

## Processi – parte IV

---

## Processi – parte IV

---

- Grafi di precedenza
- Sincronizzazione dei processi Unix mediante:
  - fork-join
  - cobegin-coend
  - semafori

## Grafi di precedenza

- Un grafo di precedenza è un grafo diretto aciclico dove i nodi rappresentano attività sequenziali e gli archi, ad esempio dal nodo  $i$  al nodo  $j$ , indicano l'ordine in cui devono essere completate le attività (l'attività  $i$  deve essere completata prima che l'attività  $j$  possa partire).
- Un nodo in un grafo di precedenza potrebbe essere:
  - Un task software
  - Un insieme di istruzioni all'interno di un programma che possono essere eseguite in modo concorrente
  - Attività che hanno luogo durante l'esecuzione di una singola istruzione macchina.

## Grafi di precedenza

- A seconda della "dimensione" di queste attività concorrenti si parla di granularità di concorrenza della rappresentazione:
  - Coarse granularity: concorrenza a livello di task di sistema operativo
  - Fine granularity: concorrenza a livello di singola istruzione o gruppo di istruzioni
  - Very fine granularity: concorrenza nell'hardware durante l'esecuzione della singola istruzione

## Esempio

- Si supponga di avere il seguente programma sequenziale:

$a := x + y$ ; /\* Statement s1 \*/

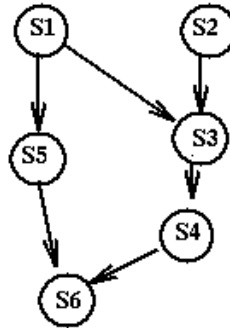
$b := z + 1$ ; /\* Statement s2 \*/

$c := a - b$ ; /\* Statement s3 \*/

$w := c + 1$ ; /\* Statement s4 \*/

$d := a + e$ ; /\* Statement s5 \*/

$w := w * d$ ; /\* Statement s6 \*/



## Esempio

- Il grafo di precedenza della slide precedente potrebbe essere eseguito usando due processori
  - Un processore potrebbe eseguire in ordine S1, S5 e S6
  - Il secondo processore S2, S3 e S4
- I due processori dovrebbero sincronizzarsi in modo tale da eseguire S3 dopo S2 e S6 dopo S4
- Questo grafo di precedenza è quello che più parallelizza le attività di un dato programma:
  - Le sole dipendenze che appaiono nel grafo sono quelle richieste dalla logica del programma

## Sincronizzazione

- Un linguaggio di alto livello per la programmazione concorrente deve fornire un insieme di costrutti linguistici che permettano di dichiarare, creare, attivare e terminare processi sequenziali. Due esempi di costrutti per la creazione/terminazione di processi sono `fork/join` e `cobegin/coend`
- N.B. anche se simile alla system call `fork()` di UNIX, la `fork` del costrutto `fork/join` è concettualmente diversa:
  - si tratta di una istruzione di un ipotetico linguaggio concorrente di alto livello.

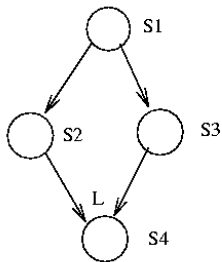
## Fork-join

- `Fork` e `Join` furono introdotte nel 1963 da Dennis e VanHorne.
- Quando viene eseguita l'istruzione:
  - `Fork(label);`  
dal programma in esecuzione è generato un secondo processo a partire dall'etichetta specificata nell'istruzione. I due processi procedono in parallelo.
- Quando viene eseguita l'istruzione:
  - `Join(count);`  
(dove `count` è una variabile intera non negativa) il valore di `count` è decrementato; se il valore risultante è positivo il processo che ha eseguito la `join` è terminato

## Grafi di precedenza e fork-join

- Qualunque grafo di precedenza può essere espresso in termini di fork e join senza perdita di concorrenza.
- N.B. Qualunque grafo di precedenza può essere eseguito come programma sequenziale a scapito della concorrenza.

## Esempio 1



```
int count = 2;
```

```
S1;  
FORK(L1);  
S2;  
GOTO L2;  
L1: S3  
L2: JOIN(count);  
S4;
```

## Esempio 2

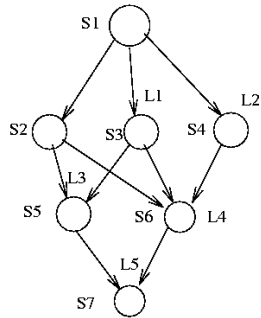


FIGURE 2a: A Precedence Graph

```

int count5 = 2;
int count6 = 3;
int count7 = 2;
  
```

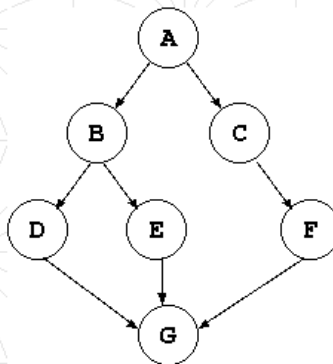
```

S1;
FORK(L1);
FORK(L2);
S2;
FORK(L4);
GOTO L3;
L1: S3;
FORK(L4);
GOTO L3;
L2: S4;
L4 : JOIN(count6);
S6;
GOTO L5;
L3: JOIN(count5);
S5;
L5: JOIN(count7);
S7;
  
```

## Esempio 3

```

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



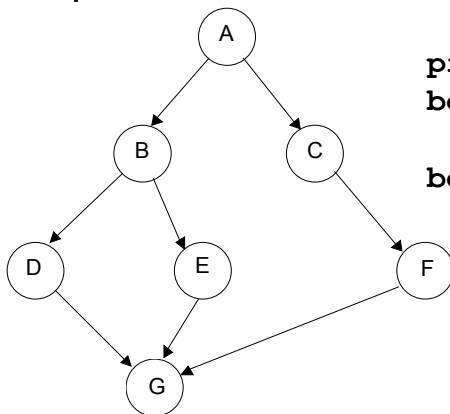
## Join tra processi

- join tra coppie di processi; la sintassi di questo tipo di join è:

- join P

dove P è un identificatore di processo (restituito dalla fork al padre quando viene creato un nuovo processo). L'esecuzione di questa istruzione fa in modo che il processo chiamante si sospenda in attesa della terminazione del processo con identificatore P (simile alla system call wait() di Unix).

## Esempio



```
var P1, P2 : process;
```

```
procedure E1;  
begin C;F; end
```

```
begin
```

```
  A;
```

```
  P1 := fork E1;
```

```
  B;
```

```
  P2 := fork E;
```

```
  D;
```

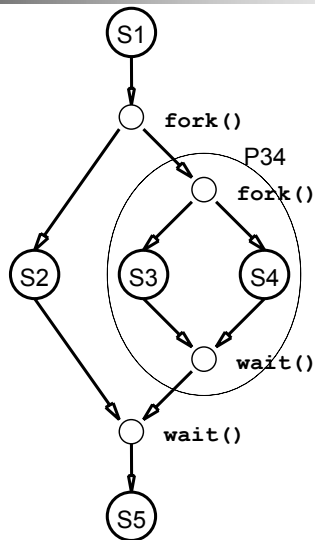
```
  join P1;
```

```
  join P2;
```

```
  G;
```

```
end
```

# Sincronizzazione di processi UNIX



# Sincronizzazione di processi UNIX

```
#include <sys/types.h>
#include <sys/wait.h>

main() {
    pid_t  childpid;

    printf( "parent - S1 - pid = %d\n", getpid() );
    if ( ( childpid = fork() ) == -1 )
        err_sys( "can't fork" );
    if ( childpid == 0 ){
        printf( "child: child pid = %d, parent
                pid = %d\n", getpid(), getppid() );
        P34();
    }
}
```

## Sincronizzazione di processi UNIX

```
else {
    printf( "parent - S2 - pid = %d\n",
           getpid() );
    printf( "parent: child pid = %d, parent
           pid = %d\n", childpid, getpid() );
    printf( "wait - P34 - \n" );
    while ( wait( (int *)0 ) != childpid );
        /* wait for child termination */
    printf( "parent - S5 - pid = %d\n",
           getpid() );
    exit( 0 );
}
}
```

## Sincronizzazione di processi UNIX

```
P34()
{
    pid_t pid4;

    printf( "fork - P34 - \n" );
    if ( ( pid4 = fork() ) == -1 )
        err_sys( "can't fork" );
    if ( pid4 == 0 ){
        printf( "- S4 - pid = %d, parent
               pid = %d\n", getpid(), getppid() );
    }
}
```

## Sincronizzazione di processi UNIX

```
else
{
    printf( "- S3 - pid = %d, parent
           pid = %d\n", getpid(), getppid() );
    printf( "wait - S3 - \n" );
    while ( wait( (int *)0 ) != pid4 )
        ; /* wait for child termination */
    printf( "end of - P34 -\n" );
}
exit( 0 );
}
```

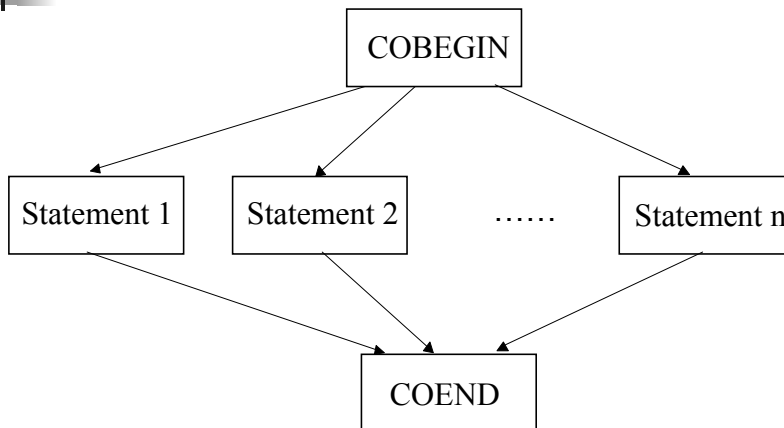
## Cobegin-coend

- I costrutti fork-join permettono di realizzare qualsiasi grafo di precedenza fra task.
- Il costrutto cobegin-coend, è più restrittivo del fork-join, infatti non permette di realizzare qualsiasi grafo di precedenza, tuttavia garantisce una buona strutturazione del codice (l'uso di fork-join è paragonabile all'uso del goto per salti incondizionati nei linguaggi di programmazione sequenziali).
- La sintassi del comando è:
  - cobegin S1 S2 S3 ... coend

## Cobegin-coend

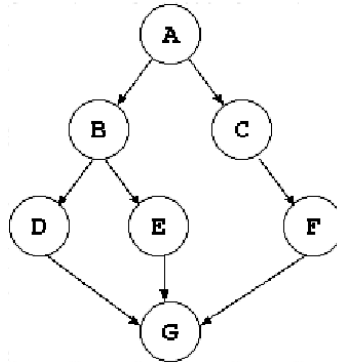
- L'esecuzione dell'istruzione causa l'attivazione dei tre processi che eseguono in parallelo le istruzioni S1, S2, S3, e la sospensione del processo padre fino alla terminazione di tutti i processi attivati.
- Le istruzioni S1, S2, S3 possono a loro volta essere costrutti di tipo cobegin-coend, ovvero i costrutti cobegin-coend possono essere annidati.

## Cobegin-coend



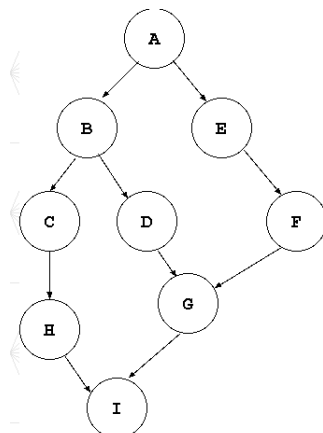
# Esempio

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



# Esercizio

- Realizzare mediante i costrutti fork-join e cobegin-coend il seguente grafo di precedenza



## Semafori

---

- Il tipo di dato semaforo è utilizzabile per risolvere problemi di sincronizzazione di processi concorrenti che interagiscono secondo il modello a memoria condivisa.
- I semafori sono delle risorse di sistema e come tali devono essere:
  - allocati,
  - gestiti
  - deallocati

## Semafori

---

- Non esistono in Unix primitive di gestione dei semafori come *wait* e *signal* bensì delle funzioni con le quali queste primitive possono essere realizzate
- Per utilizzare i semafori è necessario includere i seguenti file:
  - <sys/ipc.h>
  - <sys/msg.h>
  - <sys/sem.h>
- Per una corretta gestione degli errori sarebbe necessario includere anche:
  - <sys/types.h>
  - <unistd.h>
  - <errno.h>

## Semafori: creazione

- In Unix non è possibile allocare un semaforo singolarmente ma è necessario crearne un insieme.
- Tale insieme viene identificato tramite un identificatore di struttura di semafori.
- Ogni semaforo all'interno dell'insieme è identificato con un numero progressivo a partire da 0
- Per creare un semaforo si usa la funzione `semget`:
  - `sem_id semget (key_t chiave, int num_sem, short flags)`

## Semafori: creazione

- La `semget` restituisce un numero non negativo (se l'insieme di semafori può essere allocato) corrispondente all'identificativo dell'insieme di semafori (altrimenti ritorna `-1`):
  - Il codice di errore viene posto in `errno`
- Il campo chiave può assumere un qualunque valore, ma esiste una funzione C creata apposta per la sua inizializzazione: `ftok`
  - Se il campo chiave assume il valore `IPC_PRIVATE` un insieme di semafori viene comunque creato

## Semafori: creazione

- Il campo `num_sem` indica il numero di semafori appartenenti all'insieme:
  - Deve essere maggiore di 0
- Il campo `flags` specifica il comportamento da seguire nella gestione dell'insieme di semafori:
  - È possibile gestire l'accesso alla risorsa semaforo nel modo solito dei file
  - Alcuni valori sono predefiniti in `<sys/ipc.h>`:
    - `IPC_CREAT` crea il semaforo
    - `IPC_EXCL` verifica se il semaforo esiste
    - I valori possono essere messi in OR aritmetico `0644 | IPC_CREAT`

## Semafori: operazioni di controllo

- La funzione `semctl` permette di operare varie funzioni di controllo sui semafori:
  - Leggere lo stato di un semaforo
  - Impostare il valore
  - Rimuovere l'insieme di semafori dal sistema
  - ...
- La sintassi è la seguente:
  - `int semctl (int sem_id, int num_sem, operazione, union semun *argomenti)`

## Semafori: operazioni di controllo

- `sem_id` è l'identificatore dell'insieme di semafori
- `num_sem` indica il semaforo dell'insieme
- Il campo operazione indica il tipo di operazione che si vuole eseguire sul semaforo:
  - `IPC_RMID`: rimuove l'insieme di semafori dall'insieme
  - `GETVAL`: legge il valore del semaforo
  - `SETVAL`: imposta il valore del semaforo
  - `GETPID`: legge il pid dell'ultimo processo che ha agito sul semaforo
- Il campo argomenti serve per passare alla funzione i valori da impostare o per il ritorno dei valori letti. Il campo più importante della union è val:
  - Rappresenta il valore a cui il semaforo va impostato (`SETVAL`)
  - Rappresenta il valore corrente del semaforo (`GETVAL`)

## Semafori: operazioni

- La funzione che permette di realizzare le primitive wait-signal (o up-down) è `semop`
- La sintassi è la seguente:
  - `int semop (int sem_id, struct sembuf *operazione, int num_elementi)`
- `sem_id` è l'identificatore dell'insieme di semafori
- La struct `sembuf` operazione è il campo (vettore) che specifica l'operazione e il semaforo sul quale l'operazione deve essere applicata:
  - Nella struct `sembuf` sono presenti i seguenti campi:
    - `short sem_num`
    - `short sem_op`
    - `short sem_flg`

## Semafori: operazioni

- Il campo `sem_num` è il numero del semaforo all'interno dell'insieme
- Il campo `sem_op` specifica l'operazione da eseguire:
  - Se  $< 0$  l'operazione da eseguire è una wait
  - Se  $> 0$  l'operazione da eseguire è equivalente ad una signal
- Il campo `sem_flg` può assumere i valori:
  - `SEM_UNDO`: ripristina il vecchio valore di `sem_val` quando il processo termina
  - `IPC_NOWAIT`: non sospende il processo ma ritorna  $-1$ ; molto utile se si vogliono realizzare operazioni di wait e signal non bloccanti

## Semafori: operazioni

- Il campo `num_elementi` indica quanti elementi sono presenti nel vettore `struct sembuf *operazione`

# Esempio

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <sys/errno.h>
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
/* union semun is defined by including <sys/sem.h> */
#else
/* according to X/OPEN we have to define it ourselves */
union semun {
    int val; /* value for SETVAL */
    struct semid_ds *buf; /* buffer for IPC_STAT, IPC_SET */
    unsigned short int *array; /* array for GETALL, SETALL */
    struct seminfo *__buf; /* buffer for IPC_INFO */
} pippo;
#endif
struct sembuf pluto;
```

# Esempio

```
main(){
int semaforo;

semaforo = semget(IPC_PRIVATE, 1, 0777);
if(semaforo > 0){
    printf("Creato semaforo: %d\n", semaforo);
    printf("Leggo il valore del semaforo\n");
    semctl(semaforo,0,GETALL,&pippo);
    printf("Valore: %d\n",pippo.val);
    printf("Setto il valore del semaforo a 3\n");
    pippo.val = 3;
    semctl(semaforo,0,SETVAL,pippo);
    printf("Leggo il valore del semaforo\n");
    semctl(semaforo,0,GETALL,&pippo);
    printf("Valore: %d\n",pippo.val);
    pluto.sem_num = 0;
    pluto.sem_op = -1;
```

## Esempio

```
printf("Faccio una wait sul semaforo\n");
semop(semaforo,&pluto,1);

printf("Leggo il valore del semaforo\n");
semctl(semaforo,0,GETALL,&pippo);
printf("Valore: %d\n",pippo.val);

pluto.sem_num = 0;
pluto.sem_op = 1;
printf("Faccio una signal sul semaforo\n");
semop(semaforo,&pluto,1);

printf("Leggo il valore del semaforo\n");
semctl(semaforo,0,GETALL,&pippo);
printf("Valore: %d\n",pippo.val);
}
}
```

## Semafori: multiprocessore

- Sistemi monoprocesso:
  - Disable interrupt come prologo all'entrata in Regione Critica.
  - Enable interrupt come prologo all'uscita della Regione Critica.
- Sistemi multiprocessore con memoria comune
  - Istruzione test-and-set su una variabile di lock
  - Se il byte è 0 la Regione Critica è libera
  - Se il byte è 1 la Regione Critica è occupata
- L'istruzione testa il contenuto della variabile e lo pone a 1 in un solo ciclo indivisibile

## Test and set

- Controlla e modifica atomicamente il contenuto di un byte.

```
char Test-and-Set(char *target)
{
    Test-and-Set = *target;
    *target = TRUE;
}
```