LEZIONE 5

LA DESCRIZIONE DELLA CONCORRENZA

IN QUESTA LEZIONE IMPAREREMO...

- l'istruzione fork-join
- l'istruzione cobegin-coend

Esecuzione parallela

L'esecuzione di un processo non sequenziale richiede un sistema specifico che permetta la codifica e l'esecuzione dei programmi, cioè necessita di:

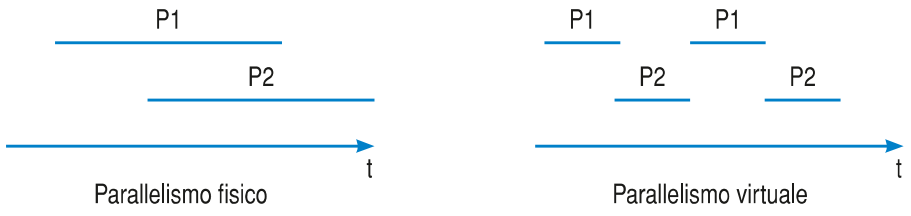
- un elaboratore non sequenziale;
- un linguaggio di programmazione non sequenziale.

Elaboratore non sequenziale

L'elaboratore non sequenziale è una macchina che deve essere in grado di eseguire più operazioni contemporaneamente e, come abbiamo detto, sostanzialmente questo si può ottenere in due modi differenti:

- architettura parallela, cioè sistemi multielaboratori;
- sistemi monoprocessori multiprogrammati.

Nei primi il parallelismo è fisico, mentre nei secondi il parallelismo è virtuale.



Nei nostri esempi tratteremo sempre il secondo caso in modo da poter scrivere e collaudare i programmi sui nostri PC che hanno tutti un sistema operativo multiprogrammato.

Linguaggi non sequenziali

Per scrivere programmi non sequenziali (o concorrenti) è inoltre necessario avere a disposizione dei particolari costrutti che permettano la descrizione delle attività parallele: non tutti i linguaggi hanno tali figure strutturali e quelli che consentono la descrizione delle attività concorrenti prendono il nome di linguaggi di programmazione concorrente (o non sequenziale).

La scrittura di un programma concorrente si basa sul concetto di scomposizione sequenziale.

Quindi il programmatore deve dapprima individuare le attività parallelizzabili e descriverle in termini di sequenze di istruzioni (blocchi sequenziali) e, successivamente mediante i linguaggi concorrenti, descrivere come tali moduli possono essere eseguiti in parallelo.

SCOMPOSIZIONE SEQUENZIALE

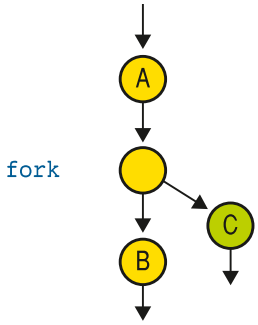
Un processo non sequenziale può essere scomposto in un insieme di processi sequenziali che possono essere eseguiti contemporaneamente.

I linguaggi di programmazione concorrenti di alto livello contemplano nuove istruzioni come il costrutto fork-join e cobegin-coend, che permettono di dichiarare, creare, attivare e terminare processi sequenziali.

Fork-join

Le istruzioni fork e join furono introdotte nel 1963 da Dennis e VanHorne per descrivere l'esecuzione parallela di segmenti di codice mediante la scomposizione di un processo in due processi e la successiva "riunione" in un unico processo.

Fork



In rifermento al grafo delle precedenze, la fork corrisponde alla biforcazione di un nodo in due rami: l'esecuzione di una fork coincide con la creazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante.

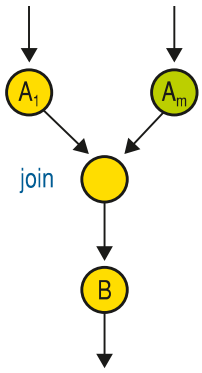
Con un linguaggio di pseudocodifica è possibile tradurre il grafo nel seguente segmento di codice:

```
/* processo padre: */
inizio
…
A: <istruzioni>;
p2 = fork figlio1;
 B: <istruzioni>;
…
fine.

/* codice nuovo processo:*/
void figlio1()
{
C: <istruzioni>;
}
```

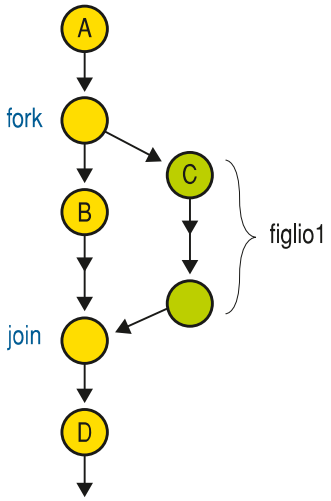
Join

La join è l'istruzione che viene eseguita quando il processo creato tramite la fork, ha terminato il suo compito, si sincronizza con il processo che lo ha generato e termina la sua esecuzione. ▶
Con un linguaggio di pseudocodifica è possibile tradurre il grafo nel seguente segmento di codice:




```
…
join p2;
B:<istruzioni>
…
```

Il programma completo rappresentato nel diagramma a lato ▶ viene codificato in pseudolinguaggio con:

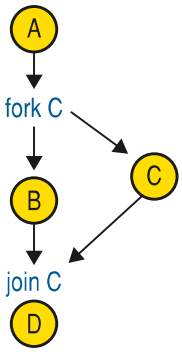



```
/* processo padre: */
inizio
…
A:<istruzioni>;
p2 = fork figlio1;      // inizia l'elaborazione parallela
   B: <istruzioni>;
join p2;                // termina l'elaborazione parallela
D:<istruzioni>;
…
fine

/* codice nuovo processo:*/
void figlio1 ()
{
   C:<istruzioni>;
}
```

Se ipotizziamo che ogni nodo sia una *routine*, possiamo meglio comprendere il funzionamento della istruzione fork con il seguente schema, dove sono messi in evidenza i due processi:

- P1 esegue la Routine A;
- con la fork attiva il processo 2 e in parallelo si eseguono la routine B e la routine C;
- P1 attende con la join la terminazione di P2;
- P1 esegue la routine C.



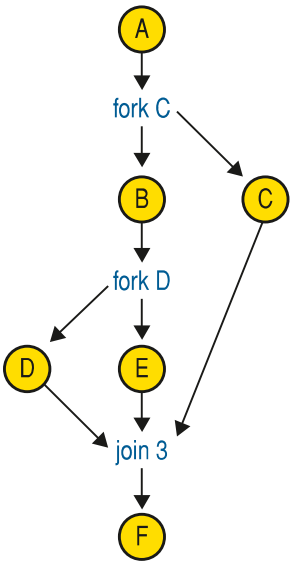
In questa definizione, la fork restituisce un identificatore di processo, che viene successivamente accettato da una join, in modo che sia specificato con quale processo si intende sincronizzarsi. Lo svantaggio di questa definizione è che la join può ricongiungere *solo due flussi di controllo*.

Join (count)

Esiste anche una formulazione estesa dell'istruzione join:

```
join(count);
```

dove count è una variabile intera non negativa e indica il numero di processi che si "riuniscono" in quel punto: viene utilizzata nel caso di terminazione congiunta di un numero superiore a due processi, come nel seguente grafo delle precedenze ▶



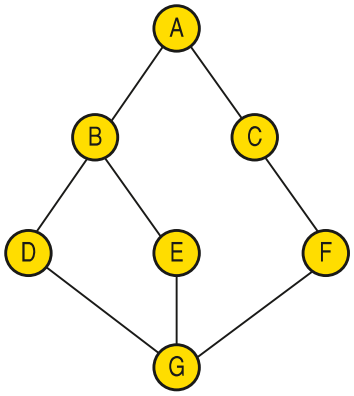
che viene codificato con questo segmento di programma:

```
/* processo padre: */
inizio
…
A:<istruzioni>;
p2 = fork C;            // inizia l'elaborazione parallela con P2
   B: <istruzioni>;
   p3 = fork D;         // inizia l'elaborazione parallela con P3
      E: <istruzioni>;
join 3;                 // termina l'elaborazione parallela di 3 processi
F:<istruzioni>;
…
fine.
```

Nella letteratura la sintassi della pseudocodifica a volte è leggermente differente da quella appena presentata: spesso viene semplificata omettendo le <istruzioni> e indicando semplicemente con le lettere maiuscole A,B,C oppure con S1,S2,…, Sn il blocco o la sequenza di istruzioni, e tale simbologia sarà adottata anche da noi per le prossime codifiche.

A volte viene anche messa una label per indicare dove il processo termina ed effettua la join, come nel seguente esempio: ▶

```
begin
   cont : = 3 ;
   A ;
   FORK E1 ;
   B ;
   FORK E2 ;
   D ;
   goto E3 ;
E1 : C ;
   F ;
   goto E3 ;
E2 : E ;
E3 : JOIN cont ;
   G ;
end.
```



Il linguaggio di shell UNIX/Linux ha due system call *simili* ai costrutti di alto livello definiti da Dennis e VanHorne:

- fork(): è l'istruzione utilizzata per creare un nuovo processo;
- wait(): corrisponde alla istruzione join e viene utilizzata per consentire a un processo di attendere la terminazione di un processo figlio.

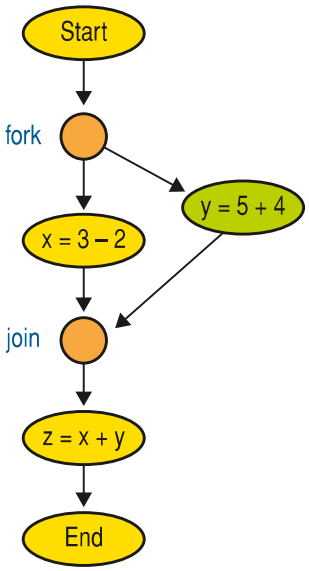
Sono però concettualmente diverse dato che sono *chiamate di sistema* mentre quelle da noi descritte sono istruzioni di un ipotetico *linguaggio concorrente di alto livello*: possono essere comunque utilizzate didatticamente per realizzare programmi concorrenti e alcuni esempi di codifiche in linguaggio C che le utilizzano in ambiente Linux sono riportati a titolo di esempio in una lezione dedicata al laboratorio a conclusione di questa unità di apprendimento.

ESEMPIO 10

Scriviamo un programma parallelo che esegue la seguente espressione matematica:

$$z = (3\text{-}2) * (5\text{+}4)$$

Le operazioni tra parentesi possono essere parallelizzate ottenendo il seguente grafo: ▶



che viene codificato in:

```
/* processo padre: */
inizio
   p2 = fork(figlio1);      // inizia l'elaborazione parallela
   x=3-2;
   join p2;                 // termina l'elaborazione parallela
   z=x+y;
…
fine

/* codice nuovo processo:*/
int figlio1()
{
   y=5+4;
   return y
}
```

Cobegin-coend

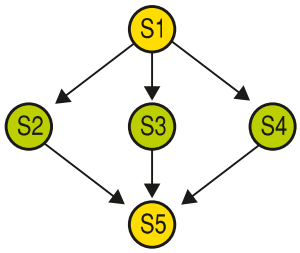
Il secondo costrutto che utilizziamo per descrivere la concorrenza è il cobegin-coend: è un costrutto che permette di indicare il punto in cui N processi iniziano contemporaneamente l'esecuzione (cobegin) e il punto che la terminano, confluendo nel processo principale (coend).

La sintassi del comando è:

```
cobegin <elenco delle attività parallele>
coend
```

ESEMPIO 11

Il grafo delle precedenze di figura: ▶
può essere descritto mediante il costrutto cobegin-coend con la seguente pseudocodifica:



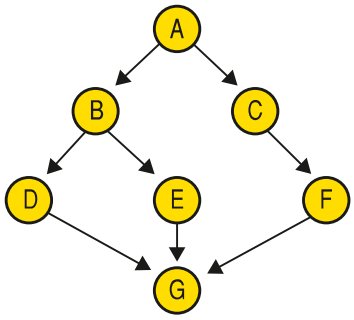

```
inizio
   S1
   cobegin
      S2        //attività parallele
      S3
      S4
   coend
   S5
fine
```

Al termine dell'esecuzione della sequenza S1 vengono attivati tre processi che eseguono in parallelo le sequenze di istruzioni S2, S3, S4: il processo padre S1 si sospende e rimane in attesa della loro terminazione.

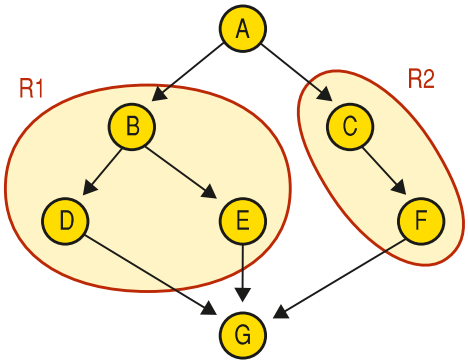
All'interno dei singoli processi possono a loro volta essere presenti delle istruzioni di cobegin-coend, cioè è possibile avere l'annidamento di più costrutti cobegin-coend.

ESEMPIO 12

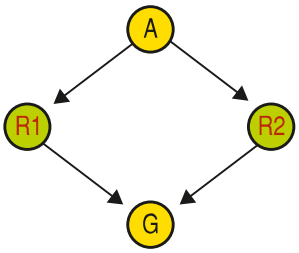
Codifichiamo il seguente grafo delle precedenze utilizzando il costrutto cobegin-coend.



Osservando il grafo delle precedenze possiamo notare come è sostanzialmente composto da due rami che possono essere evidenziati come nella seguente figura



Chiamando R1 e R2 le due sequenze di istruzioni, il grafo può essere semplificato come segue:



Il codice in pseudocodifica è quindi il seguente:

Processo P1	Processo R2	Processo R3

inizio
  A
  cobegin
  R1
  R2
  coend
  G
fine
	inizio
	  B
	  cobegin
	  D
	  E
	  coend
	fine
		inizio
		  C
		  F
		fine

Il costrutto cobegin-coend è decisamente più semplice da utilizzare e il codice che si ottiene è più strutturato e quindi più comprensibile di quello scritto con le istruzioni di fork-join che trasformano il codice in strutture che assomigliano ai vecchi programmi che utilizzavano l'istruzione di salto incondizionato "goto label", censurata dalla programmazione strutturata per la generazione di ◀ **"spaghetti code"** ▶ incomprensibili.

◀ **Spaghetti code** Spaghetti code is a derogatory term for computer programming that is unnecessarily convoluted, and particularly programming code that uses frequent branching from one section of code to another (using many GOTOs or other "unstructured" constructs). Spaghetti code sometimes exists as the result of older code being modified a number of times over the years. ▶

CAMMINO PARALLELO

Indichiamo con cammino parallelo una qualunque sequenza di nodi delimitata da un nodo COBEGIN e un nodo COEND.

Nell'esempio precedente è possibile individuare due cammini paralleli, uno esterno e uno annidato al suo interno.

Equivalenza di fork-join e cobegin-coend

Il costrutto fork-join può essere sostituito col costrutto cobegin-coend solo se non ci sono strutture annidate, nel tal caso l'unica possibilità di "conversione" è quella che ha la terminazione congiunta di tutti i processi: nel caso opposto, invece, sempre tutti i programmi codificati con cobegin-coend possono essere anche codificati con fork-join.

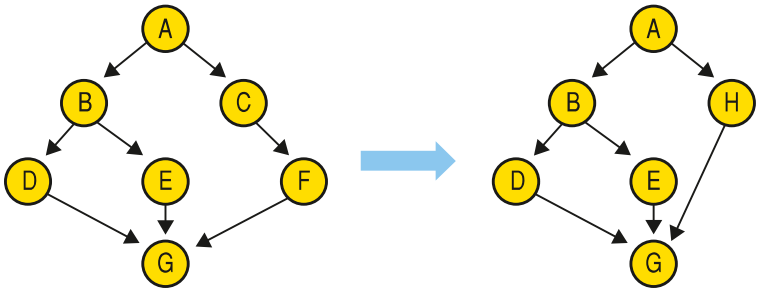
EQUIVALENZA TRA FORMALISMI

Qualunque programma parallelo può essere descritto utilizzando il costrutto fork-join, mentre non tutti i programmi paralleli possono essere descritti col costrutto cobegin-coend.

Dimostriamolo sperimentalmente tramite due semplici esempi: convertiamo un segmento descritto con cobegin-coend a fork-join e successivamente proviamo a eseguire l'operazione inversa.

ESEMPIO 13 *Da cobegin-coend a fork-join*

Scriviamo mediante l'utilizzo di fork-join l'esempio che presenta due cobegin-coend annidati.



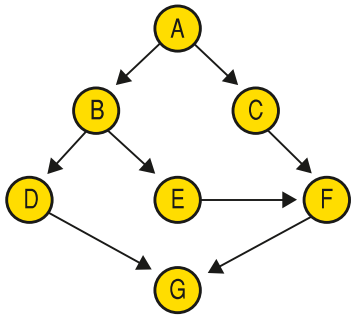
Dopo aver posto H = C+F, la pseudocodifica è la seguente:

```
/* processo padre: */
inizio
…
A
p2 = fork(H);          // inizia l'elaborazione parallela di B e H
   B
   p4 = fork(E);       // inizia l'elaborazione parallela di D e E
      D
   join p4;            // termina l'elaborazione parallela di D e E
join P2;               // termina l'elaborazione parallela del primo fork
…
fine.

/* processo figlio H */
inizio
   C
   F
fine
/* processo figlio E */
inizio
   E
fine
```

ESEMPIO 14 ***Da fork-join a cobegin-coend***

Ipotizziamo ora di dover descrivere mediante cobegin-coend la situazione rappresentata nel seguente grafo delle precedenze: ▶



Possiamo notare come il processo che esegue A si sospende per mandare in esecuzione due processi (cobegin P_B e P_C) dei quale il primo, a sua volta, si sospende per mandare in esecuzione altri due processi (cobegin P_D e P_E): è però impossibile determinare i corrispondenti coend in quanto i processi figli interagiscono tra di loro generando nuovi processi che non terminano assieme: quindi P_B non termina assieme a P_C

GRAFO STRUTTURATO

Un grafo per poter essere espresso soltanto con cobegin e coend deve essere tale che detti X e Y due nodi del grafo, tutti i cammini paralleli che iniziano da X terminano con Y e tutti quelli che terminano con Y iniziano con X (o, più sinteticamente, ogni 'sotto-grafo' deve essere del tipo one-in/one-out).
In questo caso il grafo si dice strutturato.

Il grafo dell'esempio precedente non presenta questa caratteristica e quindi, non essendo strutturato, costituisce un esempio impossibile da descrivere soltanto con cobegin e coend.

La codifica con il costrutto fork-join è invece fattibile, ma è necessario introdurre le istruzioni di salto in quanto le fork si intersecano tra loro:

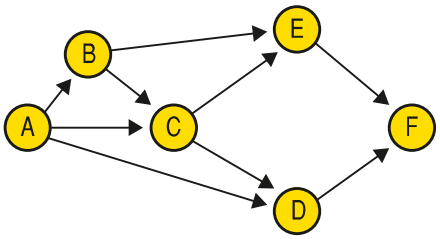
```
/* processo padre: */
inizio
…
A
p2=fork(C);        // inizia l'elaborazione parallela di B e C
   B
   P3=fork(E);     // inizia l'elaborazione parallela di D e E
      D
      goto L1
   join C;         // aspetta la terminazione di C
L1:join E2;        // aspetta la terminazione di E join C
G
…
fine.
```

La join etichettata con L1 non avviene tra gli stessi due processi che hanno fatto la prima fork, ma tra il primo processo e un processo generato dalla join tra due figli, il processo che esegue E e quello che esegue C.

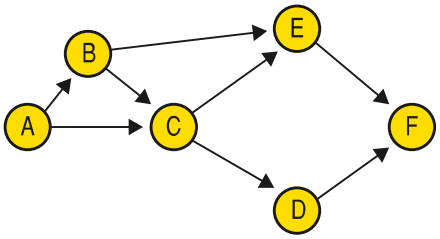
Semplificazione delle precedenze

Prima di affrontare la scrittura di un programma parallelo è sempre necessario soffermarsi ad analizzare il grafo delle precedenze per cercare di semplificarlo eliminando le precedenze implicite.

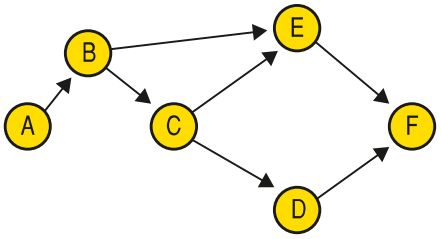
In presenza di precedenze implicite è possibile togliere un arco in quanto questo risulta ridondante ed è quindi possibile semplificare il grafo: vediamo un esempio e semplifichiamo il grafo riportato nella figura seguente:



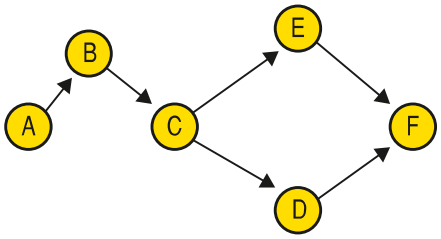
Possiamo osservare che il nodo D ha come precedenza il nodo A e anche il nodo C ha come precedenza il nodo A: quindi il nodo D ha il nodo A come precedenza diretta ma deve attendere l'elaborazione di C che a sua volta dipende da A; è possibile eliminare la precedenza tra A → D che risulta essere implicita in quella tra C → D: il grafo risulta essere trasformato nel seguente:



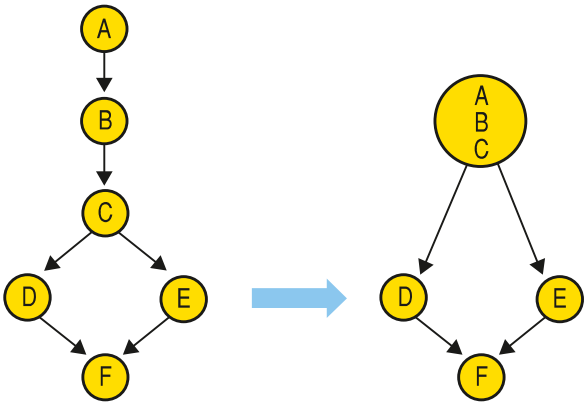
Analogo discorso può essere fatto tra B, che ha come precedenza il nodo A, e il nodo C, che anch'esso ha come precedenza il nodo A: quindi il nodo B ha il nodo A come precedenza diretta e dato che deve attendere l'elaborazione di B che a sua volta dipende da A è possibile eliminare la precedenza tra A → C che risulta essere implicita in quella tra B → C: il grafo risulta essere trasformato nel seguente:



Analogo discorso può essere fatto tra E, che ha come precedenza il nodo B, e il nodo C, che anch'esso ha come precedenza il nodo B: quindi il nodo E ha il nodo B come precedenza diretta e dato che deve attendere l'elaborazione di C, che a sua volta dipende da B, è possibile eliminare la precedenza tra B → E che risulta essere implicita in quella tra C → E; il grafo risulta essere trasformato nel seguente:



Ridisegnandolo, osserviamo che le tre operazioni A, B e C sono **sequenziali**, possono quindi essere raggruppate in un solo nodo, come si può vedere nel seguente disegno.



Il nostro grafo di partenza può essere trasformato in quest'ultimo, dove si vede che l'unica operazione che può essere parallelizzata è quella dei nodi D ed E: parallelizzare il primo grafo non porterebbe nessun vantaggio in termini di tempo di elaborazione.