

Programmazione Concorrente

Definizione:

La programmazione concorrente è l'insieme delle tecniche, metodologie e strumenti per il supporto all'esecuzione di sistemi software composti da insiemi di attività svolte simultaneamente.

Programmazione Concorrente: le origini

- La programmazione concorrente nasce negli anni 1960 nell'ambito dei **sistemi operativi**.
- Introduzione dei *canali* o **controllori di dispositivi** : consentono l'esecuzione concorrente di operazioni di I/O e delle istruzioni dei programmi eseguiti dall'unità di elaborazione centrale.
- Comunicazione tra canale ed unità centrale tramite il segnale di **interruzione**.
- Sistemi operativi multiprogrammati **time-sharing**: **interruzione** dell'esecuzione allo scadere del time slice.
- Come conseguenza dell'interruzione, parti di programmi possono essere eseguite in un **ordine non predicibile**
(->**interferenze** su variabili comuni).

- Successivamente furono introdotti i sistemi **multiprocessore**. Inizialmente costosi, ora ampiamente diffusi.
- I sistemi multiprocessore consentono a differenti processi appartenenti alla stessa applicazione di essere eseguiti in **reale parallelismo** e quindi alle applicazioni di essere eseguite più velocemente.

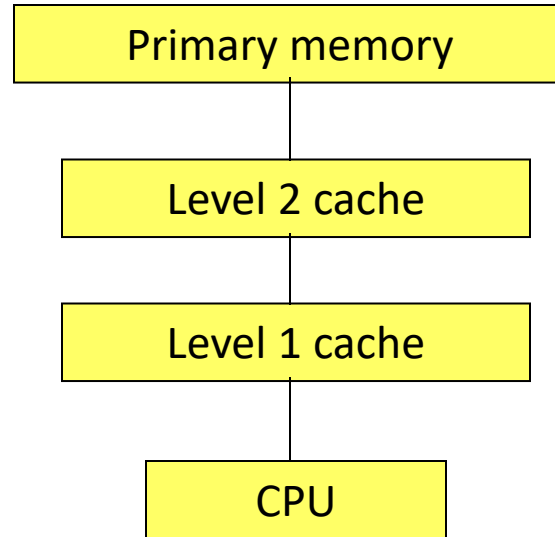
Problemi:

- Come suddividere un'applicazione in processi?
- Quanti processi utilizzare?
- Come garantire la corretta sincronizzazione delle loro operazioni?

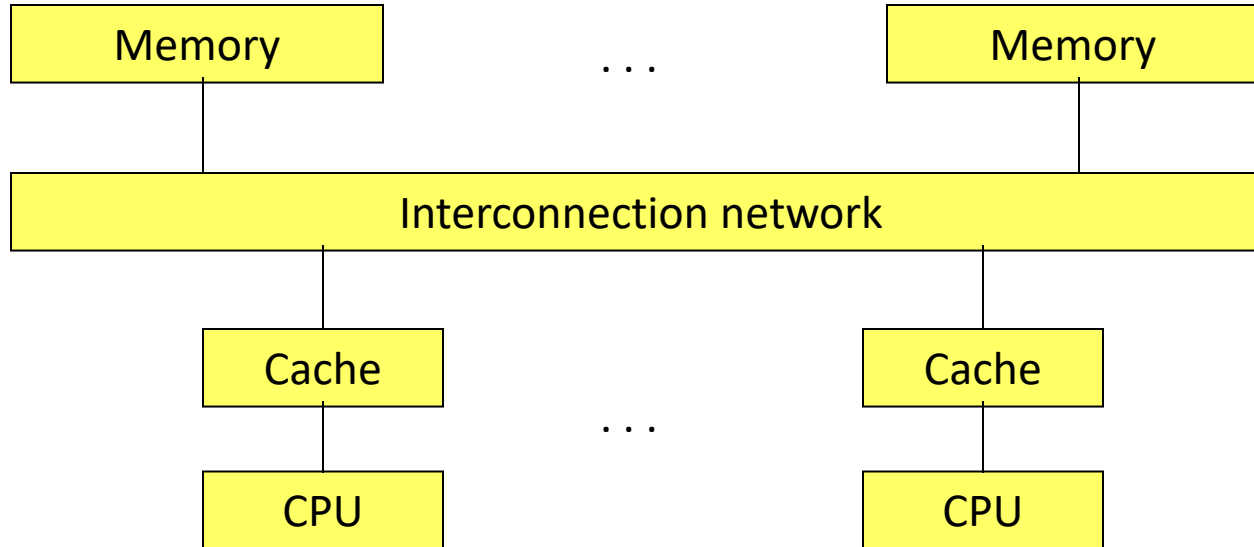
➔ Queste decisioni dipendono da:

- tipo di architettura **hardware**
- tipo di **applicazione**.

Tipi di architettura: Single processor



Shared- Memory Multiprocessors

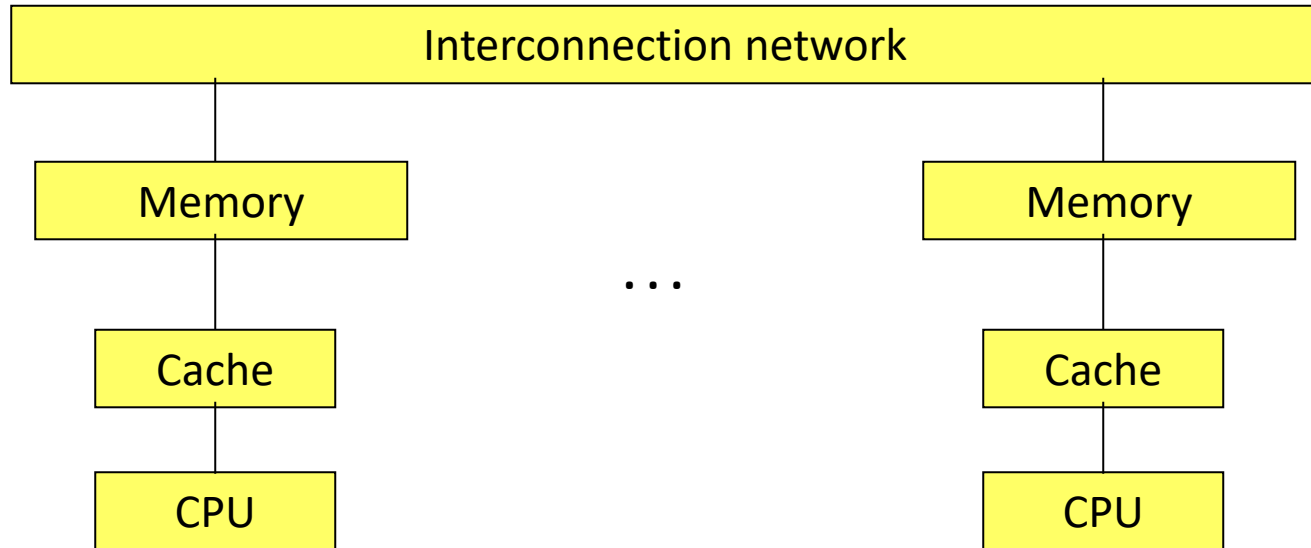


Sistemi Multiprocessore

2 modelli:

- **UMA**: sistemi a multiprocessore con un **numero ridotto di processori** (da 2 a 30 circa);
 - la rete di interconnessione è realizzata da un *memory bus* o da *crossbar switch*.
 - **UMA** (Uniform Memory Access): il tempo di accesso uniforme da ogni processore ad ogni locazione di memoria.
 - Si chiamano anche symmetric multiprocessors (SMP).
- **NUMA**: sistemi con un **numero elevato di processori** (decine o centinaia):
 - la memoria è organizzata **gerarchicamente** (per evitare la congestione del bus).
 - La rete di interconnessione è un insieme di *switches* e *memorie* strutturato ad albero. Ogni processore ha memorie che sono più vicine ed altre più lontane.
 - **NUMA** (Non Uniform Access Time): il tempo di accesso dipende dalla distanza tra processore e memoria.

Distributed-memory: Multicomputers e Network systems



Distributed-memory: classificazione

Due modelli:

- **Multicomputer** : I processori e la rete sono fisicamente vicini (nella stessa struttura): tightly coupled machine.
 - La rete di interconnessione offre un cammino di comunicazione tra i processori ad alta velocità e larghezza di banda (es. Cluster of Computers COW, Massively parallel computers).
- **Network systems**: i nodi sono collegati da una rete locale (es.Ethernet) o da una rete geografica (Internet): loosely coupled systems.

I nodi di un distributed memory system possono essere o singoli processori o shared memory multiprocessor.

Classificazione architetture: Flynn (1972)

La più usata classificazione dei sistemi di calcolo è la tassonomia di Flynn (1972).

Si basa su due concetti:

- **parallelismo a livello di istruzioni:**

- **Single instruction stream:** l'architettura è in grado di eseguire un singolo flusso di istruzioni
- **Multiple instruction streams:** possono essere eseguiti più flussi di istruzioni in parallelo

- **parallelismo a livello di dati:**

- **single data stream:** l'architettura è in grado di elaborare un singolo flusso sequenziale di dati
- **multiple data streams:** l'architettura è in grado di processare più flussi di dati paralleli

Flynn Taxonomy

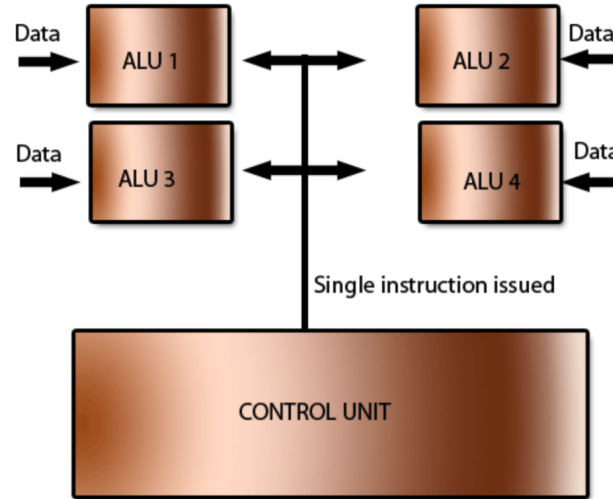
		Instruction Streams	
		one	many
Data Streams	one	SISD traditional von Neumann single CPU computer	MISD May be pipelined Computers
	many	SIMD Vector processors fine grained data Parallel computers	MIMD Multi computers Multiprocessors

Categorie di Flynn

- **SISD**: single instruction, single data. Elaboratore monoprocesso (modello Von Neumann)
- **SIMD**: architetture composte da molte unità di elaborazione che eseguono contemporaneamente la stessa istruzione su dati diversi. Es. processori vettoriali.
- **MISD**: più istruzioni in parallelo sullo stesso flusso di dati (?)
- **MIMD**: istruzioni diverse vengono eseguite in parallelo su dati diversi.
 - In questa categoria rientrano sia i **multiprocessori**, sia i **multicomputers**

SIMD

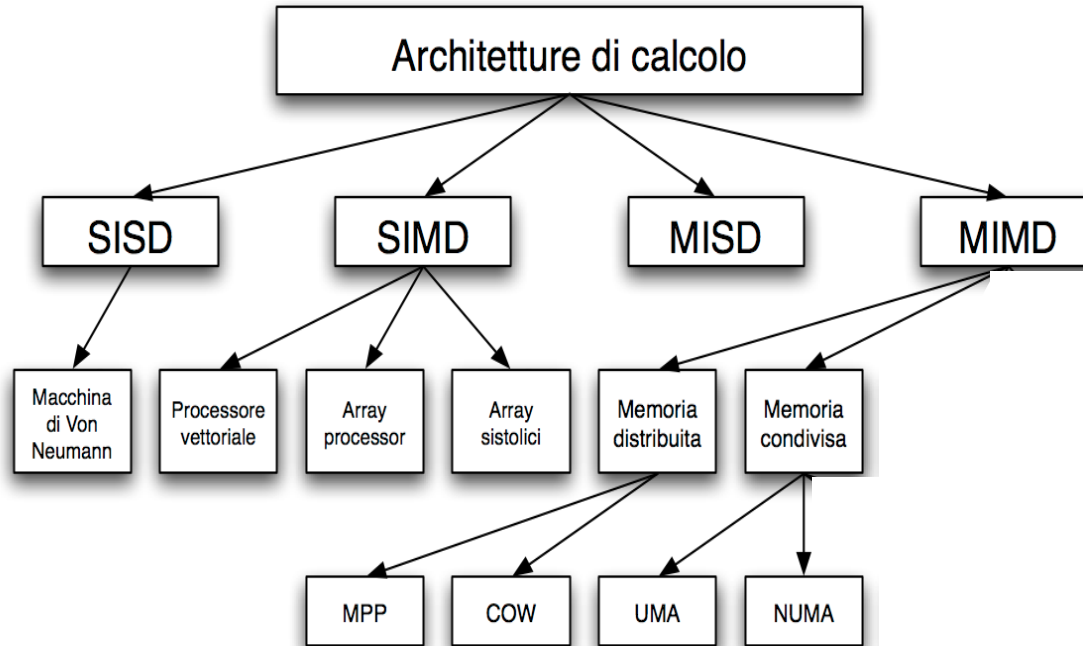
- Array processors:** architetture composte da più unità di elaborazione gestite da un'unica unità di controllo, in modo che eseguano in modo sincrono la stessa istruzione su data stream separati:



An array processor - a Single Instruction Multiple Data computer

- Vengono sfruttati per il calcolo ad alte prestazioni e sono particolarmente adatti per applicazioni di calcolo scientifico (dati in forma vettoriale/matriciale)

ARCHITETTURE: tassonomia di Flynn



Applicazioni

Tipi di applicazioni

a) multithreaded:

- Applicazioni strutturate come un insieme di processi (thread) per:
 - far fronte alla complessità
 - aumentare l'efficienza
 - semplificare la programmazione.
- I processi possono condividere variabili.
- Sono caratterizzati dal fatto che generalmente **esistono più processi che processori**.
- I processi sono schedati ed eseguiti indipendentemente.

Esempi di applicazioni:

- Applicazioni di ogni tipo..!

Tipi di Applicazioni

b) sistemi multitasking/ sistemi distribuiti:

- Le componenti dell'applicazione (task) vengono eseguite su nodi (eventualmente virtuali) collegati tramite opportuni mezzi di interconnessione (es. canali)
- I processi comunicano scambiandosi messaggi.

Esempi di applicazioni:

- File system di rete
 - Data-base systems per applicazioni bancarie etc..
 - Web server su Internet
 - Sistemi per la condivisione di risorse
 - Sistemi fault tolerant
 - Sistemi distribuiti pervasivi
-
- Tipica organizzazione : **client-server**.
 - In alternativa altri modelli: peer to peer, publisher-subscriber, etc.
- I componenti in un sistema distribuito sono spesso **multithreaded** applications

Tipi di Applicazioni

c) *Applicazioni parallele:*

- *Obiettivo:* risolvere un dato problema più velocemente (o un problema di dimensioni più elevate nello stesso tempo) sfruttando efficacemente il parallelismo disponibile a livello HW.
- Sono eseguite su *sistemi paralleli* (es. sistemi HPC, processori vettoriali) facendo uso di *algoritmi paralleli*.
- a seconda del modello architetturale, l'esecuzione è portata avanti da istruzioni/thread/processi paralleli che interagiscono utilizzando librerie specifiche.

Esempi di applicazioni:

- Applicazioni scientifiche che modellano e simulano fenomeni fisici complessi (es. previsioni del tempo, evoluzione del sistema solare, etc..)
- Elaborazione di immagini. Es: Videosorveglianza, realtà aumentata, ecc.
- Problemi di ottimizzazione di grandi dimensioni.
- elaborazione big data
- algoritmi di machine learning
- ...

Processi non sequenziali e tipi di interazione

Algoritmo, programma, processo

- **Algoritmo:** Procedimento logico che deve essere eseguito per risolvere un determinato problema.
- **Programma:** Descrizione di un algoritmo mediante un opportuno formalismo (linguaggio di programmazione), che rende possibile l'esecuzione dell'algoritmo da parte di un particolare **elaboratore**.
- **Processo:** insieme **ordinato** degli **eventi** cui dà luogo un elaboratore quando opera sotto il controllo di un programma.

Elaboratore: entità **astratta** realizzata in hardware e parzialmente in software, in grado di eseguire programmi (descritti in un dato linguaggio).

Evento: Esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere ed eseguire; ogni evento determina una **transizione di stato** dell'elaboratore

➔ Un programma descrive non un processo, ma un **insieme di processi**, ognuno dei quali è relativo all'esecuzione del programma da parte dell'elaboratore per un determinato insieme di dati in ingresso.

Processo sequenziale

Sequenza di stati attraverso i quali passa l'elaboratore durante l'esecuzione di un programma (storia di un processo o traccia dell'esecuzione del programma).

Esempio: valutare il massimo comune divisore tra due numeri naturali x e y :

- a) Controllare se i due numeri x e y sono uguali, nel qual caso il loro M.C.D. coincide con il loro valore
- b) Se sono diversi, valutare la loro differenza
- c) Tornare ad a) prendendo in considerazione il più piccolo dei due e la loro differenza

Esempio: M.C.D. di x e y (numeri naturali) [*Metodo di Euclide*]

```
int MCD(int x, int y)
```

```
{ a = x; b = y;
```

```
  while (a != b)
```

```
    if (a > b) a = a - b
```

```
      else b = b-a;
```


```
  return a;
```

```
}
```

```
...
```

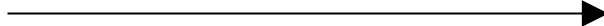
Traccia dell'esecuzione:
sequenza di stati attraversati
durante l'esecuzione

R=MCD(18,24);



x	18	18	18	18	18	18
y	24	24	24	24	24	24
a	-	18	18	18	12	6
b	-	-	24	6	6	6

**stato
iniziale**



**stato
finale**

- Più processi possono essere associati allo stesso programma (istanze).
- Ciascuno rappresenta l'esecuzione dello stesso codice con dati di ingresso (possibilmente) diversi.

Esempi:

- Il sistema operativo può generare più processi concorrenti che eseguono shell, ognuno per un diverso utente, che esegue diverse operazioni su dati diversi
- Il compilatore di un linguaggio può dare luogo a più processi, ciascuno relativo alla traduzione di un particolare programma.
- ...

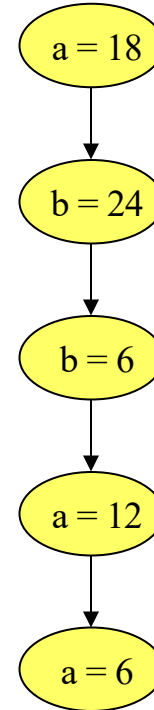
Grafo di precedenza

Un processo può essere rappresentato tramite un **grafo orientato** detto grafo di precedenza del processo, costituito da nodi ed archi orientati:

- I nodi del grafo rappresentano i singoli eventi del processo
- gli archi orientati identificano le precedenze temporali tra tali eventi

Ogni nodo rappresenta quindi un evento corrispondente all'esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere ed eseguire.

Es. MCD: essendo il processo strettamente sequenziale, **il grafo di precedenza è ad Ordinamento Totale** (ogni nodo ha esattamente un predecessore ed un successore -> ogni coppia di nodi (N1,N2) è ordinata)



Processi non sequenziali

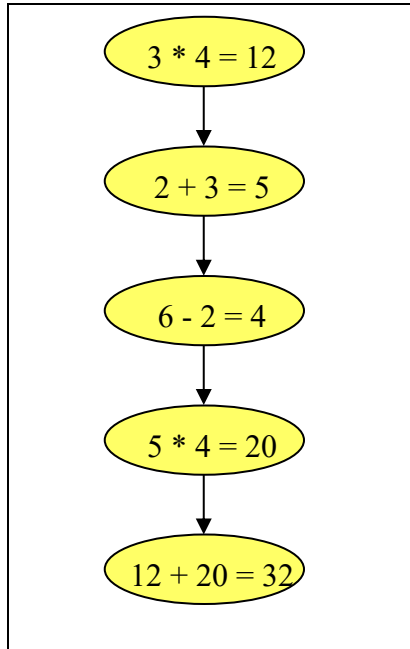
- L'ordinamento totale di un grafo di precedenza deriva dalla natura sequenziale del processo (a sua volta imposta dalla natura sequenziale dell'elaboratore).
- In taluni casi l'ordinamento totale è implicito nel problema da risolvere.
- Esistono molti esempi di problemi che potrebbero più naturalmente essere risolti tramite **processi non sequenziali**.

Processo non sequenziale: l'insieme degli eventi che lo descrive e' ordinato secondo una relazione d'ordine parziale.

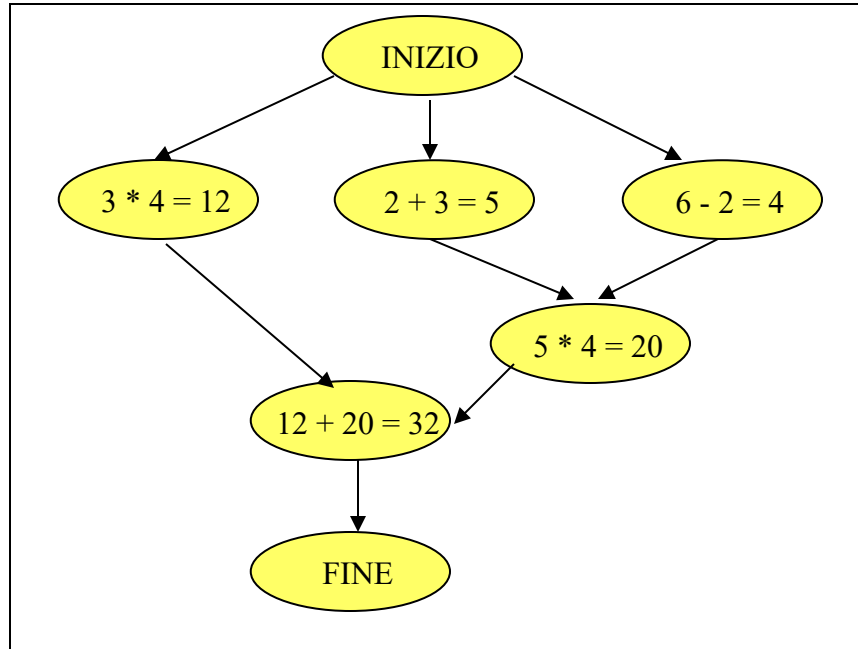
Esempio: valutazione dell'espressione

$$(3 * 4) + (2 + 3) * (6 - 2)$$

Grafo di precedenza ad
ordinamento totale:



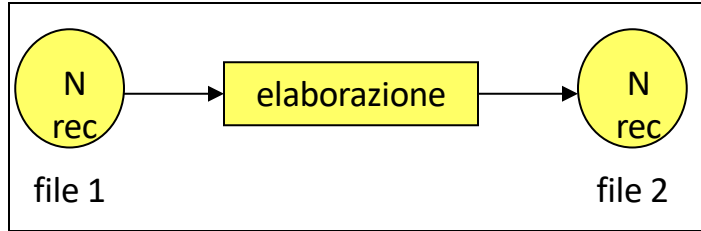
Grafo di precedenza ad
ordinamento parziale:



Esempio – segue:

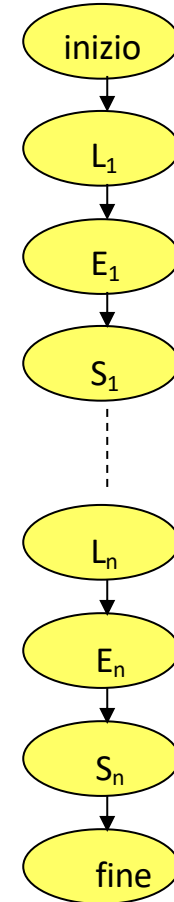
- La logica del problema non impone un ordinamento totale fra le operazioni da eseguire: ad esempio è indifferente che venga eseguito $(2 + 3)$ prima di eseguire $(6 - 2)$ o viceversa.
- Entrambe le operazioni precedenti devono invece essere eseguite prima del prodotto dei loro risultati.
- Certi eventi del processo sono tra loro scorrelati da qualunque relazione di precedenza temporale → il risultato dell'elaborazione è indipendente dall'ordine con cui gli eventi avvengono.

Esempio: Elaborazione di dati su un file sequenziale



```
buffer B;  
int i;  
for(i=1; i<=N; i++)  
{  
    lettura(B);      /* L */  
    elaborazione(B); /* E */  
    scrittura(B);    /* S */  
}
```

Grafo di precedenza ad ordinamento totale:



Esempio – segue:

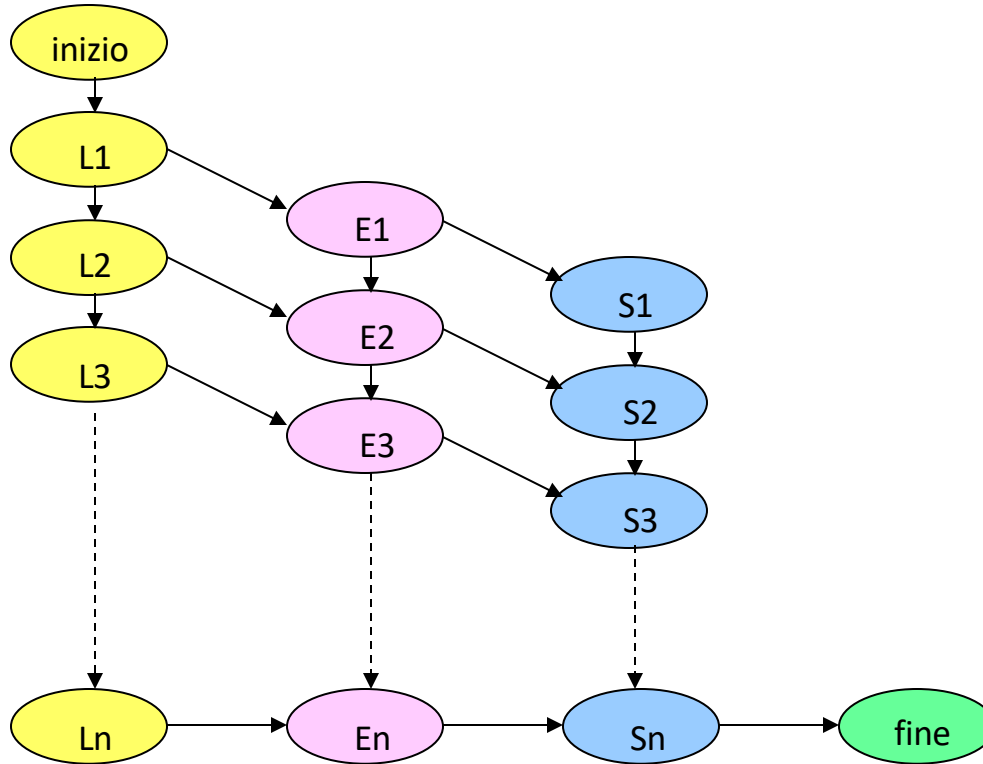
La logica del problema impone due soli vincoli

1. Le operazioni di lettura (o elaborazione o scrittura) dovranno essere eseguite in sequenza sugli N record.
2. Le operazioni di lettura, elaborazione e scrittura su ogni record dovranno essere eseguite in quest'ordine.

Ad esempio, non esiste alcuna relazione logica di precedenza tra la lettura dell' i -esimo e la scrittura dell' $(i-1)$ -esimo.

👉 Il grafo degli eventi più naturale risulta essere quello ad **ordinamento parziale**.

Esempio – segue: grafo ad ordinamento parziale



• Un arco che collega due nodi rappresenta il vincolo di precedenza tra i corrispondenti eventi

Esempio: E3 e S2 non sono legati da vincoli di precedenza →

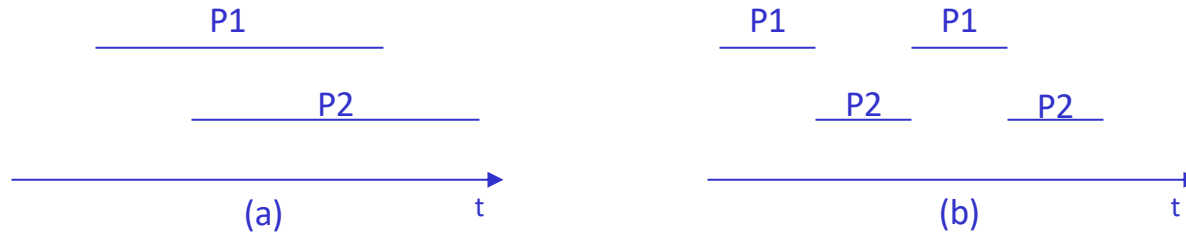
- E3 può avvenire prima di S2
- S2 può avvenire prima di E3
- E3 può avvenire contemporaneamente a S2 (PARALLELISMO)

L'esecuzione di un processo non sequenziale richiede:

- elaboratore non sequenziale
- linguaggio di programmazione non sequenziale

Elaboratore non sequenziale (in grado di eseguire più operazioni contemporaneamente):

- sistemi multielaboratori (a)
- sistemi monoelaboratori (b)



Linguaggi non sequenziali (o concorrenti): Caratteristica comune: consentire la descrizione di un insieme di attività concorrenti, tramite moduli che possono essere eseguiti in parallelo (ad esempio: processi sequenziali)

Linguaggi concorrenti

- In generale, un linguaggio concorrente permette di esprimere il (potenziale) parallelismo nell'esecuzione di moduli differenti.
- Quale dimensione minima del modulo?
 - Singola istruzione (es. CSP, Occam,...)
 - **Sequenza di istruzioni** (es. Java, Ada, go, ecc...)

Scomposizione di un processo non sequenziale

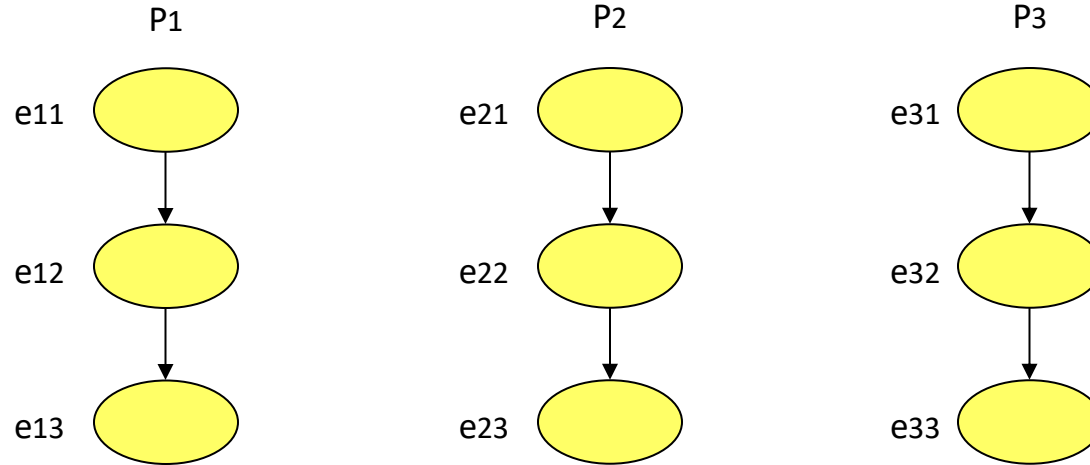
Se il linguaggio concorrente permette di esprimere il parallelismo a livello di **sequenza di istruzioni**:

- Scomposizione di un processo non sequenziale in un insieme di processi sequenziali, eseguibili “contemporaneamente”.
- Consente di dominare la complessità di un algoritmo non sequenziale

Le attività rappresentate dai processi possono essere:

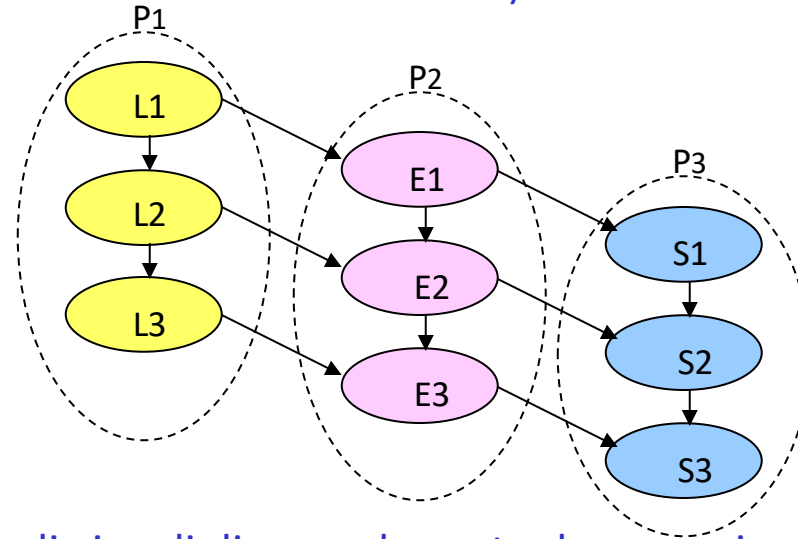
- completamente indipendenti
- interagenti

Processi indipendenti



L'evoluzione di un processo non influenza quella degli altri.

- Nel caso di grafi connessi ad ordinamento parziale, la scomposizione del processo globale in processi sequenziali consiste nell'individuare sul grafo un insieme $P1....Pn$ di sequenze di nodi (insiemi di nodi totalmente ordinati).



- la presenza di vincoli di precedenza tra le operazioni dei processi (esprese dagli archi) evidenzia i vincoli di sincronizzazione tra processi.
- In questo caso i tre processi non sono fra loro indipendenti (processi interagenti)

Processi interagenti

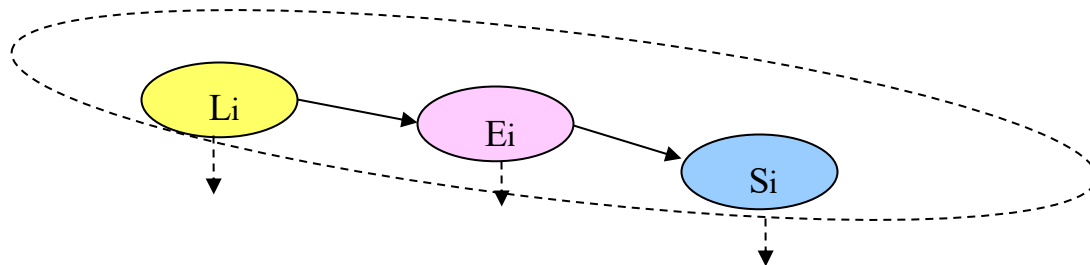
Le interazioni tra i processi di lettura, elaborazione e scrittura sono relative ad uno scambio di informazioni. (P1->P2, P2->P3)

- **Vincolo di sincronizzazione:** vincolo imposto da ogni arco del grafo di precedenza che collega nodi di processi diversi.
- Due processi, quando arrivano ad un punto di interazione corrispondente ad uno scambio di informazioni, devono sincronizzarsi, cioè ordinare i loro eventi come specificato dal grafo di precedenza.

Tipi di decomposizione

La decomposizione di un grafo di precedenza ad ordinamento parziale in un insieme di sequenze di nodi (processi) può essere fatta in vari modi.

ESEMPIO: Lettura, elaborazione e scrittura dell' i -esimo record ($i=1,2,\dots,n$)



- I vincoli di sincronizzazione sono gli archi verticali del grafo.
- La scelta più idonea del tipo di decomposizione in processi sequenziali di un processo non sequenziale è quella per la quale le interazioni tra processi sono meno numerose e meno frequenti così da agevolare l'esecuzione separata delle singole attività.

Interazione tra processi

Le possibile forme di interazione tra processi concorrenti sono:

- **Cooperazione**
- **Competizione**
- **Interferenza**

Interazione tra processi

COOPERAZIONE: comprende tutte le interazioni prevedibili e desiderate, insite cioè nella logica degli algoritmi (archi, nel grafo di precedenza ad ordinamento parziale).

Prevede scambio di informazioni:

- segnali temporali (senza trasferimento di dati)
- dati (messaggi) → **comunicazione**

→ In entrambi i casi esiste un **vincolo di precedenza** (**sincronizzazione**) tra gli eventi di processi diversi

Scambio di segnali temporali

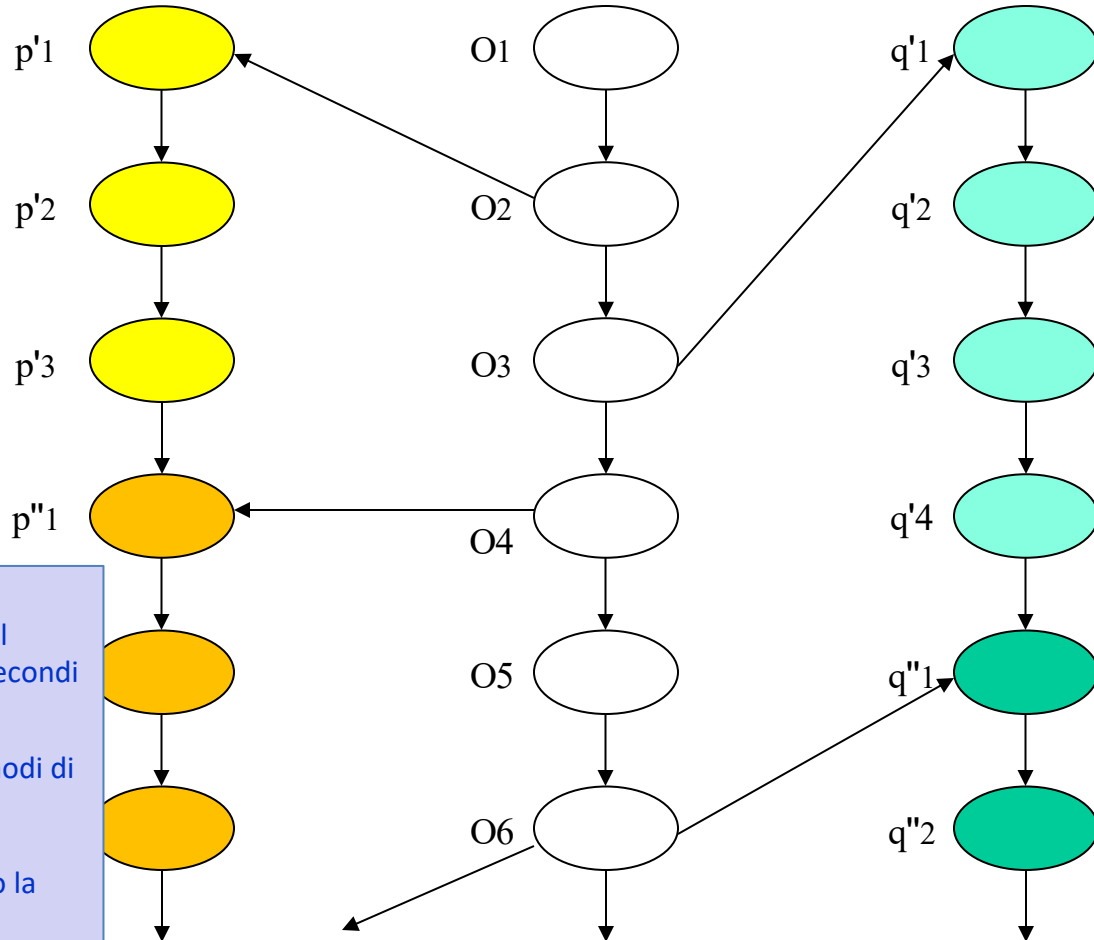
Esempio:

- Processo P, esegue la ripetizione di una sequenza di tre operazioni (p1, p2, p3) ogni due secondi.
- Processo Q, esegue la ripetizione di una sequenza di quattro operazioni (q1, q2, q3, q4) ogni tre secondi.
- Processo O (l'orologio), ha il compito di registrare il passare del tempo e di attivare periodicamente i processi P e Q inviando segnali temporali.

processo P

processo O

processo Q



- I nodi O1, O2, ... rappresentano le azioni del processo O e denotano i secondi scanditi dall'orologio (O).

- Gli archi che collegano i nodi di O con i nodi di P (Q) rappresentano i vincoli di precedenza che esprimono la temporizzazione.

- Relazione di **causa ed effetto** tra l'esecuzione dell'operazione di invio da parte del processo mittente e l'esecuzione dell'operazione di ricezione da parte del processo ricevente.

Vincolo di precedenza tra questi eventi (**sincronizzazione** dei due processi).

Comunicazione: può essere previsto uno scambio di dati.

Linguaggio di programmazione: deve fornire costrutti linguistici atti a specificare la sincronizzazione e la eventuale comunicazione tra i processi

Interazione tra processi

COMPETIZIONE

La macchina concorrente su cui i processi sono eseguiti mette a disposizione un numero limitato di risorse condivise tra i processi.

La competizione ha come obiettivo il coordinamento dei processi nell'accesso alle risorse condivise.

Ad esempio, per risorse che non possono essere usate contemporaneamente da più processi, è necessario prevedere meccanismi che regolino la competizione.

➔ Interazione prevedibile e non desiderata, ma necessaria.

Competizione

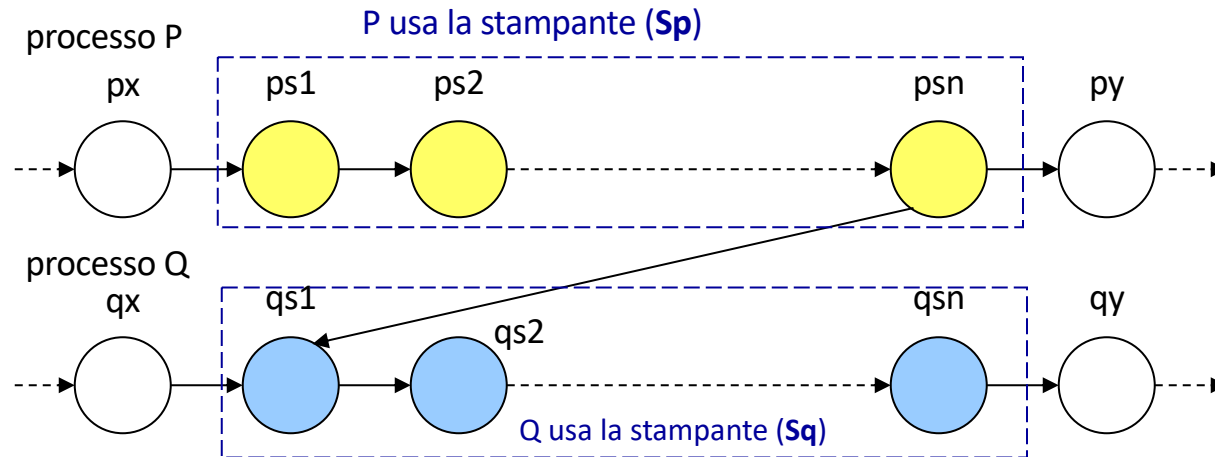
Esempio: **Mutua esclusione**

- Due processi P e Q devono usare in certi istanti una comune stampante

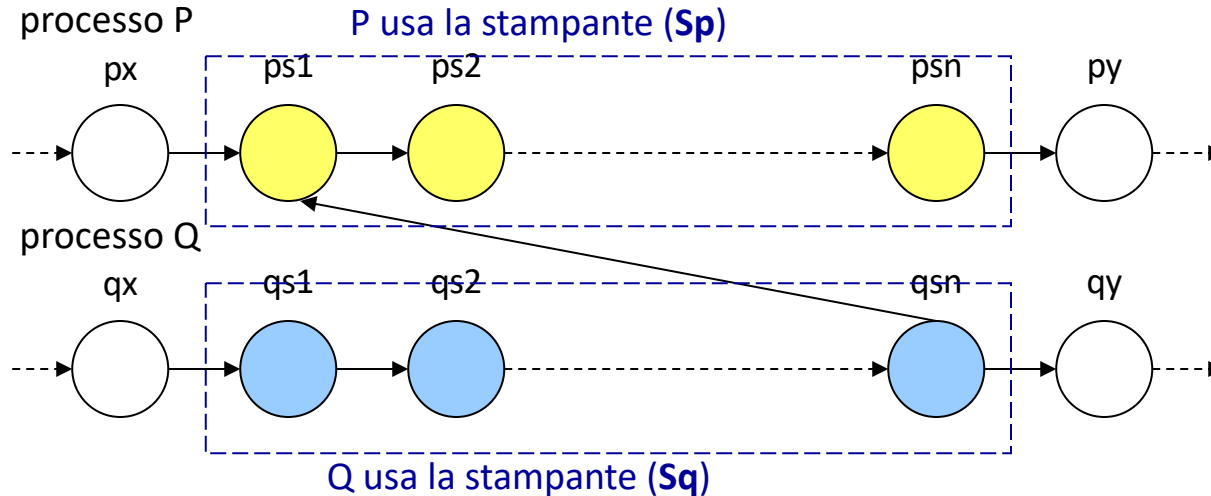
$S_p = \{ps1, ps2, \dots, psn\}$ e $S_q = \{qs1, qs2, \dots, qsn\}$

sono le sequenze di istruzioni che P e Q devono rispettivamente eseguire per produrre un messaggio sulla stampante

- Le due sequenze devono essere eseguite in modo **mutuamente esclusivo**



- L'interazione si estrinseca in un vincolo di precedenza tra eventi di processi diversi (psn deve precedere qs1), cioè in un vincolo di sincronizzazione.
- Non è (come nel caso della cooperazione) un vincolo di causa ed effetto, cioè l'ordine con cui devono avvenire due eventi non è sempre lo stesso. Basta che sia verificata la proprietà di mutua esclusione.



Sezione Critica

La sequenza di istruzioni con le quali un processo accede in modo esclusivo a un oggetto condiviso con altri processi prende il nome di **sezione critica**.

- Ad un oggetto puo` essere associata una sola sezione critica (usata da tutti i processi) o più sezioni critiche (**classe di sezioni critiche**).
- La regola di mutua esclusione stabilisce che:

Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo.

Nell'esempio precedente:

- Sp e Sq sono **sezioni critiche**;
- Sp e Sq **appartengono alla classe di sezioni critiche**, associata alla stampante;

Interazione tra processi

Cooperazione: sincronizzazione diretta o esplicita

Competizione: sincronizzazione indiretta o implicita

Interferenza: interazione provocata da errori di programmazione. Ad esempio, deadlock.

E' un'interazione non prevista e non desiderata

- può non manifestarsi, dipende dalla velocità relativa dei processi
- gli errori possono manifestarsi nel corso dell'esecuzione del programma a seconda delle diverse condizioni di velocità di esecuzione dei processi

(errori dipendenti dal tempo)

Uno degli Obiettivi fondamentali della programmazione concorrente è l'eliminazione delle interferenze.

ARCHITETTURE E LINGUAGGI PER LA PROGRAMMAZIONE CONCORRENTE

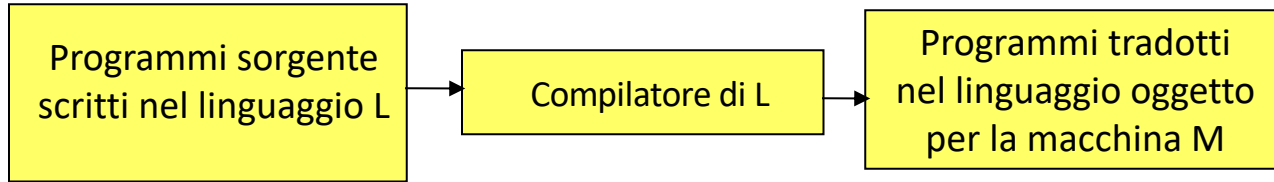
Linguaggi per la programmazione concorrente

- Disponendo di una «macchina concorrente» M (in grado di eseguire più processi sequenziali contemporaneamente) e di un linguaggio di programmazione con il quale descrivere algoritmi non sequenziali, è possibile scrivere e far eseguire programmi concorrenti.
- L'elaborazione complessiva può essere descritta come un insieme di processi sequenziali interagenti

Proprietà di un linguaggio per la programmazione concorrente

- Contenere appositi costrutti con i quali sia possibile dichiarare moduli di programma destinati ad essere eseguiti come **processi** sequenziali distinti.
- Non tutti i processi vengono eseguiti contemporaneamente. Alcuni processi vengono svolti se, dinamicamente, si verificano particolari condizioni. E' quindi necessario poter specificare quando un processo deve essere **attivato** e **terminato**.
- Occorre che siano presenti strumenti linguistici per **specificare le interazioni** che dinamicamente potranno verificarsi tra i vari processi

Architettura di una macchina concorrente



- Non sempre M ha tante unità di elaborazione quanti sono i processi da svolgere contemporaneamente durante l'esecuzione di un programma concorrente.
- M è una macchina astratta ottenuta con tecniche software (o hardware) basandosi su una macchina fisica M' generalmente più semplice (es. con un numero di unità di elaborazione generalmente minore del numero dei processi).



Oltre ai meccanismi di multiprogrammazione e sincronizzazione è presente anche il meccanismo di **protezione** (controllo degli accessi alle risorse).

- Importante per rilevare eventuali interferenze tra i processi.
- Può essere realizzato in hardware o in software nel supporto a tempo di esecuzione.
- Capabilities e liste di controllo degli accessi.

Il **nucleo** offre supporto a tempo di esecuzione di un linguaggio concorrente.

Nel nucleo sono sempre presenti due funzionalità base:

- meccanismo di **multiprogrammazione**
 - meccanismo di **sincronizzazione** e **comunicazione**
-
- Il primo meccanismo è quello preposto alla gestione delle unità di elaborazione della macchina M' (unità reali) consentendo ai vari processi eseguiti sulla macchina astratta M di condividere l'uso delle unità reali di elaborazione (**scheduling**) tramite l'allocazione in modo esclusivo ad ogni processo di un'unità virtuale di elaborazione.
 - Il secondo meccanismo è quello che estende le potenzialità delle unità reali di elaborazione, rendendo disponibile alle unità virtuali strumenti mediante i quali sincronizzarsi e comunicare.

Architettura di M

Due diverse organizzazioni logiche:

1. Gli elaboratori di M sono collegati ad un'unica memoria principale (v. sistemi multiprocessore)
2. Gli elaboratori di M sono collegati da una sottorete di comunicazione, senza memoria comune (v. sistemi multicomputer).

Le due precedenti organizzazioni logiche di M definiscono due modelli di interazione tra i processi:

1. Modello a **memoria comune**, in cui l'interazione tra i processi avviene tramite oggetti contenuti nella memoria comune (modello ad ambiente globale).
2. Modello a **scambio di messaggi**, in cui la comunicazione e la sincronizzazione tra processi si basa sullo scambio di messaggi sulla rete che collega i vari elaboratori (modello ad ambiente locale).

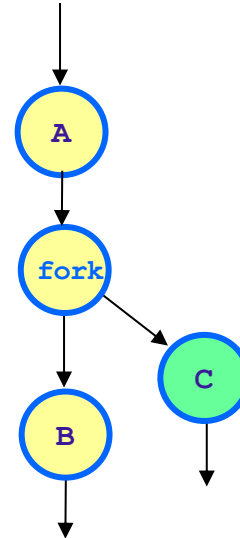
INDIPENDENZA TRA L'ARCHITETTURA DI M e M'

Costrutti linguistici per la specifica della concorrenza

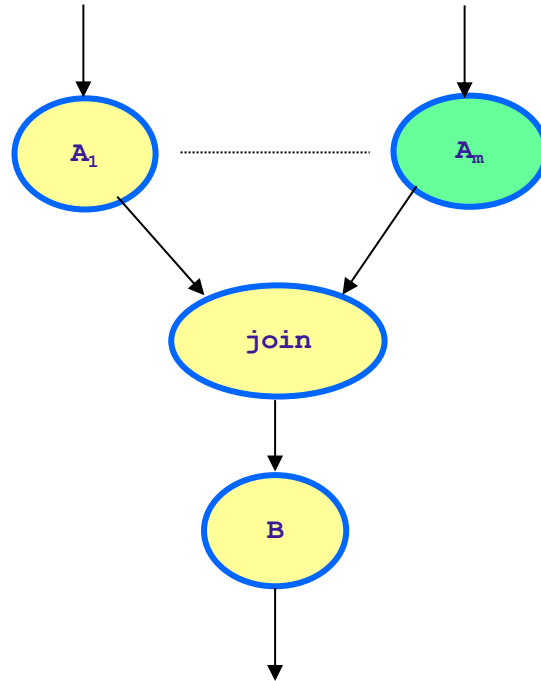
Fork/Join

L'esecuzione di una **fork** coincide con la creazione e l'attivazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante

```
/* processo p: */  
=====  
A: .....;  
   p = fork fun;  
B: .....;  
=====  
=====  
  
/* codice nuovo  
processo:*/  
void fun()  
{  
    C: .....;  
    =====  
}
```



La **join** consente di determinare quando un processo, creato tramite la **fork**, ha terminato il suo compito, sincronizzandosi con tale evento.

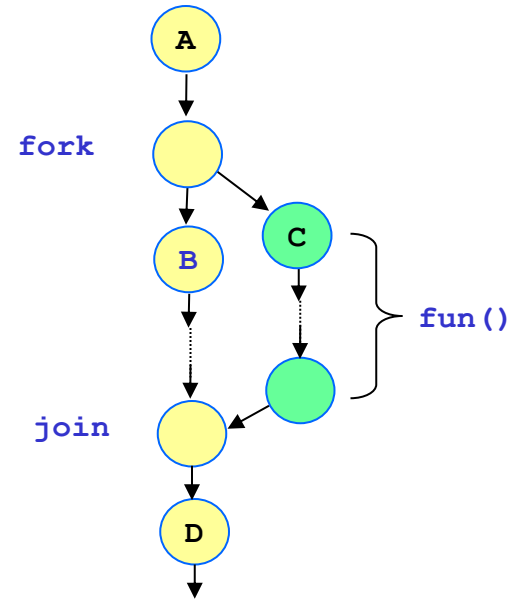


```

    /* processo p: */
    ...
A: .....;
    p = fork fun;
B: .....;
    join p;
D: .....;
    ...

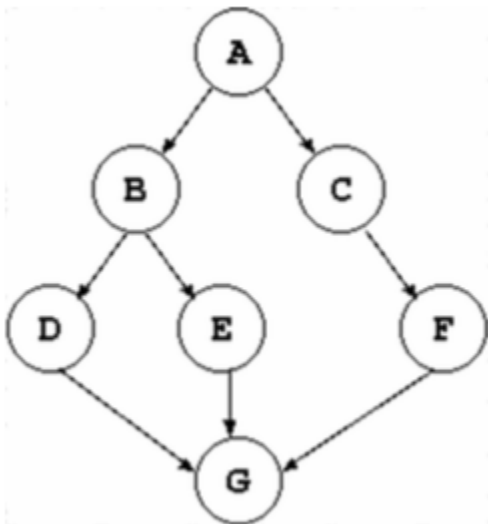
/* codice nuovo
processo:*/
void fun()
{
    C: .....;
    ...
}

```



Possibilità di denotare in modo esplicito il processo sulla cui terminazione ci si vuole sincronizzare.

Fork/Join: esempio



begin

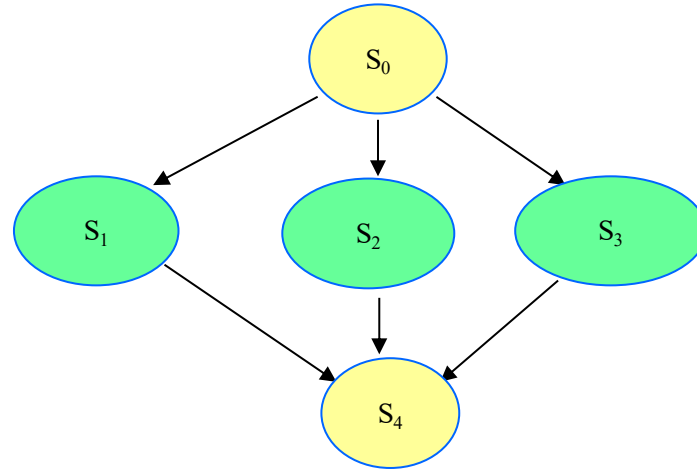
```
A;  
fork (LC) ;  
B;  
fork (LE) ;  
D;  
goto LG;  
LC: C;  
F;  
goto LG;  
LE: E;  
goto LG;  
LG: join(3) ;  
G;
```

end

Cobegin-Coend

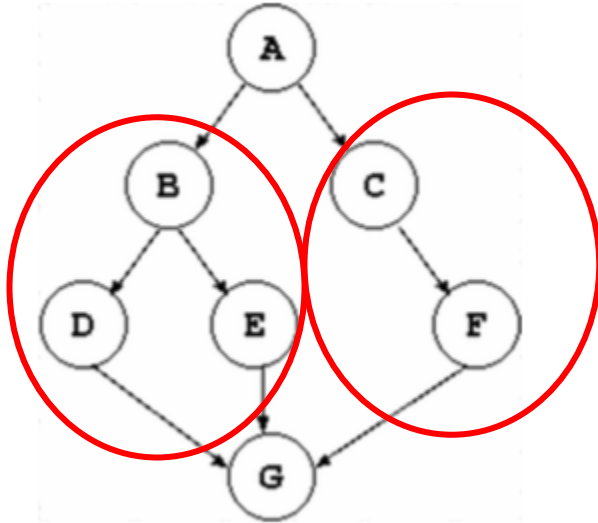
In alternativa al modello fork-join, la concorrenza può essere espressa anche nel modo seguente:

```
S0;  
cobegin  
    S1;  
    S2;  
    S3;  
coend  
S4;
```



Le istruzioni S₁, S₂,...,S_n sono eseguite in parallelo. Ogni S_i può contenere altre istruzioni **cobegin..coend** al suo interno.

Cobegin-coend: esempio



```
begin
```

```
  A;
```

```
  cobegin
```

```
    begin
```

```
      B;
```

```
      cobegin
```

```
        D;
```

```
        E;
```

```
      coend;
```

```
    end;
```

```
    begin
```

```
      C;
```

```
      F;
```

```
    end;
```

```
  coend;
```

```
  G;
```

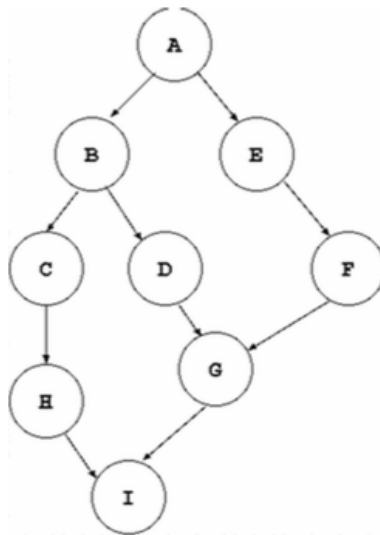
```
end;
```

fork-join vs Cobegin-Coend

S.p.d. che fork-join è un formalismo più generale di cobegin-end:

- ☞ tutti i grafi di precedenza possono essere espressi tramite fork-join
- ☞ ciò non è vero per i costrutti cobegin-end

Ad es:



Proprietà dei programmi

Programmi e proprietà

Una delle attività più importanti per chi sviluppa programmi è la verifica di correttezza dei programmi realizzati.

Cosa succede quando il programma viene eseguito?

Traccia dell'esecuzione (Storia): sequenza degli stati attraversati dal sistema di elaborazione durante l'esecuzione del programma.

Stato: insieme dei valori delle variabili definite nel programma + variabili “implicite” (es. valore del Program counter).

Programmi sequenziali: ogni esecuzione di un programma P su un particolare insieme di dati D genera sempre la stessa traccia.

La verifica può agevolmente essere svolta tramite debugging.

Programmi concorrenti: l'esito dell'esecuzione dipende da quale sia l'effettiva sequenza cronologica di esecuzione delle istruzioni contenute -> ogni esecuzione di un programma P su un particolare insieme di dati D può dare origine ad una traccia diversa. (**non determinismo**)

La verifica di proprietà di programmi concorrenti è molto più difficile:

- Il semplice debug su una (o su un numero limitato di esecuzioni) non dà alcuna garanzia sul soddisfacimento di una data proprietà.

Proprietà dei programmi

Una proprietà di un programma P è un attributo che è sempre vero, in ogni possibile traccia generata dall'esecuzione di P .

In generale, le proprietà dei programmi si classificano in due categorie:

- Safety properties**
- Liveness properties**

Proprietà SAFETY:

È una proprietà che garantisce che durante l'esecuzione di P, **non si entrerà mai** in uno stato “errato” (cioè, uno stato in cui le variabili assumono valori non desiderati).

Proprietà LIVENESS:

È una proprietà che garantisce che durante l'esecuzione di P, **prima o poi** si entrerà in uno stato “corretto” (cioè, uno stato in cui le variabili assumono valori desiderati).

Safety & Liveness

Programmi sequenziali:

Le proprietà fondamentali che ogni programma sequenziale deve avere sono:

1. **Correttezza del risultato finale:** per ogni esecuzione il risultato ottenuto è giusto -> **SAFETY**
2. **Terminazione:** prima o poi l'esecuzione termina -> **LIVENESS**

Safety & Liveness

Programmi concorrenti:

Le proprietà 1 e 2 sono fondamentali anche per i programmi concorrenti.

In aggiunta, ogni programma concorrente deve presentare altre proprietà, come :

3. Mutua esclusione nell'accesso a risorse condivise: per ogni esecuzione non accadrà mai che più di un processo acceda contemporaneamente alla stessa risorsa -> **SAFETY**

4. Assenza di deadlock: per ogni esecuzione non si verificheranno mai situazioni di blocco critico -> **SAFETY**

Inoltre, è desiderabile:

5. Assenza di starvation: prima o poi ogni processo potrà accedere alle risorse richieste -> **LIVENESS**

Verifica di proprietà nei programmi concorrenti

Il semplice testing su vari set di dati, per diverse ripetizioni dell'esecuzione non dimostra rigorosamente il soddisfacimento di proprietà.

Possibile approccio:

Specifica «formale» dei programmi concorrenti -> dimostrazione di proprietà.