

### **Il descrittore di processo (PCB)**

Il S.O. gestisce i processi associando a ciascuno di essi un struttura dati di tipo record detta descrittore di processo o **Process Control Block (PCB)**

Il PCB contiene tutte le informazioni relative a: stato globale, risorse possedute (memoria centrale, unità di I/O assegnate staticamente, file aperti, ecc.), parametri di schedulazione per l'assegnazione di alcune risorse, valori corrente dei registri del processore all'atto dell'uscita dallo stato running, posizione dell'area di swap su disco se privo della risorsa memoria centrale, informazioni di account, ecc.

### **Il descrittore di processo (PCB)**

Il PCB viene assegnato ad un processo quando viene posto nello stato new e viene deassegnato quando esce dallo stato terminated e ne viene persa traccia

Tipicamente i PCB sono disposti in code costituite da liste a puntatori. Appartengono ad una stessa coda PCB di processi caratterizzati dallo stesso stato globale (ad esempio coda dei processi ready ad uguale priorità di schedulazione, coda dei processi waiting in attesa di avviare una operazione di accesso ad un disco, coda dei processi waiting in attesa di sincronizzarsi con uno stesso processo di servizio, ecc.)

### **Il cambiamento di contesto**

L'abbandono del processore da parte di un processo provoca il cosiddetto **cambiamento di contesto** (context switch) nel senso che cambia il contesto a cui appartengono le informazioni contenute nei registri del processore.

Il contenuto dei registri appartenente al processo uscente viene salvato nel suo descrittore, e, effettuata la schedulazione del processo entrante, il contenuto dei registri del processore viene ripristinato con le informazioni prelevate dal descrittore di quest'ultimo

### **Il cambiamento di contesto**

Il cambiamento di contesto è quindi il risultato dell'esecuzione di tre procedure distinte del S.O..

**Procedure** *Salvataggio\_stato*

**Procedure** *Ripristino\_stato*

**Procedure** *Scheduling\_CPU*

**Procedure** *Context\_Switch*

**begin**

*Salvataggio\_stato, Scheduling\_CPU, Ripristino\_stato*

**end**

### Primitiva fork per la creazione di processi

La **fork(id)** è una primitiva che consente ad un processo (processo padre) di creare un altro processo (processo figlio) nello stato new o nello stato ready

Il padre e il figlio potranno condividere risorse ed in particolare potranno condividere o meno lo spazio di memoria (area istruzioni e area dati)

E' nota come **chiamata asincrona di procedura** ma, a differenza di una chiamata di procedura, le procedure chiamante e chiamata sono eseguite nei differenti contesti dei processi padre e figlio e possono quindi essere eseguite in parallelo

### Primitiva fork per la creazione di processi

A: *fork(X)*

B: <istruzione successiva alla fork>

.....

X: <prima istruzione della procedura invocata>

**var** *P*: *process*

**procedure** *X*

**begin**.....**end**

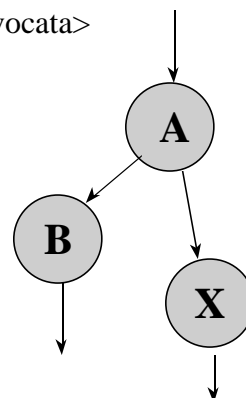
**begin**

.....

*P*:= *fork(X)*

.....

**end**



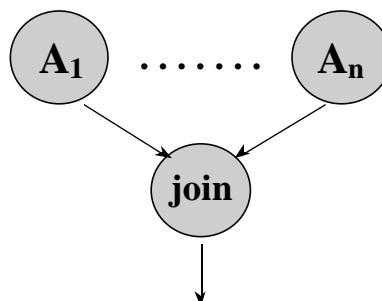
## Primitiva fork per la creazione di processi

```
Procedure fork (id)  
begin  
  if esiste-PCB-libero then  
    <alloca un descrittore libero al processo figlio>  
    <alloca spazio di memoria ed inizializzalo>  
    <inizializza descrittore>  
    <poni il processo figlio nello stato ready>  
  else  
    <poni il processo padre in attesa di un descrittore libero>  
    context_switch  
  end  
end
```

## Primitiva join(cont) per la sincronizzazione e terminazione di processi

Dati  $n$  processi la **join(cont)** è una primitiva che consente a  $n-1$  processi di terminare e all' $n$ -esimo di proseguire sincronizzandosi con la terminazione dell' $(n-1)$ esimo processo.

```
var cont: integer  
.....  
.....  
join(cont)
```



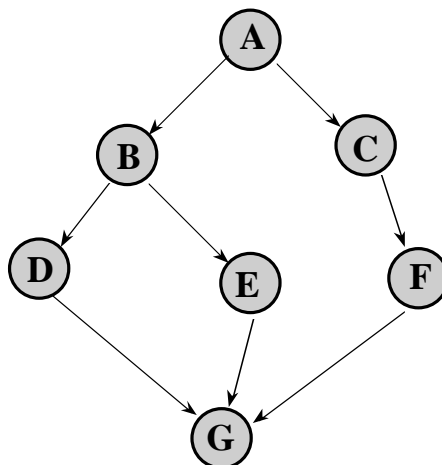
### Primitiva join(cont) per la sincronizzazione e terminazione di processi

Non consente di denotare in maniera esplicita il o i processi con cui si vuole sincronizzare.

```
procedure join(cont)
begin
  cont:=cont-1
  if cont<>0 then begin
    <termina il processo chiamante>,
    Scheduling_CPU, Ripristino_stato
  end
end
```

### Primitiva join(cont) per la sincronizzazione e terminazione di processi

```
begin
  cont:=3;
  A;
  fork (E1);
  B;
  fork (E2);
  D;
  goto E3;
E1: C;
  F;
  goto E3;
E2: E;
E3: join (cont);
  G;
end
```



### **Primitiva join(pid) per la sincronizzazione di un processo con la terminazione di un altro**

La **join(pid)** è una primitiva che consente ad un processo (di solito il processo padre) di sincronizzarsi con la terminazione di un altro (di solito un processo figlio)

```
procedure join(pid)
begin
  if stato(pid) <> terminated then
    begin
      <metti il processo chiamante in attesa che il processo
      pid termini>
      context_switch
    end
  end
end
```

### **Primitiva exit per la terminazione di un processo**

La **exit** è una primitiva che consente ad un processo di terminare

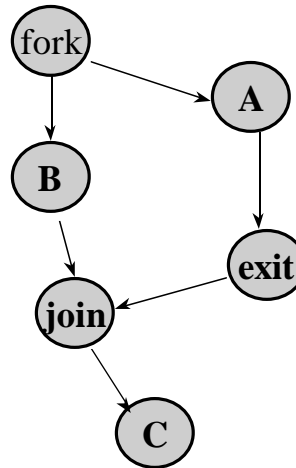
```
procedure exit
begin
  <poni il processo chiamante nello stato terminated>,
  if <processo padre in attesa della terminazione> then
    <sposta il processo padre nella coda dei ready>
    Scheduling_CPU,
    Ripristina_stato
  end
```

### Uso delle primitive fork, join e exit

```

var P: process
procedure X
begin
  A;
  exit
end
begin
  .....
  P:= fork(X)
  B;
  join (P)
  C;
end

```



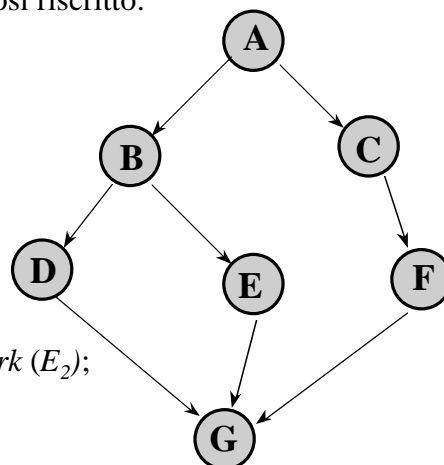
### Uso delle primitive fork, join e exit

L'esempio precedente viene così riscritto:

```

var P1, P2: process
procedure E1
begin C; F; exit; end;
procedure E2
begin E; exit; end;
begin
  A; P1:= fork (E1); B; P2:= fork (E2);
  D; join (P1); join (P2); G;
end

```



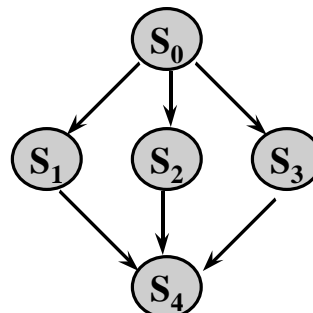
### Uso delle primitive fork, join e exit

```
var BL, BE, BS: array [1..N] of T; i: 1..N;  
    PE, PS: array[1..N] of process;  
procedure E(k:1..N);  
begin  
    if k < 1 then join (PE[k-1]);  
    Elaborazione (BE[k]);  
    BS[k] := BE[k];  
    PS[k] := fork (S(k));  
    exit;  
end;  
procedure S(k:1..N);  
begin  
    if k < 1 then join (PS[k-1]);  
    Scrittura (BS[k])  
    exit  
end;  
begin  
    for i:= 1 to N do
```

### Cobegin e coend

Trattasi di costrutti di linguaggi concorrenti per la creazione di processi figli da parte di un processo padre e la sua sincronizzazione con la loro terminazione

```
begin  
    S0  
    cobegin  
        S1  
        S2  
        S3  
    coend  
    S4  
end
```

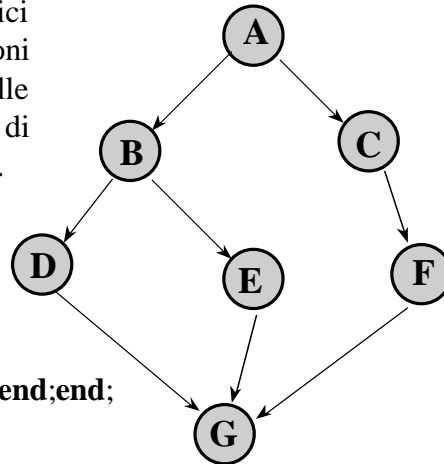




### Cobegin e coend

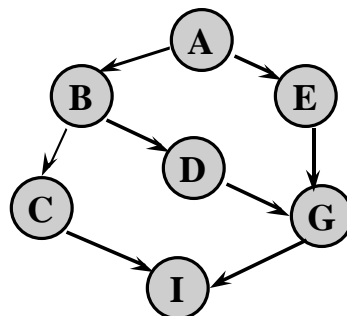
In quanto costrutti linguistici cobegin e coend sono istruzioni di più alto livello rispetto alle primitive fork e join e quindi di più agevole uso. Ad esempio...

```
begin
  A;
  cobegin
    begin C; F; end;
    begin B; cobegin D; E; coend; end;
  coend
  G;
end
```



### Cobegin e coend

Il gap esistente tra un costrutto linguistico e le primitive del S. O. viene colmato dal compilatore del linguaggio. In particolare cobegin e coend possono essere tradotti in termini di fork e join. Peraltro non è sempre vero il viceversa



## Cobegin e coend

Compilazione del costrutto in termini di *fork* e *join*:

```
var  $P_1, \dots, P_n$ : process;  
procedure  $S_1$ ;  
begin.....end;  
  
.....  
procedure  $S_n$ ;  
begin.....end;  
begin  
   $S_0$ ;  
   $P_1 := \text{fork}(S_1)$ ;  
   $P_n := \text{fork}(S_n)$ ;  
   $S_{n+1}$ ;  
end;
```

## Uso delle primitive cobegin e coend

```
var A, B, C: T; i: 2..N;  
begin  
  Lettura (A);  
  cobegin Lettura (B); Elaborazione (A); coend;  
  i:=2;  
  while i<N do  
    begin  
      C:=A; A:=B;  
      cobegin Lettura (B); Elaborazione (A); Stampa (C); coend;  
      i:=i+1  
    end;  
    C:= A; A:=B;  
    cobegin Elaborazione (A); Stampa (C); coend;  
    Stampa (A)  
  end;
```

