LEZIONE 4

ELABORAZIONE SEQUENZIALE E CONCORRENTE

IN QUESTA LEZIONE IMPAREREMO...

- il concetto di programmazione concorrente
- a realizzare il grafo delle precedenze
- il concetto di interazione tra processi

Generalità

La programmazione imperativa che si apprende nei primi corsi di informatica ha come riferimento un esecutore *sequenziale* che svolge una sola azione alla volta sulla base di un *programma sequenziale*.

ELABORAZIONE SEQUENZIALE

Con il termine elaborazione sequenziale si intende l'esecuzione di un programma sequenziale che genera un processo sequenziale con un ordinamento totale alle azioni che vengono eseguite.

Lo stesso teorema di Bhon e Jacopini indica nella sequenza una delle *tre figure strutturali fondamentali*.
L'elaborazione sequenziale è quindi un concetto fondamentale nell'informatica in quanto gli algoritmi che vengono computati sono composti da una sequenza finita di istruzioni in corrispondenza delle quali, durante la loro esecuzione, l'elaboratore passa attraverso una sequenza di stati (traccia dell'esecuzione del programma).

Anche l'esecutore dei programmi fino a ora utilizzato è una macchina sequenziale che si basa sul modello Von Neumann, cioè dotato di una sola unità di elaborazione (singola CPU).

Ma gli elaboratori non hanno tutti una sola CPU e inoltre, come abbiamo visto nello studio dei sistemi operativi, alcune attività del processore possono essere parallelizzate, come per esempio le lunghe fasi di input che provocano enorme spreco di tempo macchina soprattutto nei processi ad alta interattività con l'utente; esistono inoltre applicazioni che per loro natura necessitano di

attività parallele, come i server web, i video games a più giocatori, i robot, e per essi la codifica sequenziale delle attività propria della programmazione sequenziale non è in grado di descrivere con naturalezza queste situazioni.

Queste situazioni necessitano di un diverso modello in grado di effettuare la programmazione di un esecutore *concorrente*, ovvero di un elaboratore che è in grado di eseguire più istruzioni contemporaneamente.

PROGRAMMAZIONE CONCORRENTE

Con *programmazione concorrente* si indicano le tecniche e gli strumenti impiegati per descrivere il comportamento di più attività o processi che si intende far eseguire contemporaneamente in un sistema di calcolo (*processi paralleli*).

Siamo in una situazione di elaborazione contemporanea reale solo nel caso in cui l'esecutore sia dotato di una architettura multiprocessore, cioè con più processori che possono eseguire ciascuno un singolo programma: nei sistemi monoprocessori sappiamo che il parallelismo avviene solo virtualmente, grazie alla multiprogrammazione, e più processi evolvono "in parallelo" grazie al quanto di tempo che viene loro assegnato dalle politiche di scheduling del sistema operativo.

Il sistema operativo è per eccellenza l'esempio più eclatante di programmazione concorrente: il suo compito è quello di assegnare le risorse hardware dell'elaboratore ai processi utente che ne fanno richiesta, cercando di massimizzarne l'efficienza nella loro utilizzazione.
Le attività del sistema operativo devono essere eseguite concorrentemente in modo da consentire l'esecuzione contemporanea di più programmi utente: ogni attività interagisce con le altre sia in modo indiretto, occupando delle risorse comuni, sia in modo diretto, scambiando informazioni in merito allo stato delle risorse e dei programmi di utente al fine di realizzare la multiprogrammazione.

In un sistema multiprogrammato i programmi d'utente e le singole funzioni svolte dal sistema operativo possono essere considerati come un insieme di processi che competono per le stesse risorse.

Quindi un sistema multiprogrammato è un sistema concorrente, così definito:

SISTEMA CONCORRENTE

Per sistema concorrente intendiamo un sistema software implementato su vari tipi di hardware che "porta avanti" contemporaneamente una molteplicità di attività diverse, tra di loro correlate, che possono cooperare a un obiettivo comune oppure possono competere per utilizzare risorse condivise.

PROCESSO CONCORRENTE

Due processi si dicono concorrenti se la prima operazione di uno di essi ha inizio prima del completamento dell'ultima operazione dell'altro.

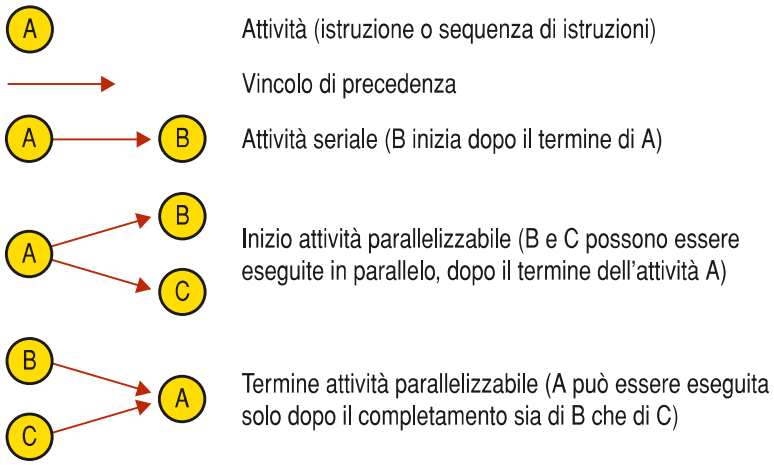
Processi non sequenziali e grafo di precedenza

Nei processi sequenziali la sequenza degli eventi che costituisce il processo è *totalmente ordinata*: se la rappresentiamo mediante un grafo orientato questo il grafo risulterà totalmente ordinato in quanto la sequenza degli eventi è ben determinata, cioè l'ordine con cui vengono eseguiti e sempre lo stesso.
Un grafo che descrive l'ordine con cui le azioni (o gli eventi) si eseguono del tempo prende il nome di grafo delle precedenze.

Nei processi paralleli, invece, l'ordinamento non è completo, in quanto l'esecutore per alcune istruzioni "è libero" di scegliere quali iniziare prima senza che il risultato sia compromesso: possiamo affermare che nella elaborazione parallela l'esecuzione delle istruzioni segue un ordinamento parziale.

Per descrivere questa libertà nella evoluzione dei processi concorrenti utilizziamo proprio il grafo delle precedenze (o diagramma delle precedenze): in un processo sequenziale il grafo delle precedenze degenera in una *lista ordinata* mentre in un processo parallelo è un *grafo orientato aciclico* e i percorsi alternativi indicano la possibilità di esecuzione contemporanea di più istruzioni.

Per la descrizione del grafo delle precedenze vengono utilizzati i seguenti simboli con i rispettivi significati:



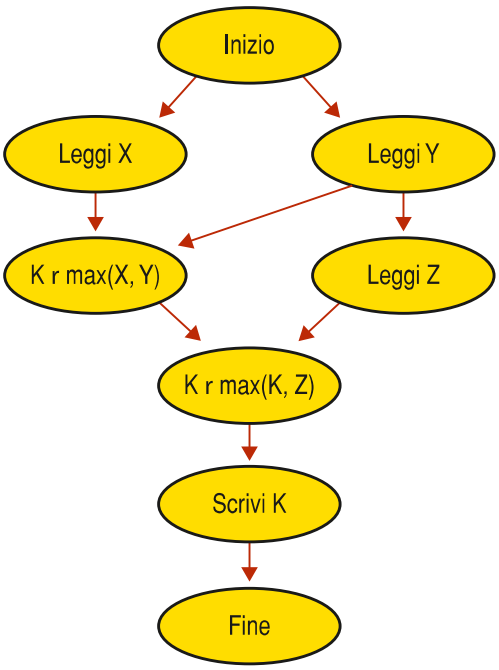
ESEMPIO 6 *Grafo delle precedenze*

Vediamo un semplice esempio di un algoritmo che legge tre numeri e ne individua il maggiore.
Per prima cosa scriviamo il codice sequenziale dell'algoritmo in *pseudocodifica (algoritmo sequenziale)*:

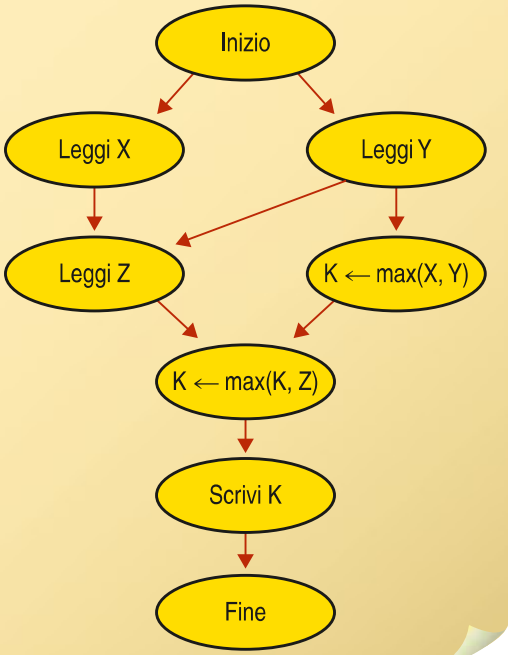
```
   inizio
1. leggi X;
2. leggi Y;
3. leggi Z;
4. K ← max(X;Y);
5. K ← max(K;Z);
6. scrivi K;
   fine
```

Ora riportiamo le istruzioni in un grafo dove esprimiamo i vincoli di effettiva precedenza per l'esecuzione delle istruzioni: *leggi X* e *leggi Y* non devono essere eseguite per forza in questo ordine, anzi, potrebbero anche essere effettuate in parallelo; anche la lettura di Z può essere eseguita dopo l'istruzione 4, oppure in parallelo con essa, mentre le istruzioni 5 e 6 devono per forza chiudere la sequenza delle operazioni.

Il grafo è così fatto: ▶



Questo non è l'unico diagramma delle precedenze possibile: la lettura di Z potrebbe essere eseguita dal ramo che effettua la lettura di X e l'operazione di calcolo del massimo tra X e Y potrebbe essere eseguita al suo posto, ottenendo ▶



che è altrettanto logicamente corretto.

I grafi ottenuti sono quindi dei grafi a ordinamento parziale.

ESEMPIO 7

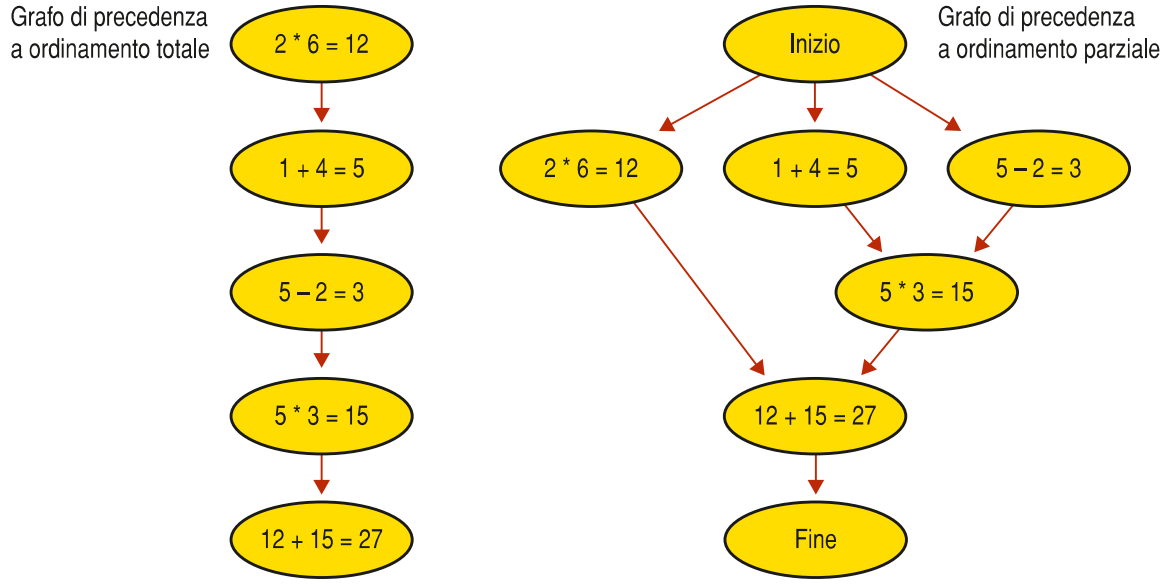
Ordinamento parziale e totale

Vediamo un secondo esempio, dove il problema da risolvere è quello di valutare una espressione matematica: (2 * 6) + (1 + 4) * (5 - 2)

Le regole di precedenza in questo caso sono dettate dalla matematica, dove dapprima si devono eseguire le operazioni all'interno delle parentesi e successivamente si deve rispettare le precedenza degli operatori (/, *, + e -).

Non esiste l'obbligo un ordinamento totale fra le operazioni da eseguire per il calcolo delle parentesi: potremmo indifferentemente eseguire prima (1 + 4) piuttosto che (2 * 6) o viceversa senza compromettere il risultato finale.

Rappresentiamo la soluzione con il grafo sequenziale a ordinamento totale per confrontarla con quello a ordinamento parziale, dove introduciamo la parallelizzazione delle attività (che in questo caso sono tutte operazioni algebriche):



Scomposizione di un processo non sequenziale

Un processo non sequenziale consiste nella elaborazione contemporanea di più processi che sono di tipo sequenziale, e quindi possono essere studiati, descritti e programmati singolarmente.

SCOMPOSIZIONE SEQUENZIALE

Un processo non sequenziale può essere scomposto in un insieme di processi sequenziali che possono essere eseguiti contemporaneamente.

Possiamo quindi affrontare lo studio dei processi paralleli scomponendoli in processi sequenziali e risolvendoli ciascuno separatamente, con la programmazione classica dei processi sequenziali.

Per poter correttamente descrivere la concorrenza è necessario distinguere le attività che i processi eseguono in due tipologie:

- attività completamente indipendenti;
- attività interagenti.

Processi indipendenti

La situazione più semplice da gestire è quella nella quale i processi sono tra loro completamente indipendenti.

PROCESSI INDIPENDENTI

L'evoluzione di un processo non influenza quella dell'altro.

Quindi sia che vengano eseguiti in sequenza che in parallelo non possono in nessun caso generare situazioni di funzionamento problematiche: l'unica accortezza è quella di rispettare per ogni processo l'ordine stabilito delle operazioni.

ESEMPIO 8 *Scomposizione in processi indipendenti*

Supponiamo di avere il seguente segmento di codice:

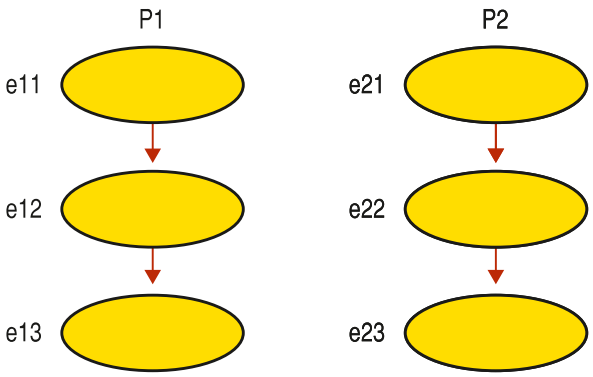
```
   inizio
1. leggi X;
2. leggi Y;
3. K ← SQRT(X);
4. W ← log(Y);
5. scrivi K;
6. scrivi W;
   fine
```

Possiamo scomporre il programma in due segmenti completamente indipendenti:

segmento 1	**segmento 2**
1. leggi X;	2. leggi Y;
3. K ← SQRT(X);	4. W ← log(Y);
5. scrivi K;	6. scrivi W;

Otteniamo due processi che eseguono ciascuno tre elaborazioni che *non hanno nulla in comune*, quindi ciascuno dei due processi può evolvere autonomamente senza interessarsi di quello che sta facendo l'altro. ▶

Il grafo di precedenze può essere scomposto in due grafi completamente autonomi.



Due processi indipendenti devono lavorare su un insieme privato di variabili e ogni variabile che ciascuno di essi modifica non può essere utilizzata da nessun altro processo: l'unico caso in cui due processi indipendenti utilizzano una variabile comune è quello in cui su tale variabile effettuano solo operazioni di lettura.

L'ultima osservazione che possiamo fare su un insieme di processi concorrenti disgiunti è che il risultato di ciascuno di essi è indipendente dalla velocità con la quale viene eseguito ma dipende unicamente dai dati di ingresso.

Processi interagenti

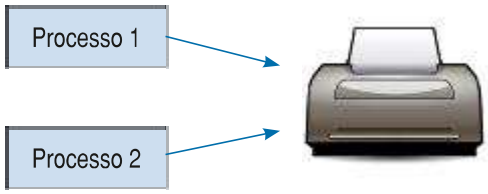
La seconda situazione è quella in cui i due (o più) processi non possono evolvere in modo completamente autonomo perché devono interagire o volontariamente o involontariamente e quindi la rappresentazione nel grafo delle precedenze dovrà necessariamente avere degli elementi comuni.

È possibile classificare le modalità di interazione tra processi in base alla loro conoscenza o meno della presenza degli altri.

A processi totalmente ignari
in questo caso i processi sono indipendenti e non sono stati progettati per lavorare assieme ma "evolvono" in un ambiente comune: interagiscono tra loro in competizione sulle risorse e si devono quindi sincronizzare.
Questa situazione viene gestita dal sistema operativo, che deve essere arbitro della loro evoluzione effettuando la sincronizzazione all'accesso delle risorse ogni volta che i processi le richiedono.

ESEMPIO 9 ***Processi in competizione***

I processi sono in competizione per l'accesso a una stampante



oppure per l'utilizzo di una tabella di dati o di file che non può essere letta da un processo mentre viene modificata da un altro.

B processi indirettamente a conoscenza uno dell'altro
è questa la situazione nella quale i processi sono a conoscenza dell'esistenza degli altri ma non ne conoscono il nome (o il ◀ PID ▶) e non possono comunicare direttamente tra loro ma devono cooperare per qualche motivo e possono scambiarsi i dati utilizzando risorse comuni, come aree di memoria condivisa.
Il sistema operativo deve offrire dei meccanismi di sincronizzazione che rendano possibile la cooperazione;

◀ **PID** In *nix, a PID is a process ID. It is generally a 15 bit number, and it identifies a process or a thread. ▶

C processi direttamente a conoscenza uno dell'altro
in questa situazione i processi devono cooperare per qualche motivo ma comunicano tra loro conoscendo i propri nomi: possono quindi effettuare direttamente lo scambio di informazioni mediante invio di messaggi espliciti.

> Il sistema operativo deve offrire dei meccanismi di comunicazione che rendano possibile la cooperazione.

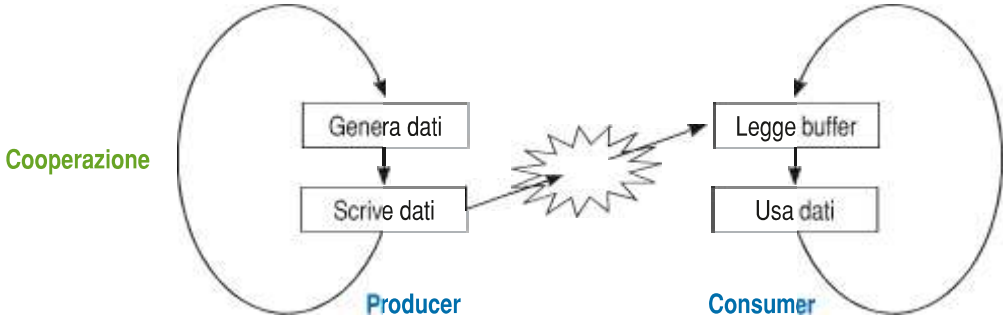
Meccanismi di comunicazione e sincronizzazione tra entità (anticipazioni)

Negli ultimi due casi i processi devono cooperare, quindi il sistema operativo deve offrire i meccanismi di sincronizzazione in modo da garantire il corretto funzionamento regolando e gestendo i vincoli sull'ordine con cui devono essere eseguite le operazioni.

> Una scorretta sincronizzazione dei processi dà luogo a una particolare categoria di errori, gli errori dipendenti dal tempo: questo tipo di errori spesso sono di difficile individuazione anche perché legati alla velocità relativa dei singoli processi e alla loro evoluzione autonoma, e quindi potrebbero manifestarsi o meno a seconda della casistica delle alternative di computazione in base alle diverse istanze di esecuzione.
> Un errore dipendente dal tempo potrebbe non ripetersi anche riavviando il sistema e riportandosi nelle medesime condizioni nelle quali si è manifestato una prima volta.

È quindi di fondamentale importanza la scelta di tecniche corrette di sincronizzazione che possono essere realizzate in tre modalità differenti:

A attraverso l'utilizzo di aree dati comuni (memoria condivisa, in inglese Shared Memory): un processo produce un dato (producer o produttore) e lo scrive nella memoria condivisa (buffer comune) in modo che l'altro processo (consumer o consumatore) lo possa leggere e utilizzare.



Spesso questa situazione viene regolata mediante un meccanismo chiamato monitor: questo si occupa di gestire la memoria in modo sincronizzato ricevendo dati creati dal *produttore* e gestendo le richieste di lettura e di risposta del *consumatore*.

B attraverso lo scambio di messaggi un processo trasmette le informazioni all'altro processo: il meccanismo a disposizione è realizzato con dei meccanismi simili a semplici operazioni di I/O che prendono il nome di InterProcess Communication (IPC).
La comunicazione diretta viene realizzata con due primitive dove ogni processo deve specificare il "nome" dell'altro processo con il quale deve comunicare

```
send(processo_destinatario, messaggio)
receive(processo_mittente, messaggio)
```

Questo è il meccanismo di più basso livello e qualsiasi sistema di interconnessione è in grado di supportarlo e può essere realizzato con due tipi di messaggi:

- messaggi sincroni: il mittente si ferma in attesa della risposta;
- messaggi asincroni: il mittente prosegue nella sua esecuzione e periodicamente (*polling*) o mediante un meccanismo di wake up (*eventi*) verifica le risposte ricevute.

C attraverso la chiamata a Procedura Remota (RPC Remote Procedure Call) un processo ha la possibilità di invocare una funzione di un servizio remoto come fosse una libreria locale: possiamo vedere questo meccanismo come una estensione del concetto client-server dove il server mette a disposizione le procedure remote che il processo cliente può invocare e tramite queste scambia informazioni o accede ad aree di memoria condivisa.

La bufferizzazione

Sia nella comunicazione diretta, sia in quella indiretta, i messaggi scambiati dai processi che comunicano risiedono in una coda temporanea.
Vi sono tre modi per implementare questa coda a seconda delle dimensioni del buffer di comunicazione:

1. *capacità zero*: in questo caso il mittente deve bloccarsi finché il destinatario riceve il messaggio;
2. *capacità limitata*: se il buffer è pieno il mittente deve bloccarsi;
3. *capacità illimitata*: il mittente non si blocca mai.

In base alla capacità del buffer il mittente non solo deve sincronizzarsi per accedere alla risorsa buffer ma a seconda del tipo di buffer ha dei comportamenti completamente differenti che vanno valutati caso per caso e che descriveremo nel seguito della nostra trattazione.