design patterns.pdf	23/11/2018
module-5.3 - service-oriented architectures.pdf	24/11/2018
module-5.4 - cloud computing.pdf	25/11/2018
module-lab-1.1 - multithreaded programming in java - introduction.pdf	26/11/2018
module-lab-1.2 - thread safety %26 liveness.pdf	27/11/2018
module-lab-1.3 - library mechanisms for thread coordination in java.pdf	28/11/2018
module-lab-1.4 - gui frameworks and concurrency.pdf	29/11/2018
module-lab-1.5 - implementing monitors in java.pdf	30/11/2018
module-lab-2.1 - java executors.pdf	1/12/2018
module-lab-2.2 - nodejs.pdf	2/12/2018
module-lab-3.1 - actors with akka.pdf	3/12/2018
module-lab-5.1 - mom.pdf	4/12/2018
module-lab-5.2 - spring.pdf	5/12/2018
module-lab-5.3 - paas-heroku.pdf	6/12/2018
Elaborato 1	10/12/2018
Elaborato 2	14/12/2018
Elaborato 3	18/12/2018
Elaborato 4	22/12/2018
Esame	

## Concurrent Programming - Intro (module 1.1)

La concorrenza rappresenta uno dei concetti principali di molti sistemi e domini applicativi.

“La **concorrenza**, nelle scienze informatiche, è una proprietà di un sistema nel quale diversi processi computazionali eseguono **nello stesso momento** e possono interagire tra loro” (Roscoe, 1997)

“La **concorrenza** riguarda gli aspetti fondamentali dei sistemi nei quali multipli agenti computazionali attivi simultaneamente interagiscono tra loro” (Rance Cleaveland, 1996)

Concurrent programming  
sviluppo di programmi con attività computazionali concorrenti

Concurrent program  
Insieme finito di programmi sequenziali che vengono eseguiti in parallelo

Note:

- Un programma sequenziale specifica un'esecuzione sequenziale di una lista di istruzioni
- L'esecuzione di un programma è chiamata **processo**
- Un programma concorrente esegue due o più programmi sequenziali in concorrenza come processi paralleli
- L'esecuzione di un programma concorrente è detta **computazione o elaborazione concorrente**

### Terminologia

Programmazione <b>concorrente</b>	Realizzazione di programmi in cui diverse esecuzioni di attività computazionali si sovrappongono nel tempo e interagiscono tra loro in qualche modo. LIVELLO: logico, astrazioni, programmazione FOCUS: organizzazione dei programmi
Programmazione <b>parallela</b>	Esecuzione di programmi che si sovrappongono nel tempo, eseguiti su processori fisici diversi. LIVELLO: fisico FOCUS: performance
Programmazione <b>distribuita</b>	Distribuzione dei processori in una rete, senza l'uso di memoria condivisa.

Secondo Rob Pike “Concurrenty is really a way of writing or structuring your program to deal with the real world”: il mondo reale può essere inteprutato come un insieme di agenti che interagiscono tra loro (non è possibile descriverne l'essenza tramite paradigmi di programmazione sequenziali); la concorrenza è la composizione di esecuzioni di calcoli indipendenti. La concorrenza è un modo per strutturare i software, scrivere codice chiaro per interagire bene nel mondo reale; un modello per la costruzione dei software semplice da comprendere, utilizzare e su cui ragionare. La concorrenza non è parallelismo: abilita il parallelismo che riguarda principalmente le performance.

Paradigmi concorrenti

Multi-threaded programming	<ul style="list-style-type: none"><li>• Condivisione dello stato / memoria</li><li>• Meccanismi di sincronizzazione: semafori e monitors</li></ul>
Message-based programming	<ul style="list-style-type: none"><li>• Nessuna condivisione di memoria</li><li>• Interazione regolata tramite scambio di messaggi</li><li>• Modalità asincrona / sincrona</li></ul>
Event-driven programming	<ul style="list-style-type: none"><li>• Il flusso di esecuzione del programma è determinato da eventi: azioni di utenti, sensori, messaggi da altri thread/processi/applicazioni</li></ul>
Asynchronous programming	<ul style="list-style-type: none"><li>• Progettazione di programmi con azioni e richieste asincrone<ul style="list-style-type: none"><li>○ Never blocking dogma</li><li>○ Future mechanisms, callbacks</li></ul></li></ul>
Reactive programming	<ul style="list-style-type: none"><li>• Il flusso di esecuzione del programma è progettato sui flussi dei dati e la propagazione dei cambiamenti</li></ul>

Computer paralleli

**Architetture multi-core:** multipli core su un singolo chip; condivisione della RAM, possibili livelli di condivisione della cache.

**Progettazione chip eterogenei:** potenziamento di un processore standard con uno o più compute engines specializzati (attached processors); ad esempio GPU, GPGPU, Field-Programmable Gate Array (FPGA), Cell processors, CUDA.

**Super computers:** tradizionalmente utilizzati da laboratori e grandi compagnie, integrano differenti tipologie di architetture e sono formati solitamente da un elevato numero di processori (IBM BlueGen/L 65536 nodi dual core).

**Cluters / Grid:** costituite da parti elementari, ogni nodo può contenere uno o pochi processori, RAM ed eventualmente memoria disco; i nodi sono collegati tramite interconnessione delle parti (Gigabit Ethernet, Myrinet, Fibre channel, ...); non vi è memoria condivisa tra le macchine, la comunicazione avviene tramite passaggio di messaggi.

**Cloud computing:** forniscono capacità di calcolo **as a service** attraverso la rete (Software as a Service, Platform as a Service, Infrastructure as a Service).

Flynn’s taxonomy

La tassonomia di Flynn categorizza i sistemi di calcolo in base al numero di **instruction stream** e **data stream**: sequenze di istruzioni o dati sui quali opera il calcolatore.

		Data stream	
		Single	Multiple
Instruction stream	Single	<b>SISD</b> Modello della macchina di Von-Neumann: un computer con un singolo processore	<b>SIMD</b> Il singolo stream di istruzioni è eseguito concorrentemente su multipli processori, ognuno sui propri dati (parallelismo a grani fini, processori vettoriali)
	Multiple	<b>MISD</b> Non esistono sistemi noti che rientrino in questa casistica	<b>MIMD</b> Ogni processore ha il proprio stream di istruzioni che operano su uno specifico stream di dati
			<b>Memoria condivisa</b> Tutti i processi (processori) condividono uno stesso spazio degli indirizzi e comunicano tra loro leggendo e scrivendo variabili condivise

## Ulteriori classificazioni MIMD relativamente alla condivisione di memoria

1. **Symmetric Multi-Processing architecture (SMP)**: tutti i processori condividono una memoria comune e accedono a tutte le locazioni di memoria con la stessa velocità.
2. **Non-Uniform Memory Access (NUMA)**: nella memoria condivisa, alcuni blocchi di memoria possono essere fisicamente più vicini e quindi associati ad alcuni processori piuttosto che ad altri.

## Ulteriori classificazioni MIMD relativamente alla distribuzione di memoria

1. **Massively Parallel Processor (MPP)**
  - a. I processori e l'infrastruttura di rete sono strettamente accoppiati e specializzati per computer paralleli
  - b. Estremamente scalabili: migliaia di processori in un singolo sistema
  - c. Utilizzati per applicazioni ad alte performance (High-Performance Computing, HPC)
2. **Clusters**
  - a. Sistemi a memoria distribuita composti di computer standard connessi tramite una rete classica (es. Beowulf clusters)
3. **Grid**
  - a. Sistemi che utilizzano risorse eterogenee e distribuite connesse tramite LAN e/o WAN, senza un punto comune di amministrazione.

## Concurrent programming: performance

Il miglioramento delle performance in computazione concorrente riguarda l'incremento del **throughput** (grazie all'utilizzo del calcolo parallelo) e l'incremento della **responsività** delle applicazioni (ottimizzando l'interazione tra CPU e attività di input/output). Una misura quantitativa per la performance è lo **speedup**.

$$S = \frac{T_1}{T_N}$$

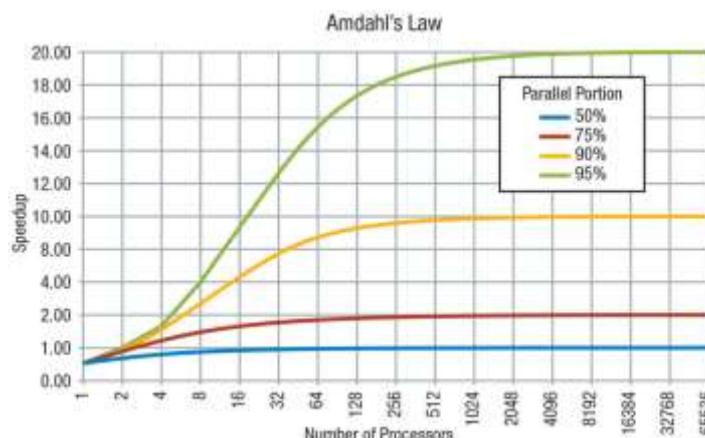
*N è il numero di processori*  
*T<sub>1</sub> è il tempo di esecuzione di un algoritmo sequenziale*  
*T<sub>N</sub> è il tempo di esecuzione del corrispondente algoritmo parallelo con N processori*

La **Legge di Amdahl** definisce il massimo speedup teorico:

$$S = \frac{1}{1 - P + P/N}$$

*P è la proporzione di programma che può essere parallelizzata*  
*(1 - P) è la proporzione di programma non parallelizzabile*  
*Teoricamente si ha massimo speedup per P = 1*

Esistono casi particolari per i quali si possono ottenere livelli "super-lineari" di speedup con  $S > N$ .



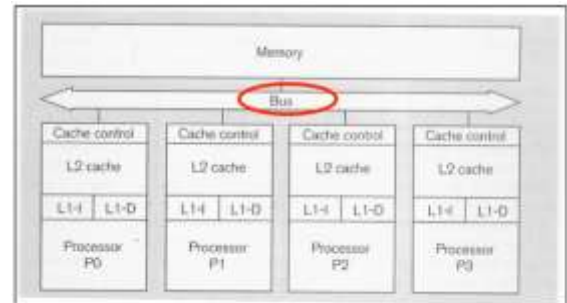
In sintesi: la sequenzialità e la serializzazione degli algoritmi contrastano le performance (ad esempio determinano blocchi o rallentamenti di esecuzione senza sfruttare le risorse hardware disponibili), tuttavia risultano talvolta necessarie per garantire correttezza degli algoritmi (safety). Per questo motivo è necessario individuare i giusti compromessi per garantire correttezza ed ottenere le migliori performance, per questi motivi l'ambito della concorrenza è oggetto di ricerca.



L'**efficienza** (definita come  $E = S/P$ ) è una misura normalizzata di speedup che indica quanto è effettivamente utilizzato ogni processore: l'efficienza ideale, pari a 1, definisce che ogni processore è utilizzato pienamente.

### Un nuovo collo di bottiglia: la memoria

L'accesso alla memoria condivisa e al bus sono potenziali colli di bottiglia in quanto può essere effettuata una sola operazione alla volta sulla memoria, ciò determina l'importanza di strutturare e utilizzare correttamente una cache tramite protocolli sempre più complessi e intelligenti che garantiscano la coerenza delle informazioni.



### Concurrent programming: progettazione e astrazione

La programmazione concorrente richiede la definizione di un livello appropriato di astrazione per la progettazione di programmi che devono interagire con l'ambiente, controllare diverse attività e gestire diversi eventi (l'astrazione OO non è sufficiente).

La concorrenza è uno strumento per la progettazione e la costruzione dei software: permette di ripensare il modo con cui si risolvono i problemi (algoritmi di base e strutture dati) e il modo in cui si progetta un sistema (diverse logiche di decomposizione, modularizzazione, incapsulamento).

La programmazione concorrente ha impatto su tutto lo spazio dell'ingegneria software: modellizzazione, progettazione, implementazione, verifica e test.

### Definizioni

**Processi:** programmi sequenziali in esecuzione.

I processi sono l'unità minima di un sistema concorrente, sono eseguiti tramite un **unico flusso di controllo**. Possono essere visti come **unità di lavoro**: sequenze di istruzioni che operano insieme come un gruppo. Sono concetti **generali e astratti**: non sono necessariamente correlati ai processi del sistema operativo. I processi vengono eseguiti in maniera completamente asincrona tra loro, non potendo effettuare assunzioni circa le loro velocità, un sistema concorrente è necessariamente legato al concetto di **non-determinismo**.

**Interazione:** qualsiasi programma (non banale) è basato su processi multipli che operano e devono interagire tra loro in qualche modo, alcuni tipi di interazione sono la **cooperazione**, la **competizione**, l'**interferenza**.

- **Cooperazione:** riguarda le interazioni aspettate e volute che fanno parte della semantica del programma concorrente
  - **Comunicazione:** riguarda la necessità di realizzare un flusso informativo tra i processi, tipicamente tramite scambio di messaggi
  - **Sincronizzazione:** riguarda la definizione di specifiche relazioni o dipendenze temporali tra azioni distinte dei processi
- **Competizione:** riguarda le interazioni che sono attese e necessarie ma non volute, è tipicamente relativa alla coordinazione dell'accesso a risorse condivise da parte di diversi processi
  - **Mutua esclusione:** regola l'accesso alle risorse condivise da parte di diversi processi
  - **Sezione critica:** regola l'esecuzione concorrente di blocchi di azioni da parte di diversi processi

**Interferenza:** riguarda le interazioni che non sono né previste né volute, producono effetti negativi solo quando il rapporto tra le velocità dei processi assume valori specifici (errori dipendenti dal tempo); "l'incubo della programmazione concorrente è rappresentato dagli **heisen-bugs** (quando il debug influenza la generazione, o meno, degli errori).

- La **race condition** o **race hazard** (o semplicemente **race**) si verifica quando due o più processi accedono concorrentemente ad una risorsa condivisa e ne modificano il contenuto generando come risultato un singolo aggiornamento dipendente dallo specifico ordine di esecuzione dei processi.

Nota: la sincronizzazione e la mutua esclusione, pur apparendo concetti simili, sono correlati ma differenti, infatti la sincronizzazione definisce una relazione temporale tra i processi mentre la mutua esclusione definisce una restrizione di accesso alla memoria condivisa (non ha senso di esistere nel caso in cui non esistano dati condivisi). La mutua esclusione richiede tipicamente qualche forma implicita di sincronizzazione (bloccare qualche azione, attendere la conclusioni di altre azioni, ...) mentre la sincronizzazione non richiede l'esistenza di dati condivisi né di mutua esclusione.

## Un esempio di sincronizzazione

*Alice e Bob vivono assieme, felicemente (e) senza cellulari o smartphone. Entrambi sono responsabili di comprare il latte quando finisce...*

Orario	Alice	Bob
5:00	Arriva a casa	
5:05	Controlla il frigorifero e nota che il latte è finito	
5:10	Esce per comprare il latte	
5:15		Arriva a casa
5:20		Controlla il frigorifero e nota che il latte è finito
5:25	Compra il latte	Esce per comprare il latte
5:30	Arriva a casa e mette il latte nel frigorifero	
5:40		Compra il latte
5:45		Arriva a casa, mette il latte nel frigor... DOH!

Una possibile soluzione prevede di utilizzare un post-it sul frigorifero. L'obiettivo è fare in modo che una sola persona compri il latte quando manca e che, se il latte manca, qualcuno si preoccupi sempre di comprarlo.

Pseudo codice (tentativo: se non ci sono note sul frigorifero e manca il latte, lascia una nota ed esci a comprarlo...):

```

1  if (no note) then
2      if (no milk) then
3          leave note
4          buy milk
5          remove note
6      fi
7  fi
    
```

Orario	Alice	Bob
5:00	Arriva a casa	
5:05	Controlla il frigorifero: non ci sono post-it	
5:10	...ops, deve andare in bagno	
5:15	...in bagno...	Arriva a casa
5:20	...in bagno...	Controlla il frigorifero: non ci sono post-it
5:21	...in bagno...	Non c'è il latte
5:22	...in bagno...	Lascia un post-it sul frigorifero
5:25	...in bagno...	Esce per comprare il latte
5:30	Torna al frigorifero e vede che non c'è il latte*	
5:40	Esce per comprare il latte	Compra il latte
5:45	Compra il latte	Arriva a casa, mette il latte nel frigorifero
5:50	Arriva a casa, mette il latte nel frigor... DOH!	

\*avendo precedentemente visto che non c'erano post-it non si preoccupa di ricontrollare.

## Situazioni critiche

Errori ed interferenze possono portare a situazioni critiche per i sistemi concorrenti:

- **Deadlock** (deadly embrace, Dijkstra): si verifica quando due o più processi si fermano in attesa della fine di un altro processo e questa non si verifica mai (riguarda il rilascio di risorse condivise bloccate)
- **Starvation** (unfairness): si verifica quando un processo è bloccato in un'attesa infinita in quanto gli viene infinitamente negato l'accesso alla risorsa di cui ha bisogno per completare la propria esecuzione
- **Livelock**: simile al deadlock tranne per il fatto che lo stato dei processi coinvolti cambia in continuazione pur non procedendo (è uno speciale caso di starvation).

## Sulle spalle dei giganti: le origini della programmazione concorrente



**Edsger Wybe Dijkstra** (1930-2002) è stato uno dei membri più influenti nella generazione dei fondatori della computer science. I suoi contributi scientifici sono fondamentali in diversi ambiti: progettazione di algoritmi, linguaggi di programmazione, progettazione software, sistemi operativi, calcolo distribuito, specifiche e verifiche formali, progettazione di argomentazioni matematiche. Alcune sue massime: “la computer science riguarda i computer quanto l’astronomia riguarda i telescopi”; “domandarsi se un computer possa pensare è interessante quanto chiedersi se un sottomarino possa nuotare”; “la programmazione Object-Oriented è una eccezionalmente cattiva idea che non poteva nascere se non in California”. Per sottolineare un aspetto del carattere di Dijkstra, Alan Kay ha detto “non so quanti di voi abbiano mai incontrato di persona Dijkstra, ma molti di voi probabilmente sanno che l’arroganza nella computer science viene misurata in nano-Dijkstras”.



**Peter Brinch Hansen** (1938-2007) e **Sir Anthony Hoare** hanno sviluppato il concetto di monitor. Brinch Hansen pubblicò la prima notazione sul monitor, adottando il concetto di classe di Simula 67 e inventò un meccanismo di accodamento. Hoare ha perfezionato le regole della ripresa dei processi. Brinch Hansen ha creato la prima implementazione di monitor, in Concurrent Pascal che è stato il primo linguaggio di programmazione concorrente: Hoare lo ha descritto come "un esempio eccezionale del meglio della ricerca accademica in questo campo". Il codice sorgente e portatile per Concurrent Pascal e il sistema operativo Solo sono stati distribuiti ad almeno 75 aziende e 100 università in 21 paesi, con conseguente adozione, porting e adattamento diffusi sia nell'industria che nel mondo accademico. Greg Andrews osservò che Concurrent Pascal e i suoi monitor "influenzarono molto la maggior parte delle successive proposte linguistiche concorrenti".

L'Architecture of Concurrent Programs di Brinch Hansen è stato il primo libro sulla programmazione concorrente, ed è stato infine pubblicato in tre lingue (inglese, giapponese e tedesco). Più di una dozzina di anni dopo la sua pubblicazione, P. J. Plauger osservò: “certo, sono stati apportati miglioramenti negli ultimi dodici anni. Abbiamo migliori algoritmi di sincronizzazione e linguaggi più fantasiosi (se non necessariamente migliori) con controllo della concorrenza. Ma non è possibile trovare una panoramica migliore sulla programmazione concorrente rispetto a quella presentata in questo libro. Almeno non l'ho trovata”.

## Breve storia

Kilburn & Howarth, nel 1961, introducono l'utilizzo degli interrupts per simulare l'esecuzione concorrente di programmi nel computer ATLAS: è la nascita della multiprogrammazione. Molti sistemi di multiprogrammazione vengono programmati in assembly senza un vero e proprio fondamento concettuale, negli anni sessanta si verifica una crisi nel mondo del software che richiedeva di capire più approfonditamente i concetti legati alla programmazione concorrente. In 15 anni gli scienziati informatici hanno scoperto concetti fondamentali, descritto nozioni di programmazione, le hanno incluse nei linguaggi di programmazione e hanno utilizzato questi linguaggi per scrivere sistemi operativi. Negli anni settanta sono stati prodotti diversi libri sui concetti di programmazione collegati alla concorrenza.

## Concetti chiave

Concetti fondamentali	Concetti legati ai linguaggi di programmazione
Processi asincroni	Dichiarazioni concorrenti
Indipendenza nelle velocità	Regioni/sezioni critiche
Pianificazione equa	Semafori
Mutua esclusione	Buffer di messaggi
Prevenzione deadlock	Regioni critiche condizionali
Comunicazione tra processi	Variabili per accordamenti sicuri
Strutture gerarchiche	Monitor
Kernel estendibili	Comunicazione tramite messaggi sincroni
	Chiamate a procedure remote (RPC)

## Pubblicazioni di riferimento

1. E. W. Dijkstra, Cooperating Sequential Processes (1965).
2. E. W. Dijkstra, The Structure of the THE Multiprogramming System (1968).
3. P. Brinch Hansen, RC 4000 Software: Multiprogramming System (1969).
4. E. W. Dijkstra, Hierarchical Ordering of Sequential Processes (1971).
5. C. A. R. Hoare, Towards a Theory of Parallel Programming (1971).
6. P. Brinch Hansen, An Outline of a Course on Operating System Principles (1971).
7. P. Brinch Hansen, Structured Multiprogramming (1972).
8. P. Brinch Hansen, Shared Classes (1973).
9. C. A. R. Hoare, Monitors: An Operating System Structuring Concept (1974).
10. P. Brinch Hansen, The Programming Language Concurrent Pascal (1975).
11. P. Brinch Hansen, The Solo Operating System: A Concurrent Pascal Program (1976).
12. P. Brinch Hansen, The Solo Operating System: Processes, Monitors and Classes (1976).
13. P. Brinch Hansen, Design Principles (1977).
14. E. W. Dijkstra, A Synthesis Emerging? (1975).
15. C. A. R. Hoare, Communicating Sequential Processes (1978).
16. P. Brinch Hansen, Distributed Processes: A Concurrent Programming Concept (1978).
17. P. Brinch Hansen, Joyce | A Programming Language for Distributed Systems (1987).
18. P. Brinch Hansen, SuperPascal: A Publication Language for Parallel Scientific Computing (1994).
19. P. Brinch Hansen, Efficient Parallel Recursion (1995)

## Linguaggi e macchine “Concorrenti”

Allo scopo di descrivere le specifiche di un programma concorrente servono **linguaggi di programmazione concorrenti** che permettano ai programmatori di scrivere programmi come insieme di istruzioni che devono essere eseguite concorrentemente. Per eseguire un programma concorrente servono **macchine concorrenti** che (possono essere astratte) sono progettate per gestire l'esecuzione di molteplici processi sequenziali utilizzando molteplici processori fisici o virtuali.

Una macchina concorrente fornisce:

- un supporto per l'esecuzione di programmi concorrenti, per la realizzazione di computazioni concorrenti
- tanti processori virtuali quanto il numero di processi che compongono la computazione concorrente

- meccanismi di base per
  - **multiprogrammazione** (generazione e gestione di processori virtuali):  
un insieme di meccanismi che rendono possibile creare nuovi processori virtuali e allocarli nei processori fisici tramite algoritmi di scheduling appropriati
  - **sincronizzazione e comunicazione**:  
si tratta di due differenti tipologie di meccanismi correlati a due differenti modelli architetturali per le macchine concorrenti:
    - **modelli a memoria condivisa**: presenza di memoria condivisa tra i processori (es. programmazione multi-thread)
    - **modelli a passaggio di messaggi**: ogni processore virtuale possiede una propria memoria, non esiste memoria condivisa tra processori, ogni comunicazione e interazione tra i processi è realizzata tramite passaggio di messaggi.
  - **controllo per l'accesso alle risorse**.

## Dalle macchine ai linguaggi di programmazione

I costrutti principali dei linguaggi che permettono di descrivere programmi concorrenti che possono essere eseguiti su macchine concorrenti sono:

- costrutti per specificare la concorrenza (creazione di molteplici processi)
- costrutti per specificare l'interazione tra i processi: sincronizzazione e comunicazione, mutua esclusione.

Esistono tre principali approcci:

- **linguaggi sequenziali + librerie con primitive concorrenti** (es. C + PThreads)
- **linguaggi progettati per la concorrenza** (es. OCCAM, ADA, Erlang)
- **approcci ibridi**:
  - paradigmi sequenziali estesi con supporto nativo per la concorrenza (es. Java, Scala)
  - librerie e pattern basati su meccanismi di base (es. `java.util.concurrent`).

## Overview su notazioni e costrutti di base

I primi costrutti proposti negli anni 1960-1970 erano **fork/join** e **cobegin/coend**. Proposte più recenti riguardano astrazioni di prima classe e costrutti per definire i processi (tasks), ad esempio sviluppati in linguaggi come ADA ed Erlang. I linguaggi maggiormente diffusi hanno implementato un supporto per la programmazione multi-threaded (es. Java) con un incremento delle capacità di programmazione asincrona e event-driven. In ambito di ricerca sono state presentate molte proposte: modelli basati sugli attori (punti di riferimento per la programmazione concorrente OO sempre più diffusi), oggetti attivi, Software Transactional Memory (STM), programmazione reattiva, programmazione orientata agli agenti.

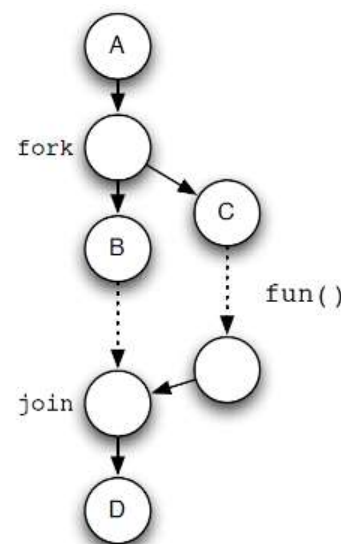
## Fork/Join

Si tratta di una delle prime notazioni (Conway 1963, Dennis 1968) utilizzate per esprimere concorrenza, adottata nei sistemi UNIX/POSIX con linguaggio MESA (1979).

La primitiva **fork** ha un comportamento simile all'invocazione di procedure con la differenza che crea un nuovo processo e lo utilizza per attivare l'esecuzione; l'input è la procedura da eseguire, l'output è l'identificatore del processo creato; il risultato della fork è una biforcazione del flusso di controllo in quanto il nuovo processo (child) è eseguito in maniera asincrona rispetto al processo che l'ha generato (parent) e agli altri processi esistenti.

La primitiva **join** rileva quando un processo creato con una fork termina la propria esecuzione e sincronizza il flusso di controllo come tramite un evento.

La Fork/Join ha il vantaggio di essere general e flessibile in quanto può essere utilizzate per generare qualsiasi tipo di applicazione concorrente. Dal punto di vista dei "contro" realizza un basso di livello di astrazione che non fornisce alcuna disciplina per strutturare processi complessi ed è soggetta ad errori; i programmi scritti con questi costrutti sono complessi da leggere ed interpretare in quanto è difficile rendersi conto di quali processi sono attivi in uno specifico punto del programma; non costituisce una rappresentazione esplicita dell'astrazione del processo per organizzare l'intero sistema.

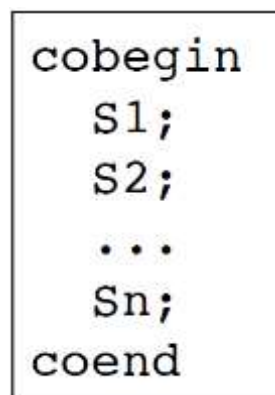


## Cobegin/Coend

Si tratta di un costrutto proposto da Dijkstra nel 1968 per fornire una disciplina per la programmazione concorrente in modo da indurre i programmatori a seguire uno specifico schema per strutturare i programmi concorrenti.

La concorrenza è espressa in blocchi:

- le istruzioni S1, S2, ..., Sn sono eseguite in parallelo
- un'istruzione Si può essere complessa quanto l'intero programma e può includere blocchi innestati di cobegin/coend
- una struttura parallela termina solo quanto tutti i suoi componenti (processi) terminano.



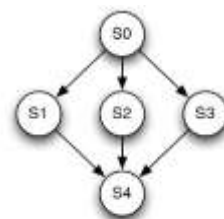
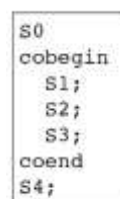
Il processo di esecuzione crea tanti processi (children) quante sono le istruzioni nel corpo e sospende la propria esecuzione finché tutti i processi non terminano.

Vantaggi:

- fornisce una disciplina più solida per la strutturazione di programmi concorrenti rispetto alle primitive fork/join
- fornisce una maggiore leggibilità dei programmi

Svantaggi:

- è meno flessibile della fork/join (come creare N processi concorrenti con N noto solo in fase di esecuzione?)



## Linguaggi di programmazione con supporto per i processi come astrazioni di prima classe

Questi linguaggi introducono una nozione di processi come entità di prima classe come **moduli** per organizzare un programma (statico) e il sistema (a runtime) e un incapsulamento esplicito del flusso di controllo (es. Concurrent Pascal, OCCAM, o i più recenti ADA, Erlang).

Per la maggior parte, i principali linguaggi di programmazione (sia procedurali come il C che orientati agli oggetti come Java) forniscono un supporto per la creazione e l'esecuzione di processi tramite librerie: senza estendere il linguaggio.

Il supporto per la programmazione multi-threaded prevede che i threads siano implementazione della nozione astratta di processo (detti anche lightweight process in contrapposizione ai heavyweight process del sistema operativo).

Java è stato il primo linguaggio ampiamente diffuso a fornire un supporto nativo per la programmazione concorrente; ha seguito un approccio conservativo: il linguaggio è comunque puramente ad oggetti senza costrutti espliciti per definire i processi arricchito tramite l'introduzione di alcune parole chiave e meccanismi per la concorrenza (blocchi sincronizzati, meccanismi di wait/notify). La nozione astratta di processo è implementata tramite un thread con una mappatura diretta attraverso il supporto per i threads del sistema operativo. Dal 2005 il supporto è stato largamente esteso tramite le Java Concurrency Utility.

## Thread in JAVA

Il modello di gestione dei thread in JAVA prevede che ogni thread sia definito da un unico flusso di controllo e condivida memoria con tutti gli altri threads (stack privata, heap); ogni programma JAVA contiene almeno un thread (quello di esecuzione della main class); ulteriori threads possono essere creati e attivati dinamicamente durante l'esecuzione del programma.

I thread sono oggetti di classi che estendono la classe Thread fornita in java.lang.package, possono essere istanziati e messi in esecuzione invocando il metodo start.

Un esempio.

```
class ClockVisualizer extends Thread {
    private int step;

    public ClockVisualizer(int step){
        this.step=step;
    }

    public void run() {
        while (true) {
            System.out.println(new Date());
            try {
                sleep(step);
            } catch (Exception ex) {
            }
        }
    }
}

class TestClockVisualizer {
    static public void main(String[] args) throws Exception {
        ClockVisualizer clock = new ClockVisualizer(1000);
        clock.start();
    }
}
```



## Programmazione multithread con C/C++ e Pthreads

La libreria Pthread (POSIX-thread) definita nel contesto POSIX (Portable Operating System Interface) fornisce un insieme di primitive di base per la programmazione multithread in C/C++:

- è implementata una nozione astratta di processo
- diversamente da Java, il corpo del processo è specificato tramite procedure
- lo standard definisce l'interfaccia e le specifiche, non l'implementazione.

Esistono API di base per la creazione di threads e la loro sincronizzazione (tutorial: <http://www.llnl.gov/computing/tutorials/pthreads/>).

L'interfaccia è definita in pthread.h; esistono due tipi di dati principali: pthread\_t (identificatore) e pthread\_attr\_t (attributi). Le funzioni principali sono

- creazione di thread (fork):  
pthread\_create(pthread\_t\* tid, pthread\_attr\_t\* attr, void\* (\*func)(void\*), void\* arg)  
pthread\_attr\_init(pthread\_attr\_t\*)
- terminazione dei thread  
pthread\_exit(int)
- join dei thread  
int pthread\_join(pthread\_t thread, void\*)

Esempio: creazione di 5 threads concorrenti.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

## Oltre ai thread

- **Framework task-based** (es. Java Executor framework)
- **Asynchronous Programming** (modelli basati su event-loop, asyncTasks everywhere)



- **Attori:** OOP concorrente, variante oggetti attivi
- **Passaggio di messaggi sincrono:** processi + passaggio di messaggi sincrono
- **Programmazione reattiva:** estensione funzionale di programmazione reattiva (Rx)
- **Calcolo parallelo:** programmazione GPGPU (es. OpenCL), architetture Lambda, parallelismo per BigData (es. MapReduce).

Modello ad attori

Introdotta originariamente dal Carl Hewitt e colleghi al MIT negli anni 70 e distribuita in contesti di Intelligenza Artificiale, è stata sviluppata da Gul Agha, Akinori Yonezawa e altri negli anni 80 e 90. Esistono molti linguaggi e framework (ACT++, Salsa, Kilim, ABCL, E, AmbientTalk, ActorFoundry, ...).

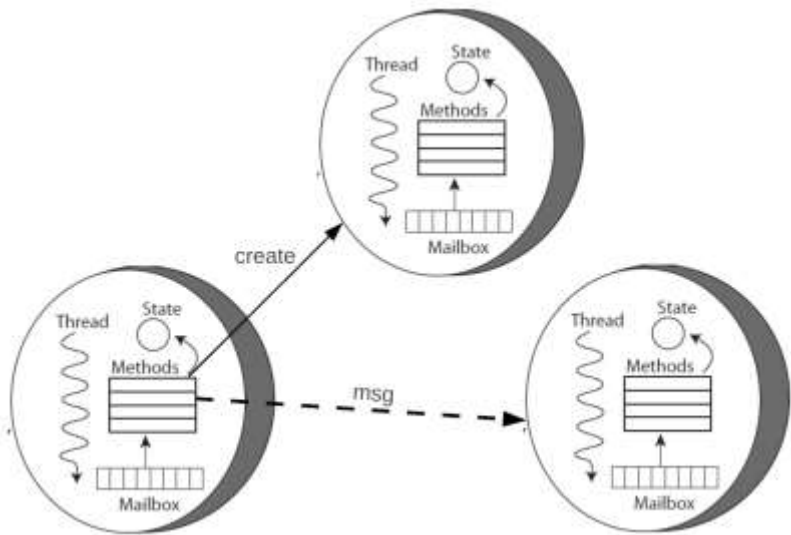
Oggi questo modello gioca un ruolo importante nel *mainstream* come modello alternativo alla programmazione multithread: ne sono esempi il framework Akka (Java/Scala), Erlang, HTML5 Web Workers, DART Isolates, ... etc.

Il modello attori prevede un passaggio di messaggi asincrono tra oggetti autonomi e puramente reattivi detti **attori**: tutto è un attore con un unico identificatore e un'unica mailbox nella quale vengono accodati i messaggi; tutte le interazioni sono regolate tramite passaggio di messaggi.

Alcune primitive: send, create, become, ...

Tutto può essere modellato con un pattern di messaggi tra attori.

L'idea astratta è visualizzata nella seguente immagine.



Doug Lea	Carl Hewitt	Gul Agha	Akinori Yonezawa

Concorrenza logica vs concorrenza fisica

I moderni paradigmi di programmazione concorrente promuovono una visione logica della concorrenza: a più alto livello rispetto alla concorrenza fisica dei thread del sistema operativo. Per esempio è possibile avere migliaia di attori in esecuzione su una stessa macchina eseguiti tramite un pool di thread eseguiti in una macchina virtuale con un numero di threads appropriato.

## Bibliografia

- [HAN-73] Per Brinch Hansen - "Concurrent Programming Concepts", ACM Computing Surveys, Vol. 5, No. 4, Dec. 1973
- [HAN-01] Per Brinch Hansen - The Invention of Concurrent Programming" in "The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls", Springer-Verlag, 2002
- [SUT-12] Sutter's Mill. Herb Sutter on software, hardware, and concurrency. "Welcome to the Jungle". <http://herbsutter.com/welcome-to-the-jungle/>
- [AND-83] Gregory Andrews and Fred Schneider - "Concepts and Notations for Concurrent Programming", ACM Computing Surveys, Vol. 15, No. 1, March 1983
- [CLE-96] Rance Cleaveland, Scott Smolka et al - "Strategic Directions in concurrency Research", ACM Computing Surveys, Vol. 28, No. 4, Dec. 1996
- [ROS-97] Roscoe, A. W. (1997). The Theory and Practice of Concurrency. Prentice Hall. ISBN 0-13-674409-5.
- [BUH-05] Peter Buhr and Ashif Harji. "Concurrent Urban Legends". Concurrency and Computation: Practice and Experience. 2005. 17:1133-1172.
- [HEW-77] C. Hewitt. Viewing Control Structures as Pattern of Passing Messages. Journal of Artificial Intelligence, 8(3):323-364, 1977
- [AGH-86] Gul Agha. Actors: A model of concurrent computation in distributed systems. MIT Press, 1986.
- [NIE-87] Oscar Nierstrasz. Active Objects in Hybrid. SIGPLAN Notices, 1987
- [LAV-96] R. Greg Lavender, Douglas C. Schmidt. Active Object An Object Behavioral Pattern for Concurrent Programming. Proc.Pattern Languages of Programs, 1996
- [GEL-92] D. Gelernter, N. Carriero. Coordination Languages and their Significance. Communications of the ACM. Vol 33, Issue 2, Feb. 1992
- [JOH06] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. 2006. Creol: a type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365, 1 (November 2006), 23-66
- [KAR09] Karmani, Shali, Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. PPPJ 09

## Modeling concurrent program execution (module 1.2)

### Realizzazione di modelli per programmi concorrenti

I modelli e le astrazioni ricoprono un ruolo importante per la scienza e l'ingegneria informatica:

- I **modelli** sono descrizioni o rappresentazioni rigorose della struttura di un programma e dei suoi comportamenti, i modelli sono realizzati al livello di astrazione più appropriato tale da includere gli aspetti rilevanti ed ignorare dettagli meno importanti
- I **diagrammi** sono utili per la progettazione dei sistemi
- I **modelli formali** vengono utilizzati per l'analisi e la verifica dei programmi.

È quindi importante utilizzare modelli, diagrammi e modelli formali per la progettazione di programmi concorrenti per definirne la struttura e i comportamenti senza specificare dettagli implementativi e realizzativi e per garantire la possibilità di ragionare sui comportamenti dinamici dei sistemi e verificarne la correttezza.

### Diagrammi degli stati e scenari di esecuzione

La modellazione di programmi concorrenti deve prima di tutto fondarsi sull'assunzione di indipendenza di velocità: le azioni atomiche dei diversi processi coinvolti in un programma concorrente vengono eseguite in ordine arbitrario, si deve modellare un programma concorrente come una sequenza arbitraria delle azioni dei processi. Ogni processo è una sequenza di azioni atomiche.

Dal punto di vista della modellazione:

- Un unico processore globale *astratto* esegue tutte le azioni
- Le istruzioni atomiche vengono eseguiti interamente, non possono essere frammentati
- Durante la computazione il control pointer indicare la prossima istruzione che può essere eseguito in ogni processo.

Una **computazione** o uno **scenario** è una delle sequenze di esecuzione che si può ottenere come risultato dell'interleaving.

Ad esempio, dati due processi **p** e **q** che condividono l'accesso ad una variabile intera **n** strutturati come segue:

integer n := 0	
p	q
integer k1 := 1	integer k2 := 1
<b>p1</b> : n := k1	<b>q1</b> : n := k2

Si ha che:

- Ogni linea etichettata rappresenta un'istruzione atomica
- Ogni processo ha una memoria privata (variabili locali, nell'esempio **k1** e **k2**)
- Ogni processo condivide una memoria (variabili globali, nell'esempio **n**)
- L'esecuzione del programma prevede due possibili scenari:
  - **p1** poi **q1** (la variabile **n** al termine dell'esecuzione è valorizzata a 2)
  - **q1** poi **p1** (la variabile **n** al termine dell'esecuzione è valorizzata a 1).

Formalmente, dato un modello, l'esecuzione di un programma concorrente può essere rappresentata da una serie di stati e di transizioni da uno stato all'altro:

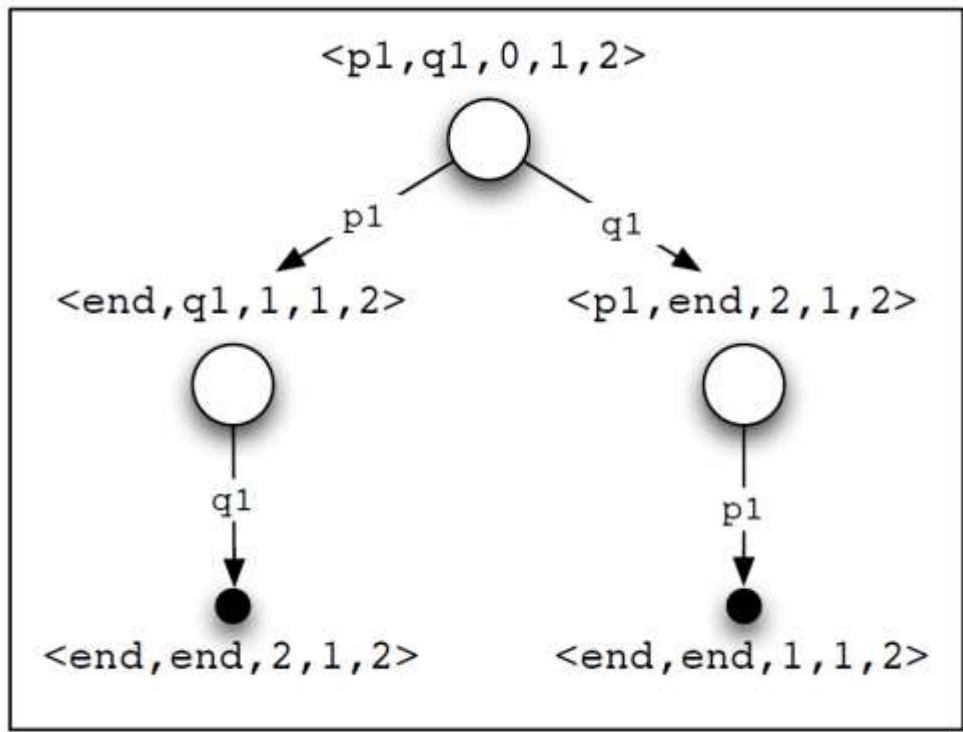
- uno **stato** è definito da una tupla che consiste di:
  - un elemento etichettato di ogni processo
  - un elemento per ogni variabile globale o locale, ogni elemento lo stesso tipo della relativa variabile
- si ha una **transizione** da uno stato **s1** ad uno stato **s2** se l'esecuzione di un'istruzione dallo stato **s1** cambia lo stato in **s2**

- l'istruzione eseguita deve essere una di quelle referenziate da un control pointer quando il programma si trova nello stato s1.

Il **diagramma degli stati** è un grafo che include tutti gli stati raggiungibili dal programma:

- gli scenari sono rappresentati come percorsi orientati lungo il diagramma dallo stato iniziale
- eventuali cicli determinano la possibilità di computazioni infinite in grafi finiti.

Il diagramma degli stati per l'esempio sopra riportato è il seguente, presenta 5 stati e due scenari.



Il diagramma degli stati per un secondo esempio è riportato di seguito

p	q
p1: print("p1")	q1: print("q1")
p2: print("p2")	q2: print("q2")

Il diagramma presenta 6 scenari differenti:

p1 p2 q1 q2

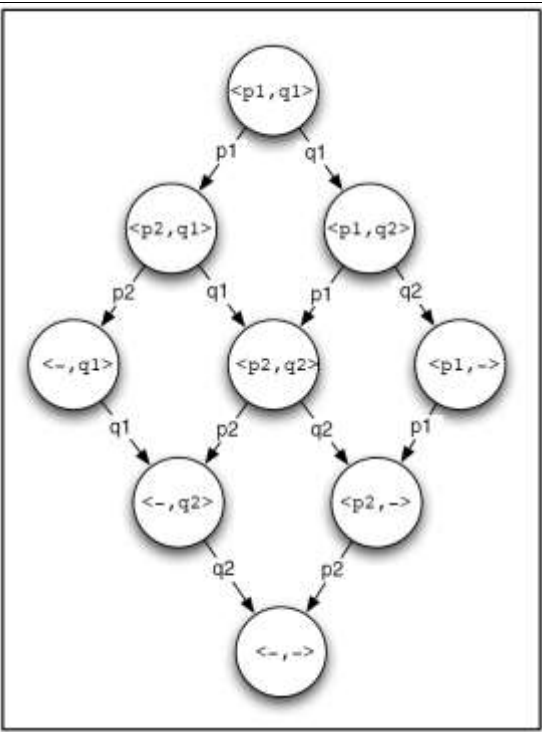
p1 q1 p2 q2

p1 q1 q2 p2

q1 q2 p1 p2

q1 p1 q2 p2

q1 p1 p2 q2



Un problema legato ai diagrammi degli stati è che all’aumentare del numero di istruzioni e di processi coinvolti, il numero degli stati e degli scenari cresce troppo rapidamente per poter essere gestito con chiarezza:

- un sistema con due processi e tre istruzioni per uno genera 20 scenari differenti
- un sistema con due processi e quattro istruzioni genera 70 scenari differenti
- ...
- 8 istruzioni per due processi determinano 12820 scenari differenti
- ...

La cosa viene amplificata al crescere del numero di processi coinvolti.

In generale, è possibile calcolare il numero *ns* di scenari prodotti da *n* processi, ognuno definito da *m<sub>i</sub>* azioni, come:

$$ns = \frac{(\sum_{i=1}^n m_i)!}{\prod_{i=1}^n (m_i)!}$$

	n = 2	3	4	5	6
m <sub>i</sub> = 2	6	90	2520	113400	2 <sup>22.8</sup>
3	20	1680	2 <sup>18.4</sup>	2 <sup>27.3</sup>	2 <sup>36.9</sup>
4	70	34650	2 <sup>25.9</sup>	2 <sup>38.1</sup>	2 <sup>51.5</sup>
5	252	2 <sup>19.5</sup>	2 <sup>33.4</sup>	2 <sup>49.1</sup>	2 <sup>66.2</sup>
6	924	2 <sup>24.0</sup>	2 <sup>41.0</sup>	2 <sup>60.2</sup>	2 <sup>81.1</sup>

L’importanza dell’atomicità delle azioni

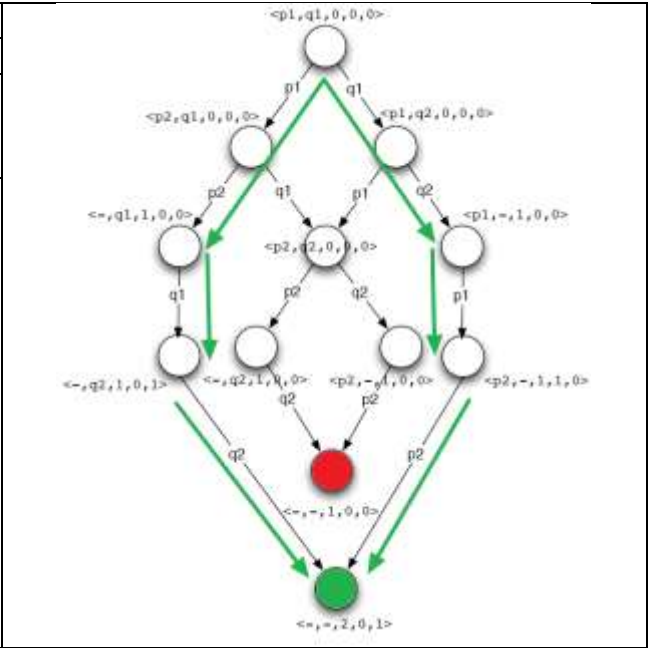
Descrivere le azioni dei processi in maniera appropriata non è solo utile a limitare il numero di stati e di scenari di un sistema concorrente, è fondamentale infatti descrivere le azioni con un corretto livello di **atomicità** allo scopo di ottenere solo scenari consistenti e validi in base alla semantica del problema da gestire.

Un semplice esempio rappresentativo riguarda l’incremento di variabili, esso può essere gestito con azioni atomiche:

integer n := 0	
p	q
p1: n := n + 1	q1: n := n + 1

Oppure con azioni non atomiche:

integer n := 0	
p	q
integer tmp; p1: tmp := n p2: n := tmp + 1	integer tmp; q1: tmp := n q2: n := tmp + 1
In questo secondo caso di generano 6 scenari dei quali non tutti portano ad un risultato corretto.	



Esempi tipici da considerare riguardano gli incrementi a livello di codice macchina.

STACK MACHINE		REGISTER MACHINE	
integer n := 0		integer n := 0	
p	q	p	q
p1: push n p2: push #1 p3: add p4: pop n	q1: push n q2: push #1 q3: add q4: pop n	p1: load R1, n p2: add R1, #n p3: store n, R1	q1: load R1, n q2: add R1, #n q3: store n, R1

La nozione “**atomico**” può essere riferita non solo alle azioni ma anche alle strutture dati:

- un oggetto può essere definito **atomico** se può trovarsi in un numero finito di stati uguale al numero di valori che può assumere: le azioni cambiano atomicamente il suo stato
- i dati primitivi nei linguaggi di programmazione concorrenti sono solitamente atomici (non sempre però: ad esempio i dati di tipo double in Java).

I tipi di dati astratti, composti da diversi oggetti più semplici, sono tipicamente non atomici (es. classi nei linguaggi OO o le strutture in C).

In questi casi è possibile identificare due tipi di base per gli stati: **stato interno e stato esterno**:

- lo stato interno è significativo per chi definisce l’oggetto
- lo stato esterno è significativo per chi utilizza l’oggetto.

La corrispondenza tra i due stati, interno ed esterno, è parziale:

- possono esistere stati interno che non hanno stati esterni corrispondenti
- gli stati interni che hanno stati esterni corrispondenti sono detti **consistenti**.

Per questi motivi l’esecuzione di un’operazione su una struttura dati non atomica può attraversare stati che non sono consistenti. Questo problema non si verifica nella programmazione sequenziale, viceversa è un problema importante nella programmazione concorrente: può accadere che un processo richieda di lavorare con un oggetto che si trova in uno stato inconsistente, mentre un altro processo sta lavorando con esso. Diventa allora necessario introdurre meccanismi appropriati che possano garantire che i processi lavorino su oggetti che si trovano sempre in stati consistenti.

Processi ciclici

In caso di processi ciclici le cose si complicano ulteriormente ed è possibile costruire differenti scenari a seconda, tra l'altro, del numero di cicli eseguiti per processo.

<code>integer n := 1</code>	
<b>p</b>	<b>q</b>
<code>p1: while (n &lt; 1)</code>	<code>q1: while (n &gt;= 0)</code>
<code>p2: n := n + 1</code>	<code>q2: n := n - 1</code>

Ad esempio è possibile realizzare un diagramma degli stati che rappresenti gli scenari per cui il ciclo del processo p viene eseguito esattamente una sola volta e il ciclo del processo q viene eseguito esattamente tre volte.

Fatte queste considerazioni è necessario tuttavia riconoscere che nella realtà i computer non hanno uno stato globale (per ragioni legate alla fisica), il ruolo dell'astrazione è allora creare modelli nei quali una certa entità globale esegue il programma concorrente esegua in maniera arbitraria le istruzioni (ordinate) di uno o di un altro processo.

Le analisi possono essere fatte ignorando il tempo ma focalizzando l'attenzione sugli ordinamenti parziali delle sequenze di azioni e sull'atomicità delle singole azioni (di fondamentale importanza è la scelta di cosa debba essere eseguito in maniera atomica e cosa no). I sistemi devono essere progettati (e i modelli devono permettere di farlo) in modo tale da garantire **robustezza** sia rispetto ai cambiamenti di hardware e software indipendentemente dalle variazioni dei tempi di esecuzione e delle performance.

I programmi concorrenti sono quindi molto sensibili all'analisi formale necessaria ad assicurare la loro **correttezza**.

## Proprietà relative alla correttezza: safety, liveness, fairness

### Correttezza

La verifica della correttezza di un programma sequenziale consiste nell'applicare unità di test basate su specifici input e verificare che l'output ottenuto sia quello previsto in modo da effettuare eventuali diagnosi, correggere bug e ri-eseguire i test. L'esecuzione di uno stesso test restituisce sempre gli stessi risultati.

La programmazione concorrente cambia le prospettive in quanto – in generale – gli stessi input possono generare differenti risultati in funzione dei diversi scenari che vengono prodotti dal sistema, alcuni scenari possono restituire i risultati corretti e altri no. Non è inoltre possibile effettuare un debug di un programma concorrente con le stesse modalità utilizzate per i programmi sequenziali, proprio perché ogni esecuzione potrebbe determinare lo svolgimento di differenti scenari.

Servono allora differenti tipi di approcci basati su modelli astratti, sulle analisi formali e sul model checking.

La **correttezza** di un programma concorrente (che potrebbe non terminare) è definita in termini di **proprietà** computazionali: condizioni che devono essere garantite in ogni possibile scenario.

Esistono due tipi di proprietà legate alla correttezza:

- **safety**
- **liveness**.

### Safety

Questa proprietà deve essere vera sempre: in ogni stato di ogni computazione. La safety è un'invariante della computazione.

Tipicamente questa proprietà viene utilizzata per specificare che non dovrebbero mai accadere cose sbagliate:

- **mutua esclusione**: un unico processo alla volta può accedere ad una regione/sezione critica
- **assenza di deadlock**: nessun processo può rimanere indefinitamente in attesa che un determinato evento accada
- ...

### Liveness

Questa proprietà deve poter eventualmente essere vera: data una certa proprietà P, deve essere vero che in ogni computazione esista un qualche stato in cui P sia verificata.

Questa proprietà viene utilizzata per definire “cose buone” che possono accadere:

- **assenza di starvation**: un processo è in grado di ottenere la risorsa della quale ha bisogno
- **assenza di dormancy**: un processo in attesa viene riattivato
- **comunicazione affidabile**: un messaggio inviato da un processo ad un altro viene ricevuto
- ...

### Fairness

Questa proprietà riguarda il fatto che “qualcosa di buono” accada *infinitamente spesso*, ad esempio che ogni processo ottiene un suo turno di esecuzione per l'intera durata di esecuzione di un sistema concorrente, o che ogni processo che effettua una richiesta ottenga una risposta.

I programmi possono avere differenti comportamenti *fairness* dipendenti dal modo con cui le singole azioni vengono schedate ed eseguite: l'interleaving dipende da una **scheduling policy**:

- **Unconditional fairness**: ogni azione atomica (senza condizioni) che è pronta per essere eseguita sarà eseguita



- **Weak fairness:** ogni azione atomica (senza condizioni) e ogni azione atomica condizionata (la cui condizione diventa e rimane vera) sarà eseguita
- **Strong fairness:** ogni azione atomica (senza condizioni) e ogni azioni atomica condizionata (la cui condizione diventa vera *infintamente spesso*) sarà eseguita.

## Il problema della sezione critica

Il problema della sezione critica è stato introdotto da Dijkstra nel 1965, può essere considerato come il più importante problema della programmazione concorrente in un contesto di competizione dei processi e come tale è anche il più studiato.

### Definizione del problema della sezione critica

È dato un sistema composto da N processi, ognuno dei quali esegue un ciclo infinito di istruzioni che possono essere divise in de sotto-sequenze:

- sezione critica (CS)
- sezione non critica (NCS).

Ogni sezione critica è, tipicamente, una sequenza di istruzioni che accedono ad un qualche oggetto condiviso.

Si devono progettare protocolli di accesso e uscita dalla sezione critica che soddisfino le seguenti proprietà:

- **mutua esclusione:** le istruzioni della sezione critica di due differenti processi non possono essere interleaved
- **assenza di deadlock:** se qualche processo richiede di accedere alla sezione critica, prima o poi uno di essi dovrà potervi accedere
- **assenza di starvation individuale dei processi:** se qualsiasi processo accede alla sezione critica, prima o poi deve accede (proprietà di attesa limitata).

Ogni soluzione proposta deve soddisfare anche la proprietà di **progresso** per la sezione critica:

- una volta che un processo inizia l'esecuzione delle istruzioni della sua sezione critica, deve poi terminarle tutte.

Le istruzioni fuori dalla sezione critica non hanno vincoli di progresso: si possono avere loop infiniti o terminazioni al di fuori della sezione critica.

### Problemi concreti basati sulla sezione critica

Il problema della sezione critica permette di modellare una serie di sistemi nei quali vengono eseguite computazioni complesse che richiedono, occasionalmente, di accedere a dati o risorse (hardware) condivisi da diversi processi (es. chioschi per il check-in in aeroporto che accedono ad un database centrale de passeggeri).

Il problema della sezione critica può essere semplicemente risolto utilizzando meccanismi di base come i semafori o i locks, disponibili nelle *concurrent machines*. Prima di affrontare questi argomenti è utile vuole focalizzare l'attenzione su una soluzione puramente algoritmica che non utilizzi meccanismi di alto livello e che utilizzi solo istruzioni di base atomiche (es. load, store).

Una prima analisi si focalizza sul problema della sezione critica in un sistema con due processi p e q che condividono qualche variabile globale.

Global variables	
P	q
Local variables	Local variables
Loop forever	Loop forever
Non-critical section	Non-critical section
Pre-protocol	Pre-protocol
Critical section	Critical section
Post-protocol	Post-protocol

Primo tentativo

Integer turn ← 1	
P	q
Loop forever	Loop forever
p1: Non-critical section	q1: Non-critical section
p2: await turn = 1	q2: await turn = 2
p3: Critical section	q3: Critical section
p4: turn ← 2	q4: turn ← 1

L’istruzione await interrompe l’esecuzione finché la condizione (turn = 1) diventa vera.  
Per verificare la correttezza della soluzione si costruisce il diagramma degli stati del programma concorrente: il programma deve soddisfare le tre proprietà. Per verificare le proprietà di correttezza è necessario esaminare l’insieme degli stati raggiungibili e le transizioni.

Gli stati sono triple <pi, qj, turn> in cui:

- pi è lo stato corrente del processo p
- qj è lo stato corrente del processo q
- turn è il contenuto della variabile condivisa.

Il diagramma viene costruito incrementale partendo dallo stato iniziale e considerando gli stati raggiungibili.  
La proprietà di mutua esclusione è verificata se nessuno degli stati accessibili presenta le istruzioni p3 e q3 contemporaneamente (<p3, q3, \_>).

Allo scopo di studiare la correttezza è possibile ridurre il numero degli stati senza alterare al semantica del programma (eliminando istruzioni non essenziali): qualsiasi istruzione eseguita all’interno delle sezione critica e all’interno di sezioni non critiche è irrilevante ai fini della verifica della correttezza e può essere ignorata.

Integer turn ← 1	
P	q
Loop forever	Loop forever
p1: await turn = 1	q1: await turn = 2
p2: turn ← 2	q2: turn ← 1

La proprietà di mutua esclusione è rispettata se non esistono stati del tipo <p2, q2, \_>, il programma modellato in questo modo permette di generare un diagramma con solo 4 stati.

Analisi della correttezza:

- **la mutua esclusione è soddisfatta:** non esistono stati del tipo <p2, q2, \_>
- **è verificata l’assenza di deadlock:** se un qualche processo tenta di accedere alla propria sezione critica (eseguendo l’istruzione di await) prima o poi riesce
- **non è verificata l’assenza di starvation:** un processo che vuole entrare in sezione critica può rimanere in attesa in maniera indeterminata aspettando un processo che esegue un loop infinito nella sua sezione non critica (es: modellazione con le 4 istruzioni, p1 può contenere un loop infinito, quando il processo q vuole eseguire q2 non riesce mai ad eseguirla).

La variabile Turn agisce come un permesso per accedere alla risorsa “sezione critica”, se il processo che detiene il permesso non lo rilascia mai, l’altro processo non potrà mai accedere alla sezione critica; inoltre entrambi i processi gestiscono una singola variabile condivisa: se un processo termina l’altro rimane bloccato.

Questo primo tentativo offre una soluzione non corretta.

Secondo tentativo

Boolean wantp ← false Boolean wantq ← false	
P	q
Loop forever p1: Non-critical section p2: await wantq = false p3: wantp ← true p4: Critical section p4: wantp ← false	Loop forever q1: Non-critical section q2: await wantp = false q3: wantq ← true q4: Critical section q4: wantq ← false

Si introducono due variabili globali differenti che indicano quanto un processo è all’interno della propria Sezione Critica. A garanzia della mutua esclusione non devono essere presenti scenari contenenti le istruzioni p4 e q4 in uno stesso stato.

La descrizione senza le istruzioni irrilevanti è:

Boolean wantp ← false Boolean wantq ← false	
P	q
Loop forever p1: await wantq = false p2: wantp ← true p3: wantp ← false	Loop forever q1: await wantp = false q2: wantq ← true q3: wantq ← false

A garanzia della mutua esclusione non devono essere presenti scenari contenenti le istruzioni p3 e q3 in uno stesso stato.

Costruendo il diagramma si può osservare che la mutua esclusione non è garantita perché è possibile raggiungere lo stato <p3, p3, true, true>, questo a causa della non atomicità dei protocolli pre-post sezione critica. In altri termini, entrambi i processi posso eseguire le loro prime istruzioni (await) accedendo alle istruzioni successive che bloccherebbero l’accesso dell’altro processo.

### Terzo tentativo

Boolean <b>wantp</b> $\leftarrow$ false Boolean <b>wantq</b> $\leftarrow$ false	
P	q
Loop forever p1: Non-critical section p2: <b>wantp</b> $\leftarrow$ true p3: await <b>wantq</b> = false p4: Critical section p4: <b>wantp</b> $\leftarrow$ false	Loop forever q1: Non-critical section q2: <b>wantq</b> $\leftarrow$ true q3: await <b>wantp</b> = false q4: Critical section q4: <b>wantq</b> $\leftarrow$ false

Spostando l'istruzione di await all'interno della sezione critica si risolve il problema della mutua esclusione, tuttavia la soluzione non esclude deadlock (o meglio, in questo caso, livelock) in quanto entrambi i processi potrebbero eseguire la seconda istruzione rimanere in attesa infinita sulle istruzioni di await.

### Quarto tentativo

Boolean <b>wantp</b> $\leftarrow$ false Boolean <b>wantq</b> $\leftarrow$ false	
P	q
Loop forever p1: Non-critical section p2: <b>wantp</b> $\leftarrow$ true p3: while <b>wantq</b> p4: <b>wantp</b> $\leftarrow$ false p5: <b>wantp</b> $\leftarrow$ true p6: Critical section p7: <b>wantp</b> $\leftarrow$ false	Loop forever q1: Non-critical section q2: <b>wantq</b> $\leftarrow$ true q3: while <b>wantp</b> q4: <b>wantq</b> $\leftarrow$ false q5: <b>wantq</b> $\leftarrow$ true q7: Critical section q4: <b>wantq</b> $\leftarrow$ false

Il deadlock della terza soluzione si verificava perché entrambi i processi provavano simultaneamente ad accedere alla sezione critica: questo tentativo fa in modo che un processo che vuole accedere alla sezione critica è in grado di capire se anche l'altro vuole accedere simultaneamente e, in tal caso, gli concede la precedenza.

Il deadlock è risolto ma esiste ancora un problema di starvation nel caso di perfetta sequenzialità delle istruzioni p4, q4, p5, q5.

L’algoritmo di Dekker

Il problema della sezione critica può essere risolto con una semplice variazione rispetto a quanto proposto nel quarto tentativo. Questa soluzione individuata per la prima volta dal matematico olandese T.J. Dekker nel 1965, utilizza una variabile turno per definire quale dei due processi che sta richiedendo simultaneamente l’accesso alla sezione critica, abbia diritto di accedervi sulla base di un turno, una schedulazione.

Boolean wantp ← false Boolean wantq ← false Integer turn ← 1	
p	q
Loop forever p1 : Non-critical section p2 : wantp ← true p3 : while wantq p4 :   if turn = 2 p5 :     wantp ← false p6 :     await turn = 1 p7 :     wantp ← true p8 : Critical section p9 : turn ← 2 p10: wantp ← false	Loop forever q1 : Non-critical section q2 : wantq ← true q3 : while wantp q4 :   if turn = 1 q5 :     wantq ← false q6 :     await turn = 2 q7 :     wantq ← true q8 : Critical section q9 : turn ← 1 q10: wantq ← false



L’algoritmo di Dekker è corretto: soddisfa la mutua esclusione e garantisce assenza di deadlock e starvation.

L’algoritmo di Peterson

Un’altra soluzione proposta nel 1981 da Peterson, Manna-Pneuli, Doran-Thomas, definisce un miglioramento dell’algoritmo di Dekker rendendolo più conciso collassando due istruzioni await in una unica con un *condizione composta*.

Boolean wantp ← false Boolean wantq ← false Integer turn ← 1	
p	q
Loop forever p1: Non-critical section p2: wantp ← true p3: turn ← 2 p4: await (wantq = false or turn = 1) p5: Critical section p6: wantp ← false	Loop forever q1: Non-critical section q2: wantq ← true q3: turn ← 1 q4: await (wantp = false or turn = 2) q5: Critical section q6: wantq ← false

Istruzioni composte atomiche

L’algoritmo di Dekker funziona su una qualsiasi architettura che sia in grado di fornire istruzioni di base load and store. In realtà l’algoritmo e il problema della sezione critica può essere estremamente semplificato se è possibile utilizzare istruzioni atomiche più complesse (fornite direttamente dalla macchina concorrente).

I principali esempi di istruzioni composte atomiche sono test-and-set, fetch-and-add, compare-and-swap.

Ad esempio, test-and-set è un’istruzione atomica definite come l’esecuzione di due istruzioni in maniera atomica: test-and-set (x,r) : < r := x, x:= 1 >

Una notazione comune per specificare l'atomicità delle istruzioni consiste nel racchiudere il gruppo di istruzioni tra parentesi angolari.

Il problema della sezione critica può essere descritto tramite test-and-set:

Integer lock $\leftarrow 0$	
p	q
Integer was_locked  Loop forever p1: Non-critical section repeat p2:   test_and_set(lock,was_locked) p3: until was_locked = 0 p4: Critical section p5: lock $\leftarrow 0$	Integer was_locked  Loop forever q1: Non-critical section repeat q2:   test_and_set(lock,was_locked) q3: until was_locked = 0 q4: Critical section q5: lock $\leftarrow 0$

Utilizzando le istruzioni composte atomiche è possibile realizzare semplicemente un meccanismo di tipo lock di base che offre una semplice soluzione al problema della sezione critica.

Il lock prevede due fasi: acquisizione e rilascio.

Lock shaderlock;	
p	q
Loop forever p1: Non-critical section p2: acquire(sharedlock) p3: Critical section p4: release(shaderlock)	Loop forever q1: Non-critical section q2: acquire(sharedlock) q3: Critical section q4: release(shaderlock)

In JAVA le istruzioni atomiche possono essere implementate utilizzando l'etichetta synchronized in blocchi che utilizzano lo stesso lock.

<pre>// process (thread) A ... Synchronized (lock) {     &lt;statement a&gt;     &lt;statement b&gt;     ... } ...</pre>	<pre>// process (thread) B ... Synchronized (lock) {     &lt;statement c&gt;     &lt;statement d&gt;     ... } ...</pre>
--	--

In questo modo le sequenze di istruzioni a, b e c, d vengono eseguite senza interruzione e senza interleave tra le diverse istruzioni, eliminando scenari che includono sequenze come a, c, b, d o c, a, d, b.

## L'algoritmo del fornaio (bakery ticket)

Una soluzione alternativa al problema della sezione critica per un sistema con N-processi, prevede di introdurre un ticket per stabilire l'ordine di accesso alla sezione critica. L'implementazione reale necessita la gestione l'eventuale overflow sulla variabile num.

Int num $\leftarrow$ 1 Int next $\leftarrow$ 1 Turn[1:n] $\leftarrow$ [0, 0, 0, ...]
<b>p[i]</b>
Loop forever p1: Non-critical section p2: < turn[i] $\leftarrow$ num; num $\leftarrow$ num + 1 > p3: await turn[i] = next p4: Critical section p5: next $\leftarrow$ next + 1

## Correttezza: verifica, test, validazione

Un **fault** è la manifestazione di un errore, ovvero di un'azione che produce un risultato non corretto.

Quando si manifesta un fault, può generarsi una **failure**, essa può essere causata da diversi fault.

Un fault può generare diverse failures.

Le failures sono relative alle specifiche.

L'attività di **test** ha l'obiettivo di individuare le cause (faults) che hanno generato una o più failures.

L'attività di **verifica** ha l'obiettivo di verificare che il sistema soddisfi le sue specifiche (si è implementato il sistema in maniera corretta?).

L'attività di **validazione** ha l'obiettivo di verificare che un sistema soddisfi le attese del cliente (si è implementato il sistema giusto?).

## Test di liveness e safety

Test di safety (sicurezza):

- verifica che il comportamento di una classe o di un sistema sia conforme alle sue specifiche
- un test di safety assume solitamente la forma di test **invariante**
- un test di safety solitamente utilizza delle **asserzioni**.

Test di liveness (progresso):

- verifica è garantita liveness nel sistema
- è molto complesso da implementare.

## Problemi e limiti dei test

Tipicamente è possibile controllare solo un sottoinsieme di tutti gli scenari possibili, inoltre è necessario affrontare il fenomeno degli **heisenbugs**: effettuare un test in un sistema concorrente può introdurre un rallentamento o una sincronizzazione (tramite artefatti dei test) che possono nascondere errori legati proprio al tempo di esecuzione.

## Test vs verifica

L'attività di test è volta a verificare che una determinata proprietà sia verificata per alcuni scenari selezionati: i test rivelano la presenza di errori, non garantiscono la loro assenza.

L'attività di verifica ha invece l'obiettivo di verificare che una proprietà sia verificata in tutti i possibili scenari. La verifica richiede l'utilizzo di tecniche formali quali, ad esempio, il model checking.

## Metodi formali per la verifica

Esistono due principali classi di tecniche formali:

- **model checking**: la verifica viene effettuata generando uno alla volta tutti gli stati del sistema e controllando il rispetto delle proprietà in ogni stato
- **inductive proofs of invariants**: le proprietà invarianti sono dimostrate per induzione basandosi sugli stati del sistema, può essere effettuata tramite tools automatici detti *sistemi deduttivi*.

Entrambe le tecniche utilizzano una qualche forma di linguaggi formali di analisi per specificare la correttezza delle proprietà.

### Correttezza delle proprietà tramite calcolo delle proposizioni

Il calcolo delle proposizioni permette di esprimere la correttezza delle proprietà tramite formule logiche che devono essere vere per poter dimostrare la proprietà in un qualche stato del sistema. Le *formulae* sono asserzioni ottenute componendo altre proposizioni utilizzando connettori logici (e, o, non, implicazione, equivalenza, ...).

Le proposizioni riguardano, nel caso della correttezza di programmi concorrenti, il valore delle variabili e il valore del control pointer durante l'esecuzione del programma, ad esempio: data la variabile booleana `wantp`, la proposizione atomica (asserzione) `wantp` è vera in un certo stato se e solo se il valore della variabile `wantp` è vero in quello stato.

Ogni etichetta di un'istruzione di un processo viene utilizzata come proposizione atomica con il significato di indicare l'istruzione a cui indirizza il control pointer di ogni processo; ad esempio la proposizione `p1` asserisce che il control pointer del processo `p` indirizza all'istruzione con etichetta `p1`.

Riprendendo l'esempio del terzo tentativo di risoluzione del problema della sezione critica, la formula  $p4 \wedge q4$  è vera se i control pointer dei due processi si trovano nella sezione critica, se esiste uno stato per cui questa formula è vera allora la proprietà di mutua esclusione sulla sezione critica non è soddisfatta. Viceversa, un programma soddisfa la proprietà di mutua esclusione se la formula  $\neg (p4 \wedge q4)$  è vera in ogni possibile stato di un qualsiasi scenario.

Boolean <b>wantp</b> ← false Boolean <b>wantq</b> ← false	
P	q
Loop forever p1: Non-critical section p2: <b>wantp</b> ← true p3: await <b>wantq</b> = false p4: Critical section p4: <b>wantp</b> ← false	Loop forever q1: Non-critical section q2: <b>wantq</b> ← true q3: await <b>wantp</b> = false q4: Critical section q4: <b>wantq</b> ← false

### Linear Temporal Logic (LTL)

I processi e i sistemi cambiano il loro stato nel tempo e così come l'interpretazione delle *formulae* circa il loro stato. Per questo motivo serve un linguaggio formale per l'analisi che consideri questi aspetti, la **logica temporale** è una logica di base piuttosto diffusa (Amir Pnueli, "The Temporal Logic of Programs", 18th Annual Symposium on Foundations of Computer Science, 1977). La logica temporale è ottenuta aggiungendo operatori in grado di manipolare le condizioni sul tempo alla logica delle proposizioni o dei predicati.

La **Linear Temporal Logic (LTL)** è in grado di esprimere proprietà che devono essere vere in un determinato stato per ogni possibile scenario seguendo un modello del tempo lineare e discreto.

La **Branching Temporal Logic** esprimono le proprietà che possono essere vere in tutti o alcuni scenari a partire da uno stato, un esempio è rappresentato dalla logica CTL (**Computational Tree Logic**).



Operatori logici

La logica LTL è basata su due operatori logici temporali di base: **always (sempre)** e **eventually (prima o poi)**.

Operatori unari

L’operatore **always** (o box) è rappresentato graficamente da un quadrato, la formula  $\Box A$  è vera in uno stato  $s_i$  di una computazione se e solo se la formula  $A$  è vera in tutti gli stati  $s_j$  con  $j \geq i$ . In certi casi si utilizza una  $G$  maiuscola al posto del simbolo quadrato ( $G = \text{Globally}$ ). L’operatore **always** può essere utilizzato per specificare proprietà di safety in quanto specifica cosa deve essere vero sempre.

L’operatore **eventually** (o diamond) è rappresentato graficamente da un rombo, la formula  $\Diamond A$  è vera in uno stato  $s_i$  di una computazione se e solo se la formula  $A$  è vera in almeno uno stato  $s_j$  con  $j \geq i$ . In certi casi si utilizza una  $F$  maiuscola al posto del simbolo quadrato ( $F = \text{Finally}$ ). L’operatore **eventually** può essere utilizzato per specificare proprietà di liveness in quanto specifica cosa deve essere vero almeno una volta nel futuro.

L’operatore **next**, rappresentato con un cerchio ( $\bigcirc A$ ), definisce che una proprietà deve essere vera nello stato successivo in ogni possibile scenario, in certi casi si utilizza una  $X$  maiuscola (**neXt**).

Operatori binari

L’operatore **until**, rappresentato tramite una  $U$  maiuscola, permette di definire che una proposizione è vera finché non lo diventa la seconda:  $A \ U \ B$  è vero in uno stato  $s_i$  di una computazione se e solo se la formula  $B$  è vera in un qualche stato  $s_j$  con  $j \geq i$  e la formula  $A$  è vera in tutti gli stati  $s_k$  con  $i \leq k < j$ .

L’operatore **Weak-until**, rappresentato tramite una  $W$  maiuscola, permette di definire che una proposizione è vera finché non lo diventa la seconda che può eventualmente non diventarlo mai:  $A \ W \ B$  è vero in uno stato  $s_i$  di una computazione se e solo se la formula  $B$  è vera in un qualche stato  $s_j$  con  $j \geq i$  e la formula  $A$  è vera in tutti gli stati  $s_k$  con  $i \leq k < j$  oppure se la formula  $B$  rimane false per sempre e  $A$  rimane vera per sempre.

Proprietà di base

- Riflessività:  $\Box A \rightarrow A$
  - Dualità:  $\neg \Box A = \Diamond \neg A$
  - Sequenza degli operatori:  $\Diamond \Box A$
- $A \rightarrow \Diamond A$
  - $\neg \Diamond A = \Box \neg A$
  - $\Box \Diamond A$

Gli operatori **always** e **eventually** possono definiti in termini di **until**:

- $\Diamond A = \text{true} \ U \ A$
- $\Box A = \neg \Diamond (\neg A) = \neg \Diamond (\neg (\text{true} \ U \ A))$

Deduzione tramite logiche temporali

La logica temporale è un sistema di logica deduttiva con i propri assiomi e le proprie regole di inferenza, può essere utilizzato per formalizzare la semantica dei programmi concorrenti e per dimostrare in maniera rigorosa le proprietà dei programmi. Esempi di teoremi:

- $(\Diamond \Box A1 \wedge \Diamond \Box A2) \rightarrow \Diamond \Box (A1 \wedge A2)$     è vero
- $(\Box \Diamond A1 \wedge \Box \Diamond A2) \rightarrow \Box \Diamond (A1 \wedge A2)$     è falso

Rifacendosi al primo tentativo di soluzione del problema della sezione critica, è possibile specificare: <b>mutua esclusione:</b> $\Box \neg (p3 \wedge q3)$ <b>progressione stato attuale:</b> $p2 \rightarrow \Diamond p3$ <b>progressione tutti gli stati:</b> $\Box (p2 \rightarrow \Diamond p3)$	Integer turn $\leftarrow 1$	
	<b>P</b>	<b>q</b>
	Loop forever p1: Non-critical section p2: await turn = 1 p3: Critical section p4: turn $\leftarrow 2$	Loop forever q1: Non-critical section q2: await turn = 2 q3: Critical section q4: turn $\leftarrow 1$

## Tecniche di verifica: model-checking

Il model checking è la più importante e utilizzata tecnica per il controllo automatico della correttezza delle proprietà di sistemi concorrenti, rappresenta un inestimabile strumento pratico per gli ingegneri software.

La strategia di verifica si basa su una ricerca esaustiva dell'intero spazio degli stati di un sistema e sulla verifica che determinate proprietà siano soddisfatte: le proprietà sono espresse come specifiche in una qualche logica temporale proposizionale riguardanti lo stato o gli stati del sistema; se il sistema soddisfa le proprietà il verificatore genera una risposta di conferma, viceversa genera una traccia utile ad identificare i bugs.

Il model checking può essere applicato sia al software che all'hardware:

- Hardware
  - Requisiti
  - Progettazione
  - Circuiti
- Software
  - Requisiti
  - Progettazione
  - Programmi

Il program model checking è l'applicazione delle tecniche di model checking ai sistemi software, solitamente nell'implementazione finale, per individuare difetti software.

Il grande problema delle tecniche di model-checking riguarda la dimensione dello spazio degli stati: è possibile gestire grafi di milioni di stati? Lo stato attuale delle tecniche prevede l'applicazione di regole per ridurre il numero di stati (utilizzando variabili che possono essere modellate in un numero limitato di valori), la costruzione incrementale del grafo (esplorando solo gli stati raggiungibili di un'esecuzione e verificando la verità delle specifiche di correttezza mentre si costruisce il diagramma, bloccando la verifica non appena si riscontra un problema) e l'uso del "model checking simbolico" che lavora con insiemi di stati.

### SPIN e PROMELA

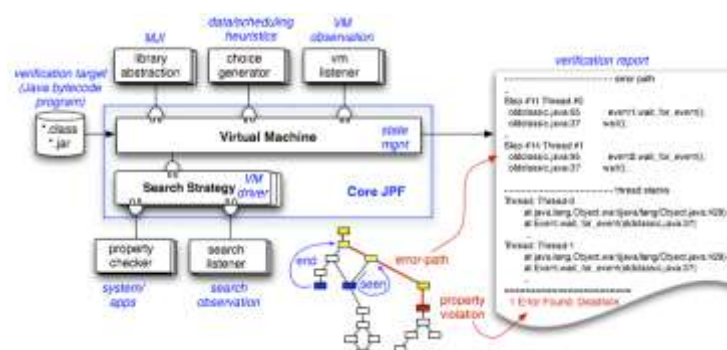
SPIN è un diffuso model checker utilizzato sia in ambito di ricerca che nello sviluppo software a livello industriale, è estremamente efficiente ed è utilizzato per modellare e progettare sistemi concorrenti e distribuiti in generale.

PROMELA è il linguaggio utilizzato in SPIN per scrivere modelli di programmi concorrenti.

### JAVA PATH FINDER (JPF)

JAVA PATH FINDER è un model-checker recente specializzato per la verifica di programmi scritti in Java. È stato sviluppato dalla NASA e utilizzato per software critici. Si tratta di un progetto open source (<http://javapathfinder.sourceforge.net/>). Si tratta di una particolare JVM che esegue programmi teoricamente in ogni possibile scenario (execution paths), verificando la violazione delle proprietà: deadlock, eccezioni non gestite, ...; se trova un errore, JPF propone l'intera esecuzione che ha portato all'errore.

La figura mostra il modello operativo di JPF.



## Tecniche di verifica: verifica induttiva delle invarianti

Un'**invariante** è una formula che deve essere invariabilmente vera in un qualsiasi punto di una qualsiasi computazione. Le invarianti possono essere verificate utilizzando l'induzione attraverso gli stati di tutte le computazioni:

- Per dimostrare che A è un'invariante è necessario:
  - Dimostrare che A è vera nello stato iniziale (caso base)
  - Assumere che A sia vera in un qualsiasi stato S (ipotesi induttiva) e dimostrare che A è vera i tutti i possibili stati successivi a S (passo di induzione).

I sistemi deduttivi sono sistemi software per la dimostrazione automatizzata del teorema.

## Process Interaction and Coordination (module 1.3)

### Meccanismi di base e astrazioni per la coordinazione

Gli algoritmi presentati per cercare di risolvere il problema della sezione critica utilizzano istruzioni nel linguaggio macchina forniti dai computers; si tratta di un livello di astrazione molto *basso* molto vicino all'implementazione fisica. L'introduzione di meccanismi di base e astrazioni per i processi di coordinazione – forniti dalle macchine concorrenti e utilizzate nei linguaggi concorrenti – permette di risolvere i problemi relativi ai programmi concorrenti in maniera più agevole. I principali costrutti sono i **semafori** e i **monitors**.

### Semafori

I semafori sono stati introdotti nel 1968 da Dijkstra, sono molto semplici da utilizzare ma molto potenti e general-purpose, questo rende possibile risolvere quasi qualsiasi problema di mutua esclusione e sincronizzazione.

I semafori sono tipi di dati primitivi forniti dalla macchina concorrente, informalmente funzionano come i semafori stradali bloccando o autorizzando l'accesso ai processi.

### Il tipo di dati semaforo

Un semaforo è un tipo di dati S composto da due campi:

- S.V è un integer  $\geq 0$
- S.L è un insieme di processi (id)

Un semaforo può essere inizializzato con:

- Un valore  $k \geq 0$  per S.V
- Un insieme vuoto  $\{\}$  per S.L

Esempio: semaphore  $S = (k, \{\})$ .

Un semaforo fornisce due operazioni atomiche:

- `wait(S)`: chiamata anche `P(S)` da Dijkstra, viene chiamata quando un processo vuole accedere ad una risorsa, quando `S.V` è negativo, l'accesso è negato.
- `signal(S)`: chiamata anche `V(S)` da Dijkstra, viene chiamata quando un processo libera una risorsa.

Con l'operazione <code>wait</code> , se il valore <code>V</code> del semaforo è $> 0$ (semaforo "verde"), allora il valore viene semplicemente decrementato. Se invece il valore è $= 0$ (semaforo "rosso"), allora il processo è bloccato. L'operazione di <code>wait</code> è <b>atomica</b> .	<pre>wait(S) = &lt; if (S.V &gt; 0)     S.V ← S.V - 1 else     S.L = S.L + {p}     p.state ← blocked &gt;</pre>
Eseguendo l'operazione <code>signal</code> , se non esistono processi in attesa, il valore del semaforo viene incrementato; viceversa si seleziona un processo <code>q</code> bloccato sul semaforo e lo si sblocca. L'operazione di <code>signal</code> è <b>atomica</b> .	<pre>signal(S) = &lt; if (S.L = {})     S.V ← S.V + 1 else     Let q ← arbitrary element on S.L     S.L ← S.L - {q}     q.state ← ready &gt;</pre>

Invariante del semaforo

Sia  $k$  il valore iniziale della componente intera del semaforo,  $\#signal(S)$  il numero di istruzioni `signal(S)` che devono essere eseguite e  $\#wait(S)$  il numero di istruzioni `wait(S)` che dovranno essere eseguite.

Un processo che è bloccato durante l'esecuzione di una `wait(S)`, non ha completato correttamente l'esecuzione dell'istruzione.

Teorema: un semaforo `S` soddisfa sempre le seguenti invarianti:

- $S.V \geq 0$
- $S.V = k + \#signal(S) - \#wait(S)$

Tipi di semafori

- **Mutex** (o semaforo binario):
  - il suo campo intero può assumere solo valori 1 e 0
  - il nome deriva dall'applicazione tipica di questi semafori per implementare la mutua esclusione
- **General** (o semaforo di conteggio):
  - il suo campo intero può contenere qualsiasi valore  $\geq 0$
- **Event**:
  - inizializzato a 0, utilizzato per scopi di sincronizzazione.

Esistono differenti definizioni di tipi di semaforo in relazione alle specifiche proprietà di liveness desiderate, in ogni caso le proprietà legate alla safety sono parte invariante del semaforo.

Principalmente si possono individuare i semafori di tipo:

- **strong o weak:** nei semafori *weak* S.L è un insieme, mentre in quelli *strong* S.L è una coda

<pre>wait(S) = &lt; if (S.V &gt; 0)     S.V ← S.V - 1 else     append(S.L, p)     p.state ← blocked &gt;</pre>	<pre>signal(S) = &lt; if (S.V &gt; 0)     S.V ← S.V - 1 else     let q ← take(S.L)     q.state ← ready &gt;</pre>
--	---

nei semafori *strong* è impossibile che si generi starvation per qualsiasi numero N di processi.

- **busy-wait:** si tratta di semafori senza S.L, le operazioni rimangono atomiche, non vi è garanzia di libertà dalla starvation; si tratta di tipi di semafori appropriati per i sistemi multiprocessore in cui i processi bloccati in attesa hanno un processore dedicato e non c'è spreco di CPU.

## Utilizzo dei semafori

I semafori sono costrutti primitivi che possono essere utilizzati nella costruzione di blocchi a basso livello per risolvere qualsiasi problema riguardante l'interazione tra i processi in architetture con memoria condivisa. In particolare possono essere utilizzati per gestire:

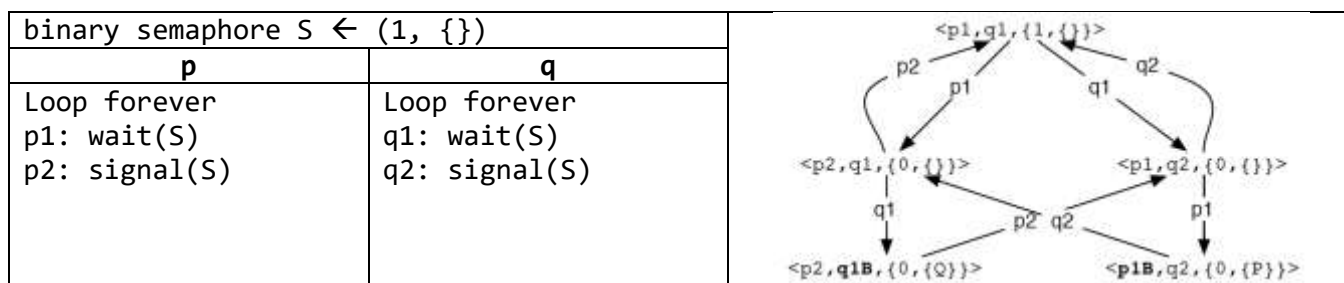
- **mutua esclusione:** ad esempio il problema della sezione critica, implementando locks
- **sincronizzazione:** semafori "event" per segnalare, barriere, ...

Il problema della sezione critica con i semafori

Utilizzando i semafori (come lock) la soluzione del problema della sezione critica diviene triviale:

binary semaphore S ← (1, {})	
p	q
Loop forever p1: Non-critical section p2: wait(S) p3: Critical section p4: signal(S)	Loop forever q1: Non-critical section q2: wait(S) q3: Critical section q4: signal(S)

La versione compatta; può essere utilizzata per verificare che la soluzione sia corretta: garantisca mutua esclusione, libertà da deadlock e starvation.



Nel diagramma gli stati etichettati con B riguardano operazioni bloccate.

La soluzione può essere estesa ad un arbitrario numero N di processi, senza però garantire l'assenza di starvation.

binary semaphore S ← (1, {})	
Any process	
Loop forever p1: Non-critical section p2: wait(S) p3: Critical section p4: signal(S)	

Utilizzo dei semafori come forma di sincronizzazione

I semafori forniscono un meccanismo di base anche per sincronizzare i processi, garantendo l’ordine di esecuzione delle attività. I semafori di tipo evento possono essere utilizzati per inviare o ricevere segnali temporali. I semafori in questo caso vengono inizializzati a (0, {}).

MERGE SORT		
Binary semaphore S1 ← (0,{}) Binary semaphore S2 ← (0,{}) Integer array A		
Sort1	Sort2	Merge
p1: sort 1 <sup>st</sup> half of A p2: signal(S1)	q1: sort 2 <sup>nd</sup> half of A q2: signal(S2)	r1: wait(S1) r2: wait(S2) r3: merge halves of A

Il problema dei produttori e dei consumatori

Si tratta di un classico problema (e pattern di progettazione) che riguarda il corretto ordine di esecuzione delle attività. Esistono due tipi di processi:

- **produttori:** eseguono le istruzioni per produrre/creare elementi dati e per inviare questi elementi ai processi consumatori
- **consumatori:** alla ricezione di elementi dati da un produttore, un processo consumatore esegue le istruzioni per *consumare*.

Esempi di produttori-consumatori:

Produttore	Consumatore
Linea di comunicazione	Web Browser
Web Browser	Linea di comunicazione
Tastiera	Sistema Operativo
Word processor	Stampante
Video gioco	Display
...	...

La comunicazione tra i processi che devono inviarsi dati può essere:

- **sincrona:** la comunicazione non può avvenire se produttore e consumatore non sono pronti
- **asincrona:** il canale di comunicazione è provvisto di una qualche capacità per memorizzare le informazioni da trasmettere; tale disaccoppiamento è molto utile per sistemi dinamici e/o aperti o quando produttori e consumatori agiscono con velocità significativamente diverse. Questo caso prevede quindi l’introduzione di un **buffer** nel quale i produttori memorizzano dati e dal quale i consumatori recuperano i dati.

La modellazione nel caso “ideale” di buffer infinito richiede che vi sia un’unica interazione che debba essere sincronizzata: il consumatore non può cercare di prelevare da un buffer vuoto.

PRODUCER/CONSUMER WITH INFINITE BUFFER	
UnboundedQueue<Item> buffer ← empty queue Semaphore nAvailItems ← (0,{})	
producer	consumer
Loop forever p1: Item e1 ← produce p2: append(buffer,e1) p3: signal(nAvailItems)	Loop forever q1: wait(nAvailItems) q2: Item e1 ← take(buffer) q3: consume(e1)

In questo caso è invariante la regola nAvailItems.V = #buffer.

nAvailItems è chiamata **risorsa semaforo**.

Il caso invece, più reale, prevede che il buffer sia in qualche modo limitato, va quindi modellata anche l’interazione per cui un produttore voglia tentare una produzione quando il buffer è pieno.

PRODUCER/CONSUMER WITH BOUNDED BUFFER	
UnboundedQueue<Item> buffer ← empty queue Semaphore nAvailItems ← (0,{}) Semaphore nAvailPlaces ← (N,{})	
Producer	consumer
Loop forever p1: Item e1 ← produce p2: wait(nAvailPlaces) p3: append(buffer,e1) p4: signal(nAvailItems)	Loop forever q1: wait(nAvailItems) q2: Item e1 ← take(buffer) q3: signal(nAvailPlaces) q4: consume(e1)

Invariante: nAvailItems + nAvailPlaces = N.

Una generalizzazione del caso con buffer limitato, richiede di considerare che la struttura dati condivisa venga utilizzata in modo non atomico (il buffer).

PRODUCER/CONSUMER WITH BOUNDED BUFFER	
UnboundedQueue<Item> buffer ← empty queue Semaphore nAvailItems ← (0,{}) Semaphore nAvailPlaces ← (N,{}) Binary semaphore mutex ← (1,{})	
Producer	consumer
Loop forever p1: Item e1 ← produce p2: wait(nAvailPlaces) p3: wait(mutex) p3: append(buffer,e1) p5: signal(mutex) p6: signal(nAvailItems)	Loop forever q1: wait(nAvailItems) q2: wait(mutex) q3: Item e1 ← take(buffer) q4: signal(mutex) q5: signal(nAvailPlaces) q6: consume(e1)

Il problema dei filosofi a cena

Il classico problema dei filosofi a cena, originariamente proposto da Dijkstra nel 1971 riguardo a un problema di sincronizzazione in cui cinque computers dovevano accedere a cinque periferiche di scrittura su nastro condivise contemporaneamente, è oggi un modello utilizzato per confrontare i vari formalismi per la descrizione e verifica di problemi concorrenti.

Il problema, in una delle sue varianti, è descritto come segue:

- cinque filosofi vivono in comunità effettuando due sole azioni: pensare e mangiare
- i pasti sono serviti su un tavolo apparecchiato con 5 piatti e 5 forchette
- al centro del tavolo è presente una ciotola di spaghetti continuamente riempita
- per mangiare gli spaghetti un filosofo ha bisogno di due forchette
- ogni filosofo può utilizzare solo le forchette alla sua sinistra e alla sua destra prendendole una alla volta.

Dining philosophers
Loop forever p1: think p2: <pre-protocol> p3: eat p3: <post-protocol>



Il problema consiste nella progettazione dei pre e post protocol per assicurare:

- che un filosofo mangi solo se ha due forchette
- mutua esclusione: due filosofi non possono raccogliere una stessa forchetta contemporaneamente
- assenza di deadlock
- assenza di starvation
- comportamenti efficienti in assenza di conflitti.

Un primo tentativo di risoluzione può essere approcciato modellando ogni forchetta come un semaforo:

- wait → raccogliere forchetta
- signal → posare la forchetta

Dining philosophers (first attempt)	È possibile dimostrare che una forchetta non è mai in mano a due filosofi contemporaneamente.
semaphore array[0..4] fork←[1,1,1,1,1]	
Loop forever p1: think p2: wait(fork[i]) p3: wait(fork[(i+1)%N]) p4: eat p5: signal(fork[i]) p6: signal(fork[(i+1)%N])	Purtroppo però questa soluzione non esclude la presenza di deadlock: mentre un filosofo raccoglie la sua forchetta di sinistra, un altro potrebbe raccogliere quella alla sua destra, e così via bloccando l’accesso, per tutti, alla seconda forchetta.

Si ricorda che un **deadlock** è una situazione in cui due o più azioni in competizione rimangono in attesa che l’altra o le altre terminino e questo non accade mai. Le condizioni necessarie affinché si verifichi un deadlock sono:

- **condizione di mutua esclusione:** una risorsa che non può essere utilizzata da più di un processo alla volta
- **condizione “hold and wait”:** i processi che già “posseggono” una risorsa possono chiedere accesso ad altre risorse
- **condizione di no preemption:** le risorse non possono essere rimosse da un processo che le sta usando, solo tale processo può rilasciarle a seguito di azioni esplicite
- **condizione di attesa circolare:** due o più processi formano una catena circolare in cui ogni processo attende una risorsa dal processo successivo che la detiene (deadly embrace, con due processi).

I deadlock possono verificarsi nei sistemi in cui tutte le 4 condizioni sono vere.

L’individuazione e ripristino dai deadlock è una metodologia adottata nei database (attraverso le transazioni che, tipicamente, acquisiscono diversi lock finché non effettuano un commit) che permette di identificare l’insieme delle transazioni che si trovano in condizione di deadlock analizzano il grafico delle dipendenze di tipo “is-waiting” (cercando cicli) e selezionare la transazione “vittima” che deve essere eventualmente abortita. Nella JVM non esiste un sistema di individuazione e ripristino automatici dei deadlock.

Dining philosophers (first solution)	Per assicurare liveness è possibile limitare il numero di filosofi che mangiano simultaneamente introducendo N-1 tickets per l’accesso al cibo.
semaphore array[0..4] fork←[1,1,1,1,1] semaphore ticket←(4, {})	
Loop forever p1: think p2: wait(ticket) p3: wait(fork[i]) p4: wait(fork[(i+1)%N]) p5: eat p6: signal(fork[i]) p7: signal(fork[(i+1)%N]) p8: signal(ticket)	Si può dimostrare che questa soluzione soddisfa tutte le proprietà.



Dining philosophers (second solution)	È possibile osservare che non si verificano deadlock se l'ultimo filosofo raccoglie le forchette dal tavolo in sequenza inversa rispetto agli altri, rompendo la "catena di attesa".
semaphore array[0..4] fork←[1,1,1,1,1]	
integer first = min(i,(i+1)%N) integer second = max(i,(i+1)%N)  Loop forever p1: think p2: wait(fork[first]) p3: wait(fork[second]) p4: eat p5: signal(fork[first]) p6: signal(fork[second])	

Filosofo	1° forchetta	2° forchetta
0	0	1
1	1	2
2	2	3
3	3	4
4	0	4

## Il problema dei lettori e degli scrittori

In questo problema i processi sono divisi in due classi:

- lettori (readers): che richiedono che nessun processo di tipo scrittore acceda alla risorsa che stanno utilizzando (ma processi di tipo lettore possono)
- scrittori (writers): che richiedono l'esclusione sia di scrittori che di lettori.

Il problema è un'astrazione di accesso al database (o altri tipi di risorse condivise) in quanto non è pericoloso che diversi processi leggano lo stesso dato ma la scrittura (o modifica) di un'informazione deve essere effettuata in condizione di mutua esclusione allo scopo di garantire consistenza dei dati.

Le soluzioni devono soddisfare le invarianti:

$$nR \geq 0$$

$$nW = 0 \vee nW = 1$$

$$(nR > 0 \rightarrow nW = 0) \wedge (nW = 1 \rightarrow nR = 0)$$

$nR$  = number of readers;  $nW$  = number of writers

READERS AND WRITERS (first attempt)	
binary semaphore rw ← (1,{}) DataBase dbase;	
Reader	Writer
Loop forever p1: wait(rw) p2: Item e1 ← read(dbase) p3: signal(rw)	Loop forever q1: wait(rw) q2: Item e1 ← create_record q3: write(dbase,e1) q4: signal(rw)

Questo primo tentativo è troppo limitativo in quanto anche ogni lettore accede in maniera esclusiva al database.

READERS AND WRITERS (solution)	
binary semaphore mutex $\leftarrow (1, \{\})$ int nr $\leftarrow 0$ binary semaphore rw $\leftarrow (1, \{\})$ DataBase dbase;	
Reader	Writer
Loop forever p1: wait(mutexR) p2: if (nr==0) p3:     wait(rw) p4: nr $\leftarrow$ nr + 1 p6: Item e1 $\leftarrow$ read(dbase) p7: wait(mutexR) p8: nr $\leftarrow$ nr - 1 p9: if (nr==0) p10: signal(rw) p11: signal(mutexR)	Loop forever q1: wait(rw) q2: Item e1 $\leftarrow$ create_record q3: write(dbase,e1) q4: signal(rw)

I lettori non utilizzano lo stesso lock degli scrittori: mutexR è utilizzato dai lettori per aggiornare la struttura dati (nr).

#### Altri problemi

- “Barber shop” problem (Dijkstra)
- “Cigarette smokers” problem (Patil, 1971)
- ...
- “The Santa Claus” problem (Bansal, 2006)

## Monitors

I semafori sono costrutti potenti sebbene agiscano ad un basso livello, motivo per cui sono difficili da utilizzare in programmi concorrenti complessi e determinano fonti di errori di programmazione.

I **monitor** sono costrutti “a più alto livello” introdotti da Brinch Hansen nel 1973 e generalizzati da Hoare nel 1974.

Un **monitor** è un’astrazione dati per la programmazione concorrente che incapsula le policy di sincronizzazione e mutua esclusione per quanto riguarda l’accesso alla loro struttura dati. I semafori includono il loro stato, le loro operazioni e le loro policy di concorrenza. Sono simili a moduli integrati a meccanismi di base per garantire la correttezza nell’accesso concorrente al modulo.

I monitor rappresentano una generalizzazione dei concetti di *kernel* o *supervisor* nei sistemi operativi in cui la sezione critica, così come l’allocazione di memoria sono centralizzati in programmi con specifici privilegi.

I monitor rappresentano anche una generalizzazione della nozioni di *oggetto* nella programmazione orientata agli oggetti (classi che incapsulano dati, operazioni e policy di sincronizzazione o mutex).

Una rappresentazione astratta di un monitor può essere:

```
monitor MonitorName {
    declaration of permanent variables
    initialization statements
    procedures (or entries)
}
```

## Proprietà dei monitor: mutua esclusione

I monitor sono istanze di tipi di dati astratti: sono visibili all'esterno solo i nomi delle operazioni (procedure):

- I monitor hanno delle interfacce
- Le interfacce forniscono l'unico punto di accesso
- Le chiamate a procedure dei monitor sono qualcosa del tipo: `MonitorName.OpName(params)`.

Le istruzioni dichiarate all'interno dei monitor non possono accedere a variabili dichiarate al di fuori del monitor stesso. Le variabili permanenti vengono inizializzate prima di una qualsiasi chiamata a procedura.

Una caratteristica dei monitor è che garantiscono implicitamente che le procedure vengono eseguite, per definizione, con mutua esclusione:

- Una procedura di un monitor è chiamata da un processo esterno
- Una procedura è attiva se qualche processo sta eseguendo un'istruzione nella procedura
- Può essere attiva, al massimo, un'istanza di una procedura di un monitor alla volta
- I processi che trovano il monitor "occupato" vengono sospesi.

## Un esempio semplice: classico contatore

```
monitor Counter {  
  
    int count;  
  
    procedure inc() {  
        count := count + 1;  
    }  
  
    procedure getValue():int {  
        return count;  
    }  
  
}
```

## Osservazioni

La mutua esclusione è implicita e non è richiesto al programmatore di utilizzare altri meccanismi (come wait e signal...):

- se diverse operazioni di uno stesso monitor vengono chiamate da più di un processo, l'implementazione assicura che vengano eseguite in mutua esclusione; ogni operazione è eseguita in maniera atomica rispetto alle altre
- se le operazioni di diversi monitor vengono chiamate, le loro esecuzioni sono intervallate.

Non esiste una coda esplicita associata alle procedure dei monitor, per questo motivo possono verificarsi problemi legati alla starvation.

## Proprietà dei monitor: sincronizzazione

I monitor offrono un supporto esplicito per la sincronizzazione: è previsto l'uso di variabili di condizione utilizzate all'interno del monitor dai programmatori per ritardare un processo che non può procedere la esecuzione in maniera sicura finché lo stato del monitor non soddisfa qualche condizione booleana. Le variabili di condizione sono utilizzate per riattivare un processo sospeso nel momento in cui le condizioni diventano vere.

Le operazioni atomiche di base sono `waitC` e `signalC`, ogni variabile di condizione è associata ad una coda FIFO di processi bloccati.

<code>waitC(cond) =</code> <code>&lt; append p to cond.queue</code> <code>  p.state := blocked</code> <code>  monitor.lock := release &gt;</code>	Sospende l'esecuzione del processo e rilascia il lock al monitor.
<code>signal(cond) =</code> <code>&lt; if cond.queue != empty</code> <code>  q := remove head of cond.queue</code> <code>  q.state := ready &gt;</code>	Sblocca un processo che aspettava il verificarsi di una condizione.

Esempio: `synchronized cell`

```
monitor SynchCell {  
  
  int value;  
  boolean available := false;  
  cond isAvail;  
  
  procedure set(int v) {  
    value := v;  
    available := true;  
    signalC(isAvail);  
  }  
  
  procedure get():int {  
    if (!available)  
      waitC(isAvail);  
    return value;  
  }  
  
}
```

Osservazioni

Esiste un collegamento esplicito tra variabili condizioni e il monitor che le incapsula: le operazioni di `wait` rilasciano il lock del monitor. Questo è essenziale per evitare che un processo che esegue una `waitC` voglia bloccare l'accesso al monitor.

Altre primitive

- `emptyC(cond)`: controlla se la coda è vuota
- `signalAllC(cond)`: agisce come `signal` ma riattiva tutti i processi che erano in attesa della condizione
- `waitC(cond, rank)`: l'attesa è ordinata per valore crescente di `rank`
- `minrank(cond)`: restituisce il valore del `rank` del processo in testa alla coda.

### Esempio: synchronized cell 2 (rivisitazione)

<pre> monitor SynchCell {      int value;     boolean available := false;     cond isAvail;      procedure set(int v) {         value := v;         available := true;         signalAllC(isAvail);     }      procedure get():int {         if (!available)             waitC(isAvail);         return value;     } }         </pre>	<p>Utilizzando signalAllC si riattivano tutti i processi in coda.</p>
---	---

### Esempio: synchronized cell 3 (rivisitazione)

<pre> monitor SynchCell {      int value;     boolean available := false;     cond isAvail;      procedure set(int v) {         value := v;         available := true;         signalC(isAvail);     }      procedure get():int {         if (!available) {             waitC(isAvail);             signal(isAvail);         }         return value;     } }         </pre>	<p>Realizza lo stesso comportamento senza l'uso di signalAllC: segnalando non appena un processo sospeso viene riattivato.</p>
---	--

Implementazione di un semaforo tramite monitor

Di seguito sono riportate due implementazioni di semafori, tramite l'utilizzo di monitor.

<pre>monitor Semaphore {      int s := &lt;InitValue&gt;;     cond notZero;      procedure wait(){         if (s=0) {             waitC(notZero);         }         s := s - 1;     }      procedure signal(){         s := s + 1;         signalC(notZero);     } }</pre>	<pre>monitor Semaphore {      int s := &lt;InitValue&gt;;     cond notZero;      procedure wait(){         if (s=0)             waitC(notZero);         else             s := s - 1;     }      procedure signal(){         if (emptyC(notZero)             s := s + 1;         else             signalC(notZero);     } }</pre>
--	--

Semafori vs Variabili condizioni nei monitor

Semaforo	Monitor
Wait può bloccare o non bloccare	WaitC blocca sempre
Signal ha sempre un effetto	SignalC non ha effetto se la coda è vuota
Signal sblocca in maniera arbitraria uno dei processi bloccati	SignalC sblocca il processo in cima alla coda
Un processo sbloccato da una signal può riprendere la sua esecuzione immediatamente	In base alla specifica semantica di segnalazione, un processo sbloccato tramite signalC deve attendere che il processo che ha eseguito la signal esca dal monitor.

Disciplina per la segnalazione

Quanto un processo esegue una signal, anche se possono esistere diversi processi pronti ad entrare in esecuzione nel monitor, solo un processo può avere accesso esclusivo in funzione della semantica base dei monitor; un unico processo rimane attivo.

Semantiche:

- **signal and continue:** il processo che esegue la signal continua l'esecuzione e il processo che viene attivato dalla signal riprende l'esecuzione in un secondo momento
- **signal and wait:** il processo che esegue la signal si interrompe, lasciando priorità di esecuzione al processo che viene attivato e rientrando in competizione con eventuali altri processi in stato d'attesa
- **signal and urgent wait (immediate resumption required):** rappresenta la soluzione più classicamente adottata nei monitor, funziona come la signal and wait, ma impone al segnalatore una priorità maggiore rispetto agli altri processi in attesa.

Più schematicamente, definiti:

- S = precedenza del processo segnalante
- W = precedenza del processo in attesa
- E = precedenza dei processi bloccati su una procedura

si può osservare:

Signal and continue	$S > W > E$	<pre>while (!B)   wait(cond) &lt;access&gt;</pre>
Signal and wait	$W > S = E$	<pre>if (!B)   wait(cond) &lt;access&gt;</pre>
Signal and urgent wait	$W > S > E$	

Utilizzo dei monitor

I monitor possono essere utilizzati per implementare qualsiasi risorsa o struttura dati da utilizzare in maniera concorrente da diversi processi e nella quale si voglia incapsulare una politica di sincronizzazione.

Il problema dei produttori e dei consumatori con i monitor

<pre>monitor BoundedBuffer {    ElemType buffer := &lt;EmptyBuffer&gt;;   cond notFull, notEmpty;    procedure put(ElemType elem) {     if (buffer is full)       waitC(notFull);     append(buffer,elem);     signalC(notEmpty);   }    procedure take(): ElemType {     if (buffer is empty)       waitC(notEmpty);     ElemType el := head(buffer);     signalC(notFull);     return el;   }  }</pre>	
Producer	Consumer
<pre>loop p1: ElemType el := produce; p2: BoundedBuffer.put(el);</pre>	<pre>loop q1: ElemType el := BoundedBuffer.take(); q2: consume(el);</pre>

È possibile utilizzare un array circolare per superare problemi di overflow.

```
monitor BoundedBuffer {  
  
    ElemType[] elems := new int[MAX_ELEMS];  
    int first := 0, last := 0;  
    cond notFull, notEmpty;  
  
    procedure put(ElemType elem) {  
        if ((last + 1) % MAX_ELEMS) = first)  
            waitC(notFull);  
        elems[last] = elem;  
        last := (last + 1) % MAX_ELEMS;  
        signalC(notEmpty);  
    }  
  
    procedure take(): ElemType {  
        if (first = last)  
            waitC(notEmpty);  
        ElemType elem = elems[first];  
        signalC(notFull);  
        return elem;  
    }  
}
```



Il problema dei lettori e degli scrittori con i monitor

Si presentano due soluzioni: RWLock1 (signal and continue) e RWLock2.

Invariante: ( numero di lettori = 0 o numero di scrittori = 0 ) e ( numero di scrittori  $\leq$  1 ).

<pre> monitor RWLock1 {      int nr, nw := 0;     cond okToRead, okToWrite;      procedure request_read() {         while (nw &gt; 0)             waitC(okToRead);         nr := nr + 1;     }      procedure release_read() {         nr := nr - 1;         if (nr = 0)             signal(okToWrite);     }      Procedure request_write() {         while (nr &gt; 0 or nw &gt; 0)             waitC(okToWrite);         nw := nw + 1;     }      procedure release_write() {         nw := nw - 1;         signalC(okToWrite);         signalC(okToRead);     }  }         </pre>	<pre> monitor RWLock2 {      int nr, nw := 0;     cond okToRead, okToWrite;      procedure request_read() {         if (nw &gt; 0)             waitC(okToRead);         nr := nr + 1;         signal(okToRead);     }      procedure release_read() {         nr := nr - 1;         if (nr = 0)             signal(okToWrite);     }      Procedure request_write() {         If (nr &gt; 0 or nw &gt; 0)             waitC(okToWrite);         nw := nw + 1;     }      procedure release_write() {         nw := nw - 1;         if (emptyC(okToRead))             signalC(okToWrite);         else             signalC(okToRead);     }  }         </pre>
<b>Readers</b>	<b>Writers</b>
<p>p1: RWLock.request_read                  p2: read the database                  p3: RWLock.release_read</p>	<p>q1: RWLock.request_write                  q2: write the database                  q3: RWLock.release_write</p>

Il problema della allocazione e gestione delle risorse con i monitor

I monitor sono utilizzati tipicamente per funzionare come allocatori di risorse, garantendo determinate politiche nell'allocazione delle risorse quando i processi ne fanno richiesta.

```
monitor ResourceAllocator {  
  
    procedure request(<Params>) {  
        < block the request until the resource or a resource is available,  
            according to some policy >  
    }  
  
    procedure release(<Params>) {  
        < possibly unblock some pending request >  
    }  
  
}
```

Esempio: shortest job next scheduling.

Invariante: variabile condizione turn ordinate per tempo e ( free => turn è vuota).

```
monitor ShortJobFirstAllocator {  
  
    bool free = true;  
    cond turn;  
  
    procedure request(int time) {  
        if (free)  
            free := false;  
        else  
            waitC(turn,time);  
    }  
  
    procedure release() {  
        if (emptyC(turn))  
            free := true;  
        else  
            signal(turn);  
    }  
  
}
```

Implementazioni di monitor tramite semafori

I monitor possono essere realizzati utilizzando i semafori, in particolare:

- un semaforo mutex per garantire la mutua esclusione
- per ogni variabile condizione, un semaforo condsem e un contatore condcount che tengano traccia del numero di processi sospesi sulla variabile.

Semantica Signal and continue	Semantica Signal and wait
<pre>procedure waitC(cond) {   condcount++;   signal(mutex);   wait(condsem);   wait(mutex); }  procedure signal(cond) {   if (condcount &gt; 0) {     condcount--;     signal(condsem);   } }</pre>	<pre>procedure waitC(cond) {   condcount++;   signal(mutex);   wait(condsem); }  procedure signal(cond) {   if (condcount &gt; 0) {     condcount--;     signal(condsem);     wait(mutex);   } }</pre>

Altri componenti di coordinazione possono essere realizzati utilizzando i monitor:

- latches
- barriers
- rendez-vous
- message boxes
- blackboards
- event services
- ...

Visual formalisms (module 1.4)

I formalismi permettono di descrivere modelli rigorosi di sistemi concorrenti tramite determinate forme di diagrammi visuali che rappresentano le dinamiche del sistema, la struttura e la dipendenza dei task.

I formalismi visuali sono molto utili nelle fasi di specifica dei requisiti, analisi e progettazione.

Di seguito sono descritti tre tipi di formalismi:

- Le Reti di Petri
- I diagrammi di stato
- I diagrammi di attività

Reti di Petri

Le Reti di Petri sono un modello formale astratto per rappresentare il flusso di informazioni: descrivono e analizzano il flusso delle informazioni e il controllo in un sistema.

Una Rete di Petri è in grado di modellare e descrivere le proprietà statiche di un sistema descrivendone le regole che permettono transizioni di stati e le proprietà dinamiche attraverso la simulazione dei comportamenti (cambiamenti di stato) di un sistema utilizzando un sistema di tokens e regole.

Da Wikipedia, arricchito:

**Una rete di Petri PT (Place/Transition)** è un grafo orientato con due tipi di nodi, **posti (places)** e **transizioni (transitions)**, connessi da **archi** diretti. I posti sono rappresentati graficamente da cerchi e le transizioni da rettangoli.

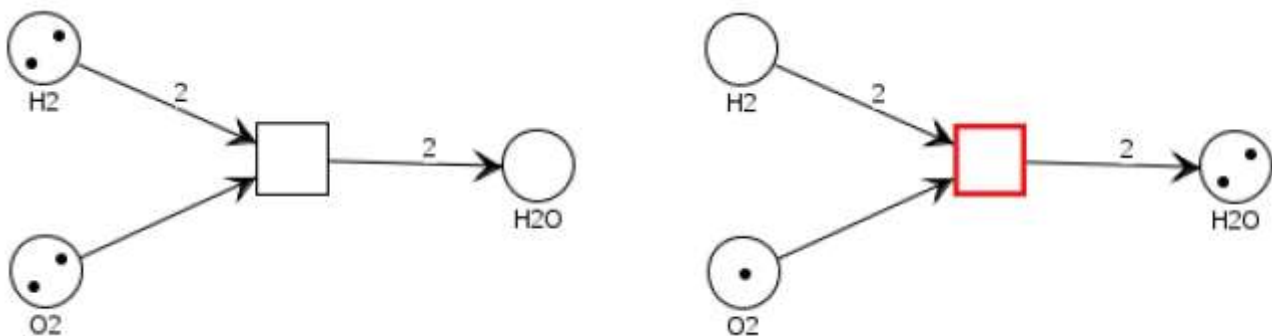
Un arco può unire solamente nodi di tipo diverso, quindi possono esserci archi tra posti e transizioni - ma non tra posti e posti o transizioni e transizioni. Un posto da cui un arco parte per finire in una transizione è detto posto di input della transizione; un posto in cui un arco arriva partendo da una transizione è detto posto di output della transizione. Una transizione senza posto di input è detta **source** e semplicemente produce dei tokens; una transizione senza posto di output è detta **sink** e consuma tokens.

I posti possono contenere un certo numero di **token** o **marche**. Una distribuzione di token sull'insieme dei posti della rete è detta **marcatura (marking)**. Le transizioni agiscono sui token in ingresso secondo una regola, detta **regola di scatto (firing)**. Una transizione è abilitata se può scattare, cioè se ci sono token in ogni posto di input. Gli archi possono essere **"pesati"**, il peso di un arco che esce da un posto indica il numero di tokens necessari per poter abilitare la transizione; il peso di un arco che arriva in un posto indica il numero di tokens prodotti dalla transizione nel posto di arrivo. Quando una transizione scatta, essa consuma token dai suoi posti di input, esegue dei task e posiziona un numero specificato di token in ognuno dei suoi posti di uscita. Ciò avviene automaticamente, ad esempio in un singolo step non-prelazionabile. L'esecuzione delle reti di Petri è non deterministica. Ciò significa due cose:

- se più transizioni sono abilitate nello stesso momento, una qualsiasi di esse può scattare
- non è garantito che una transizione abilitata scatti; una transizione abilitata può scattare immediatamente, dopo un tempo di attesa qualsiasi (a patto che resti abilitata), o non scattare affatto.

Poiché lo scatto di una transizione non è predicibile a priori, le reti di Petri sono molto adatte a modellare il comportamento di un sistema concorrente.

Esempio: reazione  $2\text{H}_2 + \text{O}_2 = 2\text{H}_2\text{O}$



## Modellazione con Reti di Petri

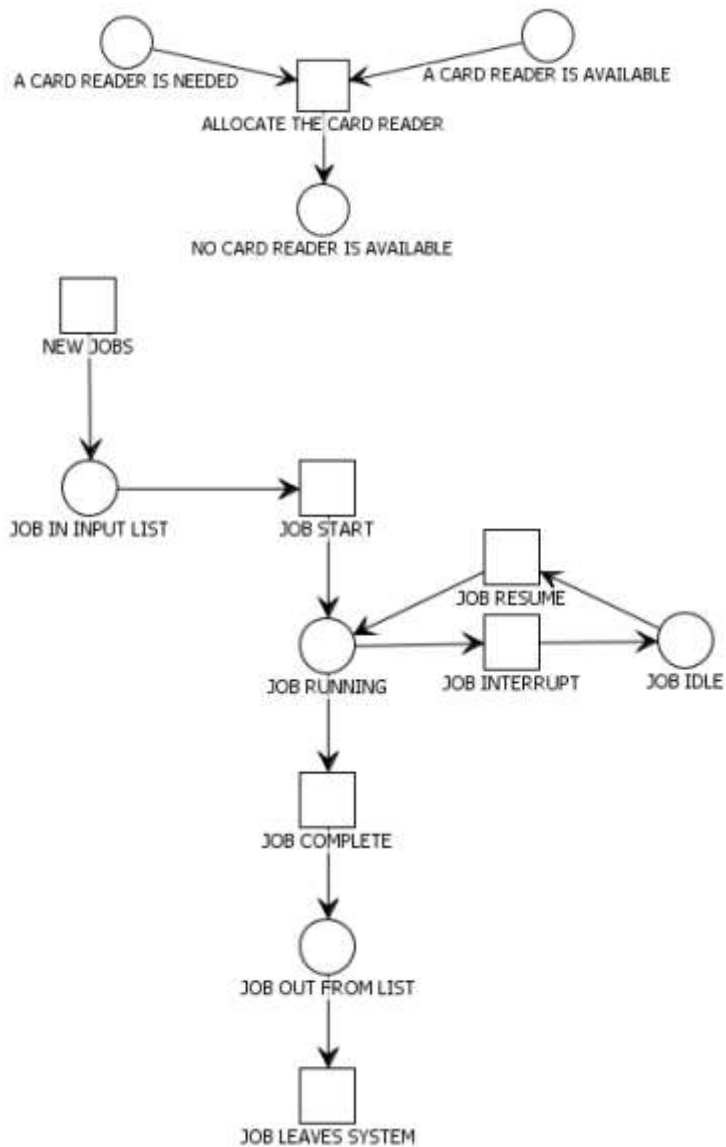
Le Reti di Petri possono essere utilizzate per modellare sistemi concorrenti in termini di:

- eventi e condizioni
- relazioni tra essi

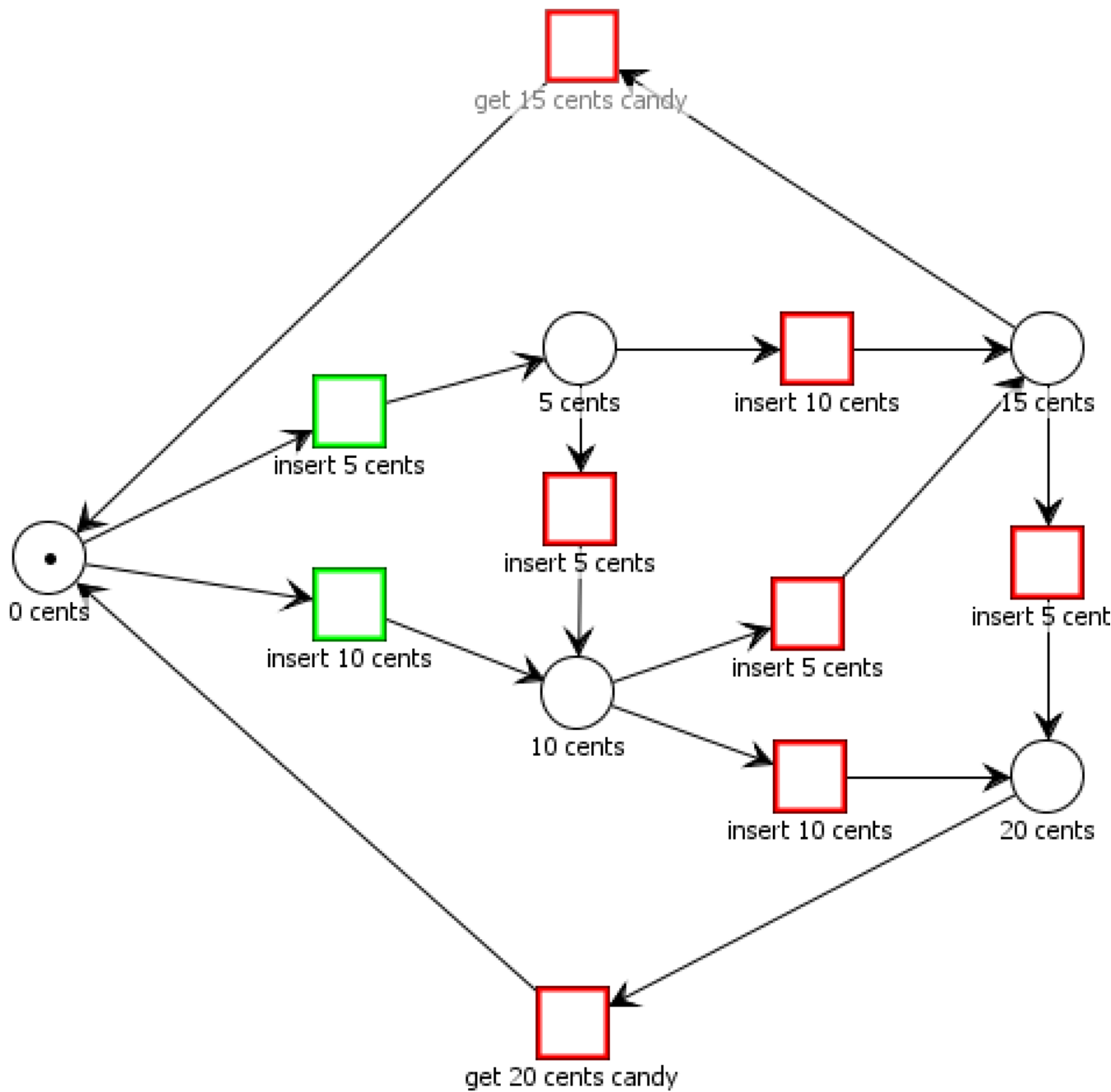
in quanto:

- in un sistema, in un determinato momento, certe condizioni sono rispettate
- il fatto che queste condizioni siano rispettate può generare l'occorrenza di determinati eventi
- l'occorrenza degli eventi può cambiare lo stato del sistema
  - determinando che alcune precedenti condizioni non siano più valide ma se ne verifichino di nuove
- l'esecuzione di una transizione equivale all'occorrenza di un evento (considerato istantaneo o, meglio, come un cambiamento atomico del sistema).

## Esempi: allocazione delle risorse



Esempio: macchina distributrice



Interpretazione

Posti e transizioni possono essere interpretate, classicamente, come da seguente tabella.

Interpretazione	Posto di origine	Transizione	Poso di destinazione
1	Pre-condizione	Evento	Post-condizione
2	Dato in input	Passo computazionale	Dato in output
3	Segnale in input	Processore di segnale	Segnale in output
4	Risorsa necessaria	Azione (task, job)	Risorsa rilasciata
5	Condizione	Clausola logica	Conclusione
6	Buffers	Processore	Buffers

Modellazione concorrenza e parallelismo

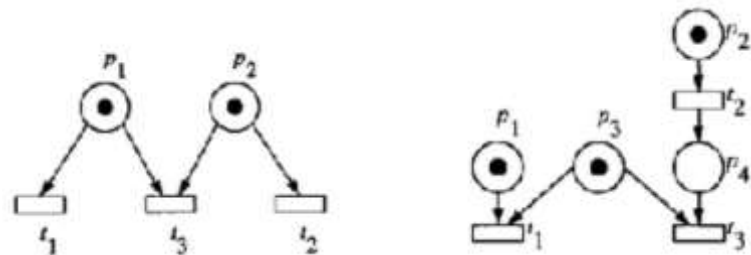
Le reti di Petri sono uno strumento ideale per modellare i sistemi distribuiti con multipli processi che operano in maniera concorrente.

Esempi:

Modellazione di due attività parallele	
Modellazione di due attività parallele con un punto di sincronizzazione	
Modellazione del calcolo $x = \frac{a+b}{a \cdot b}$	

Modellazione dei conflitti e di eventi concorrenti

Due eventi si dicono in conflitto se uno o l'altro possono verificarsi ma non entrambi.  
Due eventi sono concorrenti se entrambi possono verificarsi in qualsiasi ordine senza generare conflitti.  
Situazioni nelle quali conflitti e concorrenza appaiono in qualche modo mixate, sono dette *confusion*.



Asincronia e località

In una rete di Petri non è presente una misura di tempo o del flusso del tempo: l'unica proprietà importante del tempo, da un punto di vista logico, riguarda la definizione di un **ordinamento parziale** delle occorrenze degli eventi; possono esistere eventi che non sono vincolati ad un determinato ordinamento.

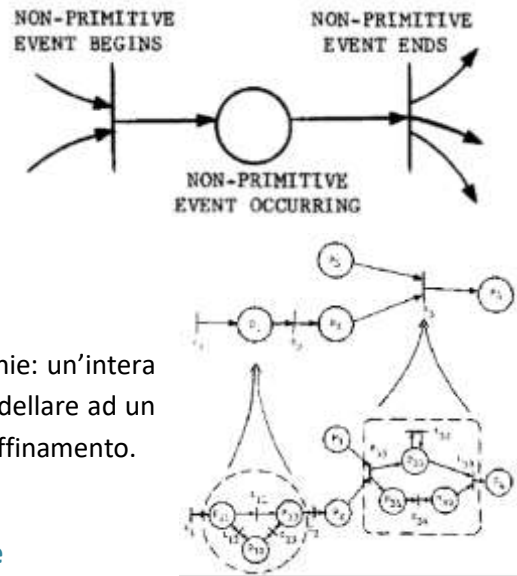
In un sistema complesso composto da sottoparti indipendenti che operano in maniera asincrona, ogni parte può essere modellata tramite una rete di Petri. L'abilitazione e l'esecuzione delle transizioni hanno effetto solo grazie a cambiamenti locali dei making nella rete di Petri (**località**).

Non determinismo

Una rete di Petri modella una sequenza di eventi (discreti) dei quali l'ordine di esecuzione è uno dei tanti possibili permessi dalla struttura di base. Se, in un determinato istante, più di una transizione può essere attiva, la scelta di quale transizione attivare è presa in maniera non deterministica in maniera casuale o seguendo regole o forze che non sono modellate.

Eventi atomici

Una transizione avviene quando si verifica un evento. Un evento è un'attività atomica senza una durata. Se è necessario modellare eventi non primitivi (con una durata), si devono utilizzare eventi multipli, ad esempio definendo eventi di inizio ed eventi di fine che racchiudono stati da esplicitare.



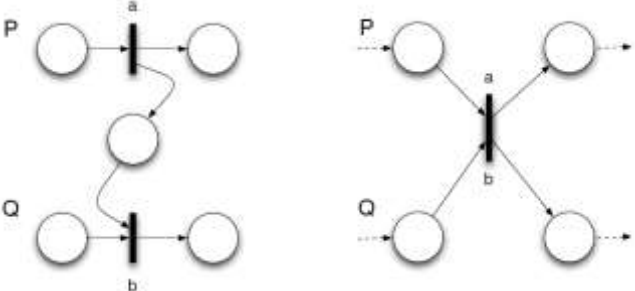
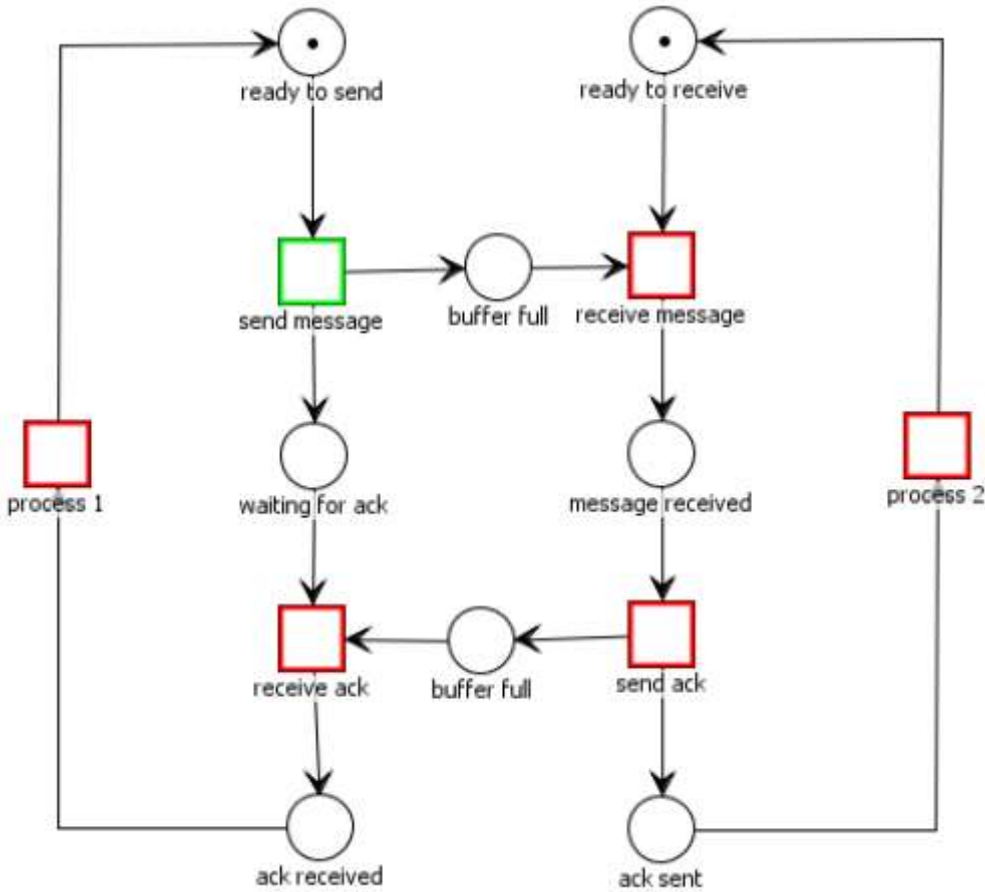
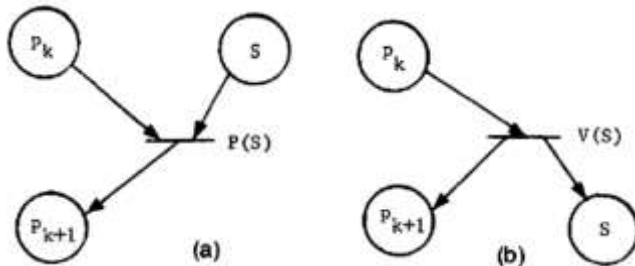
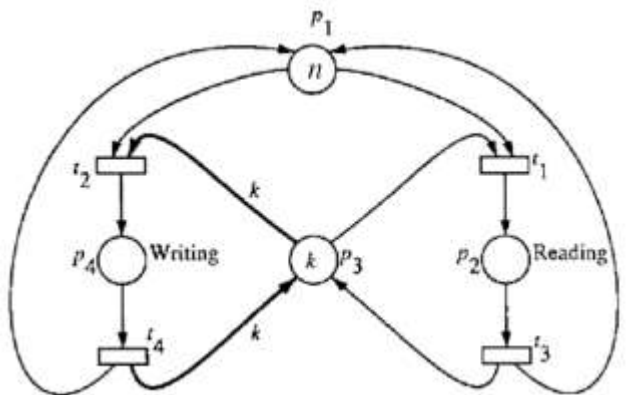
Gerarchie

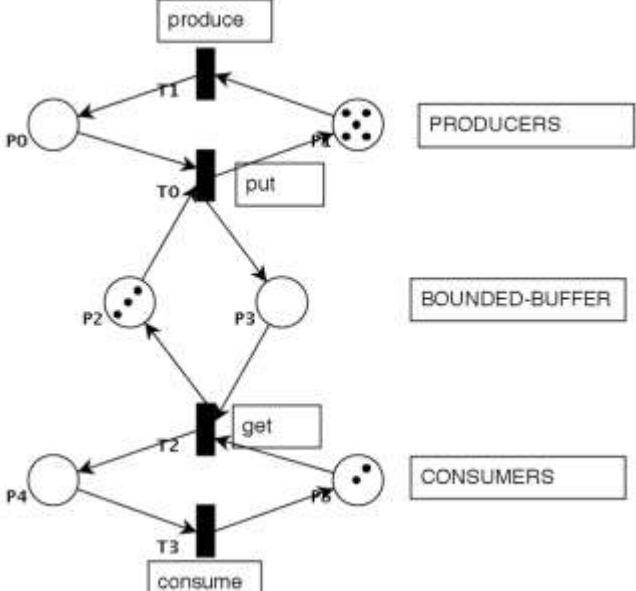
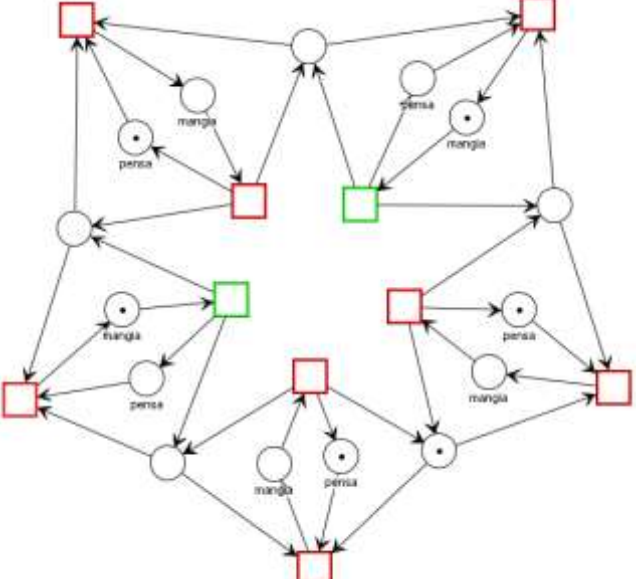
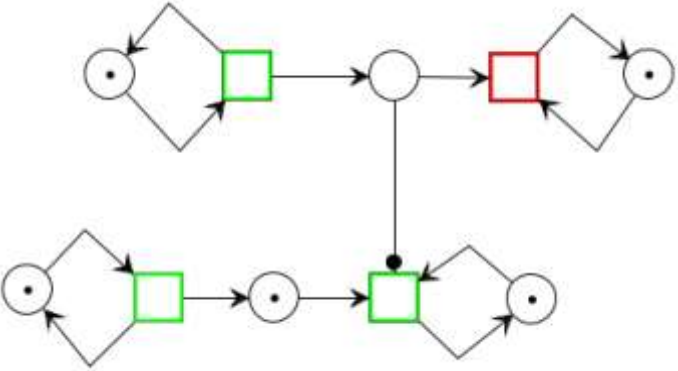
Le reti di Petri offrono un supporto naturale per modellare le gerarchie: un'intera rete può essere sostituita da una singola piazza o transizione per modellare ad un livello di astrazione più alto e viceversa per effettuare un'attività di raffinamento.

Applicazioni alla progettazione e programmazione concorrente

<p><b>Problema della sezione critica e mutua esclusione:</b> p2 e p4 rappresentano le sezioni critiche; s è il token necessario per entrare in una sezione critica.</p>	<p>The diagram shows a Petri net for mutual exclusion. It has four places: <math>p_1</math> (top left), <math>p_2</math> (bottom left), <math>p_3</math> (top right), and <math>p_4</math> (bottom right). There are two green squares representing critical sections: one between <math>p_1</math> and <math>p_2</math>, and another between <math>p_3</math> and <math>p_4</math>. There are two red squares representing entry/exit points: one between <math>p_2</math> and <math>p_1</math>, and another between <math>p_4</math> and <math>p_3</math>. A central place <math>s</math> (the semaphore) has two outgoing transitions: one to the green square between <math>p_1</math> and <math>p_2</math>, and another to the green square between <math>p_3</math> and <math>p_4</math>. Transitions also connect <math>p_1</math> to <math>p_2</math> and <math>p_3</math> to <math>p_4</math>.</p>
---	---



<div><b>Sincronizzazione</b> L'azione b del processo Q può essere eseguita solo dopo l'azione a del processo P.  Nel secondo caso, le azioni a e b dei due processi devono essere eseguite simultaneamente.</div>	<div></div>
<div><b>Comunicazione</b></div>	<div></div>
<div><b>Semafori</b> Modellati come risorse condivise:<ul style="list-style-type: none"><li>wait (P) è modellata come una transizione con il semaforo risorsa come una piazza di input</li><li>signal (Q) è modellata come una transizione con il semaforo risorsa come piazza di output</li></ul></div>	<div></div>
<div><b>Readers and writers</b> Il tokens n rappresenta n processi che possono voler leggere o scrivere una memoria condivisa rappresentata da p3.</div>	<div></div>

<p><b>Producers and consumers</b></p>	
<p><b>Dining philosophers</b></p>	
<p><b>Estensione delle reti di Petri con archi inibitori</b></p> <p>Un arco inibitore, rappresentato da un arco che termina con un piccolo cerchio, abilita la transizione solo nel caso in cui la piazza di origine non contenga tokens.</p> <p>Le reti di Petri con archi inibitori permettono di ottenere una Turing-equivalenza in termini di espressività ed indecidibilità dei problemi.</p>	

Una rete di Petri può essere descritta formalmente allo scopo di permettere analisi rigorose sulle proprietà dei problemi con esse modellati:

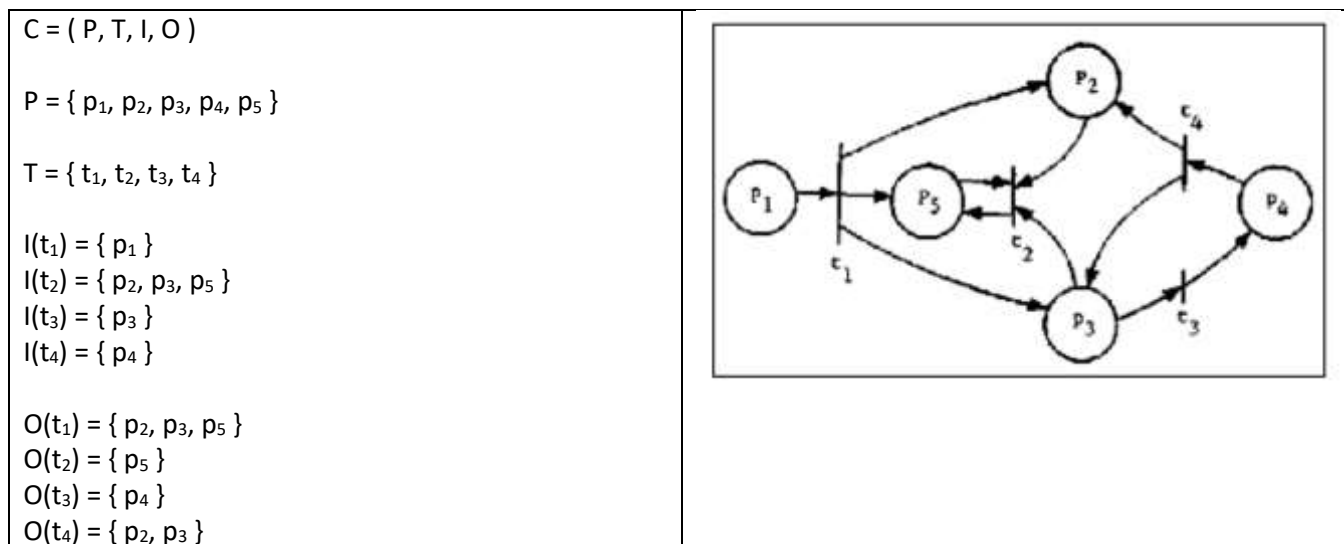
- proprietà **strutturali**: indipendenti dalle posizioni iniziali dei tokens (marking iniziale)
- proprietà **comportamentali**: dipendenti, invece, dai marking.

Una rete di Petri è formalmente definita come una tupla  $C = (P, T, I, O)$  con:

- $P$  è un insieme di piazze

- $T$  è un insieme di transizioni
- La funzione  $I$  definisce l'insieme delle piazze di input per ogni transizione  $t_i$
- La funzione  $O$  definisce l'insieme degli output per ogni transizione  $t_i$ .

Esempio.



Un **marking** è un'assegnazione dei tokens alle piazze nella rete. Può essere formalmente rappresentato:

- come un vettore di  $N$  elementi, uno per ogni piazza, rappresentante il numero di tokens per ogni piazza
- come una funzione  $\mu : P \rightarrow N$  dove  $\mu(p_i)$  è il numero di tokens nella piazza  $p_i$ .

Una rete di Petri con marking è quindi rappresentata dalla tupla  $M = (P, T, I, O, \mu)$ .

#### Regole semantiche di esecuzione

- Lo stato di una rete di Petri è definito dal suo marking, l'esecuzione di una transizione determina un cambiamento di stato nella rete.
- La funzione di "prossimo stato"  $\delta(\mu, t_i)$  è indefinita se non ci sono transizioni abilitate dal marking attuale; se  $t_i$  è abilitata,  $\mu' - \delta(\mu, t_i)$  è il marking risultante dalla rimozione di tokens dall'input di  $t_i$  e dall'aggiunta di tokens all'output di  $t_i$ .
- Data una rete di Petri e un marking iniziale, è possibile *eseguire* la rete di Petri attraverso l'esecuzione di transizioni successive:
  - L'esecuzione di una transizione  $t_i$  dal marking iniziale produce un nuovo marking  $\mu^1 = \delta(\mu^0, t_i)$
  - Da questo nuovo marking è possibile eseguire una qualsiasi transizione  $t_k$  tra quelle abilitate, producendo un nuovo marking  $\mu^2 = \delta(\mu^1, t_k)$
  - È possibile continuare ad eseguire transizione finché esiste almeno una transizione abilitata
  - Se si raggiunge un marking per cui non esistono transizioni abilitate, l'esecuzione viene interrotta.
- L'esecuzione di una rete di Petri può generare molteplici sequenze di marking e transizioni relative in maniera non deterministica.

Una rete di Petri può anche essere rappresentata, quando utile, tramite l'utilizzo di matrici:

- Una matrice indicente  $D$ 
  - Si tratta di una matrice ( $m \times n$ ) con  $m$  transizioni (righe) e  $n$  piazze (colonne) ottenuta come differenza di due matrici:
    - Una matrice  $D^-$  di input in cui l'elemento  $D^-[i,j]$  contiene il peso dell'arco che connette la piazza  $p_j$  con la transizione  $t_i$

- Una matrice  $D^+$  di output in cui l'elemento  $D^+[i,j]$  contiene il peso dell'arco che connette la transizione  $t_i$  con la piazza  $p_j$
- Una matrice di transizione  $T$ 
  - Si tratta di una matrice  $(1 \times m)$  che rappresenta l'esecuzione della rete
- Una matrice  $M$  che rappresenta il marking attuale
  - Si tratta di una matrice  $(1 \times n)$  che rappresenta l'attuale numero di tokens presenti in ogni piazza.

La computazione dell'evoluzione della rete è definita da:  $M' = T \times D + M$ .

Analisi di modelli tramite reti di Petri

Una rete è **k-limitata** se il numero di tokens che possono trovarsi in ogni piazza è limitato a  $k$ .

Una rete **safe** è una rete di Petri 1-limitata.

Una rete è **conservativa** se il numero di tokens nella rete è sempre lo stesso.

Liveness

Basandosi sull'analisi delle transizioni è possibile definire:

- **Transizioni morte** se non esistono sequenze di esecuzioni di transizioni che possono abilitarle, fanno riferimento a situazioni di deadlock
- **Transizioni potenzialmente eseguibili** se esistono sequenze di esecuzioni di transizioni che possono abilitarle, fanno riferimento a situazioni di starvation
- **Transizioni vive**, che sono potenzialmente eseguibili in qualsiasi marking raggiungibile.

Per garantire la proprietà di liveness è importante non solo che una transizione sia eseguibile in un dato marking, ma anche che rimanga potenzialmente eseguibile in tutti i raggiungibili marking successivi, se non fosse vero si potrebbero raggiungere stati con transizioni morte e, quindi, deadlock.

Estensioni

Ulteriori estensioni delle reti di Petri riguardano:

- Reti timed
  - Introducono un tempo di ritardo associato ad ogni transizione o ad ogni piazza, sono utili per valutare problemi di performance o di schedulazione
  - I delay sono definiti in maniera deterministica
  - Reti **stocastiche** hanno delay definiti in maniera probabilistica
- Reti di alto livello
  - Associano qualche tipo di informazioni simbolica/numerica ai tokens e alcune regole computazionali alle transizioni che consumano e producono tokens (reti di Petri colorate che associano un colore ad ogni token, reti con transizioni predicati, ...).

Strumenti per le reti di Petri

<http://www.informatik.uni-hamburg.de/TGI/PetriNets/>

<http://pipe2.sourceforge.net/>

**Diagrammi di stato (TO DO)**

**Diagrammi di attività (TO DO)**

## Multithreaded Programming in Java: Introduction (module lab-1.1)

Java è stato il primo linguaggio di programmazione ampiamente diffuso a fornire un supporto nativo per la programmazione concorrente, utilizza un approccio conservativo per cui tutto è un oggetto ma aggiunge meccanismi per la concorrenza. L'estensione alla concorrenza è stata implementata con la libreria **java.util.concurrent** (JSR 166 e JSR 236), essa fornisce un supporto ad alto livello realizzato tramite semafori, locks, synchronizers, ... e frameworks per i task.

### Meccanismi

La classe **Thread** (assieme ad alcune classi di utility collegate) è utilizzata per inizializzare e controllare le attività concorrenti, l'interfaccia di riferimento è **Runnable**.

Le parole chiave **synchronized** e **volatile** vengono utilizzate per controllare l'esecuzione del codice negli oggetti che possono partecipare nell'esecuzione concorrente (per gestire la mutua esclusione).

I metodi **wait**, **notify** e **notifyAll** sono definiti in **java.lang.Object** e vengono utilizzati per coordinare le attività tra i thread per una reale sincronizzazione.

### Definire i thread

Java fornisce un API di base per definire i diversi tipi di thread, per crearli e gestirne l'esecuzione in maniera dinamica:

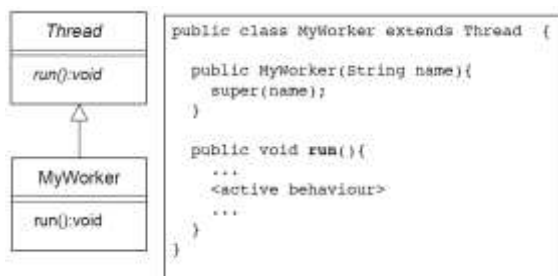
- i threads sono mappati su threads del sistema operativo con strategie che dipendono dallo specifico sistema
- tipicamente si adotta un approccio uno-a-uno.

Un thread è rappresentato tramite la classe astratta **Thread**, caratterizzata dal metodo astratto **run** che definisce il comportamento del thread: un oggetto concreto thread può essere definito estendendo la classe **Thread** ed implementando il metodo **run**.

Per avviare l'esecuzione asincrona di un thread si utilizza il metodo **start**:

- deve essere invocato sull'istanza di un oggetto thread
- termina immediatamente lanciando una nuova attività che esegue il codice definito nel metodo **run**.

Il thread termina non appena viene eseguito il metodo **run**.



### Principali elementi dell'API Thread

<b>Thread (string name)</b>	Per costruire un thread con uno specifico nome
<b>getName()</b>	Restituisce il nome del thread
<b>void sleep(ms)</b>	Sospende l'esecuzione del thread per ms millisecondi
<b>void join()</b>	Attende la terminazione del thread
<b>void interrupt()</b>	Genera un'attesa, aspetta una join o termina con un <b>InterruptedException</b> che può essere gestita dall'applicazione
<b>static Thread currentThread()</b>	Restituisce il riferimento al thread corrente in esecuzione

Generare threads

<pre>public class Test {     public static void main(String[] args) {         Thread myWorkerA = new MyWorker("worker-A");         myWorkerA.start();         Thread myWorkerB = new MyWorker("worker-B");         myWorkerB.start();     } }</pre>	<p>Note</p> <p>Il metodo che viene chiamato dell'oggetto thread è <b>start</b> non <b>run</b>, in quanto l'esecuzione di <b>run</b> sarebbe una chiamata a metodo eseguita in maniera sincrona.</p> <p>Un'applicazione Java ha sempre almeno un thread in esecuzione.</p>
---	---

Sincronizzare i threads

<p>Il metodo <b>join</b> permette di sincronizzare l'esecuzione di un thread con la terminazione di un altro thread.</p> <p>Ad esempio: <code>t.join()</code> sospende l'esecuzione del thread corrente in attesa che il thread <code>t</code> completi la sua esecuzione.</p>	<pre>MyThread t = new MyThread(); t.start(); ... t.join(); System.out.println("spawned thread terminated.");</pre>
--	--

Monitorare i threads

JConsole

<http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>

JConsole è uno strumento grafico di monitoraggio e gestione rilasciato in J2SE JDK 5.0.

Fornisce informazioni sulle performance e sull'utilizzo delle risorse delle applicazioni in esecuzione sulla piattaforma Java. È basato su tecnologia Java Management Extension (JMX).

JConsole è molto utile anche per monitorare i thread generati da un programma Java inclusi i threads della JVM (ad esempio quelli che gestiscono la garbage collection).

VisualVM

<http://visualvm.java.net>

VisualVM è un profiler, simile a JConsole, che permette di misurare e visualizzare le performance di programmi Java.

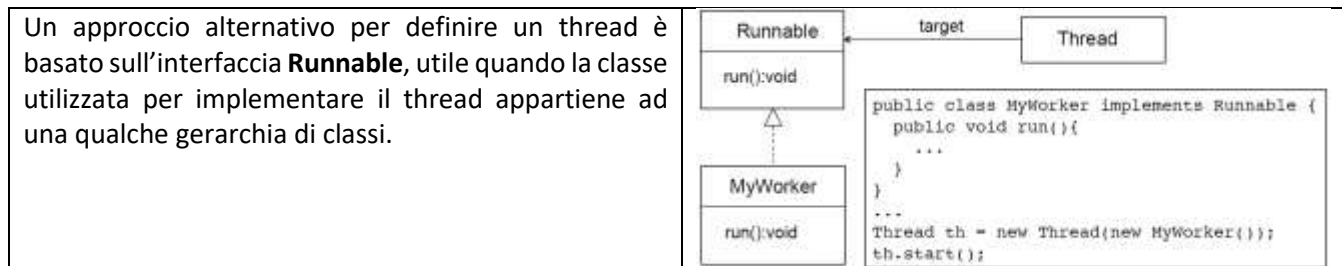
Come JConsole utilizza le strumentazioni di Java per fornire informazioni sulle performance e sull'utilizzo delle risorse, è basato sulla stessa tecnologia ed è stato rilasciato in J2SE JDK 5.0.

Permette un monitoraggio più fine rispetto a JConsole, ad esempio per quanto riguarda il monitoraggio della percentuale di CPU utilizzata dai threads e dai metodi, il tempo di esecuzione o blocco di un thread, ...

Metodi deprecati

Tutti i metodi pubblici che agiscono in maniera asincrona sul flusso di controllo del thread sono stati deprecati (`stop`, `suspend`, `resume`, `destroy`, ...) in quanto gli stessi obiettivi possono (e devono) essere raggiunti attraverso appropriati pattern di sviluppo.

## Interfaccia Runnable



## Sincronizzazione

### Implicita

Applicando la parola chiave **synchronized** come qualificatore ad un qualsiasi blocco di codice in un qualsiasi metodo, si garantisce che solo un thread alla volta potrà accedere al blocco di codice, allo scopo di prevenire un interleaving arbitrario sulle azioni nel corpo del metodo e prevenire interazioni inattese tra i thread che accedono allo stesso oggetto.

Se ne suggerisce l'utilizzo in oggetti *passivi* che sono condivisi e acceduti concorrentemente (in scrittura) da diversi threads.

### Esplicita

Per gestire la sincronizzazione esplicita tra thread viene utilizzato un insieme di meccanismi che agiscono tramite oggetti condivisi.

Il metodo **wait**: ogni metodo sincronizzato in ogni oggetto può contenere una wait che sospende l'esecuzione del thread.

Il metodo **notifyAll**: tutti i threads sospesi possono essere ripristinati grazie all'invocazione del metodo notifyAll sull'oggetto target. Il metodo notifyAll stesso deve essere incluso in un metodo o blocco **synchronized**.

Il metodo **notify**: uno dei thread sospesi viene scelto arbitrariamente e riattivato con l'applicazione del metodo notify. Anche il metodo notify deve essere contenuto in un blocco o metodo **synchronized**.

## Disciplina di programmazione

I threads devono essere pensati come oggetti che incapsulano il loro stato, il loro comportamento ed il controllo del loro comportamento: i metodi dell'oggetto dovrebbero essere chiamati solo dal thread che rappresenta l'oggetto e l'uso di metodi pubblici dovrebbe essere ridotto al minimo.

L'interazione deve essere promossa attraverso l'utilizzo di oggetti passivi condivisi evitando chiamate a metodi pubblici che violerebbero l'incapsulamento del loro controllo.

È necessario separare nettamente le entità attive da quelle passive:

- le entità attive sono agenti responsabili di compiere alcune attività
- le entità passive sono gli oggetti condivisi e manipolati da qualche agente allo scopo di eseguire un qualche compito.

## Attività di laboratorio

Il codice e i sorgenti degli esercizi e degli assignments sono disponibili su BitBucket utilizzando git.

Per clonare il repository:

```
git clone https://aricci303@bitbucket.org/aricci303/pcd1718.git
```

Per aggiornarlo:

```
git pull
```



Nota: i comandi devono essere eseguiti dalla directory clonata pcd1718.

“Hello, concurrent world!”

Ri-scrivere il primo programma concorrente in Java (module-lab-1.1).

- one thread, two threads - the main thread
- running the example, in Eclipse and from the shell
- start vs. run methods → run passa il flusso di controllo all’oggetto Thread, start genera un nuovo flusso di controllo

First look to concurrent program debugging - Using Eclipse JDT

- main thread, dynamic threads, stacks
- asynchronous execution (with 2 threads) - how to deal with it?

First Look at Concurrent Programs

Non-determinism – Test00 program.

- first basic discipline about structuring multi-threaded concurrent programs
- focus on interleaving of worker behavior

Non-Terminating Processes – Test01 program.

- focus on cyclic behavior of workers
- monitoring concurrent programs: JConsole, VisualVM

CPU utilization & CPU bound behaviours - Test02 program

- focus on how thread behaviour can affect CPU usage & reactivity
- profiling concurrent programs: JConsole, VisualVM

How many threads? - Bouncing-Ball demo

- focus on the scalability problem, physical vs. logical concurrency

First Look at Synchronization: join

Joining threads - Test03 program

First utilities

- how many processors? Method availableProcessors() in Runtime - Test04 program
- Measuring time. Method currentTimeMillis() and nanoTime() in System - Test04 program

JAVA è il primo linguaggio di programmazione mainstream ad aver fornito supporto nativo per la programmazione concorrente, l’approccio conservativo (tutto è un oggetto) è stato esteso anche ai meccanismi per la concorrenza con la **java.util.concurrent library** (JSR 166 e JSR 236) per fornire un supporto alla programmazione concorrente id alto livello (semafori, locks, synchronizers, ..., task frameworks).

Thread	Utilizzato per inizializzare e controllare le attività concorrenti
Runnable Interface	Interfaccia implementata da Thread
Synchronized Volatile	Keyword utilizzate per controllare l’esecuzione del codice in oggetti che possono essere eseguiti da thread multipli (es. per gestire la mutua esclusione)
wait notify notifyAll	Metodi definiti in <b>java.lang.Object</b> utilizzati per coordinare le attività tra i thread, per la vera e propria sincronizzazione

Java fornisce una API di base per definire nuovi tipi di thread e per creare e gestire (almeno parzialmente) l’esecuzione dei thread:



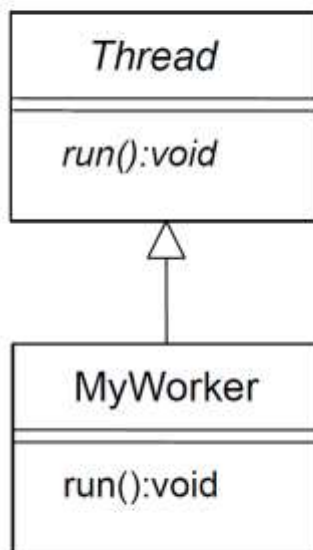
- I thread sono mappati all'interno dei thread del sistema operativo che dipendono dallo specifico sistema
- L'approccio tipicamente utilizzato è uno-a-uno.

Ogni thread, rappresentato dalla classe astratta Thread, deve estendere la classe Thread e implementare il metodo **run** che definisce il comportamento del thread.

Per eseguire un thread in maniera asincrona è necessario utilizzare il metodo **start**:

- Deve essere invocato sull'istanza dell'oggetto thread
- Esegue immediatamente la return e mette in esecuzione ciò che è specificato nel metodo **run**.
- Il thread termina non appena termina l'esecuzione del metodo **run**.

- Thread class is provided in the package `java.lang`



```
public class MyWorker extends Thread {

    public MyWorker(String name){
        super(name);
    }

    public void run(){
        ...
        <active behaviour>
        ...
    }
}
```

Le principali features della classe Thread:

- Thread (String name): per creare un thread con uno specifico nome
- String getName(): per ottenere il nome assegnato al thread
- void sleep(long ms): per sospendere l'esecuzione per ms millisecond
- void join(): attende la terminazione del thread
- void interrupt(): determina una terminazione di sleep, wait o join con un InterruptedException
- static Thread currentThread(): per ottenere il riferimento al thread in esecuzione

Esempio

```
Thread myWorkerA = new MyWorker("worker-A");
myWorkerA.start();

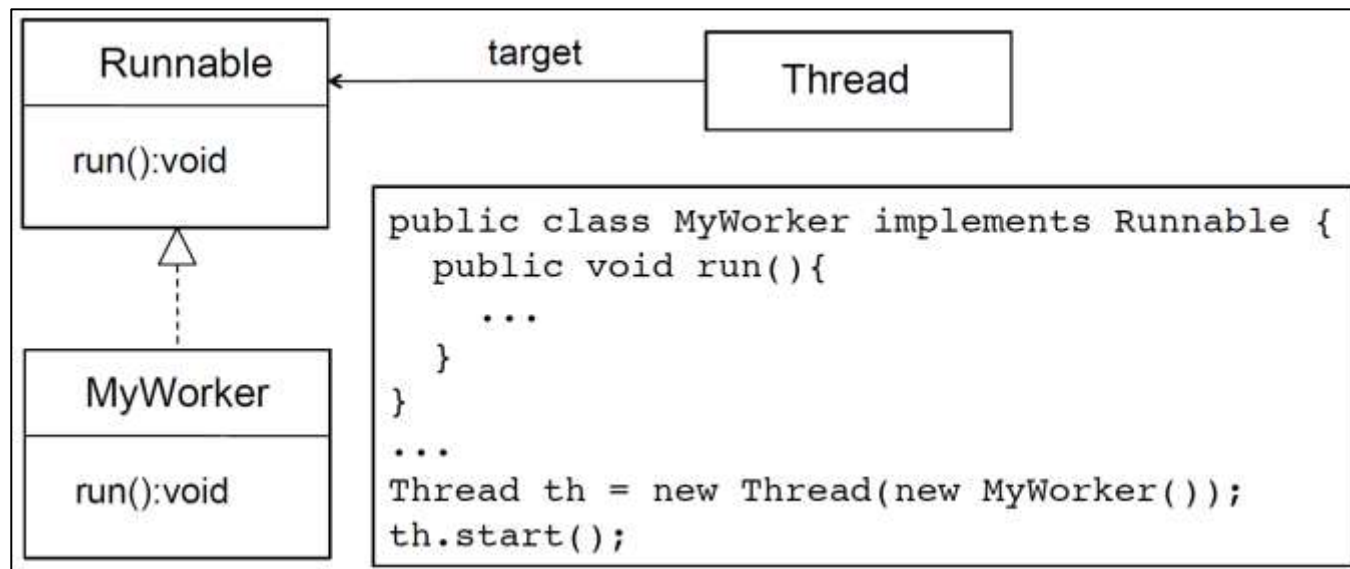
Thread myWorkerB = new MyWorker("worker-B");
myWorkerB.start();

...

myWorkerA.join();
myWorkerB.join();
```

```
System.out.println("myWorker A and myWorker B threads terminated");
```

È anche possibile sviluppare un thread implementando l'interfaccia Runnable.



## Sincronizzazione implicita

Utilizzando la keyword **synchronized** come qualificatore in un certo metodo, un solo thread alla volta potrà ottenere l'accesso all'oggetto per l'esecuzione del metodo.

Si suggerisce di utilizzarlo in oggetti passivi che sono condivisi e concorrentemente acceduti (per update) da thread multipli.

## Sincronizzazione esplicita

I seguenti metodi (che devono trovarsi all'interno di metodi sincronizzati) permettono di gestire in maniera esplicita la sincronizzazione tra thread:

- **wait**: sospende l'esecuzione del thread corrente
- **notifyAll**: tutti i thread che erano in attesa dell'oggetto sono riattivati
- **notify**: un thread (arbitrario) in attesa dell'oggetto viene riattivato.

## Monitoraggio dei thread

JConsole è uno strumento grafico di monitoraggio e gestione dei thread Java fornito dalla J2SE JDK 5.0.

<http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>

È particolarmente utile per monitorare i thread prodotti da un programma JAVA (inclusi i thread della VM).

Simile a JConsole è VisualVm che permette di misurare e visualizzare le performance di un programma JHava.

<http://visualvm.java.net>

Fornito dalla J2SE JDK 5.0 permette di effettuare analisi più affinate di JConsole (% CPU utilizzata per metodo, durata del blocco o dell'esecuzione di un thread, ...).

## Disciplina per la programmazione

I thread devono essere visti come oggetti che incapsulano stato, comportamento e controllo del comportamento:

- i metodi dell'oggetto dovrebbero essere chiamati solo dal thread che rappresenta l'oggetto minimizzando l'uso di metodi pubblici.

L'interazione deve essere promossa tramite oggetti passivi condivisi senza chiamare metodi pubblici delle loro interfacce (violando l'incapsulamento).

Deve essere perseguita una forte separazione concettuale tra entità attive (agenti responsabili di realizzare le attività) e passive (oggetti condivisi, manipolati dagli agenti).

## Thread Safety & Liveness (module lab-1.2)

Un aspetto centrale della programmazione concorrente consiste nella scrittura di codice (di classi) **thread-safe**.

Una classe è **thread-safe** se continua a comportarsi correttamente quando viene acceduta da molteplici threads:

- senza far riferimento allo scheduling o all'interleaving di esecuzione di questi thread da parte dell'ambiente di esecuzione
- senza ulteriori sincronizzazioni o coordinazione realizzati nel codice chiamante.

La **correttezza** significa che una classe è conforme alle proprie specifiche. Buone specifiche definiscono:

- invarianti che delimitano lo stato di un oggetto
- post-condizioni che descrivono gli effetti delle operazioni.

Le classi di thread-safe incapsulano la sincronizzazione necessaria in modo tale che i clients non debbano fornirle/implementarle in maniera autonoma.

Scrivere codice thread-safe riguarda la gestione degli accessi agli stati, in particolare agli stati shared e mutable:

- **shared**: variabili o oggetti che possono essere acceduti da più di un thread,
- **mutable**: variabili o oggetti che possono variare durante la loro esistenza.

Se diversi threads accedono allo stesso oggetto mutable senza una sincronizzazione appropriata, il programma è guasto (race conditions). Esistono tre strade per correggerlo:

- non condividere variabili tra thread
- rendere lo stato delle variabili immutabile
- utilizzare la sincronizzazione per accedere allo stato delle variabili.

Gli oggetti **stateless** sono sempre thread-safe: le azioni di n thread che accede ad un oggetto stateless non possono agire sulla correttezza di operazioni in altri threads.

La sicurezza di un thread è indeterminata quando l'accesso in lettura e scrittura di oggetti stateful è condivisa.

L'esecuzione concorrente di sequenze di istruzioni non atomiche che potrebbero essere considerate atomiche genera una **race conditions**: essa si verifica quando la correttezza di una computazione dipende dai tempi relativi o dall'interleaving di molteplici threads a tempo di esecuzione.

Alcuni esempi:

- **lost updates**: esecuzione concorrente di operazioni di lettura, modifica e scrittura, non atomiche (es. count++)
- **check and act**: quando l'osservazione di uno stato è utilizzata per prendere una decisione sull'azione successiva, se le attività di "check and act" non sono atomiche, lo stato potrebbe cambiare dopo il check ma prima dell'act (es. if (file X doesn't exists) then create file X)

Le azioni atomiche compound sono queste sequenze di operazioni che devono essere eseguite in maniera atomica.

In JAVA, le azioni compound possono essere realizzate utilizzando blocchi o metodi **synchronized**.

Un blocco **synchronized** ha due parti:

- un riferimento ad un oggetto che servirà come lock
- un blocco di codice per cui il lock fa da condizione di guardia.

I blocchi atomici sfruttano il lock incorporato in ogni oggetto Java, chiamato lock intrinseco funziona come guardia per il blocco.

Il lock è acquisito automaticamente e poi rilasciato da un thread rispettivamente quando l'esecuzione entra ed esce dal blocco:

- se il lock è già stato acquisito, il thread viene sospeso e aggiunto all'**entry set**
- quando un thread esce dal blocco, un altro thread dell'entry set viene selezionato e riattivato
- non sono specificate policy di ordinamento
- se il lock non viene rilasciato dal thread all'interno del blocco, i threads nell'entry set rimangono bloccati per sempre (starvation).

Per i metodi e i campi statici, il lock è associato al relativo oggetto Class.

Per i metodi synchronized, l'oggetto utilizzato come lock è **this**.

Quando un lock è acquisito "per thread" si parla di **lock reentrancy** (a differenza di lock acquisiti "per invocazione"). In JAVA i lock intrinseci sono rientranti: se un thread prova ad acquisire un lock che già detiene, la richiesta ha successo. Questa caratteristica facilita l'incapsulamento del comportamento bloccante e, quindi, lo sviluppo di codice concorrente orientato agli oggetti.

```
public class Widget {  
    public synchronized doSomething() { ... }  
}  
public class LoggingWidget extends Widget {  
    public synchronized void doSomething() {  
        System.out.println(toString()+" : calling doSomething");  
        super.doSomething();  
    }  
}
```

In assenza di reentrancy, l'esempio precedente genererebbe una situazione di deadlock.

L'uso inappropriato di blocchi atomici potrebbe generare problemi di performance: la sequenzializzazione ha infatti un forte impatto sullo speedup. È quindi necessario prendere le migliori decisioni riguardo le parti che devono essere progettate ed implementate come sezioni critiche, minimizzando i tempi di ritenzione dei lock senza però scendere a compromessi rispetto alla safety.

L'utilizzo di lock può assicurare sicurezza dei thread, tuttavia un uso indiscriminato di lock può causare situazioni di deadlock. L'utilizzo di pool di thread e semafori può limitare il consumo di risorse, tuttavia, un fallimento nella corretta identificazione delle attività che devono essere limitate può generare deadlock sulle risorse.

## Deadlock

Un **deadlock** è una situazione in cui due o più azioni concorrenti rimangono bloccate l'una in attesa della terminazione delle altre, e questa non avviene mai.

Coffman definisce le condizioni necessarie affinché si generi un deadlock:

- condizione di **mutua esclusione**: una risorsa che non può essere usata da più di un processo alla volta
- condizione **hold and wait**: i processi che già detengono una risorsa potrebbero richiedere ulteriori risorse
- condizione di **no preemption**: le risorse non possono essere rimosse da un processo che le detiene; le risorse possono essere rilasciate solo dalle azioni esplicite dei processi
- condizione di **circular wait**: quando due o più processi formano una catena circolare e attendono una risorsa bloccata dal processo successivo nella catena.

I deadlock possono verificarsi solo se tutte le 4 condizioni rimangono vere.

Due processi in circular wait rappresentano quello che viene definito deadly embrace.

L'identificazione e il recupero delle situazioni di deadlock comprendono strategie tipicamente adottate nella gestione dei databases: essi sono progettati per identificare e ripristinare situazioni di deadlock, le transazioni – solitamente – acquisiscono lock fino alla commit. È possibile identificare l'insieme delle transazioni che sono oggetto di deadlock analizzando il grafico delle dipendenze "is waiting" alla ricerca di cicli.

Nella Java Virtual Machine non esistono meccanismi di identificazione e recupero automatico: se i threads entrano in deadlock è necessario terminare l'applicazione.

È possibile effettuare una diagnosi grazie al supporto per il thread dump fornito dalla JVM inviando al processo JVM un SIGQUIT (kill -3), premendo CTRL-\ in Unix o CTRL-Break in Windows. Il contenuto del dump riporta lo stack delle chiamate per ogni thread in esecuzione e le informazioni di locking (quale lock è tenuto da ogni thread, in qualche stack frame è stato acquisito il lock, ...).

#### Un semplice esempio

```
public class LeftRightDeadlock {
    private final Object left = new Object();
    private final Object right = new Object();

    public void leftRight() {
        synchronized(left) {
            synchronized(right) {
                doSomething();
            }
        }
    }

    public void rightLeft() {
        synchronized(right) {
            synchronized(left) {
                doSomethingElse();
            }
        }
    }

    private void doSomething() {
        System.out.println("something.");
    }

    private void doSomethingElse() {
        System.out.println("something else.");
    }
}
```

#### Lock definiti dinamicamente.

```
public class Testla {

    private static final int NUM_THREADS = 20;
    private static final int NUM_ACCOUNTS = 5;
    private static final int NUM_ITERATIONS = 10000000;
    private static final Random gen = new Random();
    private static final Account[] accounts = new Account[NUM_ACCOUNTS];

    public static void transferMoney(Account from, Account to, int amount)
        throws InsufficientBalanceException {

        synchronized (from) {
            synchronized (to) {
                if (from.getBalance() < amount)
                    throw new InsufficientBalanceException();
                from.debit(amount);
                to.credit(amount);
            }
        }
    }
}
```

```
    }  
    }  
    ...  
}  
  
public class Testla {  
  
    private static final int NUM_THREADS = 20;  
    private static final int NUM_ACCOUNTS = 5;  
    private static final int NUM_ITERATIONS = 10000000;  
    private static final Random gen = new Random();  
    private static final Account[] accounts = new Account[NUM_ACCOUNTS];  
  
    public static void transferMoney(Account from, Account to, int amount)  
        throws InsufficientBalanceException {...}  
  
    static class TransferThread extends Thread {  
        public void run() {  
            for (int i = 0; i < NUM_ITERATIONS; i++){  
                int fromAcc = gen.nextInt(NUM_ACCOUNTS);  
                int toAcc = gen.nextInt(NUM_ACCOUNTS);  
                int amount = gen.nextInt(10);  
                try {  
                    transferMoney(accounts[fromAcc], accounts[toAcc], amount);  
                } catch (InsufficientBalanceException ex) {}  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        for (int i = 0; i < accounts.length; i++){  
            accounts[i] = new Account(1000);  
        }  
        for (int i = 0; i < NUM_THREADS; i++){  
            new TransferThread().start();  
        }  
    }  
}
```

**Ordering locks:** i deadlock si possono generare se due threads tentano di acquisire i locks i ordini diversi, se tentassero di ottenere i lock nello stesso ordine non si genererebbe la condizione di dipendenza ciclica e si eviterebbe il deadlock. Un programma è libero da deadlock lock-ordering se tutti i suoi threads acquisiscono i locks di cui hanno bisogno sempre secondo un prefissato ordinamento globale.

```
public class AccountManager {  
  
    private final Account[] accounts;  
  
    public AccountManager(int nAccounts, int amount){  
        accounts = new Account[nAccounts];  
        for (int i = 0; i < accounts.length; i++){  
            accounts[i] = new Account(amount);  
        }  
    }  
  
    public void transferMoney(int from, int to, int amount)  
        throws InsufficientBalanceException {  
  
        int first = from;  
        int last = to;  
  
        if (first > last){  
            last = first;  
            first = to;  
        }  
    }  
}
```

```
    }

    synchronized (accounts[first]) {
        synchronized (accounts[last]) {
            if (accounts[from].getBalance() < amount)
                throw new InsufficientBalanceException();
            accounts[from].debit(amount);
            accounts[to].credit(amount);
        }
    }
}
}
```

**Deadlock tra oggetti cooperanti:** alcuni deadlock più *sottili* possono verificarsi tra oggetti cooperanti nei quali non esistono metodi che acquisiscono lock in maniera esplicita, ad esempio nel pattern *Observer*, si possono verificare differenti flussi di esecuzione dei metodi tra oggetti osservanti e osservati.

```
interface IObserved {
    int getState();
    void register(IObserver obj);
}

class MyEntityA implements IObserved {

    private List<IObserver> obsList;
    private int state;

    public MyEntityA() {
        obsList = new ArrayList<IObserver>();
    }

    public void register(IObserver obs) {
        obsList.add(obs);
    }

    public synchronized int getState() {
        return state;
    }

    public synchronized void changeState1() {
        state++;
        for (IObserver o: obsList){
            o.notifyStateChanged(this);
        }
    }

    public synchronized void changeState2() {
        state--;
        for (IObserver o: obsList){
            o.notifyStateChanged(this);
        }
    }
}
```

```
interface IObserver {
    void notifyStateChanged(IObserved obs);
}

class MyEntityB implements IObserver {

    List<IObserved> obsList;

    public MyEntityB(){
        obsList = new ArrayList<IObserved>();
    }

    public synchronized void observe(IObserved obj){
        obsList.add(obj);
        obj.register(this);
    }

    public synchronized
        void notifyStateChanged(IObserved obs) {
        synchronized(System.out){
            System.out.println("state changed: "+obs.getState());
        }
    }

    public synchronized int getOverallState() {
        int sum = 0;
        for (IObserved o: obsList){
            sum += o.getState();
        }
        return sum;
    }
}

class MyThreadA extends Thread {

    MyEntityA obj;

    public MyThreadA(MyEntityA obj){
        this.obj = obj;
    }

    public void run(){
        while (true){
            obj.changeState1();
            obj.changeState2();
        }
    }
}
```



```
class MyThreadB extends Thread {

    MyEntityB obj;

    public MyThreadB(MyEntityB obj){
        this.obj = obj;
    }

    public void run(){
        while (true){
            log("overall state: "+obj.getOverallState());
        }
    }

    private void log(String msg){
        synchronized(System.out){
            System.out.println("[ "+this+" ] "+msg);
        }
    }
}

public class Test2 {
    public static void main(String[] args) {

        MyEntityA objA = new MyEntityA();
        MyEntityB objB = new MyEntityB();
        objB.observe(objA);

        new MyThreadA(objA).start();
        new MyThreadB(objB).start();
    }
}
```

In un programma che non acquisisce mai più di un lock alla volta non si possono generare situazioni di deadlock lock-ordering. Se è necessario acquisire locks multipli, l'ordinamento dei lock deve essere parte integrante della progettazione: minimizzando il numero di interazioni potenzialmente bloccanti e documentando un protocollo di lock-ordering per i lock che possono essere richieste contemporaneamente. Tecniche alternative prevedono l'uso di *timed locks* identificando i deadlock ed effettuando ripristini utilizzando le tryLock delle classi Lock al posto dei lock intrinseci.

## Starvation

La starvation si manifesta, tipicamente, quando si utilizzano le *priorità*. Il supporto di base dei thread per le priorità in Java è deprecato in quanto dipendente dalla piattaforma e soggetto a problemi di liveness.

## Poor responsiveness

La scarsa responsività (ad esempio eseguendo tasks a lungo termine nel thread di gestione della GUI) può essere causato da una cattiva gestione dei lock: se un thread detiene un lock per molto tempo (ad esempio interagendo con una grande collection ed eseguendo molte attività per ogni elemento) gli altri threads che richiedono l'accesso alla stessa collection possono dover aspettare molto tempo.

## Livelock

Quando un thread non può procedere nella propria esecuzione perché tenta di eseguire ripetutamente una operazione che fallisce sempre si ha una situazione di livelock. Una possibile soluzione suggerisce l'introduzione di una qualche forma di casualità nel meccanismo di retry allo scopo di *rompere* la sincronizzazione che genera il livelock.

## Library Mechanisms for Thread Coordination in Java (module lab-1.3)

Le librerie della piattaforma Java (5.0 & 6.) includono un ricco insieme di blocchi per la costruzione di codice concorrente come `collections thread safe` e una varietà di sincronizzatori che possono coordinare il flusso di threads cooperanti.

### Collezioni synchronized

Il wrapper `synchronized`, creato tramite metodo `factory` della `Collection.synchronizedXXX`, permette di ottenere thread-safety tramite incapsulamento dello stato e sincronizzazione di ogni metodo pubblico; in sintesi permette di ottenere safety tramite la serializzazione di tutti gli accessi alle informazioni di stato.

Presenta tuttavia alcuni problemi:

- richiede l'uso di ulteriori meccanismi di lock per gestire azioni composte (iterazioni, navigazione, operazioni condizionali, ...)
- l'oggetto che deve essere utilizzato per il lock client-side è la stessa `collection synchronized`
- si possono verificare problemi di performance:
  - in caso di lock per operazioni lunghe
  - quando la concorrenza viene fortemente limitata.

### Collezioni concurrent

Le collezioni `concurrent` sono state introdotte dalla versione 5.0 e sono progettate per gestire l'accesso concorrente da parte di molteplici threads con lo scopo di accrescere la scalabilità e le performance rispetto alle `collection synchronized`.

Le classi principali sono:

- **ConcurrentHashMap**: sostituiscono le implementazioni basate su `hash map synchronized`
- **CopyOnWriteArrayList**: sostituiscono le implementazioni di `List synchronized`
- **Queue** e **BlockingQueue**: interfacce con diverse implementazioni disponibili.

### Blocking queue

Le `blocking queue` forniscono metodi bloccanti **put** e **take** e gli equivalenti metodi *timed offer* e *poll*:

- quando la coda è piena, la `put` si blocca finché non si genera spazio disponibile nella coda
- quando la coda è vuota, la `take` si blocca fin quando un elemento non è disponibile.

La coda può essere **bounded** o **unbounded** (mai piena).

Le code `bounded` costituiscono un blocco di base per lo sviluppo del pattern di progettazione *producer-consumer*.

Alcune altre classi che implementano **BlockingQueue**: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `PriorityBlockingQueue`, ...

### Dequeues and work stealing

Le strutture dati **Deque** e **BlockingDeque**, introdotte da Java 6.0, implementano delle code a doppia entrata che permettono inserimenti e rimozioni efficienti sia dalla testa che dalla coda (implementazioni: **ArrayDeque** e **LinkedBlockingDeque**).

Vengono utilizzate per il pattern di progettazione *work stealing*, simile a *producers-consumers* in questo caso però ogni consumer ha una propria coda (deque): se un consumer esaurisce il lavoro sulla propria deque, può *rubare* lavoro partendo dalla coda di una deque di qualche altro consumer.

Permette una scalabilità maggiore rispetto a *producers-consumers* in quando i workers non contendono per l'accesso ad una coda di lavoro condivisa, accedono più spesso alla propria deque riducendo i contenziosi.

## Aggregated operation and Lambda

Il Collections Framework ha subito una revisione importante in Java 8 aggiungendo operazioni aggregate basate sulle nuove facility di streams ed espressioni lambda.

Come risultato, la classe **ConcurrentHashMap** introduce più di 30 nuovi metodi in questa release:

- metodi `forEach`: `forEach`, `forEachKey`, `forEachValue`, `forEachEntry`
- metodi `search`: `search`, `searchKeys`, `searchValues`, `searchEntries`
- un gran numero di metodi di riduzione: `reduce`, `reduceToDouble`, `reduceToLong`, ...

## Synchronizers

Un synchronizer è un oggetto che coordina il flusso di controllo dei thread basandosi sul proprio stato (blocking queue possono funzionare come synchronizer).

Si tratta di un blocco di sviluppo codice molto importante nelle applicazioni concorrenti in quanto si tratta di un componente passivo che incapsula funzionalità di coordinazione.

Tutti i synchronizers condividono un certo numero di proprietà strutturali:

- incapsulano lo stato che determina se i thread che arrivano al synchronizer debba essere autorizzato a *passare* o se debba attendere
- forniscono metodi per manipolare lo stato
- forniscono metodi per far attendere in maniera efficiente il synchronizer prima di entrare nello stato desiderato.

I principali tipi di synchronizers forniti nelle librerie Java sono:

- locks
- semaphores
- latches
- barriers.

## Locks

Forniscono funzionalità di blocco esplicite.

Interfaccia Lock:

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

Implementazione di ReentrantLock, tipico utilizzo:

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // update shared object state
    // catch exception and restore invariants if necessary
} finally {
    lock.unlock();
}
```

## Polled and timed lock acquisition

Allo scopo di ottenere un sistema più sofisticato per il recupero in caso di generazione errori è possibile utilizzare tryLock per l'acquisizione di lock polled o timed.

```
public boolean transferMoney(Account from, Account to, Amount am)
    throws InsufficientFundException, InterruptedException {
    while (true) {
        if (from.lock.tryLock()) {
            try {
                if (to.lock.tryLock()) {
                    try {
                        if (from.getBalance().compareTo(am) < 0) {
                            throw new InsufficientFundException();
                        } else {
                            from.debit(am);
                            to.credit(am);
                            return true;
                        }
                    } finally {
                        to.lock.unlock();
                    }
                }
            } finally {
                from.lock.unlock();
            }
        }
    }
}
```

## Explicit vs Intrinsic Locks

I lock intrinseci (blocchi synchronized) lavorano bene in molte situazioni ma hanno alcune limitazioni funzionali:

- non è possibile interrompere un thread che sta aspettando di acquisire un lock
- non è possibile tentare di acquisire un lock senza essere disposti ad aspettare per sempre.

In questi casi è possibile utilizzare dei lock espliciti:

- che possono gestire le interruzioni
- che possono specificare limitazioni sui tempi di attesa.

È tuttavia necessario che gli sviluppatori seguano una disciplina ferrea per evitare comportamenti anomali.

## Semaphores

Si tratta dell'implementazione di base del costrutto di Dijkstra.

La classe **Semaphore** viene creata specificando un numero di permessi virtuali, implementa i metodi **acquire** e **release**, permette di sostenere la fairness.

## Latches

Un latch è un synchronizer che può ritardare il progresso di un thread fino a quando esso non raggiunge il suo stato terminale. Funziona come un gate:

- fin quando il latch non raggiunge lo stato terminale, il gate rimane chiuso e nessun thread può passare
- allo raggiungimento dello stato terminale il gate si apre permettendo il passaggio a tutti i threads
- una volta che il latch raggiunge lo stato terminale non può cambiare nuovamente il suo stato, rimanendo per sempre aperto.

La classe **CountDownLatch** viene inizializzata con uno specifico valore di contatore, utilizza un metodo **countDown** per decrementare il proprio contatore e il metodo **await** per far sì che il thread attenda finché il latch non azzeri il proprio contatore.

Un latch viene utilizzato per assicurare che determinate attività non procedano fino a che altre attività temporizzate non completino la propria esecuzione.

Alcuni esempi:

- assicurare che una computazione non proceda finché le risorse di cui necessita non siano state interamente inizializzate (un binary latch per ogni risorsa)
- assicurare che un servizio non entri in esecuzione finché un altro servizio da cui dipende non sia avviato (utilizzando un binary latch per ogni risorsa)
- attendere che tutte le parti coinvolte in un'attività siano pronte a procedere.

## Barriers

Le barriere sono meccanismi simili ai latches in quanto bloccano un gruppo di thread finché un qualche evento non si verifica; la differenza chiave è che in questo caso tutti i thread devono arrivare al punto di *barriera* per poter procedere la propria esecuzione: i latches attendono il verificarsi di un evento, le barriere attendono il sopraggiungere di tutti gli altri threads.

## GUI Frameworks and Concurrency (module lab-1.4)

Le Graphical User Interfaces (GUI) sono da sempre realizzate utilizzando un unico thread dedicato (event dispatch thread, EDT) che si occupa di raccogliere gli eventi e gestire le variazioni dell'interfaccia grafica per l'utente.

Per raggiungere una thread safety per le GUI si deve passare attraverso un confinamento del thread: tutti gli oggetti GUI devono essere acceduti unicamente dall'event dispatch thread, gli sviluppatori devono assicurarsi che questi oggetti siano appropriatamente confinati. Gli eventi devono essere processati sequenzialmente dall'EDT.

Problemi e sfide: se un task richiede molto tempo per la propria esecuzione, gli altri task devono attendere bloccando, di fatto, la GUI. Per questo motivo i tasks devono completare rapidamente. In generale i tasks lunghi devono fornire un feedback durante il loro progresso e al loro completamento, è quindi necessaria una gestione dell'EDT da parte di questi task.

In Swing, tutti i componenti (JButton, JTable, ...) e i data model (Table Model, Tree Model, ...) sono confinati all'event thread: ogni codice che accede a questi oggetti deve essere eseguito tramite l'event thread. Esistono tuttavia alcune eccezioni, alcuni metodi thread-safe:

- **SwingUtilities.invokeLater()**: permette di verificare se il thread corrente è l'EDT
- **SwingUtilities.invokeLater()**: permette di schedare un Runnable per l'esecuzione sull'evento thread
- **SwingUtilities.invokeAndWait()**: permette di schedare un Runnable bloccando il thread corrente fino al suo completamento
- **Metodi per accodare e ridisegnare o rivalidare**
- **Metodi per aggiungere o rimuovere dei listeners.**

L'event thread di Swing può essere pensato come un unico **single-threaded Executor** che processa i task dall'event queue: `invokeLater` e `invokeAndWait` vengono utilizzati per sottomettere nuovi task da eseguire.

## Esempio di un semplice Executor per la GUI

```
public class GuiExecutor extends AbstractExecutorService {
    // Singletons have a private constructor and a public factory
    private static final GuiExecutor instance = new GuiExecutor();
    private GuiExecutor() { }
    public static GuiExecutor instance() { return instance; }
```

```
public void execute(Runnable r) {
    if (SwingUtilities.isEventDispatchThread())
        r.run();
    else
        SwingUtilities.invokeLater(r);
}
// Plus trivial implementations of lifecycle methods
}
```

I task brevi possono essere eseguiti direttamente nell'event thread.

```
final Random random = new Random();
final JButton button = new JButton("Change Color");
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setBackground(new Color(random.nextInt()));
    }
});
```

Il controllo non lascia mai l'event thread: l'evento originato nel GUI toolkit è consegnato all'applicazione, l'applicazione che modifica la GUI in risposta alle azioni dell'utente.

Tasks a computazione lunga possono volere fornire dei feedback all'utente oppure no. Nel primo caso, un esempio di implementazione è il seguente.

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setEnabled(false);
        label.setText("busy");
        backgroundExec.execute(new Runnable() {
            public void run() {
                try {
                    doBigComputation();
                } finally {
                    GuiExecutor.instance().execute(new Runnable() {
                        public void run() {
                            button.setEnabled(true);
                            label.setText("idle");
                        }
                    });
                }
            }
        });
    }
});
```

Una semplice implementazione, invece, per un task che non vuole fornire output all'utente, è la seguente:

```
ExecutorService backgroundExec = Executors.newCachedThreadPool();
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        backgroundExec.execute(new Runnable() {
            public void run() { doBigComputation(); }
        });
    }
});
```

Java 6.0 fornisce classi ausiliarie per rendere più semplice la programmazione di task a lungo termine che possono interagire con la GUI. La classe **SwingWorker** fornisce un supporto diretto per notificare la cancellazione, il progresso e il completamento dei tasks.

```
class SwingWorker<T,V> implements RunnableFuture<T> {
    ...
    // to be overridden
    protected abstract T doInBackground();
    protected void done()
    protected final void publish(V... chunks)
    protected void process(List<V> chunks);
    ...
    // to be directly used
    boolean cancel(boolean mayInterruptIfRunning);
    protected void setProgress(int progress);
    ...
}
```

Un supporto per l'esecuzione e il conseguente aggiornamento dell'interfaccia utente per tasks asincroni è dato da:

- **doInBackground**: incapsula il corpo computazionale del task affinché venga eseguito in maniera asincrona rispetto alle attività della GUI, calcolando il risultato e generando un'eccezione se non è in grado di completare; incapsulata da un qualche thread, non nella Swing EDT
- **done**: incapsula l'azione da fare sulla GUI quando il task è completo, viene eseguita dalla Swing EDT
- **publish(V... chunks)**: è utilizzata da dentro la `doInBackground` per consegnare risultati intermedi di elaborazione all'EDT
- **process(List<V> chunks)**: riceve data chunks dal metodo `publish` in maniera asincrona sull'EDT.

### Esempio: Swing Worker Test

```
class CounterTask extends SwingWorker<Integer, Integer> {

    protected Integer doInBackground() throws Exception {
        int i = 0;
        int sum = 0;
        int maxCount = 10;
        while (!isCancelled() && i < maxCount) {
            sum+=i;
            i++;
            publish(new Integer[] { i });
            setProgress(100 * i / maxCount);
            Thread.sleep(1000);
        }
        return sum;
    }

    protected void process(List<Integer> chunks) {
        for (int i : chunks)
            System.out.println("Step "+i);
    }

    protected void done() {
        if (isCancelled()){
            System.out.println("Task cancelled.");
        } else {
            System.out.println("Task completed.");
        }
    }
}
```

```
}

public class SwingWorkerTest {
    public static void main(String[] args) {
        JTextArea textArea = new JTextArea(10, 20);
        JProgressBar progressBar = new JProgressBar(0, 100);
        CounterTask task = new CounterTask();

        JButton startButton = new JButton("Start");
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { task.execute(); });

        JButton cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { task.cancel(true); });

        task.addPropertyChangeListener(new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent evt) {
                if ("progress".equals(evt.getPropertyName())) {
                    progressBar.setValue((Integer) evt.getNewValue());
                }
            }
        });

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(startButton);
        buttonPanel.add(cancelButton);
        JPanel cp = new JPanel();
        LayoutManager layout = new BoxLayout(cp, BoxLayout.Y_AXIS);
        cp.setLayout(layout);
        cp.add(buttonPanel);
        cp.add(new JScrollPane(textArea));
        cp.add(progressBar);
        JFrame frame = new JFrame("SwingWorker Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setContentPane(cp);
        frame.pack();
        frame.setVisible(true);
    }
}
```

## Implementing Monitors in Java (module lab-1.5)

L'implementazione di monitors in Java può seguire due approcci di sviluppo differenti:

- sfruttando i meccanismi di basso livello di Java (synchronized, wait, notify, notifyAll)
- sfruttando i meccanismi di alto livello forniti da java.util.concurrent

### Primo approccio (meccanismi di basso livello)

È possibile sfruttare due tipi di meccanismi di base:

- blocchi di sincronizzazione: le istruzioni per entrare nei blocchi sincronizzati sono chiamate **monitorenter** e **monitorexit**; talvolta i locks intrinseci di Java sono chiamati monitor lock oppure monitors.
- insieme di meccanismi utilizzati per sincronizzazione esplicita tra threads attraverso oggetti condivisi
  - metodo **wait**: qualsiasi metodo sincronizzato in qualsiasi oggetto può contenere una chiamata wait che sospende il thread corrente



- metodo **notify**: on thread tra quelli in attesa sull'oggetto target (scelto arbitrariamente) viene ripristinato a seguito della chiamata del metodo notify che deve anch'esso essere contenuto in un metodo o blocco sincronizzati
- metodo **notifyAll**: tutti i thread in attesa sull'oggetto target vengono ripristinati, anche questo metodo deve essere contenuto in un metodo o blocco sincronizzati.

## Semantica wait

Una chiamata wait determina le seguenti azioni:

- se il thread corrente è stato interrotto, allora il metodo esce immediatamente generando un'**InterruptedException**, viceversa il thread viene bloccato
- la Java Virtual Machine inserisce il thread nell'insieme di attesa (inaccessibile) associato all'oggetto target
- il synchronization lock per l'oggetto target viene rilasciato ma tutti gli altri locks appartenenti al thread rimangono mantenuti

## Semantica notify e notifyAll

Una chiamata notify determina le seguenti azioni:

- Se esiste, viene selezionato un thread (in maniera arbitraria e senza nessuna garanzia sulla scelta), questo viene rimosso dall'insieme di attesa associato all'oggetto target
- Il thread deve ri-ottenere il lock di sincronizzazione per l'oggetto target per poter ripristinare
- Il thread viene ripristinato.

NotifyAll opera come notify eccetto per il fatto che gli step sono eseguiti simultaneamente per tutti i threads, tuttavia – considerando che devono acquisire il lock – i threads continueranno uno alla volta.

## Wait interruption

Se viene invocato Thread.interrupt per un thread sospeso in attesa, si applica lo stesso meccanismo di notify ad eccezione del fatto che, dopo la riacquisizione del lock, il metodo genera un'**InterruptedException** e lo stato di interruzione del thread viene impostato a false.

Se un interrupt e un notify si verificano contemporaneamente non c'è alcuna garanzia su quale azione abbia la precedenza.

## Wait-timed version

La timed-version del metodo wait: wait(long msecs) e wait(long msecs, int nanosecs) accetta argomenti che specificano il tempo massimo, desiderato, di permanenza nell'insieme di attesa. Opera nello stesso modo della versione non timed ad eccezione del fatto che se una wait non viene notificata entro il tempo massimo, viene rilasciata automaticamente.

Non esiste uno stato che differenzi se si sta eseguendo una wait o una timed-wait.

Non è garantito un tempo esatto di ripristino al raggiungimento del tempo di permanenza desiderato e non c'è garanzia riguardo la granularità.

## Monitor Pattern

Un oggetto che implementa il pattern monitor incapsula tutto il suo stato e lo protegge tramite lock intrinseci dell'oggetto stesso.

Regole:

- Ogni metodo pubblico deve essere implementato come synchronized
- Non deve avere campi pubblici
- Il codice del monitor deve accedere e utilizzare solo oggetti completamente confinati dentro al monitor

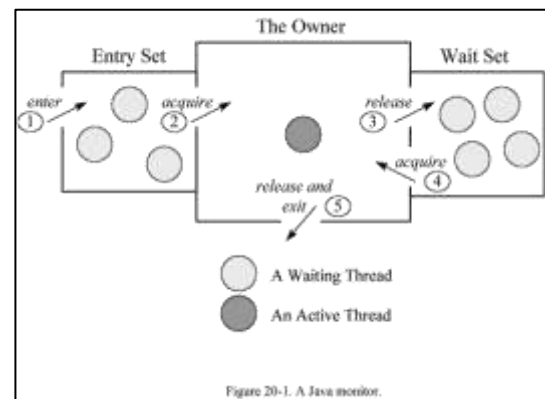
- È possibile utilizzare una sola variabile condizionale: l'oggetto stesso

La semantica di segnalazione è un variante della strategia signal-and-continue:  $E = W < S$ .

**Entry set:** insieme dei threads sospesi in attesa di ottenere un lock

**Wait set:** insieme dei threads che hanno eseguito una wait e attendono di essere notificati per riprendere la propria esecuzione.

### Esempio 1: un contatore (semplice)



```
public class Counter {  
  
    private int count;  
  
    public Counter() {  
        count = 0;  
    }  
  
    public synchronized void inc() {  
        count++;  
    }  
  
    public synchronized int getValue() {  
        return count;  
    }  
  
}
```

### Esempio 2: celle sincronizzate

```
public class SynchCell {  
  
    private int value;  
    private boolean available;  
  
    public SynchCell() {  
        available = false;  
    }  
  
    public synchronized void set(int v) {  
        value = v;  
        available = true;  
        notifyAll();  
    }  
  
    public synchronized void int get() {  
        while (!available) {  
            try {  
                wait();  
            } catch (InterruptedException ex) {}  
        }  
        return value;  
    }  
  
}
```

## Alcune osservazioni sulle limitazioni del supporto di base di Java

In caso di molteplici predicati condizionali, questi devono essere associato alla stessa ed unica variabile di condizione. Questo determina il fatto che diversi threads con ruoli differenti che entrano in attesa di specifici differenti eventi, possono dover aspettare sulla stessa variabile di condizione.

Inoltre la semantica della wait include “spurious wake up” che non si verificano in risposta di nessuna chiamata notify di nessun thread (verificare la documentazione di Java).

Per questi motivi:

- Per riattivare il thread desiderato, tutti i thread in attesa devono essere riattivati
- Un thread in attesa sulla variabile condizionale può esser riattivato anche se lo specifico predicato condizionale non è verificato.

Uno schema di base per un’implementazione sicura prevede:

- L’inserimento della wait in un loop while che verifichi lo specifico predicato condizionale
- L’utilizzo di notifyAll anziché notify.

### Esempio 3: Somma di operandi, molteplici threads in attesa su differenti predicati

```
public class SynchAdder {
    private int x, y;
    boolean xAvailable, yAvailable;

    public SynchAdder(){
        xAvailable = yAvailable = false;
    }

    public synchronized void setFirstOperand(int x){
        while (xAvailable) {
            wait();
        }
        this.x = x;
        xAvailable = true;
        if (xAvailable && yAvailable){
            notifyAll();
        }
    }

    public synchronized void setSecondOperand(int y){
        while (yAvailable) {
            wait();
        }
        this.y = y;
        yAvailable = true;
        if (xAvailable && yAvailable){
            notifyAll();
        }
    }

    public synchronized int getSum() throws InterruptedException {
        while (!(xAvailable && yAvailable)){
            wait();
        }
        xAvailable = yAvailable = false;
        notifyAll();
        return x + y;
    }
}
```

## Implementazione di un buffer “bounded”

```
public class BoundedBuffer<Item> {
    private int first;
    private int last;
    private int count;
    private Item[] buffer;

    public BoundedBuffer(int size){
        first = 0;
        last = 0;
        count = 0;
        buffer = (Item[])new Object[size];
    }

    public synchronized void put(Item item) throws InterruptedException {
        while (isFull()){
            wait();
        }
        last = (last + 1) % buffer.length;
        count++;
        buffer[last] = item;
        notifyAll();
    }

    public synchronized Item get() throws InterruptedException {
        while (isEmpty()){
            wait();
        }
        first = (first + 1) % buffer.length;
        count--;
        notifyAll(); //IS THIS NECESSARY?
        return buffer[first];
    }

    public synchronized boolean isEmpty(){
        return count == 0;
    }

    public synchronized boolean isFull(){
        return count == buffer.length;
    }
}
```

## Secondo approccio (meccanismi di alto livello)

Sfruttando lock espliciti tramite le classi **ReentrantLock** e **Condition** e implementando le variabili di condizione fornite dalla libreria **java.util.concurrent**.

La classe **ReentrantLock** è un’implementazione di un blocco rientrante analogo ai mutex (semafori).

La classe **Condition** rappresenta le variabili condizione che devono essere utilizzate solo all’interno dei blocchi protetti da un **Reentrant Lock**. Un oggetto condizione è strettamente accoppiato ad un blocco rientrante.

Il metodo per creare una condizione è **public Condition newCondition()** che restituisce un’istanza **Condition** da utilizzare con l’istanza **Lock**.

Per implementare i monitors:

- Si usa un **ReentrantLock mutex** per ogni monitor
- Si racchiude ogni metodo tra le chiamate **mutex.lock** e **mutex.unlock**
- Per ogni condizione da utilizza, la si crea tramite il blocco **mutex**.

## Esempio 2

Versione precedente	Versione rivisitata
<pre> public class SynchCell {      private int value;     private boolean available;      public SynchCell() {         available = false;     }      public synchronized void set(int v) {          value = v;         available = true;         notifyAll();     }      public synchronized void int get() {          while (!available) {             try {                 wait();             } catch (InterruptedException ex) {}         }          return value;     } } </pre>	<pre> public class SynchCell2 {      private int value;     private boolean available;     private Lock mutex;     private Condition isAvail;      public SynchCell2() {         available = false;         mutex = new ReentrantLock();         isAvail=mutex.newCondition();     }      public void set(int v) {         try {             mutex.lock();             value = v;             available = true;             isAvail.signalAll();         } finally {             mutex.unlock();         }     }      public void int get() {         try {             mutex.lock();             if (!available) {                 try {                     isAvail.await();                 } catch                 (InterruptedException ex) {}             }             return value         } finally {             mutex.unlock();         }     } } </pre>

## Bounded buffer

```
public class BoundedBuffer<Item> {
    private int first, last, count;
    private Item[] buffer;
    private Lock mutex;
    private Condition notFull, notEmpty;

    public BoundedBuffer(int size){
        first = last = count = 0;
        buffer = (Item[])new Object[size];
        mutex = new ReentrantLock(); // new ReentrantLock(true) for fair mutex
        notFull = mutex.newCondition();
        notEmpty = mutex.newCondition(); //Condition taken from same lock
    }

    public void put(Item item) throws InterruptedException {
        try {
            mutex.lock();
            while (count == buffer.length){
                notFull.await();
            }
            last = (last + 1) % buffer.length; count++; buffer[last] = item;
            notEmpty.signal();
        } finally {
            mutex.unlock();
        }
    }

    public Item get() throws InterruptedException {
        try {
            mutex.lock();
            while (count == 0){
                notEmpty.await();
            }
            first = (first + 1) % buffer.length; count--;
            notFull.signal();
            return buffer[first];
        } finally {
            mutex.unlock();
        }
    }

    public boolean isEmpty() throws InterruptedException {
        try {
            mutex.lock();
            return count == 0;
        } finally {
            mutex.unlock();
        }
    }

    public boolean isFull(){
        try {
            mutex.lock();
            return count == buffer.length;
        } finally {
            mutex.unlock();
        }
    }
}
```

## Semantica di signaling e effetti indesiderati

Sia nel caso di implementazione che utilizza **synchronized+wait/notify/notifyAll** che nel caso di quella che utilizza **ReentrantLock+Condition**, la semantica è del tipo  $E = W < S$ .

L'effetto potrebbe determinare comportamenti inattesi, infatti, dati

- un thread T1 che è in attesa su una condizione C all'interno di un monitor
- un thread T2 anch'esso all'interno del monitor
- un thread T3 in attesa di accedere al monitor

supponendo che T2 segnali sulla variabile C, allora, T1 è ripristinato ma, siccome  $E = W$ , T1 compete con T3 per ottenere il lock sul monitor; pertanto, in funzione di differenti implementazioni, T3 potrebbe procedere e T1 rimanere in attesa.

### Esempio

```
class MyMonitor {
    private ReentrantLock mutex;
    private Condition c;

    public MyMonitor() {
        mutex = new ReentrantLock();
        c = mutex.newCondition();
    }

    public void m1() throws InterruptedException {
        try {
            mutex.lock();
            System.out.println("First thread inside, going to wait");
            c.await();
            System.out.println("First thread unblocked.");
            Thread.sleep(5000);
        } finally {
            mutex.unlock();
        }
    }

    public void m2() throws InterruptedException {
        try {
            mutex.lock();
            System.out.println("Second thread inside");
            Thread.sleep(5000);
            System.out.println("Second thread inside, going to signal");
            c.signal();
            System.out.println("Second thread inside signaled.");
        } finally {
            mutex.unlock();
        }
    }

    public void m3() throws InterruptedException {
        try {
            mutex.lock();
            System.out.println("Third thread inside.");
            Thread.sleep(5000);
        } finally {
            mutex.unlock();
        }
    }
}
```

```
class MyThread1 extends Thread {
    private MyMonitor mon;
    public MyThread1(MyMonitor mon){
        this.mon = mon;
    }

    public void run(){
        log("First thread started.");
        mon.m1();
    }

    private void log(String msg){
        synchronized(System.out){
            System.out.println(msg);
        }
    }
}

class MyThread2 extends Thread {
    private MyMonitor mon;
    public MyThread2(MyMonitor mon){
        this.mon = mon;
    }

    public void run(){
        log("Second thread started.");
        mon.m2();
    }

    private void log(String msg){
        synchronized(System.out){
            System.out.println(msg);
        }
    }
}

class MyThread3 extends Thread {
    private MyMonitor mon;
    public MyThread3(MyMonitor mon){
        this.mon = mon;
    }

    public void run(){
        log("Third thread started.");
        mon.m3();
    }

    private void log(String msg){
        synchronized(System.out){
            System.out.println(msg);
        }
    }
}

public class TestSemantics {
    public static void main(String[] args) throws Exception {
        MyMonitor mon = new MyMonitor();
        new MyThread1(mon).start();
        new MyThread2(mon).start();
        new MyThread3(mon).start();
    }
}
```



L'output all'esecuzione è il seguente:

```
First thread started.  
First thread inside, going to wait  
Second thread started.  
Second thread inside  
Third thread started.  
Second thread inside, going to signal  
Second thread inside signaled.  
Third thread inside.  
First thread unblocked.
```

## From Threads to Tasks (module-2.1)

L'analisi e la progettazione orientate ai task si fondano sull'idea di analizzare il problema per identificare le concorrenze identificando le possibili decomposizioni di attività e dati e le loro dipendenze.

Il concetto di **task** riguarda un'unità di lavoro astratta, indipendente e discreta collocata al livello di astrazione del dominio applicativo del problema. Il concetto di **task** è disaccoppiato dalla nozione di **thread**: un task viene eseguito da qualche thread o processo del sistema.

### Task-oriented analysis

L'analisi orientata ai task si compone di due fasi:

1. Analisi di decomposizione di task e dati
2. Analisi delle dipendenze.

### Task decomposition

Seguendo il principio *divide-et-impera*, un problema può essere risolto con una collezione di attività indipendenti o quasi indipendenti ottenute decomponendo una soluzione più elaborata. I tasks devono essere sufficientemente indipendenti in modo tale che la gestione delle dipendenze occupi solo una piccola parte del tempo complessivo di esecuzione del programma.

Dopo aver individuato i tasks, i dati possono essere decomposti in base alla decomposizione dei tasks.

Esempio: sistemi basati su workflow.

### Data decomposition

In modo alternativo, quando è più semplice, è possibile decomporre i dati in unità che possono essere gestite in maniera relativamente indipendente, in questo caso la decomposizione in tasks segue la decomposizione dei dati. Si tratta di una decomposizione in grado di scalare in maniera appropriata con il numero di processori.

Esempio: moltiplicazione di matrici.

### Dependency analysis

L'analisi delle dipendenze permette di individuare gruppi e ordinamenti di esecuzione dei task in accordo con i tipi di dipendenze esistenti allo scopo di ottenere soluzioni corrette e massimizzare le performance.

I possibili tipi di dipendenze riguardano:

- Dipendenze temporali
- Dipendenze su dati o risorse.

### Task-oriented design & programming

I tasks sono astrazioni di progettazione e programmazione di più alto livello rispetto ai threads (la concorrenza è gestita a livello logico, non fisico).

I task sono mappati sui threads in base alla specifica architettura scelta.

Gli elementi principali di un'architettura a task sono:

- chi sottomette il task
- chi esegue il task

Il punto chiave è che il task è eseguito in maniera asincrona rispetto alle attività di sottomissione dei tasks.

I principali esempi di architetture orientate ai task sono:

- master-workers (fork-join)
- filer-pipeline.

## Master-workers architecture

L'architettura è composta da un agente **master** e da un insieme, possibilmente dinamico, di agenti **workers** che interagiscono tramite un appropriato medium funzionante come un *contenitore di tasks*.

L'agente **master**:

- decompone il task globale in sotto-tasks
- assegna i sotto-task inserendone le descrizioni nel medium
- raccoglie i risultati.

L'agente **worker**:

- recupera il task dal medium
- esegue il task
- comunica i risultati

Il medium *contenitore di tasks* è tipicamente implementato come una blackboard o un bounded buffer.

### Collecting results

La raccolta dei risultati prodotti dai singoli tasks è uno dei principali problemi, riguarda la modalità tramite la quale un master aggrega i risultati elaborati in maniera asincrona.

I principali approcci prevedono:

- l'aggregazione tramite un *contenitore dei risultati*
- l'aggregazione attraverso un monitor collettore/aggregatore specificatamente progettato
- l'aggregazione attraverso il meccanismo dei **future**.

### Future mechanism

Il meccanismo dei **future** prevede che, quando si esegue una computazione asincrona tramite un task, viene creato un oggetto che rappresenta il risultato futuro (o lo stato) della computazione. Tale oggetto viene immediatamente restituito.

L'oggetto potrebbe fornire operazioni per:

- verificare lo stato del task (poll)
- interrompere l'esecuzione quando è richiesto l'accesso al risultato del task
- cancellare un task in esecuzione, se possibile
- catturare gli errori o le eccezioni relativi al task in esecuzione.

### Executor abstraction and policies

L'astrazione **executor** ha l'obiettivo di incapsulare i task **workers** e il contenitore di task:

- i **masters** sottomettono i tasks da eseguire inviandoli all'executor (fase di sottomissione)
- l'**executor** implementa una policy di esecuzione:
  - progettata e messa a punto in base ai requisiti
  - utilizzando, ad esempio, un pool di N+1 workers threads per l'esecuzione dei tasks su N cores.

Il Java Executor Framework è un esempio di implementazione concreta.

Le policy di esecuzione hanno lo scopo di definire:

- quando un task sarà eseguito e da quale thread
- in quale ordine i tasks dovrebbero essere messi in esecuzione (FIFO, LIFO, priorità, ...)
- quanti tasks possono essere eseguiti in maniera concorrente
- quale task deve essere selezionato come vittima in caso di sovraccarico del sistema
- quali azioni dovrebbero essere messe in atto prima o dopo l'esecuzione di un task.

L'uso di strumenti di gestione delle risorse permette di individuare le policy ottimali che dipendono dalle risorse di calcolo disponibili e dai requisiti di quality of services.

La gestione del pool di thread utilizza una coda di lavoro, ogni thread segue questo comportamento:

- loop:
  - richiesta del prossimo task da eseguire
  - esecuzione del task

### Master-workers variant: fork-join

In alcuni casi, la decomposizione in task non può essere fatta unicamente da un master ma sono i workers che, dinamicamente, determinano quali sotto-attività mettere in campo (esempio: merge-sort, metodi di quadratura adattiva per il calcolo di integrali, ...).

Il fork-join si sviluppa applicando l'architettura master-workers ricorsivamente (i workers diventano masters producendo sotto-tasks e raccogliendo i risultati parziali). Si tratta dell'approccio effettivo per implementare una strategia divide-et-impera. Può essere realizzato sia utilizzando un contenitore di tasks sia direttamente.

### Filter-pipeline architecture

Questa architettura utilizza una catena lineare di agenti che interagiscono attraverso quale canale, bounded buffer o contenitore di tasks.

Un **agente generator** si trova all'inizio della catena e genera i dati che devono essere processati dalla pipeline.

Uno o più **agenti filter** – agenti intermedi all'interno della catena – consumano le informazioni in input da un canale e generano informazioni di output in un altro canale.

Un **agente sink** termina la catena e aggrega i risultati.

Si tratta di un'architettura utilizzata ad esempio nell'immagine processing.

## Asynchronous Programming (module-2.2)

La **programmazione asincrona** è uno stile di programmazione molto importante al giorno d'oggi: in generale riguarda l'esecuzione asincrona, la gestione delle richieste e dei processi. Si pone ad un livello di astrazione più alto rispetto ai thread del sistema operativo. È supportata dalla maggior parte dei frameworks e delle piattaforme di sviluppo.

Si tratta tuttavia di un oggetto ancora in evoluzione.

Gli approcci principali di programmazione asincrona possono essere classificati in:

- **framework task-oriented**
- **programmazione event-driven**
  - architetture del flusso di controllo basate su cicli di eventi
  - pattern "reactor"
- **funzioni asincrone e stile "continuation passing"**
  - handler degli eventi utilizzati per la continuazione

### Event-driven programming

In questo approccio il flusso di controllo del programma è determinato da **eventi** (azioni dell'utente, output di sensori, messaggi da altri programmi o threads, ...) che invocano l'esecuzione asincrona tramite **event handlers**.

Le interfacce grafiche e le applicazioni che richiedono di essere *reattive* implementano questo approccio.

### Thread vs Event-driven programming

*"Why Threads are a bad idea (for most purposes)"; 1996, John Ousterhout • Sun Microsystems Laboratories*

Threads		Event-driven	
Sviluppati nell’ambito dei sistemi operativi. Evoluzione in strumenti user-level. Proposti come soluzioni per una varietà di problemi. Qualsiasi programmatore dovrebbe saper manipolare i thread?			
Problema: la programmazione con i threa è molto complessa. I thread dovrebbero essere utilizzati quando la concorrenza (multi core CPI) è realmente un requisito.		Alternativa: gli eventi. Molti problemi gestibili con i thread possono essere risolti più facilmente con gli eventi.	
<b>What are threads?</b> I thread rappresentano una soluzione generale per gestire la concorrenza; costituiscono stream multipli e indipendenti che condividono uno stato, sono gestiti tramite uno scheduling e richiedono l’utilizzo di meccanismi di sincronizzazione		<b>What is event-driven programming?</b> Un’unico stream di esecuzione (senza concorrenza sulla CPU, sui core) agisce in funzione degli eventi (callbacks): l’event-loop aspetta gli eventi e invoca gli handlers. Non esiste uno schedling preventivo. Generalmente gli eventi sono computazioni brevi.	
<b>What are threads used for?</b> I thread sono utilizzati nei sistemi oerativi (un kernel thread per ogni processo utente), in applicazioni specifiche per risolvere problemi più rapidamente (un thread per CPU/core), nei sistemi distribuiti per processare le richieste concorrentemente (I/O overlap), nelle interface grafiche (garantire responsività durante l’esecuzione di computazioni lunghe, multimedia, animazioni).		<b>What are events used for?</b> Principalmente utilizzato nelle interfacce grafiche (nelle quali gli handler per gli eventi utente implementano i comportamenti opportuni), è anche usato nei sistemi distribuiti (un handler per ogni input).	
<b>What’s wrong with threads?</b> La programmazione con i thread è molto complessa per la maggior parte dei programmatori, anche per gli esperti lo sviluppo non è banale.		<b>Problems with events</b> <b>Long-running</b> handlers: handlers che implementano comportamenti con lunghi tempi di computazione rendono le applicazioni non responsive. In questi casi è possibile valutare alcune tecniche: suddivere i comportamenti in sotto task utilizzando gli eventi per assemblare i risultati, interrompere gli handlers (es. per operazioni I/O), richiamare periodicamente l’event loop (aumento della complessità). <b>Local state across events:</b> non è possibile mantenere uno stato <i>locale</i> , gli handlers devono necessariamente ritornare valori <b>No CPU concurrency:</b> richiesta in applicazioni scientifiche specifiche <b>Event-driven I/O not always well supported:</b> ad esempio possono verificarsi problemi nella gestione del buffering in scrittura.	
<b>Why thread are hard?</b> <b>Sincronizzazione:</b> è necessario coordinare l’accesso ai dati condivisi tramite meccanismi di blocco <b>Deadlock:</b> i blocchi possono determinare dipendenze circolari con conseguente blocco del sistema <b>Debug:</b> la dipendenza dai dati, dal tempo rendono difficile il processo di identificazione e risoluzione degli errori <b>Thread break abstraction:</b> non è possibile progettare i moduli in maniera indipendente <b>Callback don’t work with locks</b> <b>Achieving good performance is hard:</b> i meccanismi di blocco riducono la concorrenza, blocchi a grana fine incrementano la complessità e riducono normalmente le performance, le performance sono comunque limitate dalle attività di scheduling e context switch del sistema operativo. <b>Threads not well supported:</b> la portabilità tra sistemi operativi non è garantita, le librerie standard non sono thread-safe, esistono pochi strumenti di debug			
Threads vs Events			
I thread gestiscono bene la concorrenza.	← CONCORRENZA	Gi eventi evitano la concorrenza il più possibile	
	DEBUG →	Il processo di debug è più semplice nella programmazione ad eventi: la dipendenza	

		temporale è correlata agli eventi, non allo scheduling interno. È facile individuare problemi di memoria corrotta o responsività.
I thread implementando davvero la concorrenza su sistemi multicore / multi CPU	← VELOCITÀ →	Gli eventi determinano computazioni più rapide su singole CPU non prevedendo locking né context swtiching.
I thread sono legati al sistema operativo	PORTABILITÀ →	

In conclusione entrambi gli approcci hanno caratteristiche specifiche e possono essere implementati per risolvere diversi tipi di problemi: i **threads** sono importanti nello sviluppo di sistemi altamente concorrenti (applicazioni scientifiche, ...) ma devono essere evitati quando si vogliono gestire interfacce grafiche, sistemi distribuiti, low-end servers; quando possibile sarebbe utile isolare l'uso dei threads in specifici application kernel per governare meglio la complessità.

In generale la concorrenza è altamente complessa e per questo motivo sarebbe sensato cercare di evitarla quando possibile: i threads sono molto più potenti degli eventi ma raramente sono davvero necessari; la programmazione con i threads è molto complessa: iniziare sempre con lo sviluppo di applicazioni tramite eventi ed utilizzare i threads solo quando le performance diventano critiche.

Event loop, callbacks and event handlers

Nella programmazione ad eventi:

- i comportamenti sono organizzati in insiemi **event handlers** che incapsulano la computazione da eseguire quando un certo evento si verifica
- si usa una coda per mantenere traccia degli eventi generati sia dall'ambiente che dagli stessi event handlers
- un programma è concettualmente rappresentato da un loop

```
loop {
    Event ev = evQueue.remove()
    Handler handler = selectHandler(ev)
    Execute(handler)
}
```
- l'esecuzione di computazioni asincrone è atomica: gli eventi che si verificano mentre un handler è in esecuzione sono accodati nella coda eventi.

Attenzione

Siccome gli handlers devono essere eseguiti senza blocchi come operazioni atomiche non è permesso l'uso di primitive bloccanti e comportamenti *bloccanti* devono essere rimpiazzati da richieste o computazioni asincrone tramite le quali si possono generare eventi nel futuro.

L'event-loop è inserito come parte integrante dell'esecuzione a runtime, per questo motivo gli sviluppatori non si devono preoccupare di creare cicli o gestirli: devono solo specificare in che modo selezionare ed eseguire gli event handlers.

La programmazione event-driven è anche chiamata **programmazione senza stack** perché l'event handling non è una chiamata a procedura, non esiste punto di return, non è necessario alcuno stack.

L'event-loop è un singolo flusso di controllo: la concorrenza è ottenuta tramite l'alternanza di esecuzione degli event handlers, per sfruttare core multipli è possibile generare multipli event-loop.

Per gestire handlers che prevedono computazioni lunghe, senza generare blocchi di sistema e garantire responsività, è possibile utilizzare funzioni non bloccanti eseguite in maniera asincrona specificando la funzione di callback che sarà chiamata dal processo quando il risultato sarà disponibile.

Una **callback** rappresenta un event handler che viene generato quando una funzione completa la propria esecuzione, essa definisce una *continuazione* della compuaZIONE (**Continuation Passing Style, CPS**).

Nello stile CPS una funzione prende in input, oltre ai propri parametri, un ulteriore argomento: una funzione da eseguire al termine della computazione.

Sync version	Async Version (findUserById e loadUserPic)
<pre>function loadUserPic(userId) {   var user = findUserById(userId);   return loadPic(user.picId); }</pre>	<pre>function loadUserPic(userId, ret) {   findUserById(userId, function(user) {     loadPic(user.picId, ret);   }); }  loadUserPic('john', function(pic) {   ui.show(pic); });</pre>

Le callbacks sono solitamente implementate come **chiusure** (closures): tramite questa tecnica legata al binding dello scope per cui le funzioni sono oggetti di prima classe, una closure è un record che memorizza una funzione assieme a tutto il suo ambiente.

L’invocazione delle callback può avvenire:

- tramite un flusso di controllo separato che esegue concorrentemente al flusso di controllo che ha generato la richiesta (inversione del controllo, problemi di corsa critica)
- nello stesso flusso di controllo che ha immesso la richiesta (event loop model, modello di esecuzione delle moderne web apps).

Funzione bloccante	Funzione non bloccante
<pre>var ajaxRequest = new XMLHttpRequest; ajaxRequest.open('GET',url); ajaxRequest.send(null); while (ajaxRequest.readyState === XMLHttpRequest.UNSENT){} // !DEADLOCK!</pre>	<pre>var ajaxRequest = new XMLHttpRequest; ajaxRequest.open('GET',url); ajaxRequest.send(null); ajax.onreadystatechange = function() { // what to do next ... }</pre>

In Javascript, ad esempio, è possibile registrare una funzione da eseguire in maniera asincrona dopo un determinato intervallo di tempo

```
for (var i = 1; i <= 3; i++){  
  console.log("setting timer..." + i);           (A)  
  setTimeout(function(){ console.log(i); }, 0)      (B immesso nell'event loop)  
}  
  
Output sulla console:  
Setting timer...1  (A)  
Setting timer...2  (A)  
Setting timer...3  (A)  
4                  (B)  
4                  (B)  
4                  (B)
```

Esempio in Node.js

```
var sys = require("sys"),  
    http = require("http"),  
    url = require("url"),  
    path = require("path"),  
    fs = require("fs");  
  
http.createServer(function(request, response) {  
  var uri = url.parse(request.url).pathname;  
  var filename = path.join(process.cwd(), uri);  
  path.exists(filename, function(exists) {
```

```
if(exists) {
  fs.readFile(filename, function(err, data) {
    response.writeHead(200);
    response.end(data);
  });
} else {
  response.writeHead(404);
  response.end();
}
});
}).listen(8080);

sys.log("Server running at http://localhost:8080/");
```

Punti chiave della programmazione asincrona “continuation-based”

L'utilizzo di un singolo thread per coordinare e gestire molteplici tasks asincroni (eventualmente eseguiti da threads esterni), garantisce un baso utilizzo di memoria rispetto alle soluzioni multi-thread (uno stack per ogni thread). Inoltre garantisce l'assenza di corse critiche a basso livello in quanto non vi sono informazioni di stato condivise per cui un solo thread accede allo stato e garantisce l'assenza di deadlocks a basso livello in quanto gli handlers asincroni non si interrompono mai.

Problemi con la programmazione asincrona

Oltre ai benefici, esistono alcuni problemi ben noti riguardo la programmazione asincrona ad eventi e lo stile Continuation Passing, spesso riferiti come **callback hell**:

- **asynchronous spaghetti**: la computazione è frammentata in esecuzioni asincrone “sparse” incidendo negativamente sulla struttura e sulla modularità dei programmi
- **pyramid of doom / callback hell**: le callbacks sono innestate negli handler per diversi livelli incrementando la complessità e riducendo la leggibilità del codice con conseguenze legate alla riusabilità e estensibilità.

```
step1(function(result1) {
  step2(function(result2) {
    step3(function(result3) {
      // and so on ...
    })
  })
})
```

*“I love async, but I can't code like this”  
(a complaint of a developer on the Node.js Google Group)*

Promise-Async Await (module-2.3)

Una soluzione (parziale) al probelma delle callback (callback hell) prevede l'adozione del meccanismo delle **promises** (1976, Daniel Friedman and D. Wise).

Le promises rappresentano l'eventualee completamento e il risultato di una singola operazione asincrona: incapsulano le azioni asincrone che agiscono sul risultato del calcolo (non disponibile immediatamente); una promise può essere rifiutata o risolta una sola volta ed è immutabile. Le promises hanno la proprietà chiave di permettere l'appiattimento delle callback.

Esempio:

Sync version	Promise Version
<pre>function loadUserPic(userId) {   var user = findUserById(userId);   return loadPic(user.picId); }</pre>	<pre>var promisedPic = loadUserPic('john');  promisedPic.then(function(pic) {   ui.show(pic); });</pre>

Un esempio sulla pyramid of doom: pattern per specificare una esecuzione sequenziale in un framework asincrono

Funzioni anonime	Funzioni definite
------------------	-------------------



<pre>h() {   asyncFunc1(params1, (res1) -&gt; {     ...     asyncFunc2(params2, (res2) -&gt; {       ...       asyncFunc3(params3, (res3) -&gt; {         ...       }     }   } }</pre>	<pre>asyncFunc1(params1, func) { ... } asyncFunc2(params2, func) { ... } asyncFunc3(params3, func) { ... }  h() {   asyncFunc1(params1,     asyncFunc2(params2,       asyncFunc3(params3))) ; }</pre>
---	---

Per superare il problema della piramide si sfrutta l'idea di passare i parametri ma non la continuazione.

Di base un'asyncFunc non ritorna nulla perché il risultato della sua esecuzione viene utilizzato nella callback.

Il concetto di **promise** è simile al concetto di **future**: permette di specificare cosa sarà effettuato con il risultato.

In event loop non si effettua mai il polling (che invece si fa con le future).

Le promise hanno un metodo chiamato "then" per specificare la continuazione.

```
asyncFunc1(params1) { ... }
asyncFunc2(params2) { ... }
asyncFunc3(params3) { ... }

h() {
  Promise p1 = asyncFunc1(params1);
  Promise p2 = p1.then(asyncFunc2(params2));
  Promise p3 = p2.then(asyncFunc3(params3));
  ...
}
```

Quando la promise è creata si trova in stato **pending**, quando la funzione asincrona ha successo la promise passa in stato **fulfilled** o **resolved**

In caso di errore nell'esecuzione della funzione asincrona, la promise entra in stato **rejected** ed è possibile definire le azioni da compiere in caso di reject.

```
asyncFunc1(params1) { ... }
asyncFunc2(params2) { ... }

h() {
  Promise p1 = asyncFunc1(params1);
  Promise p2 = p1.then(asyncFunc2(params2), asyncFuncReject(paramsReject));
  ...
}
```

Esempio realizzato in javascript (su repl.it)

<pre>function delayWithRandom(t,p){   var promise = new Promise(     function (resolve,reject){       setTimeout(function(){         var r = Math.random();         if (r &gt; p) {           resolve(r);         } else {           reject(r);         }       },t);     }   );   return promise; }</pre>	<pre>var promise = delayWithRandom(1000,0.5);  promise.then(function(r){   console.log("success: ",r); }, function(r){   console.log("fail: ",r); });</pre>
--	---

Esempio realizzato in javascript (su repl.it)

```
var asyncFunc = function(t) {
  var myPromise = new Promise(function(resolve, reject) {
    console.log("pre");
    setTimeout(function() {
      console.log("doing something...");
      resolve(new Date().getTime());
      console.log("...done something");
    }, t);
    console.log("post");
  });
  return myPromise;
}

function reject() {
  console.log(timestamp + ': reject');
}

var p = asyncFunc(3000);

console.log(p);

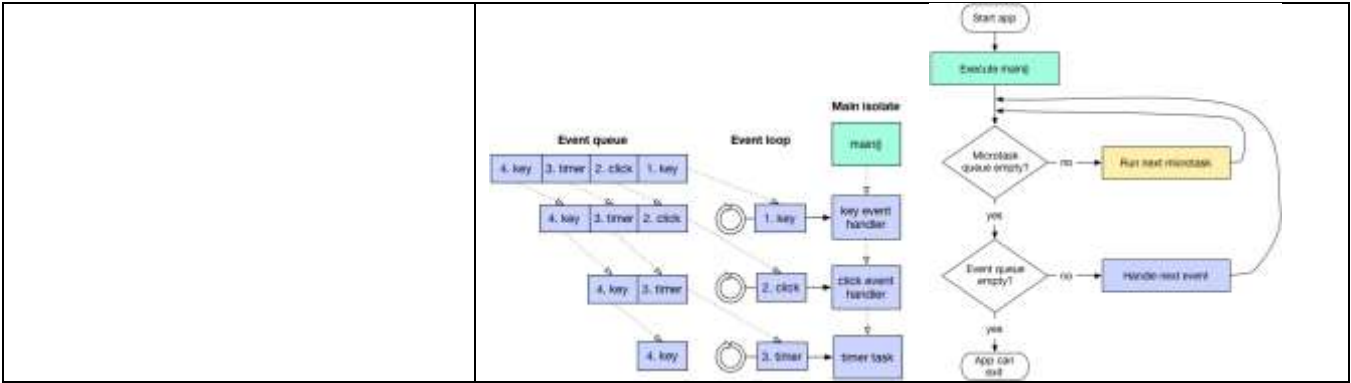
p.then(function(timestamp) { console.log(timestamp + ': resolve'); },
      function(timestamp) { console.log(timestamp + ': reject'); } );
```

Pattern / techinche

<p><b>Promise chaining</b></p> <p><i>Obiettivo</i> Appiattimento pyramid of doom</p> <p><i>Tecnica</i> Il metodo ritorna una promessa che può essere accodata</p>	<pre>var promise = new Promise(   function(resolve, reject) {     resolve(1);   } );  var p1 = promise.then(function(val){   console.log(val); // 1   return val + 2; }); p1.then(function(val){   console.log(val); // 3 });</pre>
<p><b>Chaining Async Functions</b></p> <p><i>Obiettivo</i> Accodare nuove funzioni asincrone</p> <p><i>Tecnica</i> Il valore di ritorno di una funzione è una promise</p>	<pre>function delayWithRand(t){   var promise = new Promise(     function (resolve, reject){       setTimeout(function(){         var r = Math.random();         resolve(r);       }, t);     }   );   return promise; }  var promise = delayWithRand(1000); promise.then(function(r){   console.log("success: ", r);   return delayWithRand(1500); }).then(function(r){   console.log("success: ", r);   return delayWithRand(500); }).then(function(v){   console.log(v); });</pre>

<p><b>Propagation values (and errors)</b>  <i>Obiettivo</i>                      Passaggio informazioni  <i>Tecnica</i>                      Il valore di ogni promise in una catena è qualsiasi cosa il precedente evento (complete/error) ritorni</p>	<pre>var myProm1 = delayWithRandomSucc(1000,0.5); var myProm2 = myProm1.then(function(v) {     console.log(v);     return "done"; }); myProm2.then(function(v) {     console.log(v); });</pre>
<p><b>Sequential Execution Flow Pyramid flattening</b>  <i>Obiettivo</i>                      Semplificare il codice, appiattare la piramide  <i>Tecnica</i>                      Esecuzione sequenziale</p>	<pre>var promise = new Promise(     function(resolve, reject) {         resolve(1);     } );  promise.then(function(val) {     console.log(val); // 1     return val + 2; }).then(function(val) {     console.log(val); // 3 });</pre>
<p><b>Parallel execution flow</b>  <i>Obiettivo</i>                      Eseguire callbacks in ordine non predefinito  <i>Tecnica</i>                      Tutte le callbacks sono chiamate quando una Promise è risolta</p>	<pre>var myPromise = asyncFunc1(...);  myPromise.then(function(v) {     console.log(v);     return asyncFunc2(...); });  myPromise.then(function(v) {     console.log(v);     return asyncFunc3(...); });</pre>
<p><b>Promisifying existing code</b>  <i>Obiettivo</i>                      Semplificare la gestione di un codice asincrono  <i>Tecnica</i>                      Eseguire Promise function nel metodo e gestire la resolve e la reject</p>	<pre>function get(url) {     // Return a new promise.     return new Promise(function(resolve, reject) {         // Do the usual XHR stuff         var req = new XMLHttpRequest();         req.open('GET', url);         req.onload = function() {             // Called even on 404 etc so check the status             if (req.status == 200) {                 // Resolve promise with response text                 resolve(req.response);             } else {                 // Otherwise reject with the status text                 reject(Error(req.statusText));             }         };         // Handle network errors         req.onerror = function() {             reject(Error("Network Error"));         };         req.send(); // Make the request     }); }  // configuring the promise.. get('story.json').then(function(response) {     console.log("Success!", response); }, function(error) {     console.error("Failed!", error); });</pre>

<p><b>Combining promises (ALL)</b>  <i>Obiettivo</i>                      Comporre un set di chiamate asincrone indipendenti da eseguire in parallelo con un punto di sincronizzazione  <i>Tecnica</i>                      Metodo all: la callback è chiamata quando tutte le promise sono completate o almeno una è fallita</p>	<pre>var totalProm = Promise.all(promiseList); totalProm.then(function(values) {...})  var myProm1 = delayWithRand(1000); var myProm2 = delayWithRand(2000); var myTotalProm = Promise.all([myProm1, myProm2]);  myTotalProm.then(function(values) {     console.log(values[0]);     console.log(values[1]); });</pre>
<p><b>Combining promises (RACE)</b>  <i>Obiettivo</i>                      Comporre un set di chiamate asincrone indipendenti da eseguire in parallelo per completare un'azione il prima possibile  <i>Tecnica</i>                      Metodo race: la callback è chiamata quando una (la prima) promise è completata o è fallita                       Esempio: ricerca di informazione in rete... appena si trova l'informazione è possibile stamparla senza attendere il termine di tutte le promise</p>	<pre>var totalProm = Promise.race(promiseList); totalProm.then(function(value) {...},     function(value) {...});  var myProm1 = delayWithRand(1000); var myProm2 = delayWithRand(2000);  var myTotalProm = Promise.race([myProm1, myProm2]);  myTotalProm.then(function(value) {     console.log(value); });</pre>
<p><b>Promise error handling</b>  <i>Obiettivo</i>                      Introdurre un miglioramento relativo alle callbacks  <i>Tecnica</i></p> <ul style="list-style-type: none"> <li>• ogni handler che restituisce un valore e non genera un errore passa un resolve alla Promise successiva</li> <li>• ogni handler che genera un'eccezione passa un reject alla promise successiva</li> <li>• viene passato un stato non gestito alla prossima Promise</li> </ul>	
<p><b>Immediately resolved promise</b>  <i>Obiettivo</i>                      Rendere immediata la risoluzione di una promise, ad esempio per accodare una callback da eseguire nel prossimo ciclo di event loop  <i>Tecnica</i>                      Metodo resolve</p>	<pre>console.log("this cycle"); Promise     .resolve("done")     .then(function(v) {         console.log("next cycle " + v);     }); console.log("end of this cycle");</pre>
<p><b>Tasks and microtasks</b>                      Task lunghi rischiano di rendere l'eventloop non responsivo, i microtasks sono stati introdotti come versione "light" dei task: hanno precedenza sugli altri</p>	<pre>Promise.resolve().then(function() { ... });</pre>



I problemi delle Promises

Eagerness e il problema dei parametri

Non è possibile passare parametri alle funzioni utilizzate in un “then”, le promesse sono “eager” (avide): iniziano a lavorare non appena vengono costruite.

<pre>function delay(t){   var promise = new Promise(     function (resolve,reject){       setTimeout(function(){ resolve(); },t);     });   return promise; }  function getTime(){   var time = new Date().getTime();   console.log(time); }</pre>	
<pre>getTime(); delay(1000)   .then(getTime)   .then(delay(1000))   .then(getTime);</pre>	<pre>getTime(); delay(1000)   .then(getTime)   .then(function(){ return delay(1000); })   .then(getTime)</pre>

Utilizzo nei loop

Le promises non possono essere utilizzate nei cicli o nelle iterazioni. In questo esempio si nota come si ottenga la stessa stampa ad ogni iterazione;

```
function delay(t){
  var promise = new Promise(
    function (resolve,reject){
      setTimeout(function(){
        resolve();
      },t);
    });
  return promise;
}

for (var i = 0; i < 3; i++){
  delay(1000).then(function(){
    var time = new Date().getTime();
    console.log(i + " > " + time);
  })
}
```

Cancellazione

Le promises non possono essere cancellate, non esiste controllo sui task asincroni con conseguente possibile spreco di risorse

### Async & Await

Recente estensione del linguaggio che fornisce uno stile di programmazione sincrona mantenendo un core asincrono.

Utilizzando l’etichetta `async` una funzione può essere definita come asincrona.

Chiamando la funzione, il codice è eseguito come una funzione normale (stesso ciclo di eventi) e restituisce una `promise`.

<code>console.log("pre");</code>	<code>→→→ output →→→</code>	<code>pre</code>
<code>console.log(myFunc);</code>	<code>→→→ output →→→</code>	<code>[AsyncFunction]</code>
<code>console.log(myFunc());</code>	<code>→→→ output →→→</code>	<code>Promise {}</code>
<code>console.log("post");</code>	<code>→→→ output →→→</code>	<code>post</code>

#### Esempio

<pre>var test = async function() {   return 1+2; } var testa = function() {   return 1+2; }  test().then(function(v) {   console.log ("v="+v); })  console.log("pre"); console.log("-----"); console.log("testa:"); console.log(" "+testa); console.log("-----"); console.log("testa():"); console.log(" "+testa()); console.log("-----"); console.log("test:"); console.log(" "+test); console.log("-----"); console.log("test():"); console.log(" "+test()); console.log("-----"); console.log("post");</pre>	<pre>//OUTPUT ON CONSOLE  pre ----- testa:   function () {return 1 + 2;} ----- testa():   3 ----- test:   async function () {return 1 + 2;} ----- test():   [object Promise] ----- post v=3</pre>
---	---

L’operatore `await` deve essere utilizzato nel corpo di funzioni `async`, può essere applicato ad una qualsiasi `promise` ed ha l’effetto di sospendere l’esecuzione della funzione finché la `promise` non passa in stato *fulfilled*. Non blocca il flusso di controllo.

```
var delay = function(t) {
  var promise = new Promise(
    function (resolve,reject){
      setTimeout(function(){
        resolve();
      },t);
    });
  return promise;
};
```

```
var myFunc = async function() {
  console.log("here1");
  await delay(3000);
  console.log("here2");
  await delay(2000);
  console.log("here3");
  return 1+2;
}

myFunc().then(function(v) { console.log(v) ; });
```

L'await <promise> converte una promise in un risultato o in un'eccezione quando ottiene un risultato dalla funzione sincrona:

- se la promise è risolta, l'operatore await estrae il valore e lo restituisce
- se la promise è rejecte, l'operatore await genera un'eccezione con il valore (rejected).

Il comportamento indotto dall'operatore await è simile a una return che può essere ripresa per continuare l'esecuzione del codice successivo:

- quando una funzione esegue un await il flusso ritorna al programma chiamante che riceve l'oggetto promise create dalla funzione asincrona in uno stato *pending*
- lo stato della computazione è memorizzato per essere ripristinato quando il controllo ritorna alla funzione
- la funzione asincrona è ripristinata in maniera asincrona non appena l'event loop estrae l'evento corrispondente dalla coda degli eventi.

I benefici di un programma asincrono gestito con await riguardano soprattutto la semplificazione del codice superando i problemi di eagerness e parametri e di utilizzo nei cicli.

```
function delay(t){
  var promise = new Promise(
    function (resolve,reject){
      setTimeout(function(){
        resolve();
      },t);
    });
  return promise;
}

async function waitFor(t){
  console.log("waitFor - pre");
  await delay(t);
  console.log("waitFor - post");
}

async function main(){
  console.log("Before");
  for (var i = 0; i < 3; i++){
    console.log("Stepping "+i+"...");
    waitFor(1000);
    console.log("Step "+i);
  }
  console.log("After");
}

main();
```

## Problemi

L'operatore await può essere utilizzato solo nel corpo di funzioni async, pertanto è necessario introdurre almeno una funzione async.

Le funzioni async non possono ripartire (dopo un await) se il flusso di controllo sta eseguendo altri handlers.

È necessario applicare una disciplina di progettazione ferrea per evitare di realizzare programmi che, mixando approcci sincroni e asincroni, abbiano comportamenti difficili da interpretare.

<pre> async function pause(t) {   console.log("before");   var promise = new Promise(     function (resolve, reject) {       setTimeout(function () {         resolve();       }, t);     });   await promise;   console.log("after"); } </pre>	
<pre> console.log("before call"); pause(1000); console.log("after call"); </pre>	<pre> async function main() {   console.log("before call");   await pause(1000);   console.log("after call"); } main(); </pre>
<pre> //output on console before call before after call =&gt; undefined after </pre>	<pre> //output on console before call before =&gt; Promise {} after after call </pre>

Per comporre più funzioni async in un await è necessario utilizzare le API promise.

```

function delay(t) {
  var promise = new Promise(
    function (resolve, reject) {
      setTimeout(function () {
        resolve();
      }, t);
    });
  return promise;
}

async function f() {
  await delay(1000);
  return 1+2;
}

async function g() {
  await delay(999);
  return 2+3;
}

async function compose() {
  console.log("before");
  r = await Promise.race([f(), g()]);
  console.log("done: " + r);
}

compose();

```



## Reactive Programming (module-2.4)

Lo stile continuous passing e le promises assumono che la computazione asincrona debba fornire i suoi risultati in un unico momento. Tuttavia è normale che le applicazioni necessitino di gestire **streams** di dati o eventi.

Il paradigma di programmazione reattiva è orientato attorno alla nozione di **flusso dati** e alla propagazione dei cambiamenti, semplificando la gestione di streams asincroni di dati ed eventi (observer pattern).

Introdotta originariamente negli anni '90 gode oggi di parecchie attenzioni come approccio effettivo per lo sviluppo di applicazioni web responsive, in particolar modo nell'ambito dei Big data.

### Manifesto sistemi reattivi

[www.reactivemanifesto.org](http://www.reactivemanifesto.org)

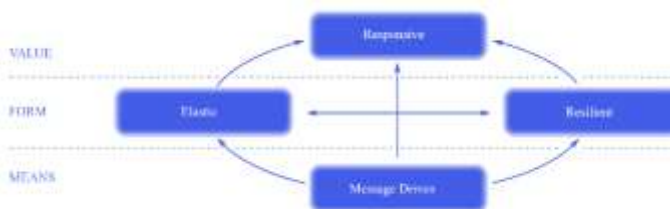
I Sistemi Reattivi sono:

**Responsivi:** Il sistema, se è in generale possibile dare una risposta ai client, la dà in maniera tempestiva. La responsività è la pietra miliare dell'usabilità e dell'utilità del sistema; essa presuppone che i problemi vengano identificati velocemente e gestiti in modo efficace. I sistemi responsivi sono focalizzati a minimizzare il tempo di risposta, individuando per esso un limite massimo prestabilito di modo da garantire una qualità del servizio consistente nel tempo. Il comportamento risultante è quindi predicibile, il che semplifica la gestione delle situazioni di errore, genera fiducia negli utenti finali e predispone ad ulteriori interazioni con il sistema.

**Resilienti:** Il sistema resta responsivo anche in caso di guasti. Ciò riguarda non solo i sistemi ad alta disponibilità o mission-critical: infatti, accade che ogni sistema che non è resiliente si dimostrerà anche non responsivo in seguito ad un guasto. La resilienza si acquisisce tramite replica, contenimento, isolamento e delega. I guasti sono relegati all'interno di ogni componente, isolando così ogni componente dagli altri e quindi garantendo che il guasto delle singole porzioni del sistema non comprometta il sistema intero. Il recupero di ogni componente viene delegato ad un altro componente (esterno) e l'alta disponibilità viene assicurata tramite replica laddove necessario. I client di un componente vengono dunque alleviati dal compito di gestirne i guasti.

**Elastici:** Il sistema rimane responsivo sotto carichi di lavoro variabili nel tempo. I Sistemi Reattivi possono adattarsi alle variazioni nella frequenza temporale degli input incrementando o decrementando le risorse allocate al processamento degli stessi. Questo porta ad architetture che non hanno né sezioni contese né colli di bottiglia, favorendo così la distribuibilità o la replica dei componenti e la ripartizione degli input su di essi. I Sistemi Reattivi permettono l'implementazione predittiva, oltre che Reattiva, di algoritmi scalabili perché fondati sulla misurazione real-time della performance. Tali sistemi, raggiungono l'elasticità in maniera cost-effective su commodity hardware e piattaforme software a basso costo.

**Orientati ai Messaggi:** I Sistemi Reattivi si basano sullo scambio di messaggi asincrono per delineare per ogni componente il giusto confine che possa garantirne il basso accoppiamento con gli altri, l'isolamento e la trasparenza sul dislocamento e permetta di esprimere i guasti del componente sotto forma di messaggi al fine di delegarne la gestione. L'utilizzo di uno scambio esplicito di messaggi permette migliore gestibilità del carico di lavoro, elasticità e controllo dei flussi di messaggi mediante setup e monitoraggio di code di messaggi all'interno del sistema e mediante l'applicazione di feedback di pressione laddove necessari. Uno scambio di messaggi trasparente rispetto al dislocamento rende possibile, ai fini della gestione dei guasti, l'utilizzo degli stessi costrutti e semantiche sia su cluster che su singoli host. Uno stile di comunicazione non bloccante fa sì che l'entità ricevente possa solo consumare le risorse, il che porta ad un minor sovraccarico sul sistema.



I sistemi software di grandi dimensioni si compongono di sistemi più piccoli e dunque dipendono dalle proprietà Reattive dei loro costituenti. Ciò significa che i Sistemi Reattivi si fondano su principi di design che considerano tali proprietà ad ogni livello della scala architeturale, rendendo così componibili i sottosistemi Reattivi. I sistemi software più grandi al mondo hanno architetture basate sui principi Reattivi e riescono ad operare servendo miliardi di utenti ogni giorno: è giunto il momento di utilizzare questi principi con consapevolezza e dall'inizio del processo di design, evitando di riscoprirli nuovamente ogni volta.

Sono state introdotte estensioni reattive a tutti i maggiori linguaggi e sistemi di programmazione.

Con la programmazione reattiva un programmatore può definire gli elementi che reagiscono ad ogni evento, le astrazioni definite sono valori di prima classe e possono essere trasferiti o composti nei programmi.

Esistono due tipi di astrazioni reattive:

1. **event streams**: modellano valori variabili nel tempo (discreti o continui, sono astrazioni asincrone agganciate al flusso di dati sequenziali o intermittenti provenienti da eventi ricorrenti (es. mouse)
2. **behaviours (or signals)**: hanno un valore continuo nel tempo, sono astrazioni che rappresentano un ininterrotto flusso di dati provenienti da un evento costante (es. timer)

Esempio di timer in FlapJax

```
var timer = timerB(100); //behaviour: produce un valore ogni 100ms
var seconds = liftB(      //behaviour: converte i ms in secondi
  function (time){
    return Math.floor(time / 1000);
  }, timer );

insertDomB(seconds, 'timer-div'); //la variabile seconds inserita in un div
```

## Lifting

I valori nelle espressioni che dipendono da valori reattivi devono essere anch'essi reattivi; allo stesso modo una variabile assegnata a qualche espressione che implica behaviours o event streams deve essere reattiva.

Il processo di conversione di una variabile normale in una variabile reattiva è chiamato **lifting**.

Allo scopo di computare correttamente tutte le espressioni reattive una volta che un event stream o un behaviour è "triggerato", molte librerie costruiscono un grafico delle dipendenze: quando un'espressione cambia, le dipendenze sono ricalcolate e i loro valori aggiornati.

Il lifting può essere implicito o esplicito.

## Composizione di event streams e behaviours

Una caratteristica essenziale nella programmazione reattiva è data dalla composizione: essa permette di evitare problemi di gestione delle callback (es. la gestione di tre diverse callback per separare gli eventi di pressione di un pulsante sul mouse, movimento o rilascio di un pulsante può essere effettuata componendo i comportamenti in un singolo event stream).

Esempio in FlapJax

```
var saveTimer = timerE(10000); //10 seconds
var saveClicked = extractEventE('save-button', 'click');
var save = mergeE(saveTimer, saveClicked);
save.mapE(doSave); //save received data
```

- saveTimer è un event stream che genera un valore ogni 10 secondi
- saveClicked è un event stream che genera un valore ad ogni click del mouse
- save, compone i due event streams, e genera un valore in ognuno dei due casi
- per ogni evento di ognuno dei due streams viene eseguita la funzione doSave.

## Reactive extensions (Rx)

Rx è una libreria che compone programmi asincroni e event-based utilizzando le collection observable.

Possiede tre proprietà fondamentali: asynchronous and event based, composizione, observable collections.

Rx è stata introdotta originariamente come libreria su .NET, ora è disponibile per molti linguaggi di programmazione.

Fornisce tre proprietà di base:

- **Asynchronous and event based**  
Nella mission di Rx c'è la semplificazione dei modelli di programmazione asincroni e basati su event quali aspetti chiave per lo sviluppo di interfacce utente, web app, cloud app reattive.
- **Composition**  
Semplificazione della composizione di computazioni asincrone.
- **Observable collections**  
Rx sfrutta la conoscenza attiva i modelli di programmazione simili a LINQ, che vedono le computazioni asincrone come data sources observable. Un mouse *diventa* un database di movimenti e clicks, i data sources asincroni sono composti utilizzando svariati combinatori che permettono l'applicazione di filtri, proiezioni, joins, operazioni basate sul tempo, ...

## Principi

Utilizzando Rx è possibile:

- Rappresentare multipli data streams asincroni ("osservabili") provenienti da diverse fonti
- Definire relazioni tra gli event stream e degli "osservatori".

## Interfacce Observable e Observer

```
interface IObservable<T> {  
    IDisposable Subscribe(IObserver<T> observer);  
}
```

```
interface IObserver<T> {  
    void OnNext(T value);  
    void OnError(Exception error);  
    void OnCompleted();  
}
```

```
interface IDisposable {  
    void Dispose();  
}
```

Un semplice esempio: quando un observer si sottoscrive ad una sequenza observable, il thread che richiama il metodo Subscribe può essere diverso dal thread in cui la sequenza esegue fino al suo completamento. La chiamata Subscribe è asincrona e il chiamante non è bloccato durante l'osservazione.

```
...  
IObservable<int> source = Observable.Range(1, 10);  
IDisposable subscription = source.Subscribe(  
    x => Console.WriteLine("OnNext: {0}", x),  
    ex => Console.WriteLine("OnError: {0}", ex.Message),  
    () => Console.WriteLine("OnCompleted"));  
Console.WriteLine("Press ENTER to unsubscribe...");  
Console.ReadLine();  
subscription.Dispose();  
...
```

Bridging with existing .NET events

Lo stream di eventi dei movimenti del mouse di una form Windows viene convertita in una sequenza osservabile. Ogni volta che un evento mouse-move viene generato, il subscriber riceve una notifica OnNext. È possibile esaminare il valore di EventArgs di ogni notifica e recuperare la posizione del mouse-move.

```
var lbl = new Label();
var frm = new Form { Controls = { lbl } };
IObservable<EventPattern<MouseEventArgs>>
    move = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
move.Subscribe(evt => { lbl.Text = evt.EventArgs.Location.ToString(); });
```

Bridging with existing asynchronous sources

```
Stream inputStream = Console.OpenStandardInput();
var read = Observable.FromAsyncPattern<byte[], int, int, int>
    (inputStream.BeginRead, inputStream.EndRead);
byte[] someBytes = new byte[10];
IObservable<int> source = read(someBytes, 0, 10);
IDisposable subscription = source.Subscribe(
    x => Console.WriteLine("OnNext: {0}", x),
    ex => Console.WriteLine("OnError: {0}", ex.Message),
    () => Console.WriteLine("OnCompleted"));

Console.ReadKey();
```

BeginRead e EndRead per un’oggetto Stream che usa il pattern IAsyncResult, vengono convertiti in una funzione che restituisce una sequenza osservabile.

Querying Observable sequences using LINQ operators

Combining streams  var source1 = Observable.Range(1,3); var source2 = Observable.Range(1,3); source1.Concat(source2).Subscribe(Console.WriteLine);	Il risultato è la sequenza 1, 2, 3, 1, 2, 3. Questo perché con l’operatore Concat la seconda sequenza non si attiva fin quando la prima non ha terminato di effettuare la push di tutti i suoi valori. Il Subscriber legge poi tutti i valori dalla sequenza risultante.
Merge streams  var source1 = Observable.Range(1,3); var source2 = Observable.Range(1,3); source1.Merge(source2).Subscribe(Console.WriteLine);	Il risultato è la sequenza 1, 1, 2, 2, 3, 3. L’operatore Merge infatti attiva le due sequenze contemporaneamente. La sequenza risultante sarà completa solo dopo che l’ultima sequenza sorgente avrà terminato di effettuare la push dei suoi valori.
Porojction  var seqNum = Observable.Range(1,5); var seqString = from n in seqNum select new string('*', (int)n); seqString.Subscribe( str => { Console.WriteLine(str); }); Console.ReadKey();	Mappatura di uno stream di numero di uno stream di stringhe.

Altri operatori utili e importanti permettono di filtrare, effettuare operazioni a tempo, gestire eccezioni, ...

In generale:

- **Transform data:** utilizzando metodi come l'aggregazione e la proiezione
- **Compose data:** utilizzano metodi come zip e selectMany
- **Query data:** utilizzando metodi come where e any.

## RxJava

RxJava è un'implementazione delle Reactive Extensions per la Java VM. Estende il pattern observer per supportare sequenze di dati e/o eventi e aggiunge operatori che permettono di mettere insieme sequenze dichiarativamente, astruendo dai problemi che riguardano cose come la gestione dei thread a basso livello, la sincronizzazione, la sicurezza, le strutture dati concorrenti e l'input/output non bloccante.

### RxJava and Reactive Stream

“lo scopo di Reactive Streams è di fornire uno standard per la processazione asincrona degli stream senza la pressione del ‘non bloccante’”.

Reactive Streams è uno standard ed una specifica per librerie orientate agli streams che:

- Processino un potenzialmente illimitato numero di elementi nella sequenza
- Permettano il passaggio di elementi tra i componenti

Link: <https://github.com/reactive-streams/reactive-streams-jvm/>

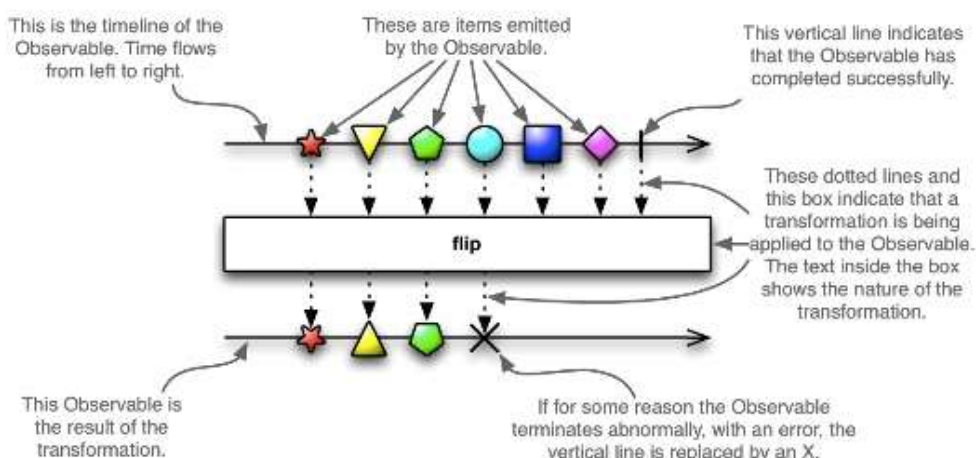
### RxJava Operators

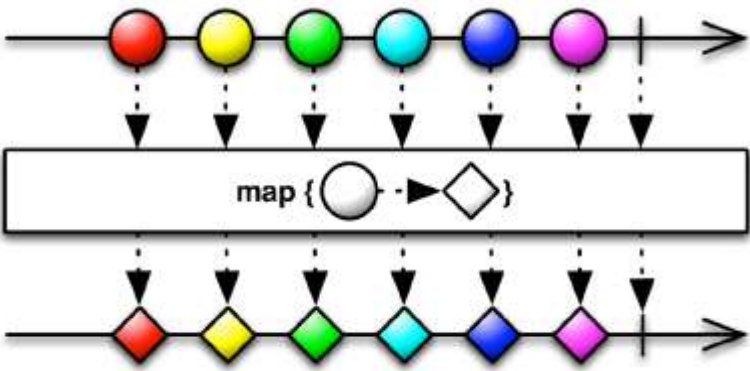
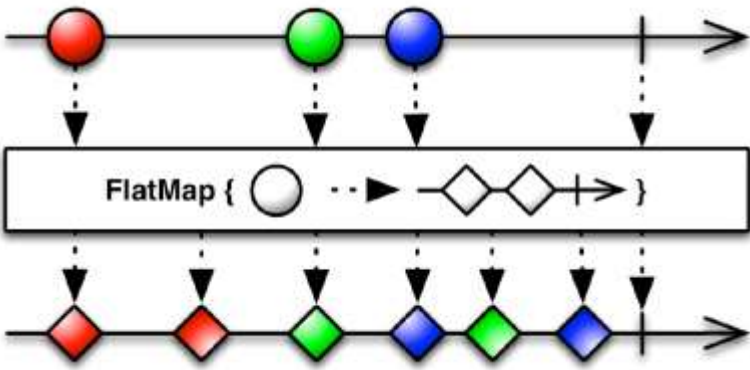
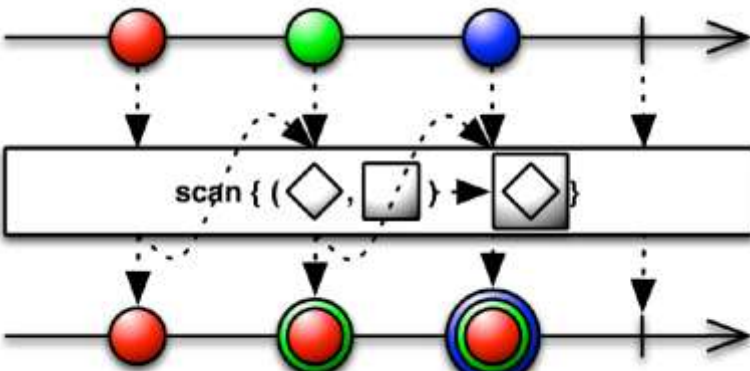
La documentazione completa è reperibile tramite il link <http://reactivex.io/documentation/operators.html>.

Alcuni esempi:

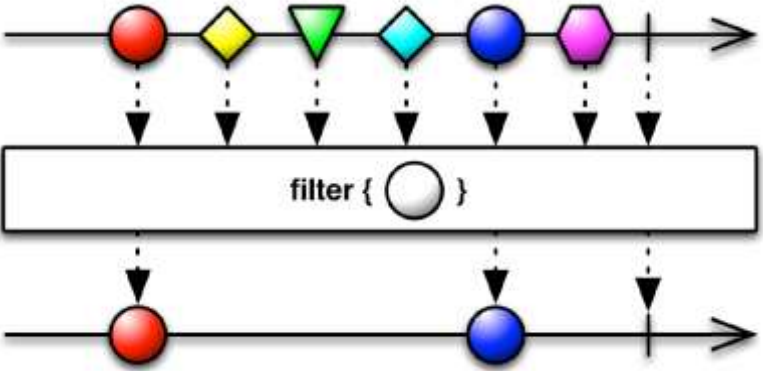
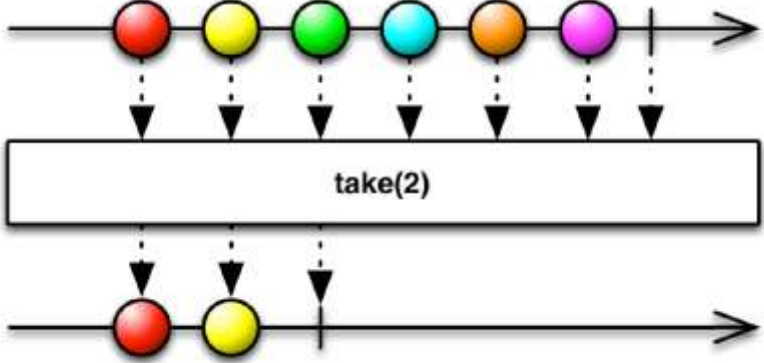
- **Creating observables:** operatori per originare nuove osservazioni
- **Transforming observables:** operatori per trasformare elementi che vengono emessi da un osservabile
- **Filtering observables:** operatori che emettono elementi selezionati da una fonte osservabile
- **Combining observables:** operatori che creano un singolo osservabile operando su molteplici fonti
- **Conditional and boolean operators:** operatori che valutano una o più osservabili o elementi emessi da fonti osservabili
- **Mathematical and aggregate operators:** operatori che agiscono sull'intera sequenza di elementi da un osservabile
- **Backpressure operators:** strategie per gestire fonti osservabili che producono elementi più rapidamente di quando l'observer sia in grado di consumarli.

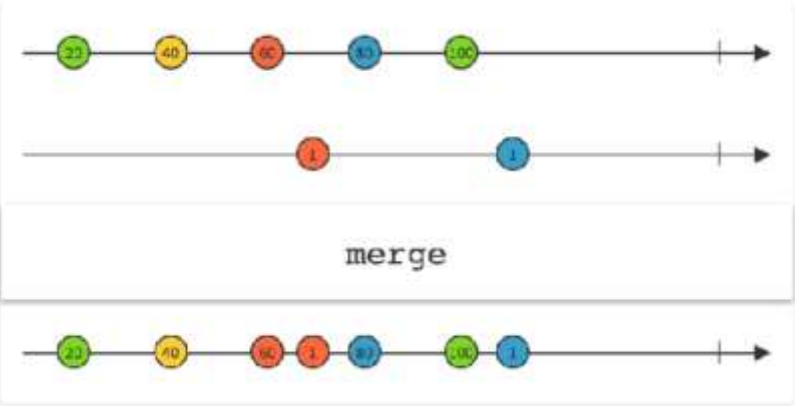
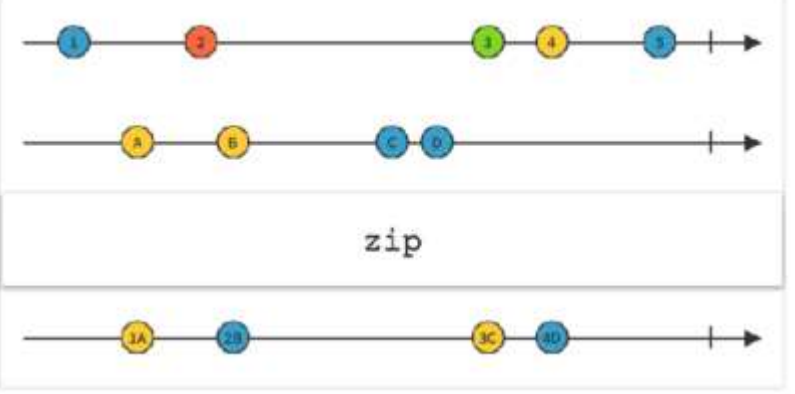
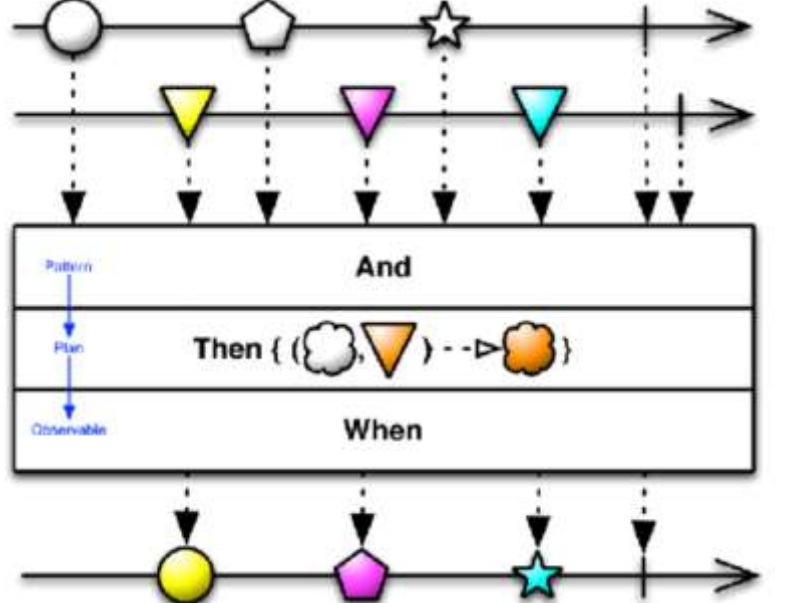
### Diagramma Marble



Transforming Operators	
<p>Operatore MAP</p> <p>Trasforma gli elementi emessi da un osservabile applicando una funzione di mapping</p>	
<p>Operatore FLAT MAP</p> <p>Trasforma gli elementi emessi da un osservabile in osservabili, poi appiattisce le emissioni in un singolo elemento osservabile</p>	
<p>Operatore SCAN</p> <p>Applica una funzione ad ogni elemento emesso da un osservabile, sequenzialmente, ed emette il valore successivo.</p>	
<p>Operatore Buffer: periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time</p> <p>Operatore GroupBy: divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key</p> <p>Operatore Window: periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time</p>	



Filtering Operators	
<div>Operatore FILTER</div> <div>Emette solo gli elementi, da un osservabile, che superano un predicato di test</div>	
<div>Operatore TAKE</div> <div>Emette solo i primi N elementi da un osservabile</div>	
<div>Operatore Skip:</div> <div>Operatore Debounce:</div> <div>Operatore Distinct:</div> <div>Operatore ElementAt:</div> <div>Operatore First:</div> <div>Operatore IgnoreElements:</div> <div>Operatore Last:</div> <div>Operatore Sample:</div> <div>Operatore SkipLast:</div> <div>Operatore TakeLast:</div>	<div>suppress the first n items emitted by an Observable</div> <div>only emit an item from an Observable if a particular timespan has passed without it emitting another item</div> <div>suppress duplicate items emitted by an Observable</div> <div>emit only item n emitted by an Observable</div> <div>emit only the first item, or the first item that meets a condition, from an Observable</div> <div>do not emit any items from an Observable but mirror its termination notification</div> <div>emit only the last item emitted by an Observable</div> <div>emit the most recent item emitted by an Observable within periodic time intervals</div> <div>suppress the last n items emitted by an Observable</div> <div>emit only the last n items emitted by an Observable</div>

<b>Combining Operators</b>	
<b>Operatore MERGE</b>  Combina molteplici osservabili in uno unico cominando tutte le emissioni	
<b>Operatore ZIP</b>  Combina le emissioni da molteplici osservabili utilizzando una specifica funzione per emettere un singolo elemento per ogni combinazione in base ai risultati della funzione	
<b>Operatori AND/THEN/WHEN</b>  Combinano insieme di elementi emessi da due o più osservabili tramite un Pattern and Plan intermedio	
<b>Operatore Join:</b>	combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable
<b>Operatore CombineLatest:</b>	when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function
<b>Operatore StartWith:</b>	emit a specified sequence of items before beginning to emit the items from the source Observable
<b>Operatore Switch:</b>	convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables



## Asynchronous Programming: challenges

Programmazione reattiva o estensioni Reactive non possono essere considerate la soluzione definitiva per la programmazione asincrona, sono soluzioni efficaci per gestire stream di dati o di eventi asincroni con stile funzionale ma non costituiscono un modello di programmazione asincrona general-purpose.

La sfida oggi consiste nell'integrazione di tutti questi approcci e queste tecniche:

- Synch + asynch, push + pull
- Sfruttando la concorrenza (multipli event loop comunicanti, attori)
- Escogitando modelli che funzionano anche nel caso di sistemi distribuiti (reactive distributed programming).

## Java Executors (module-lab-2.1)

Strutturare un programma concorrente in task prevede di gestire:

- il concetto di **task** come unità di lavoro indipendente, astratta, discreta, separata dalla nozione di thread
- il pattern **division of labor** (noto anche come strategia divide-et-impera):
  - identificando i confini dei task (non devono dipendere dallo stato, dai risultati, da effetti collaterali di altri task)
  - scegliendo una politica di esecuzione dei task in base agli obiettivi:
    - miglior throughput
    - miglior responsività
    - grace-full degradation

Gli obiettivi:

- semplificare l'organizzazione del programma
- facilitare il recupero di errori fornendo confini naturali per le transazioni
- promuovere la concorrenza.

## Esecuzione sequenziale dei task

### Esempio: WebServer

L'implementazione con thread singolo:

```
class SingleThreadWebServer {  
  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            Socket connection = socket.accept();  
            handleRequest(connection);  
        }  
    }  
}
```

presenta alcuni problemi:

- scarse performance e basso throughput
- bassa responsività
- basso utilizzo delle risorse.

Un policy del tipo "one-thread-per-task" che crei threads in base alle richieste senza specifiche limitazioni:

```
class ThreadPerTaskWebServer {  
  
    public static void main(String[] args) throws IOException {
```

```
ServerSocket socket = new ServerSocket(80);
while (true) {
    final Socket connection = socket.accept();
    Runnable task = new Runnable() {
        public void run() {
            handleRequest(connection);
        }
    };
    new Thread(task).start();
}
```

presenta i seguenti pro e contro:

- pro
  - aumento della responsività
  - aumento del throughput
  - maggiore utilizzo delle risorse
- contro
  - peso del ciclo di vita del thread: creazione ed eliminazione dei thread non sono operazioni gratuite
  - consumo di risorse: i thread attivi consumano risorse di sistema (memoria)
  - stabilità: esiste un limite fisico al numero di thread che possono essere creati
- rischio di concorrenza:
  - sebbene possa apparire prototipato correttamente, in particolari condizioni di carico intenso potrebbe fallire completamente.

## Politiche più flessibili: l'executor framework

L'astrazione task-oriented è stata introdotta in Java con il JDK 5.0 (**java.util.concurrent**).

Essa fornisce un supporto per separare la sottomissione dei task dalla loro esecuzione: i task sono unità logiche di lavoro eseguite in maniera asincrona dai threads.

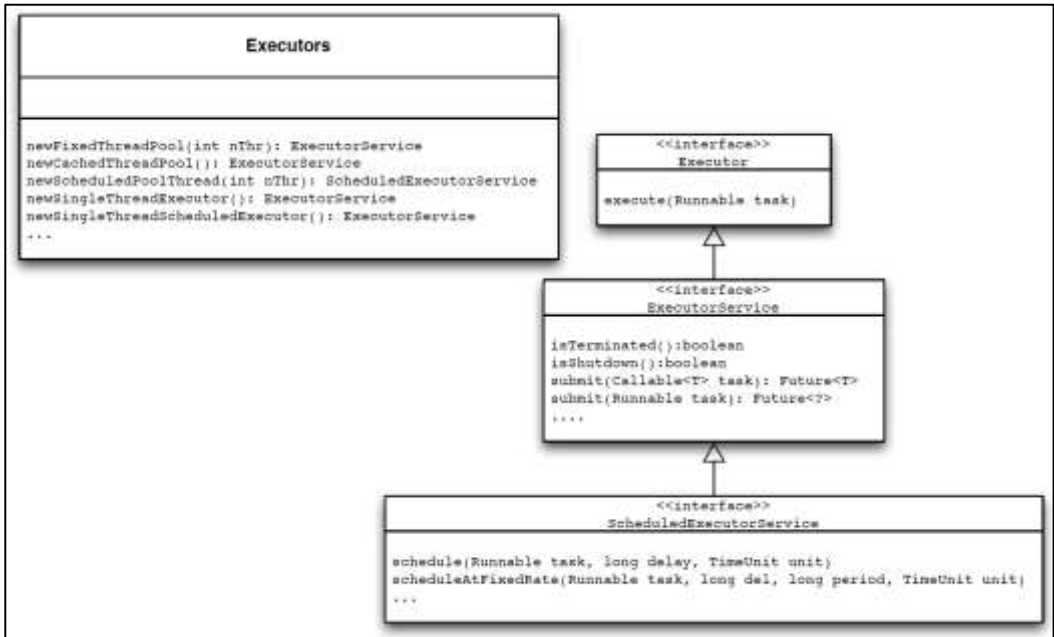
Implementa un pattern di tipo producer-consumer:

- i produttori sono attività che sottomettono i task
- i consumers sono i thread che eseguono i task.

I **task devono essere indipendenti**: un task non può attendere l'esecuzione di un altro task (potrebbe generare deadlock).

L'interfaccia **Executor** utilizza il metodo **execute** per sottomettere i task descritti tramite l'interfaccia **Runnable** (implementando il metodo **run** per definire il comportamento).

La classe **Executors** è un utility da usare per raccogliere dati statistici, effettuare gestione e monitoraggio dell'applicazione; è una factory per gli executors concreti che implementano specifiche politiche di esecuzione.



Le **politiche di esecuzione** riguardano il cosa, il dove, il quando, il come riguarda all’esecuzione dei task:

- quando un task sarà eseguito e da qualche thread?
- in quale ordine i task dovrebbe essere eseguiti (LIFO, FIFO, priorità, ...)?
- quanti task possono essere eseguiti concorrentemente?
- se un task deve essere terminato perché il sistema è sovraccarico, quale task dovrebbe essere scelto come vittima?
- quali azioni dovrebbero essere intraprese prima o dopo l’esecuzione di un task?

Le politiche ottimali dipendono dalle risorse computazionali disponibili e dai requisiti legati alla qualità del servizio.

La gestione del pool di task avviene tramite una coda di lavoro e un thread consumer che implementa il seguente comportamento: loop { request next task todo => execute it => go back and wait for another task }.

Gli esecutori concreti disponibili sono:

FixedThreadPool	<ul style="list-style-type: none"><li>• un thread per ogni task fino al massimo numero fornito in input</li><li>• mantiene costante la dimensione del pool di threads</li></ul> <pre>class TaskExecutionWebServer {     private static final int NTHREADS         = Runtime.getRuntime().availableProcessors() + 1;     private static final Executor exec         = Executors.newFixedThreadPool(NTHREADS);     public static void main(String[] args) throws IOException {         ServerSocket socket = new ServerSocket(80);         while (true) {             Socket connection = socket.accept();             Runnable task = new Runnable() {                 public void run() {                     handleRequest(connection);                 }             };             exec.execute(task);         }     } }</pre> <p>⇒ performance migliori</p>
-----------------	---

	⇒ forte impatto sulla stabilità dell'applicazione (ddos)
CachedThreadPool	<ul style="list-style-type: none"><li>• termina i thread inattivi se il pool size è maggiore della richiesta corrente</li><li>• crea nuovi thread se necessario</li><li>• senza limiti</li></ul>
SingleThreadExecutor	<ul style="list-style-type: none"><li>• singolo thread worker, rimpiazzato se termina in errore</li><li>• garanzia di esecuzione dei task in accordo con l'ordine della coda</li></ul>
ScheduledThreadPool	<ul style="list-style-type: none"><li>• thread pool di dimensioni fissate</li><li>• supporto per l'esecuzione rimandata e periodica</li></ul>

L'interfaccia **ExecutorService** estende l'interfaccia **Executor** definendo delle politiche di shutdown:

- void shutdown();
- List<Runnable> shutdownNow();
- boolean isShutdown();
- boolean isTerminated();
- boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;

Gli stati dell'**Executor** sono:

- **running**
- **shutting down**
  - il metodo shutdown è utilizzato per il graceful shutdown:
    - non sono accettati nuovi task
    - i task in esecuzione sono autorizzati a completare
  - il metodo shutdownNow è utilizzato per il shutdown brusco:
    - tenta di cancellare i tasks in esecuzione
    - non avvia alcun task accodato
  - è possibile gestire le attività dopo l'arresto dei task
- **terminated**: quanto tutti i task hanno completato la loro esecuzione.

```
class WebServerLifeCycle {  
  
    private static final ExecutorService exec = ...;  
  
    public void start() throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (!exec.isShutdown()) {  
            Socket conn = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(conn);  
                }  
            };  
            try {  
                exec.execute(task);  
            } catch (RejectedExecutionException ex) {  
                if (!exec.isShutdown()) {  
                    log("task submission rejected", ex);  
                }  
            }  
        }  
    }  
  
    public void stop() {  
        exec.shutdown();  
    }  
}
```

```
}

void handleRequest(Socket connection){
    Request req = readRequest(connection);
    if (isShutdownRequest(req)) {
        stop();
    } else {
        dispatchRequest(req);
    }
}
}
```

## Callable task

L'ExecutorService può gestire dei "callable" tasks che rappresentano una computazione differita che completa con un qualche tipo di risultato (call al posto di run).

Il metodo **call** incapsula il comportamento del task effettuando un qualche tipo di computazione e restituendo un qualche tipo di valore (oggetto).

Il metodo **submit** permette di sottomettere un callable task ed ottenere un oggetto di tipo **future** tramite il quale è possibile gestire il risultato del task.

L'interfaccia **Future** fornisce i metodi per verificare quando un task ha completato la propria esecuzione o quando è stato cancellato e per gestire i risultati e cancellare il task.

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException,
        CancellationException();
    V get(long timeout, TimeUnit unit) throws InterruptedException,
        ExecutionException, CancellationException();
}
```

Un'attività è cancellabile se il codice esterno può completare prima della suo normale termine, principalmente a causa di:

- richieste di cancellazione da parte dell'utente (GUI)
- raggiungimento di un certo limite di tempo massimo di esecuzione
- eventi applicativi (es. a seguito dell'individuazione di una prima soluzione valida)
- errori
- chiusura dell'applicazione.

## Approcci asincrono e cooperativo

L'approccio asincrono determina prelazione e problemi di sicurezza legati alla consistenza degli stati (le API nella classe Thread sono deprecate).

L'approccio **cooperativo** (sincrono) prevede la segnalazione di richieste di stop, la gestione delle richieste (cancellation flag) e la definizione delle politiche di cancellazione per ogni task (come altro codice può richiedere la cancellazione, quando il task controlla se è stata richiesta una cancellazione e quali azioni il task deve effettuare in risposta ad una richiesta di cancellazione).

```
public class PrimeGenerator implements Runnable {

    private final List<BigInteger> primes = new ArrayList<BigInteger>();

    private volatile boolean cancelled;
```

L'attributo **volatile** deve essere utilizzato quando un campo può essere acceduto concorrentemente da diversi thread.

```
public void run() {
    BigInteger p = BigInteger.ONE;
    while (!cancelled) {
        p = p.nextProbablePrime();
        synchronized (this) {
            primes.add(p);
        }
    }
}

public void cancel() { cancelled = true; }

public synchronized List<BigInteger> get() {
    return new ArrayList<BigInteger>(primes);
}
}

List<BigInteger> aSecondOfPrimes() throws InterruptedException {

    PrimeGenerator gen = new PrimeGenerator();

    new Thread(gen).start();

    try {
        SECONDS.sleep(1);
    } finally {
        gen.cancel();
    }
    return gen.get();
}

class StuckedPrimeProducer extends Thread {

    private BoundedBuffer buffer;

    private volatile boolean cancelled;

    public StuckedPrimeProducer(BoundedBuffer buf) {
        buffer = buf;
        cancelled = false;
    }

    public void run() {
        BigInteger p = BigInteger.ONE;
        while (!cancelled) {
            BigInteger value = p.nextProbablePrime();
            buf.put(value);
        }
    }

    public void cancel() { cancelled = true; }
}
```

## Interruption

Per evitare blocchi infiniti è possibile agire sui thread interrompendone l'esecuzione.

Le politiche di interruzione devono essere applicate ad ogni thread: si parla di idiomi di gestione delle interruzioni.

Idioma comune:

- explicit shutdown state
- cancel method
- interruption check in loop
- management of InterruptedException

Esempio di interruption policy.

```
class PrimeProducer extends Thread {

    private BlockingQueue<BigInteger> queue;

    public PrimeProducer(BlockingQueue<BigInteger> q) {
        queue = q;
    }

    public void run() {
        BigInteger p = BigInteger.ONE;
        try
            while (!Thread.currentThread().isInterrupted()) {
                BigInteger value = p.nextProbablePrime();
                queue.put(value);
            } catch (InterruptedException ex) {
                // allow thread to exit
            }
        }

    public void cancel() {
        interrupt();
    }
}
```

## Executors shutdown

Esistono due modalità per interrompere l'ExecutorService:

- il metodo **shutdown** è utilizzato per un'interruzione "coordinata":
  - non vengono accettati nuovi task
  - si attende la terminazione dei task in corso o in coda di sottomissione
- il metodo **shutdownNow** è utilizzato per un'interruzione "brusca":
  - tenta di cancellare tutti i task in corso
  - non avvia nessun task nemmeno quelli non ancora entrati in esecuzione.

I task sottomessi dopo un'operazione di shutdown vengono gestiti tramite un reject execution handler.

### Esempio "LifecycleWebServer"

```
class LifecycleWebServer {
    private final ExecutorService exec = ...
    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                Socket sock = conn.accept();
                exec.execute(new Runnable() {
```

```
        public void run(){ handleRequest(conn); }
    });
} catch (RejectedExecutionException ex){
    if (!exec.isShutdown()){
        log("task submission rejected.");
    }
}
}

public void stop(){ exec.shutdown(); }

void handleRequest(Socket conn){
    Request req = readRequest();
    if (isShutdownRequest(req)) {
        stop();
    } else { dispatchRequest(req); }
}
```

Il web server può essere terminato in due modi:

1. Tramite il metodo stop()
2. Tramite una richiesta client che invia una richiesta formattata in modo speciale

L'ExecutorService fornisce il metodo **awaitTermination** che permette di attendere il completamento dei task prima di effettuare lo shutdown.

Esempio: LogService

```
public class LogService {

    private final ExecutorService exec = Executors.newSingleThreadExecutor();
    private final PrintWriter writer = ...

    public void stop() throws InterruptedException {
        try {
            exec.shutdown();
            exec.awaitTermination(Long.MAX_VALUE, TimeUnits.SECONDS);
        } finally {
            writer.close();
        }
    }

    public void log(String msg){
        try {
            exec.execute(new WriteTask(msg));
        } catch (RejectedExecutionException ignored){}
    }
}
```

La tecnica denominata **Poison Pills** utilizza oggetti posizionati in strutture dati condivise per segnalare la terminazione di alcuni processi o attività.

```
class Consumer extends Thread {

    private BoundedBuffer buffer;
    private boolean shutdown;

    public Consumer(BoundedBuffer buf){
        buffer = buf;
        shutdown = false;
    }

    public void run(){
```



```
while (!shutdown) {
    try {
        Item item = buffer.get();
        if (!item.getDescr().equals("stop")) {
            // process item
        } else {
            shutdown = true;
        }
    } catch (InterruptedException ex) {
    }
}

public void shutdown() {
    shutdown = true;
    interrupt();
}
```

## Algoritmi map-reduce, il pattern Fork-Join

Molti algoritmi di programmazione parallela richiedono che i tasks creino sotto-tasks e comunichino tra loro per completare.

L'algoritmo map-reduce prevede due fasi:

- una fase di map che divide il problema in problemi più piccoli e indipendenti
- una fase di reduce che raccoglie i risultati delle elaborazioni effettuate nella fase di map e consolida i risultati finali.

Mentre gli Executors possono essere applicati facilmente quando la topologia dello spazio dei dati è conosciuta, gli algoritmi map-reduce sono più adatti nei casi in cui essa viene esplorata e "conosciuta" durante l'esecuzione.

Questo tuttavia genera dipendenze tra task, quando un Callable attende il risultato di un altro Callable, si trova in uno stato di attesa ed elimina la possibilità di accodare un altro Callable per l'esecuzione, sprecando tempo e risorse.

Il framework Fork-Join è stato introdotto nel package `java.util.concurrent` in Java SE 7 per rendere effettivi gli algoritmi di tipo map-reduce.

Un nuovo tipo di executor è il **ForkJoinPool**, dedicato ad eseguire istanze che implementano **ForkJoinTask**.

Gli oggetti che implementano **ForkJoinTask** supportano la creazione di sotto-task e l'attesa che i sotto-task terminino; l'executor è in grado di distribuire i task tra i thread al suo interno "rubando" risorse operative quando un task è in stato di attesa.

L'API **ForkJoinTask** contiene:

- il metodo **fork()** per sottomettere un nuovo **ForkJoinTask** da uno esistente
- il metodo **join()** per attendere il suo completamento
- il metodo **invokeAll(Collection<? Extends Callable<T>> tasks)** per lanciare ed attendere il completamento di una serie di **ForkJoinTasks**.

Esistono due specializzazioni di **ForkJoinTask**:

- **RecursiveAction**: esecuzioni che non ritornano un valore
- **RecursiveTask**: esecuzioni che ritornano un valore.

## Bibliografia

Rance Cleaveland, S. S. (1996, December). Strategic Directions in concurrency Research. *ACM Compunting Surveys*, 28(4).

Roscoe, A. W. (1997). The Theory and Practice of Concurrency. *ISBN 0-13-674409-5*.

## Message Passing Models (module 3.1)

L’idea alla base di questi sistemi risiede nel fatto che l’unico modo per abilitare l’interazione tra i processi è lo scambio di messaggi (unicamente tramite le primitive **send** e **receive** e senza utilizzare ulteriori astrazioni come oggetti condivisi, monitors, locks, semafori o altro).

Il modello, solitamente utilizzato per la programmazione distribuita, viene considerato anche nella programmazione concorrente in generale in linguaggi, framework e tecnologie ampiamente diffuse per evitare le insidie introdotte dalla programmazione multi-threaded e i relativi meccanismi di sincronizzazione e mutua esclusione. Il modello Message Passing è implementato in HTML5 tramite i Web Workers, nei frameworks ad attori, nei linguaggi più recenti (Go, DART, ...).

Le prime origini di questo modello risalgono al 1970 quando Brinch Hansen introdusse i messaggi asincroni progettati per i sistemi operativi del computer Danish RC4000. Nel 1971 Bob Balzer introdusse la nozione di porte di comunicazione che si trovano alla base del modello attuale. La comunicazione sincrona è stata introdotta da Hoare nel 1978 attraverso il formalismo CSP. Nel 1973 Carl Hewitt e il suo studente Gul Agha, nell’anno successivo, introdussero il modello ad attori basato sul modello message passing asincrono. Recentemente si è esplorata la possibilità di utilizzo del modello nella progettazione e implementazione dei sistemi operativi; sistemi multi-core e strategie many-code determinano la possibilità di utilizzare questo modello come alternativa al classico multi-thread.

### Primitive di base per la comunicazione

<b>Tipi di canale</b>
<b>chan ch(type id1, ..., type idn)</b> <ul style="list-style-type: none"><li>ch è il nome del canale</li><li>type e id sono i tipi e i nomi dei campi del messaggi trasmessi tramite il canale</li><li></li></ul>
<b>Primitive di comunicazione</b>
<b>send ch(expr1, expr2, ...)</b> Invio di messaggio composto dalle espressioni expr1, expr2, ...; tali espressioni devono avere tipo compatibile con i corrispondenti campi della dichiarazioni di ch. <b>receive ch(var1, var2, varn)</b> Ricezione di un messaggio dal canale ch; le var1, var2, ..., devono avere tipo compatibile con i corrispondenti campi della dichiarazione

L’accesso al contenuto di ogni canale è atomico; i canali sono globali.

### Comunicazione sincrona vs comunicazione asincrona

Nella **comunicazione sincrona** l’invio di un messaggio su un canale rimane bloccato finché il messaggio non viene ricevuto sullo stesso canale, fin quando un messaggio è inserito nel canale la ricezione è bloccata.

Nella **comunicazione asincrona** i canali utilizzano un buffer FIFO in cui i messaggi sono accodati, l’invio di un messaggio avviene non appena il messaggio viene accodato nel canale, la ricezione è bloccata finché un messaggio è disponibile nel canale.

### Schemi di comunicazione

Il più semplice schema di comunicazione è il modello **one-to-one**: un canale può essere utilizzato solo da una coppia di processi, tipicamente in maniera sincrona.

Un modello **many-to-many**, più generale, utilizza uno stesso canale per molteplici processi che inviano e ricevono messaggi; la ricezione dei messaggi è competitiva e non deterministica.

Lo schema **many-one**, infine, prevede che un canale permetta a molteplici processi (senders) di inviare messaggi ad un unico processo (receiver), tipico utilizzo tramite porte.

Esempio.

<pre> chan requestA(int value) chan requestB(int value) chan requestC(int value)         </pre>		
P	Q	R
<pre> r1, r2: integer ... send requestA(5) send requestB(6) receive response(r1) receive response(r2) write(r1+r2) ...         </pre>	<pre> r: integer ... receive requestA(r) send response(r*2) ...         </pre>	<pre> r: integer ... receive requestB(r) send response(r+1) ...         </pre>

Esempio: producer and consumer, interazione tramite un singolo canale

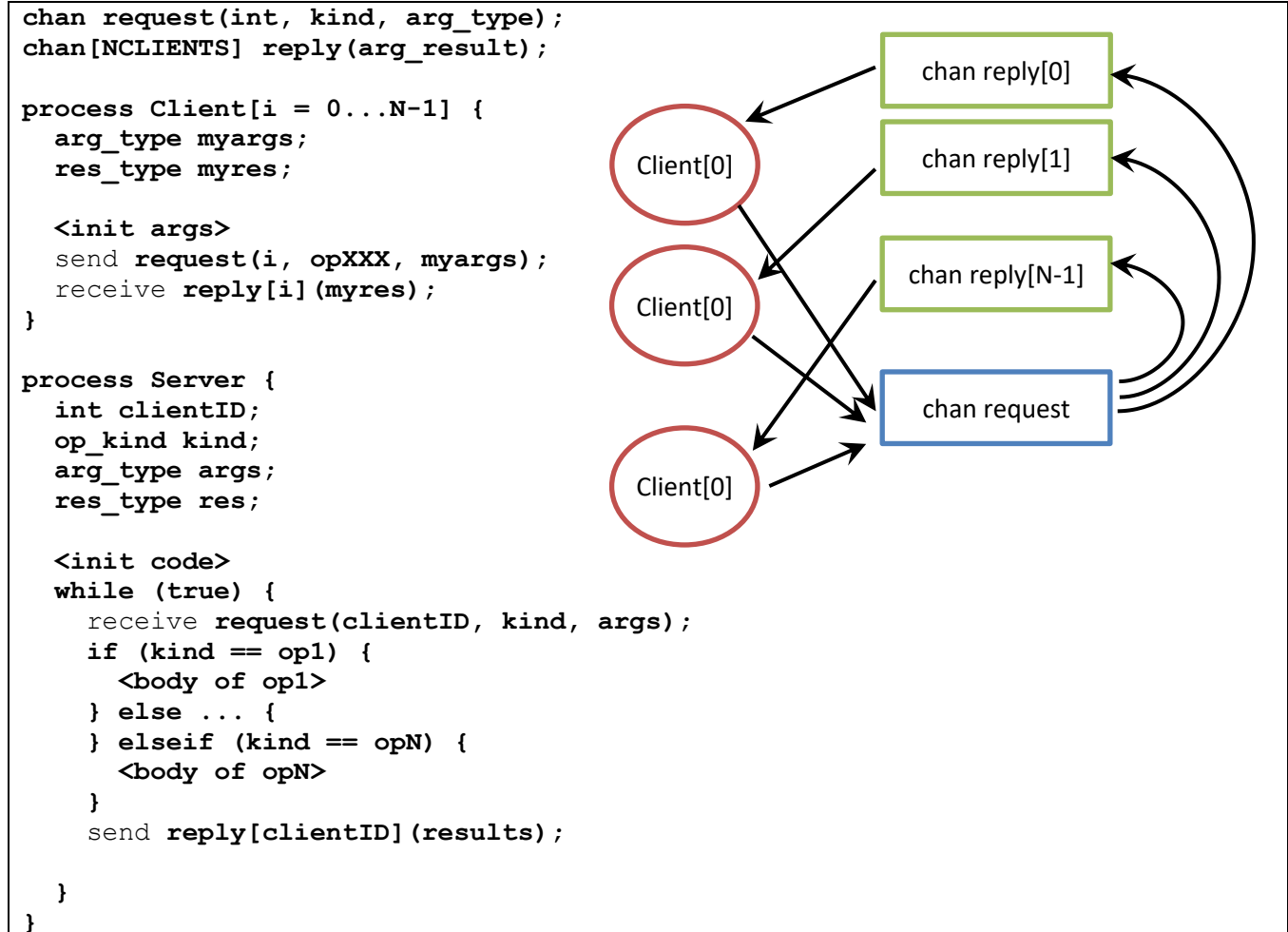
<pre> channel buf(int)         </pre>	
PRODUCER	CONSUMER
<pre> integer x  loop forever: p1: x ← produce p2: send buf(x)         </pre>	<pre> integer y  loop forever: q1: receive buf(y) q2: consume(y)         </pre>

Esempio: una PIPELINE, un processo che assembla linee di caratteri

<pre> chan input(char) , output(char[MAXLINE])  process CharToLine {     char line[MAXLINE+1];     int i = 0;     while (true) {         receive input(line[i]);         while (line[i]!=CR and i&lt;MAXLINE) {             i = i + 1;             receive input(line[i]);         }         line[i] = EOL;         send output(line);         i = 0;     } }         </pre>
--

### Esempio: interazione client-server

Il server processa le richieste in attesa nel canale di richiesta e fornisce le risposte su un canale dedicato; agisce come un **active monitor**, tipicamente utilizzato per implementare processi allocatori di risorse.



### Esempio: Allocatore di risorse passivo e attivo

MONITOR BASED	PROCESS BASED
<pre> monitor ResAllocator {     int avail = MAXUNITS;     set units = &lt;initial value&gt;;     cond free;      procedure res_id acquire() {         res_id id;         if (avail==0) {             waitC(free);         } else {             avail = avail - 1;         }         iId = remove(units);         return id;     }      Procedure release(res_id id) {         insert(units,id);         if (emptyC(free)) {             avail = avail + 1;         } else {             signalC(free);         }     } }         </pre>	<pre> chan request(int clientID, int type); chan[N] reply(res_id id);  process ResAllocator {     int clientID;     int avail = MAXUNITS;     queue pending; #initially empty     set units = &lt;initial value&gt;;     op_kind kind;     res_id id;      while(true) {         receive request(clientID,kind);         if (kind==ACQUIRE) {             if (avail&gt;0) {                 avail = avail - 1;                 id = remove(units);                 send reply[clientID](id);             } else {                 insert(pending,clientID);             }         } elseif (kind==RELEASE) {             if (empty(pending)) {                 insert(units,id);                 avail = avail + 1;             } else {                 remove(pending,clientID);                 send reply[clientID](id);             }         }     } }         </pre>
	<pre> process Client[i = 0...N-1] {     arg_type myargs;     res_id id;     ...     send request(i,ACQUIRE);     receive reply[i](id);     #...use resource     send request(i,RELEASE); }         </pre>
DIFFERENZE	
Permanent variables	Local server variable
Procedure identifiers	Request channel and op kinds
Procedure call	Send request(...) Receive reply(...)
Monitory entry	Receive request(...)
Procedure return	Send reply(...)
Wait statement	Save pending requests
Signal statement	Retrieve and process pending requests
Procedure bodies	Cases in which statement on op kind

### Comunicazione utilizzando condizioni “guardia”

Dijkstra, nel 1974, introdusse l'utilizzo delle guardie per affrontare il problema di come riceve messaggi che possono arrivare da molteplici canali contemporaneamente, sia nel caso sincrono che in quello asincrono.

Il meccanismo realizza una ricezione selettiva utilizzando un'istruzione *select*.

Nell'istruzione:  $B ; C \rightarrow S$

- B è un'espressione booleana, se omessa il suo valore è sempre vero
- C è un'istruzione di comunicazione (tipicamente receive)
- S è un blocco o una lista di istruzioni.

Insieme, B e C costituiscono ciò che è chiamato **guardia**:

- Una guardia **ha successo** se l'espressione B è vera e l'esecuzione di C non genera un ritardo
- Una guardia **fallisce** se l'espressione B è falsa
- Una guardia **blocca** se l'istruzione C non può ancora essere eseguita.

Istruzione IF	Semantica
<pre>if B1;C1 → S1; [] B2;C2 → S2; [] B3;C3 → S3; fi</pre>	<p>Primo: valutazione delle espressioni booleane della guardia</p> <ul style="list-style-type: none"><li>• Se tutte sono false, l'if termina senza effetto</li><li>• Se almeno una è vera, se ne sceglie in maniera non deterministica tra quelle vere</li><li>• Se tutte le guardie bloccano, si attende finché una guardia possa eseguire</li></ul> <p>Secondo: si esegue l'istruzione C per la guardia scelta</p> <p>Terzo: si esegue il blocco S.</p>
Esempio	<pre>if nReqs &lt; max; receive computeSum(a,b,i) → nReqs+=1; send result[i] (a+b) [] nReqs &lt; max; receive computeMul(a,b,i) → nReqs+=1; send result[i] (a*b) fi</pre>

Istruzione LOOP	Semantica
<pre>do B1;C1 → S1; [] B2;C2 → S2; [] B3;C3 → S3; od</pre>	<p>In questo caso il processo di selezione viene ripetuto fino a che tutte le guardie falliscono.</p>
Esempio	<pre>process Copy(chan in(char), chan out(char)) {   char buffer[10];   int front = 0, rear = 0, count = 0;   do count &lt; 10; receive in(buffer[rear]);     → count++; rear = (rear+1)%10;   [] count &gt; 0; send out(buffer[front])     → count--; front = (front+1)%10; }</pre>

## Producers and consumers

<pre> process BoundedBufferManager {     int nItems = 0;     int maxElems = ...;     Queue&lt;ItemType&gt; queue = ...;     ItemType item;     boolean ack = true;     chan replyChan;      do nItems &lt; maxElems; receive put(item,replyChan)         → queue.add(item);         nItems++;         send replyChan(ack);     [] nItems &gt; 0; receive get(replyChan);         → ItemType el = queue.remove();         nItems--;         send replyChan(el);     od }         </pre>	
<pre> Process Producer(chan myChan) {     boolean ack;      loop {         ItemType el = produce();         send put(el,myChan);         receive myChan(ack);     } }         </pre>	<pre> Process Consumer(chan myChan) {     loop {         ItemType el;         send get(myChan);         receive myChan(el);         consume(el);     } }         </pre>

## Interazione peer-to-peer

In questo modello la responsabilità dell'interazione è decentralizzata e delegata ai diversi elementi/processi (peers): la coordinazione è realizzata tramite protocolli di interazione per lo scambio di messaggi tra pari.

Ad esempio, nell'"exchanging values problem" un valore intero  $v$  è memorizzato in ognuno degli  $N$  processi del sistema; l'obiettivo di ogni processo è di individuare il più piccolo e il più grande degli  $N$  valori distribuiti.

Di seguito vengono proposte tre tipi di soluzione:

- **Centralizzata:** un processo si occupa di coordinare l'azione, ricevere i valori da tutti gli altri e determinare il valore minimo e massimo
- **Simmetrica:** ogni processo esegue lo stesso algoritmo e determina in parallelo il valore minimo e il valore massimo
- **Ad anello:** i processi si coordinano e organizzano in un anello logico, ogni processo  $P[i]$  riceve messaggi dai suoi predecessori e invia messaggi ai suoi successori; la soluzione si sviluppa in due fasi: determinazione del valore minimo e massimo e diffusione dell'informazione a tutti i processi



<p>Soluzione Centralizzata</p> <pre> chan values(int), results[n](int smallest, int largest);  process P[0] { #coordinator     int v = ...;     int new; smallest = v; largest = v; #initial state      for i in [1...n-1] {         receive values(new);         if (new &lt; smallest) smallest = new;         if (new &gt; largest) largest = new;     }      for i in [1...n-1] { #send results to other processes         send results[i](smallest, largest);     } }  process[i] {     int v = ...; smallest; largest;     send values(v);     receive results[i](smallest, largest); } </pre>	<p>Caratteristiche:</p> <ul style="list-style-type: none"> <li>• Basso numero di messaggi</li> <li>• Collo di bottiglia nel processo coordinatore</li> <li>• Ricezione ritardata sul coordinatore</li> </ul>
<p>Soluzione simmetrica</p> <pre> chan values[n](int);  process P[i=0 to n-1] {     int v = ...;     int new; smallest = v; largest = v; #initial state      for j in [1...n-1], j!=i {         send values[j](v);     }      for k in [1...n-1] {         receive values[i](new);         if (new &lt; smallest) smallest = new;         if (new &gt; largest) largest = new;     } } </pre>	<p>Caratteristiche:</p> <ul style="list-style-type: none"> <li>• Alto numero di messaggi <math>N*(N-1)</math></li> <li>• Massimizzazione della concorrenza</li> </ul>
<p>Soluzione ad anello</p> <pre> chan values(int), result[n](int smallest, int largest);  process P[0] { #coordinator     int v = ...;     int new; smallest = v; largest = v; #initial state     send values[1](smallest, largest);     receive values[0](smallest, largest);     send values[1](smallest, largest); }  process P[i=1 to n-1] {     int v = ...;     int smallest; largest;     receive value[i](smallest, largest);     if (v &lt; smallest) smallest = v;     if (v &gt; largest) largest = v;     send values[(i+1)%n](smallest, largest);     receive values[i](smallest, largest);     send values[(i+1)%n](smallest, largest); } </pre>	<p>Caratteristiche:</p> <ul style="list-style-type: none"> <li>• Basso numero di messaggi</li> <li>• Concorrenza limitata</li> </ul>

## I filosofi a cena

Di seguito si affronta il problema dei filosofi a cena in un ambiente distribuito:

- Ogni filosofo è eseguito in uno specifico nodo della rete
- Le “forchette” (risorse) sono anch’esse distribuite.

Il problema in generale consiste nel risolvere i conflitti tra processi in un sistema distribuito.

### Soluzione centralizzata

Si utilizza un processo “cameriere” che agisce come allocatore di risorse (le posate), è responsabile di evitare deadlock e garantire fairness.

```
chan getForks(int,int,chan);
chan releaseForks(int,int,chan);

process Waiter[i:0...N-1] {
  List<Request> pending = ...;
  boolean availForks[0...N-1] = {false, ...};

  do receive getForks(fork1, fork2, ReplyChanID) {
    if (availForks[fork1] && availForks[fork2]) {
      availForks[fork1] = false;
      availForks[fork2] = false;
      send ReplyChanID(fork1, fork2);
    } else {
      Pending.add(new Request(Reply, fork1, fork2));
    }
  }

  [] receive releaseFork(fork1, fork2) {
    availForks[fork1] = true;
    availForks[fork2] = true;
    foreach (Request r in pending) {
      if (availForks[r.fork1] && availForks[r.fork2]) {
        pending.remove(r);
        availForks[r.fork1] = true;
        availForks[r.fork2] = true;
        send req.ReplyChanID(req.fork1, req.fork2);
      }
    }
  }

  od
}

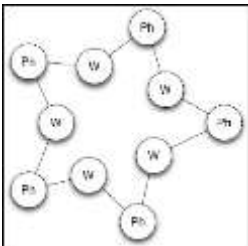
process Philosopher[i:0...N-1, chan reply]{
  int first = i;
  int second = (i+1)%N;

  loop {
    think();
    send getForks(first, second, reply);
    receive reply(first, second);
    eat();
    send releaseForks(first, second);
  }
}
```

Soluzione distribuita

Si introduce un processo cameriere che gestisce ogni forchetta (risorsa) che funge da allocatore di risorsa.

Si adotta un appropriato protocollo di coordinazione tra i filosofi e i camerieri per evitare deadlocks seguendo una strategia gerarchica.



```
chan getFork[0...N-1];
chan getForkReply[0...N-1];
chan releaseFork[0...N-1];
chan releaseForkReply[0...N-1];

process Waiter(i:0...N-1) {
  loop {
    receive getFork[i] ();
    send    getForkReply[i] ();
    receive releaseFork[i] ();
    send    releaseForkReply[i] ();
  }
}

process Philosopher[i...N-1] {
  int first = i; second = (i+1)%N;

  if (second < first) {
    first = second;
    second = i;
  }

  loop {
    think();

    send    getFork[first] ();
    receive getForkReply[first] ();
    send    getFork[second] ();
    receive getForkReply[second] ();

    eat();

    send    releaseFork[first] ();
    receive releaseForkReply[first] ();
    send    releaseFork[second] ();
    receive releaseForkReply[second] ();

  }
}
```

Synchronous Message Passing

La semantica **sincrona** per il passaggio di messaggi definisce che la send blocchi l’esecuzione finché il messaggio non viene ricevuto sul canale (non richiede la presenza di buffers).

SENDER	RECEIVER
...	...
p(i): send ch(msg)	q(j): receive ch(msg)
...	...

La comunicazione è un’interazione atomica: i dati sono trasferiti da una canale ch che si attiva solo quando il control pointer del sender raggiunge l’istruzione send e il control pointer del receiver si trova sull’istruzione receive.

Esempio producer-consumer

chan ch(int)	
PRODUCER	CONSUMER
x: integer	y: integer
loop forever:	loop forever:
p1: x ← produce	q1: receive ch(y)
p2: send ch(x)	q2: consume(y)

Grazie alla semantica sincrona, i dati vengono trasferiti solo quando il control pointer del PRODUCER è su P2 e del CONSUMER è su Q1.

Linguaggio GO: comunicazione sincronizzata

[https://golang.org/doc/effective\\_go.html#sharing](https://golang.org/doc/effective_go.html#sharing)

GO (golang) è un linguaggio a tipizzazione statica sviluppato inizialmente dagli Robert Griesemer, Rob Pike e Ken Thompson per Google nel 2007.

La sintassi è derivata da quella del C, con aggiunta di garbage collection, type safety, qualche funzionalità di tipizzazione dinamica, tipi nativi addizionali – come ad esempio array a lunghezza variabile e mappe chiave-valore – e disponibilità di molte librerie standard. GO è un linguaggio molto veloce. È stato sviluppato con un insieme di primitive per la concorrenza: processi light-weight (goroutines), channels e l’istruzione select.

GO incoraggia un approccio per cui le i valori condivisi (da condividere) viaggiano in un canale mai condiviso da thread separati. Solo la **goroutine** ha accesso al valore in un qualsiasi dato momento, evitando per definizione la competizione per l’accesso al dato. Questo concetto è sintetizzato nello slogan: “Do not communicate by sharing memory; instead, share memory by communicating”.

Si può pensare a questo modello considerandolo un tipico approccio single-threaded, per cui un unico programma in esecuzione nella CPU non ha bisogno di primitive di sincronizzazione. Aggiungendo un altro processo, anch’esso non ha bisogno di sincronizzazione. Quando questi due processi devono comunicare, se la comunicazione è fatta attraverso un elemento di sincronizzazione non servo altri supporti. Questo elemento di sincronizzazione risiede negli Hoare’s Communicating Sequential Processes (CSP).

Rendez-vous

Si tratta di un modello di sincronizzazione estesa in cui un processo sender aspetta non solo che il receiver riceva la richiesta ma anche che venga consegnata una risposta con un risultato. Il nome *rendez-vous* richiama l’immagine di due persone che scelgono un luogo dove incontrarsi: la prima persona che arriva deve attendere l’arrivo della seconda.

Si definiscono due ruoli:

- **Processi chiamanti (calling processes)**
  - Chiamano un *entry* di un processo accettante
  - Devono conoscere l’identità del processo accettante e il nome dell’*entry*
  - Utilizzano la primitiva **call**
- **Processi accettanti (accepting processes)**
  - Accettando chiamate su specifiche *entries* e inviano risultati in risposta.

CLIENT	SERVER
integer param, result	integer p, r
loop forever	loop forever
p1: param ← ...	q1: accept service(p,r)
p2: call server.service(param,result)	q2: <do the service with p>
p3: use(result)	q3: r ← <result>

Rendez-Vous in ADA

Ada è un linguaggio di programmazione sviluppato verso la fine degli anni settanta su iniziativa del Dipartimento della Difesa (DOD) degli Stati Uniti.

Ada combina principi e tecniche provenienti da diversi paradigmi di programmazione, in particolare programmazione modulare, programmazione orientata agli oggetti, programmazione concorrente e calcolo distribuito. Sebbene l'interesse del DOD vertesse principalmente sullo sviluppo di applicazioni militari, Ada è un linguaggio general purpose che si presta all'utilizzo in qualsiasi dominio applicativo. L'origine militare si rivela però nella presenza di caratteristiche fortemente orientate alla sicurezza del codice; per questo motivo, il linguaggio viene ancora oggi usato in molti contesti in cui il corretto funzionamento del software è critico, come astronautica, avionica, controllo del traffico aereo, finanza e dispositivi medici.

I compilatori Ada impiegati per lo sviluppo di software mission-critical devono seguire un processo di certificazione secondo lo standard internazionale ISO/IEC 18009 (Ada: Conformity Assessment of a Language Processor), implementato nella suite Ada Conformity Assessment Test Suite (ACATS), parte integrante del processo di certificazione svolto da laboratori autorizzati dalla Ada Compiler Assessment Authority (ACAA).

Il nome iniziale del linguaggio doveva essere DOD-1, ma venne in seguito cambiato in Ada in onore di Ada Lovelace, illustre matematica dei primi anni del XIX secolo, accreditata come la prima programmatrice della storia per aver sviluppato un algoritmo per il calcolo dei numeri di Bernoulli sulla macchina analitica di Charles Babbage.

Ada offre un supporto diretto per chiamate tramite *entry* come meccanismo di comunicazione di base tra i tasks. L'istruzione *accept* è utilizzata nell'implementazione delle entries dichiarate nell'interfaccia pubblica del task.

Calling Task	Called Task
<pre>: T.E(actuals) --calls the rendezvous</pre>	<pre>task T;   entry E(formals);   : --declaration begin   :     accept E(formals);       : -- body of rendezvous     End accept;   : end;</pre>

Esempio

Codice ADA
<pre>with Ada.Text_IO; use Ada.Text_IO;  procedure HotDog is    task Gourmet is     entry Make_A_Hot_Dog;   end Gourmet;    task body Gourmet is   begin     Put_Line("I am ready to make a hot dog for you");     for Index in 1..4 loop       accept Make_A_Hot_Dog do         delay 0.8;         Put("Put hot dog in bun ");         Put_Line("and add mustard");       end Make_A_Hot_Dog;     end loop;     Put_Line("I am out of hot dogs");   end Gourmet;  begin   for Index in 1..4 loop     Gourmet.Make_A_Hot_Dog;     delay 0.1;     Put_Line("Eat the resulting hot dog");     New_Line;   end loop;   Put_Line("I am not hungry any longer"); end HotDog;</pre>
Risultato dell'esecuzione
<pre>-- Result of execution I am ready to make a hot dog for you Put hot dog in bun and add mustard Eat the resulting hot dog Put hot dog in bun and add mustard Eat the resulting hot dog Put hot dog in bun and add mustard Eat the resulting hot dog Put hot dog in bun and add mustard I am out of hot dogs Eat the resulting hot dog I am not hungry any longer</pre>

Remote Procedure Call (RPC)

Il modello message passing viene utilizzato per realizzare l'astrazione di chiamata a procedura remota nei contesti distribuiti abilitando un client a richiedere un servizio da un server che potrebbe essere eseguito in processori differenti: il client chiama una procedura sul server effettuando una normal chiamata, viene creato un processo per gestire la chiamata. A differenza dei rendez-vous, RPC prevede la partecipazione attiva dei due processi in una comunicazione sincrona.

Alcuni esempi: JAVA RMI (Java Remote Method Invocation), CORBA (Common Object Request Broker), ADA con Distributed Systems Annex (libreria).

## Implementazioni esistenti per message passing basato to canali

### Message Passing Interface (MPI)

Libreria di routine per message passing, proposta come metodologia standard da una larga platea di vendors, sviluppatori e utenti. I programmi distribuiti che sono scritti in linguaggi sequenziali (C, Fortran, Java, ...), comunicano e si sincronizzano chiamando funzioni nella libreria MPI.

### Parallel Virtual Machine (PVM)

Si tratta di un package software che permette l'utilizzo, come fosse una sola macchina, di una collezione eterogenea di computers collegati tra loro in rete.

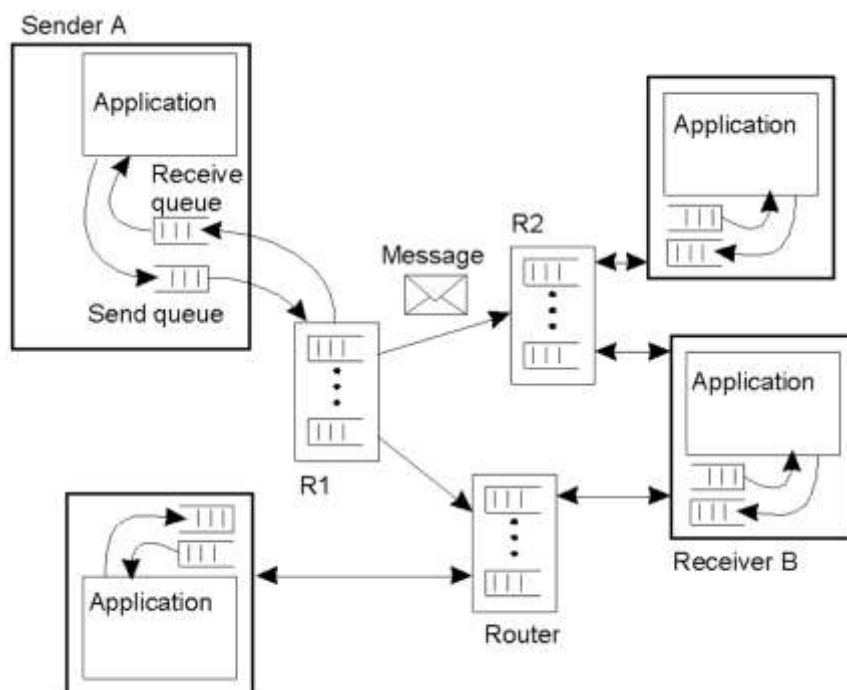
### Middleware orientato ai messaggi (MOM)

Questo modello abilita una comunicazione basata su messaggi tra applicazioni eterogenee e distribuite. Utilizza una comunicazione asincrona, basata su code/argomenti, punto-punto e tramite pattern publish/subscribe. È largamente utilizzata nelle architetture orientate ai servizi.

Esiste un gran numero di implementazioni: WebSphere MQ, TIBCO, SonicMQ, Herald, Hermes, SIENA, Gryphon, JEDI, REBECCA, OpenJMS...

Esistono anche degli standard proposti:

- **Advanced Message Queuing Protocol (AMQP)**  
Standard OASI che definisce il protocollo e i formati utilizzati tra le applicazioni in modo da rendere le implementazioni interoperabili.
- **eXtensible Messaging and Presence Protocol (XMPP)**  
Protocollo di comunicazione per middleware orientati ai messaggi basati su XML.



## ATTORI

<CONTINUARE pcd3.1/44>