

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;

    for(i=0; i<MAXPAROLA; i++)
        freq[i]=0;

    if(argc != 2)
    {
        fprintf(stderr, "ERRORE: serve un parametro con il nome del file\n");
        exit(1);
    }
    f = fopen(argv[1], "r");
    if(f==NULL)
    {
        fprintf(stderr, "ERRORE: impossibile aprire il file %s\n", argv[1]);
        exit(1);
    }

    while( fgets( riga, MAXRIGA, f ) != NULL )
```

# Processi

## Introduzione ai processi

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

# Algoritmi, programmi e processi

... Ricordare lezione introduttiva ...

## ❖ Algoritmo

- Procedimento logico che, in un numero finito di passi, permette la risoluzione di un problema

## ❖ Programma

- Formalizzazione di un algoritmo attraverso un linguaggio di programmazione
- Entità passiva, ovvero file (eseguibile) su disco

## ❖ Processo

- Astrazione di un programma in esecuzione
- Entità attiva
  - Sequenza di operazioni effettuate da un programma in esecuzione su un determinato insieme di dati

## Processi sequenziali e concorrenti

### ❖ Esecuzione sequenziale

- Le azioni sono eseguite una **dopo** l'altra
  - Ogni nuova istruzione inizia terminata la precedente
  - Esiste una totale relazione di ordinamento
- Comportamento deterministico
  - Dato uno stesso input si genera sempre lo stesso output, indipendentemente
    - Dal momento di esecuzione
    - Dalla velocità di esecuzione
    - Da quanti altri processi sono in esecuzione sul sistema

Azioni sequenziali

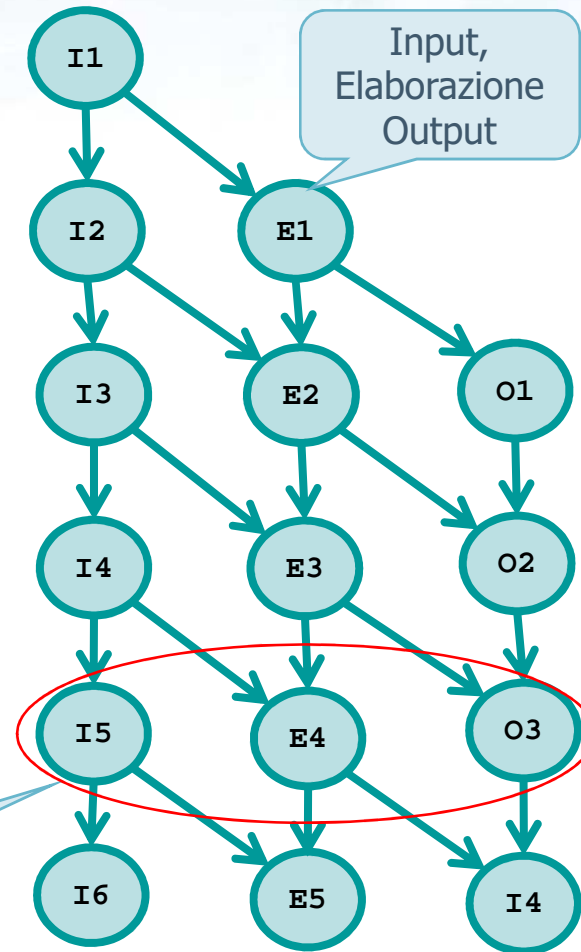


Input,  
Elaborazione  
Output

## Processi sequenziali e concorrenti

### ❖ Esecuzione concorrente

- Più istruzioni possono essere eseguite allo **stesso** istante
  - Non esiste relazione d'ordine
  - Comportamento non deterministico
- La concorrenza è
  - Fittizia (illusoria)
    - Sistemi mono-processore
  - Reale (parallelismo)
    - Sistemi multi-processore o multi-core



## Processi

### ❖ Al bootstrap vengono eseguiti numerosi processi

#### ➤ Automaticamente

- Daemon Process
- Attesa messaggi di posta elettronica
- Controllo e scan virus e simili
- ...

#### ➤ Su richiesta esplicita dell'utente

- Gestione stampanti
- Gestione WEB server (con richieste parallele dall'esterno)
- ...

Eseguiti al bootstrap,  
terminati allo shut-down.  
Svolgono attività ricorrenti.

## Processi

### ❖ Normalmente è possibile

➤ Identificare e controllare un processo esistente

ID & system call:  
pid, getpid, getppid, etc.

➤ Creare un nuovo processo

- Il processo creatore assume il ruolo di processo **padre** e quello creato di processo **figlio**
- È possibile creare un **albero di processi**

System call:  
fork, exec, system

➤ Attendere, sincronizzare e terminare processi esistenti

System call:  
exit, wait, waitpid

## Identificazione di un processo

- ❖ Ogni processo possiede un identificatore univoco
  - PID o Process Identifier
- ❖ Il PID è normalmente un intero non negativo
  - Nonostante l'identificatore sia univoco UNIX li riutilizza
  - Essendo univoci possono essere inclusi dal processo per generare oggetti unici
    - Per esempio nel caso diversi processi in esecuzione concorrente, scrivano file simili il PID può essere utilizzato per creare nomi di file univoci
      - `sprintf (filename, "file-%d", getpid());`

Identifica il processo chiamante

## Identificazione di un processo

- ❖ Alcuni identificatori sono riservati
  - **0** riservato per lo schedatore dei processi
    - Noto sotto con il nome di swapper
    - Viene eseguito a livello kernel
  - **1** riservato per il processo di **init**
    - Invocato alla fine del bootstrap
    - Viene eseguito a livello utente ma con privilegi di super-user
    - Non termina (muore) mai
    - Diventa automaticamente il padre di ogni processo rimasto orfano

SO recenti: "jobs started are not reparented to PID1 (init), but to a custom init -user, owned by the same user ..."



## Identificazione di un processo

```
#include <unistd.h>

pid_t getpid();
pid_t getppid();
uid_t getuid();
gid_t getgid();
```

Non esiste una system call per ottenere il PID di un figlio

- ❖ All'identificatore (PID) di un processo sono associati altri identificatori
- ❖ Le system call precedenti ritornano l'identificatore
  - Del processo chiamante
  - Del padre del processo chiamante
  - Dell'utente del processo chiamante
  - Del gruppo del processo chiamante

## Identificazione di un processo

```
#include <unistd.h>

uid_t  getuid();
gid_t  getgid();
```

```
#include <unistd.h>

uid_t  geteuid();
gid_t  getegid();
```

- ❖ UID e GID hanno dei corrispondenti Effective-UID e Effective-GID
  - Un processo di UID (GID) definito può cambiare identità assumendo un EUID (EGID) diverso
    - Esempio
      - Il comando **passwd** permette di cambiare la password di un utente; questo richiede permessi di root; quindi il processo **passwd** con l'UID dell'utente assume uno EUID di root per effettuare l'operazione

## Creazione di un processo

- ❖ Windows e UNIX/Linux utilizzano procedure diverse
  - Nelle Windows API un processo viene creato mediante la system call **CreateProcess**
    - In pratica esegue un nuovo processo specificandone l'eseguibile
    - In genere il nuovo processo è distinto dal chiamante
  - In UNIX/Linux un nuovo processo viene creato mediante la system call **fork**
    - In pratica si procede per clonazione/duplicazione del processo corrente

## Creazione di un processo

- ❖ In Windows la `CreateProcess` assume lo stile tipico delle API Windows
  - Verboseità, elevato numero di parametri, tipizzazione elevata, etc.

```
BOOL CreateProcess (  
    LPCTSTR lpImageName,  
    LPTSTR lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpsaProcess,  
    LPSECURITY_ATTRIBUTES lpsaThread,  
    BOOL bInheritHandles, DWORD dwCreate,  
    LPVOID lpvEnvironment, LPCTSTR lpCurDir,  
    LPSTARTUPINFO lpsiStartInfo,  
    LPPROCESS_INFORMATION lppiProcInfo  
);
```

## Creazione di un processo

- ❖ In UNIX/Linux la **fork** genera un processo detto **processo figlio**
  - Il figlio è una copia identica al padre tranne che per il Process ID (**PID**) ritornato dalla fork
    - Il padre riceve l'ID del figlio
      - Ogni processo può avere più figli e li distingue in base al loro PID
    - Il figlio riceve il valore 0
      - Può identificare il proprio padre mediante la system call `getppid`
  - In pratica la `fork` viene richiamata una volta ma ritorna due volte una nel padre e una nel figlio

## Creazione di un processo

```
#include <unistd.h>

pid_t fork (void);
```

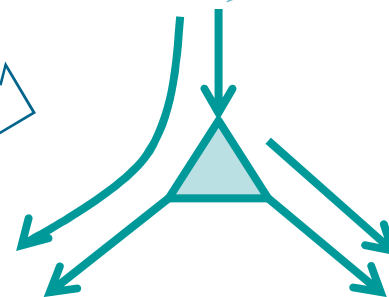
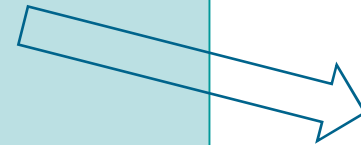
Varianti: vfork, rfork, clone

### ❖ Valore restituito

- Se l'operazione si conclude correttamente
  - Il PID del figlio nell'istanza di codice del padre
  - Il PID zero nell'istanza di codice del figlio
- Il valore -1 se non è possibile allocare un nuovo processo
  - Generalmente si è raggiunto il limite sul numero di processi

## Creazione di un processo

```
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
switch (pid) {
  case -1:
    // Fork failure
    ...
    exit (1);
  case 0:
    // Child
    ...
  default:
    // Father
    ...
}
...
```



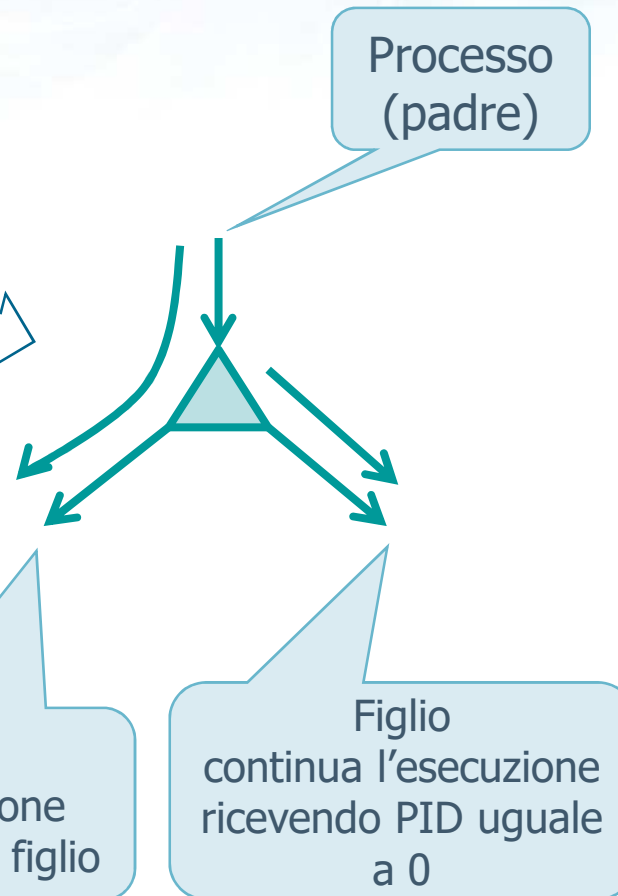
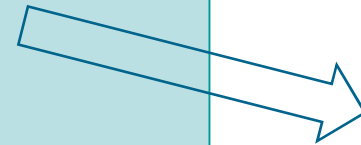
Processo  
(padre)

Padre  
continua l'esecuzione  
ricevendo il PID del figlio

Figlio  
continua l'esecuzione  
ricevendo PID uguale  
a 0

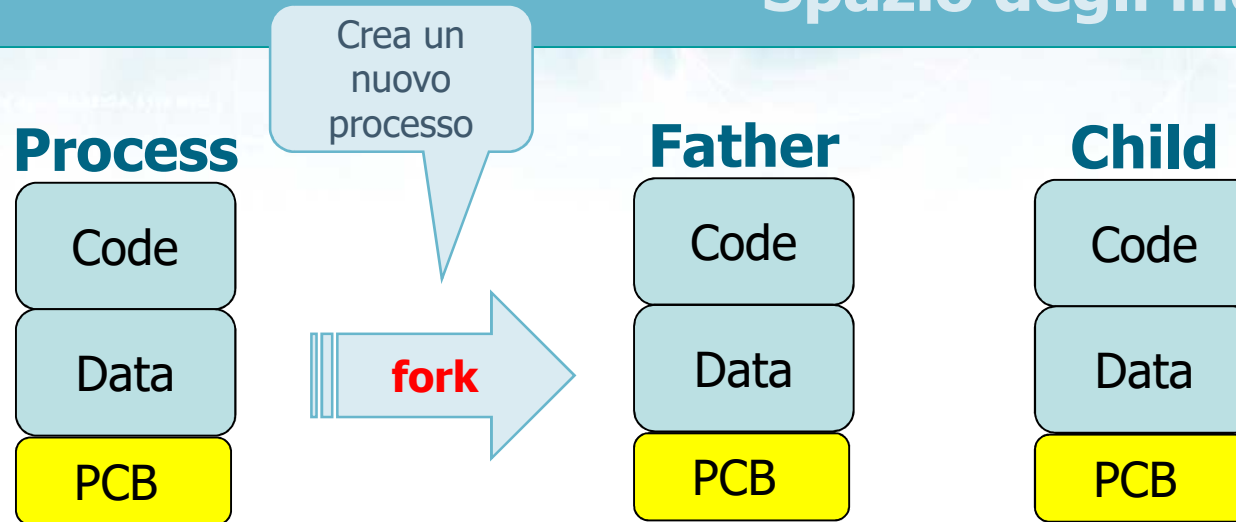
## Creazione di un processo

```
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
if (pid > 0) {
    // Father
    ...
} else {
    // Child
    ...
}
...
}
```





## Spazio degli indirizzi



### ❖ Padre e figlio

- Condividono il codice
- Subito dopo la fork
  - Hanno una copia dei dati identica
  - Possono modificare (distruggere, creare, etc.) i propri dati in maniera indipendente

## Esempio

- ❖ Scrivere un programma concorrente in grado di
  - Generare un processo figlio
  - Far terminare
    - Il processo padre prima del figlio
    - Il processo figlio prima del padre

La system call  
unsigned int sleep (unsigned int sec)  
mette il processo in wait per (almeno) sec secondi

- ❖ Visualizzare per ogni processo che termina il PID del processo e del processo padre

Chi è il padre del padre?

Se il padre termina prima,  
chi è il padre del figlio?

## Esempio

```
#include <unistd.h>
```

```
...
```

```
printf ("Main :                               ");
```

```
printf ("PID=%d; My parent PID=%d\n",  
        getpid(), getppid());
```

```
...
```

```
pid = fork();
```

```
if (pid == 0){
```

```
    sleep (tC);
```

```
    printf ("Child : PIDreturned=%d          ", pid);
```

```
    printf ("PID=%d; My parent PID=%d\n",  
            getpid(), getppid());
```

```
} else {
```

```
    sleep (tF);
```

```
    printf ("Father: PIDreturned=%d ", pid);
```

```
    printf ("PID=%d; My parent PID=%d\n",  
            getpid(), getppid());
```

```
}
```

```
tC = atoi (argv[1]);  
tF = atoi (argv[2]);
```

Child

Father

## Esempio

```
➤ ps
  PID TTY          TIME CMD
 2088 pts/10        00:00:00 bash
 2760 pts/10        00:00:00 ps
```

Stato della shell  
(ps: print process status)

Child awaits 2 secs  
Father awaits 5 secs

```
➤ ./e03-fork 2 5
Main   :                               PID=2813; My parent PID=2088
Child  : PIDreturned=0                 PID=2814; My parent PID=2813
Father: PIDreturned=2814               PID=2813; My parent PID=2088
```

Osservare i PID  
crescenti ...

Il child rimane **zombie** per  
3 secondi

## Esempio

```
➤ ps
  PID TTY          TIME CMD
 2088 pts/10        00:00:00 bash
 2760 pts/10        00:00:00 ps
```

Stato della shell  
(ps: print process status)

Child awaits 5 secs  
Father awaits 2 secs

```
➤ ./e03-fork 5 2
Main      :                PID=2815; My parent PID=2088
Father: PIDreturned=2816 PID=2815; My parent PID=2088
➤ Child  : PIDreturned=0   PID=2816; My parent PID=1
```

Osservare i PID  
crescenti ...

Il child rimane **orfano** e  
viene ereditato dal  
processo init

## Esercizio

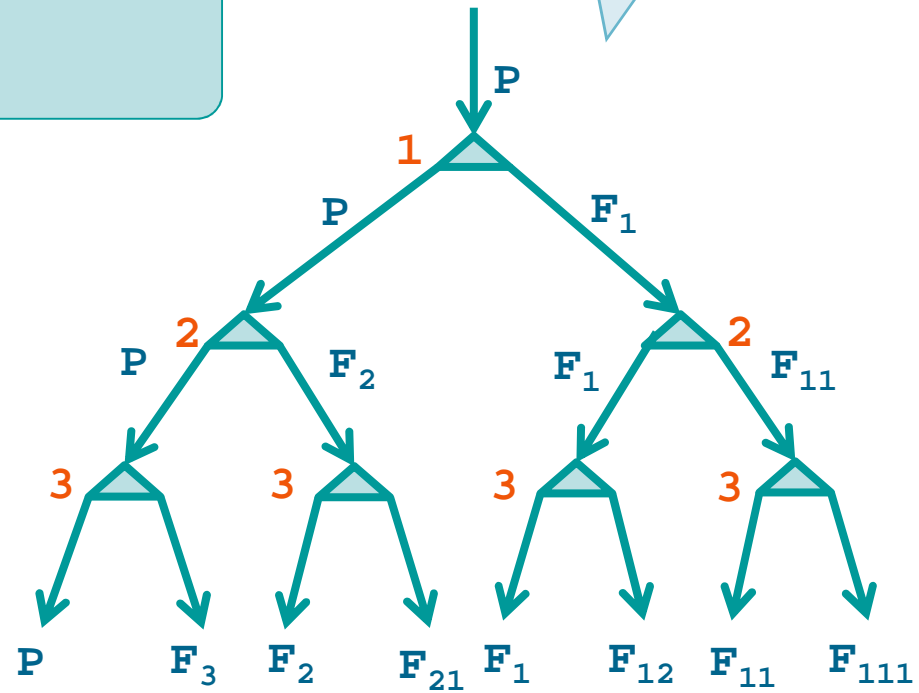
- ❖ Dato il seguente programma disegnare
  - Il grafo che rappresenta il flusso di controllo (Control Flow Graph, CFG)
  - L'albero di generazione dei processi

```
int main () {  
    /* fork a child process */  
    fork();  
  
    /* fork another child process */  
    fork();  
  
    /* fork a last one */  
    fork();  
}
```

# Soluzione

```
int main () {  
    fork (); // 1  
    fork (); // 2  
    fork (); // 3  
}
```

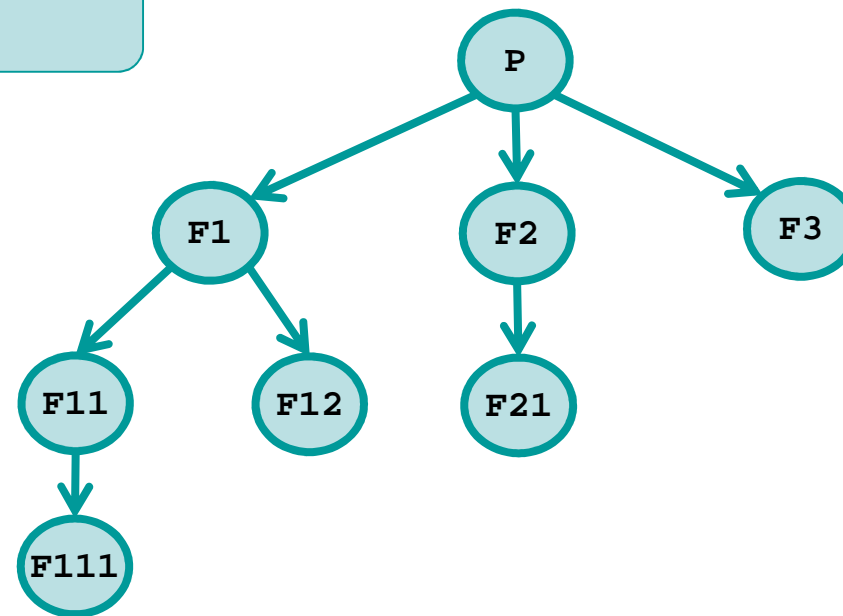
Control Flow Graph (CFG)



# Soluzione

```
int main () {  
    fork (); // 1  
    fork (); // 2  
    fork (); // 3  
}
```

Albero di generazione dei processi





## Esercizio

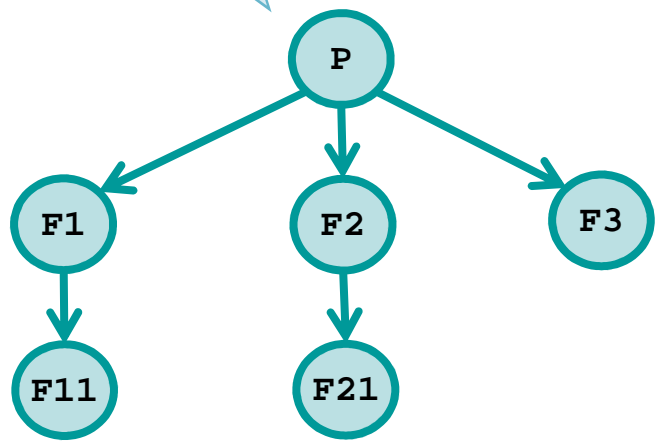
- ❖ Dato il seguente programma disegnare
  - Il grafo che rappresenta il flusso di controllo (Control Flow Graph, CFG)
  - L'albero di generazione dei processi

```
pid = fork (); /* call #1 */  
  
if (pid != 0)  
    fork (); /* call #2 */  
  
fork (); /* call #3 */
```

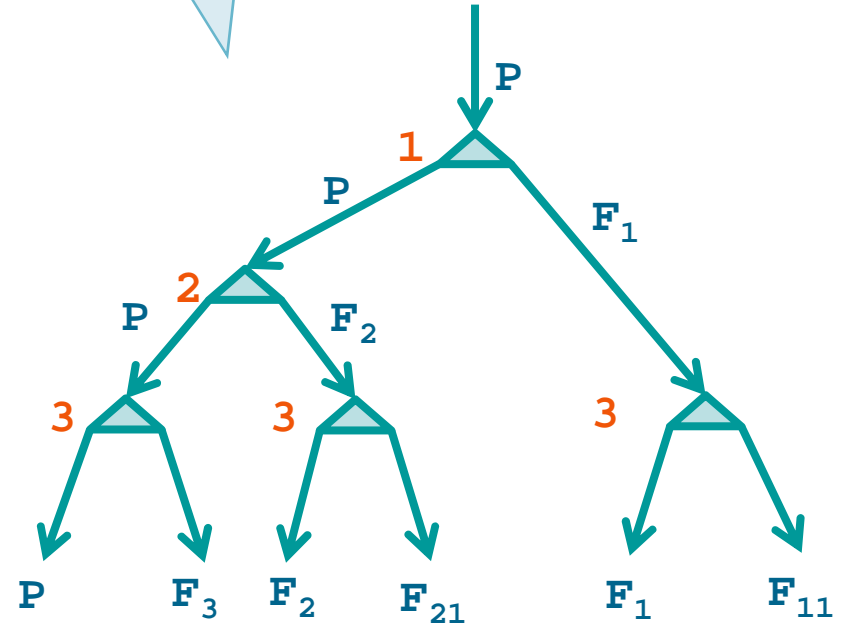
# Soluzione

```
pid = fork (); // 1  
if (pid != 0)  
    fork (); // 2  
fork (); // 3
```

Albero di generazione dei processi



Control Flow Graph (CFG)



## Esercizio

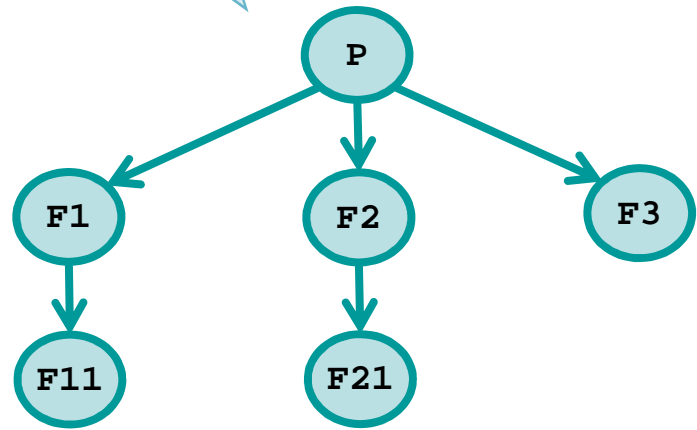
- ❖ Dato il seguente programma disegnare
  - Il grafo che rappresenta il flusso di controllo (Control Flow Graph, CFG)
  - L'albero di generazione dei processi

```
pid = fork() /* call #1 */  
  
fork();     /* call #2 */  
  
if (pid != 0)  
    fork(); /* call #3 */
```

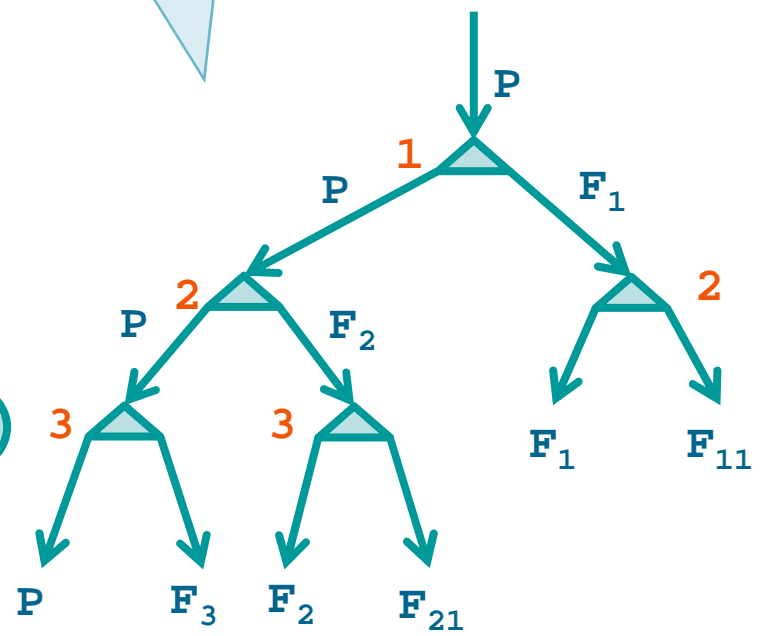
# Soluzione

```
pid = fork (); // 1  
fork ();      // 2  
if (pid != 0)  
    fork ();  // 3
```

Albero di generazione dei processi



Control Flow Graph (CFG)



## Esercizio

- ❖ Dato il seguente programma disegnare
  - Il grafo che rappresenta il flusso di controllo (Control Flow Graph, CFG)
  - L'albero di generazione dei processi

```
#include <stdio.h>

int main () {
    int i;
    for (i=0; i<2; i++) {
        printf("i: %d \n", i);
        if (fork()) /* call #1 */
            fork(); /* call #2 */
    }
}
```

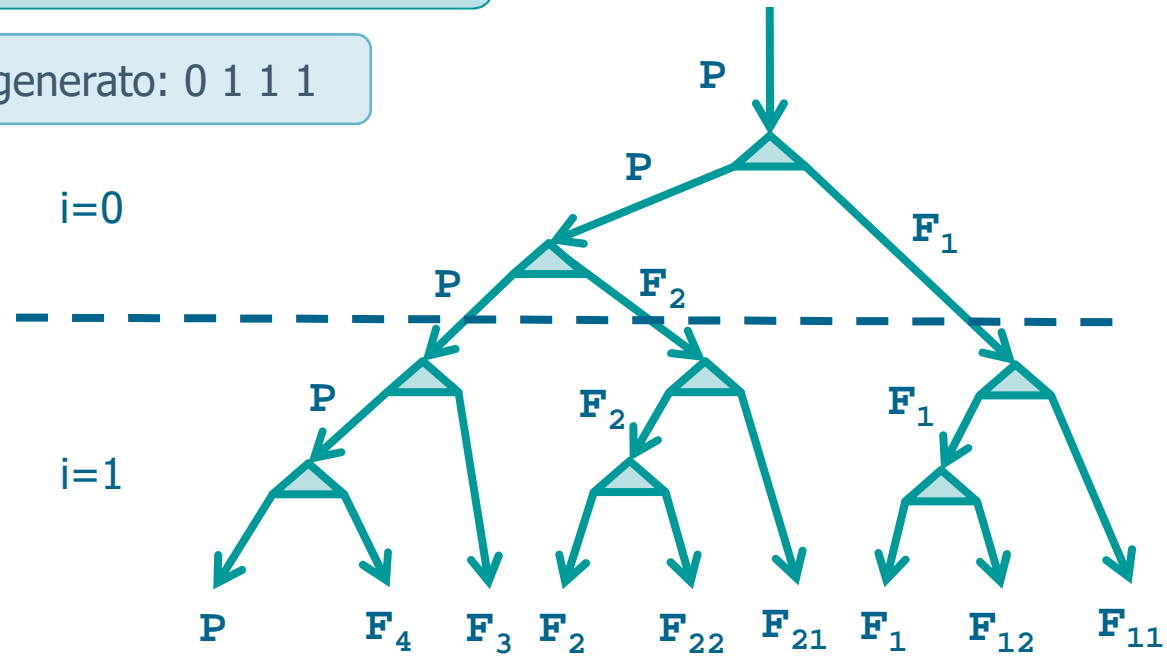
fflush (stdout);  
oppure  
setbuf (stdout, 0);

# Soluzione

```
for (i=0; i<2; i++) {  
    printf("i: %d \n", i);  
    if (fork()) // 1  
        fork(); // 2  
}
```

Output generato: 0 1 1 1

Control Flow Graph (CFG)

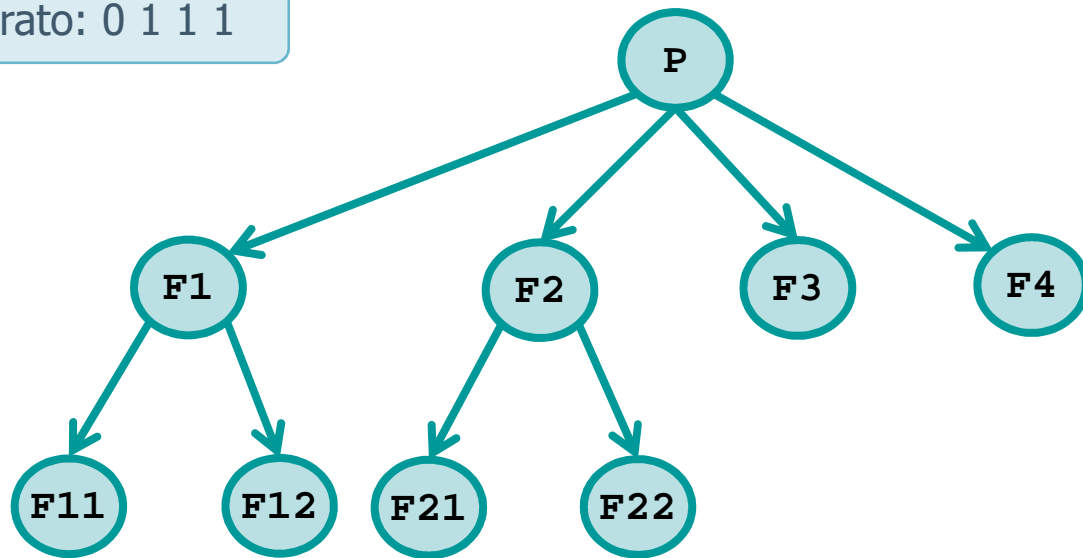


# Soluzione

```
for (i=0; i<2; i++) {  
    printf("i: %d \n", i);  
    if (fork()) // 1  
        fork(); // 2  
}
```

Albero di generazione dei processi

Output generato: 0 1 1 1



## Esercizio

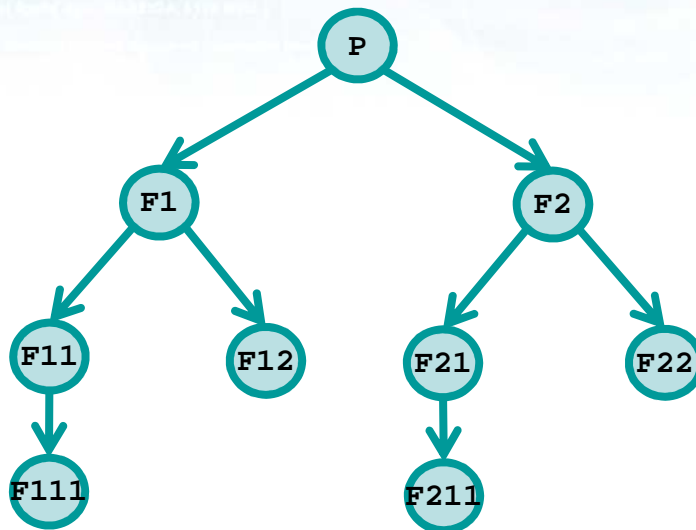
- ❖ Dato il seguente programma riportarne l'output e l'albero di generazione dei processi

```
int main() {
    int a, b=5, c;
    a = fork(); /* #1 */
    if (a) {
        a = b; c = split(a, b++);
    } else {
        fork(); /* #2 */
        c = a++; b += c;
    }
    if (b > c) {
        fork(); /* #3 */
    }
    printf("%3d", a+b+c);
    return 0;
}
```

```
int split(int a, int b) {
    a++;
    a = fork(); /* #4 */
    if (a) {
        a = b;
    } else {
        if (fork()) /* #5 */ {
            a--;
            b += a;
        }
    }
    return a+b;
}
```



# Soluzione



P	a	b	c	a+b+c
P	5	6	10	21
F1	1	5	0	6
F2	5	6	3	14
F11	1	5	0	6
F12	1	5	0	6
F21	5	6	5	16
F22	5	6	3	14
F111	1	5	0	6
F211	5	6	5	16

## Esercizio

- ❖ Scrivere un programma concorrente che
  - Dato un valore intero **n**
  - Sia in grado di generare **n** processi figlio
- ❖ Ciascun processo figlio visualizzi il proprio PID e termini

## Soluzione 1

```
int i, n;

scanf ("%d", &n);
for (i=0; i<n; i++) {
    fork();
    printf ("Proc %d (PID=%d)\n",
           i, getpid());
}

exit (0);
```

# Soluzione 1

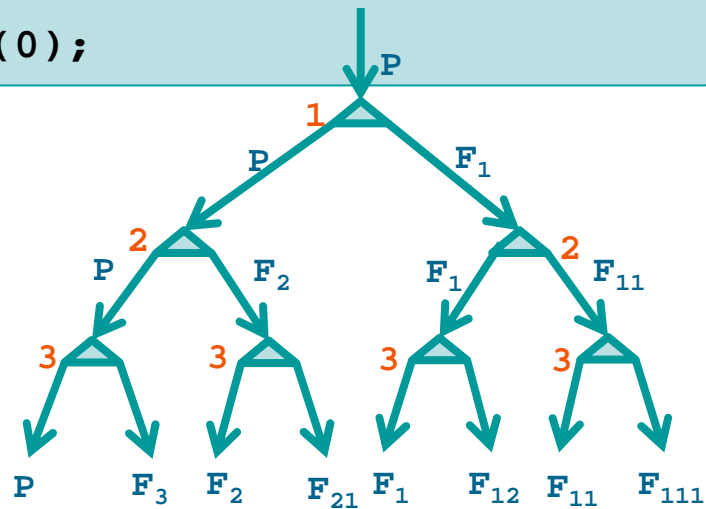
```

int i, n;

scanf ("%d", &n);
for (i=0; i<n; i++) {
    fork();
    printf ("Proc %d (PID=%d)\n",
           i, getpid());
}

exit (0);

```



Output generato con n=3

```

Proc 0 (PID=3188)
Proc 1 (PID=3188)
Proc 2 (PID=3188)
Proc 2 (PID=3191)
Proc 1 (PID=3190)
Proc 2 (PID=3190)
Proc 0 (PID=3189)
Proc 1 (PID=3189)
Proc 2 (PID=3189)
Proc 2 (PID=3192)
Proc 2 (PID=3194)
Proc 1 (PID=3193)
Proc 2 (PID=3193)
Proc 2 (PID=3195)

```

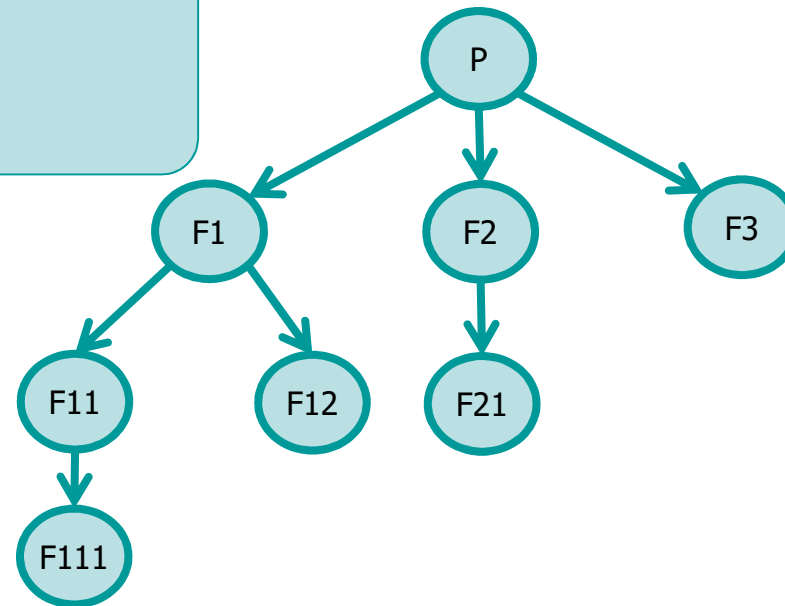
# Soluzione 1

```
int i, n;  
  
scanf ("%d", &n);  
for (i=0; i<n; i++) {  
    fork();  
    printf ("Proc %d (PID=%d)\n",  
           i, getpid());  
}  
  
exit (0);
```

Generati 7 processi  
(oltre a quello iniziale)

**Soluzione 1  
Errata**

Albero dei processi  
con n=3



## Soluzione 2

```
int i, n;
...
scanf ("%d", &n);
printf ("Start PID=%d\n",
        getpid());
for(i=0; i<n; i++) {
    if (fork() == 0) {
        printf ("Proc %d (PID=%d)\n",
                i, getpid());
        break;
    }
}
printf ("End PID=%d (PPID=%d)\n",
        getpid(), getppid());

exit(0);
```

## Soluzione 2

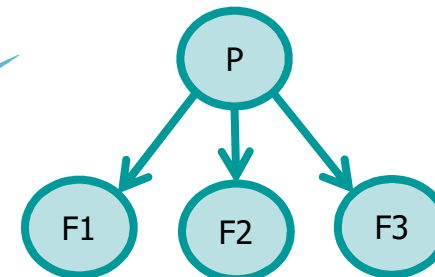
```
int i, n;
...
scanf ("%d", &n);
printf ("Start PID=%d\n",
        getpid());
for(i=0; i<n; i++) {
    if (fork() == 0) {
        printf ("Proc %d (PID=%d)\n",
                i, getpid());
        break;
    }
}
printf ("End PID=%d (PPID=%d)\n",
        getpid(), getppid());

exit(0);
```

```
> ps
PID TTY          TIME CMD
088 pts/10      00:00:00 bash

> ./u04s01e06-fork
Start PID=3225
End PID=3225 (PPID=2088)
Proc 2 (PID=3228)
End PID=3228 (PPID=1314)
Proc 1 (PID=3227)
End PID=3227 (PPID=1314)
Proc 0 (PID=3226)
End PID=3226 (PPID=1314)
```

Albero dei processi  
e output con n=3



## Risorse

- ❖ Il processo figlio è una nuova entry nella tabella dei processi
- ❖ Le **risorse** del processo possono
  - Essere condivise completamente tra padre e figli
    - Stesso spazio di indirizzamento
  - Essere condivise in parte
    - Spazi di indirizzamento parzialmente sovrapposti
  - Non essere condivise affatto
    - Spazi di indirizzamento separati



- ❖ In UNIX/Linux padre e figlio **condividono**
  - Il codice sorgente (C)
  - Tutti i descrittori dei file
    - In particolare stdin, stdout e stderr
    - Effettuare operazioni concorrenti di I/O implica avere un I/O inter-allacciato
      - Duplicando i file descriptor si duplicano i puntatori e si effettuano R/W a partire dalla stessa posizione
  - Lo user ID, il group ID, etc.
  - La root e la working directory
  - Le risorse del sistema e i limiti di utilizzo
  - I segnali

## Risorse

- ❖ In UNIX/Linux padre e figlio si **differenziano** per
  - Il valore ritornato dalla fork
  - Il PID
    - Il padre conserva il proprio PID
    - Il figlio ottiene un PID nuovo
  - Lo spazio dati, lo heap e lo stack
    - In realtà i moderni SO utilizzano la tecnica "**copy-on-write**"
    - La memoria è duplicata solo quando strettamente necessario, ovvero quando uno dei due processi effettua una scrittura
    - Il valore iniziale delle variabili viene ereditato

## Esempio

```

char c, str[10];

c = 'X';
if (fork()) {
    // parent (!=0)
    c = 'F';
    strcpy (str, "parent");
    sleep (5);
} else {
    // child (==0)
    strcpy (str, "child");
}

fprintf(stdout, "PID=%d; PPID=%d; c=%c; str=%s\n",
        getpid(), getppid(), c, str);
    
```

Utilizzo (R/W) di  
una variabile  
globale

```

PID=2777; PPID=2776; c=X; str=child
PID=2776; PPID=2446; c=F; str=parent
    
```

Output

## Terminazione di un processo

- ❖ Esistono **5** metodi standard per terminare un processo
  - Eseguire una **return** dalla funzione principale
  - Eseguire una **exit**
  - Eseguire una **\_exit** oppure una **\_Exit**
    - Sinonimi definiti in ISO C o POSIX
    - Hanno effetti simili ma non indentici alla **exit**
  - Richiamare **return** dal main dell'ultimo thread del processo
  - Richiamare **pthread\_exit** dall'ultimo thread del processo

## Terminazione di un processo

- ❖ Esistono **3** metodi anomali per terminare un processo
  - Richiamare la funzione **abort**
    - Questo è un sottocaso del successivo in quanto l'abort genera il segnale SIGABORT
  - Ricevere un segnale (**signal**)
    - Di terminazione
    - Non gestito
  - Cancellare l'ultimo thread del processo

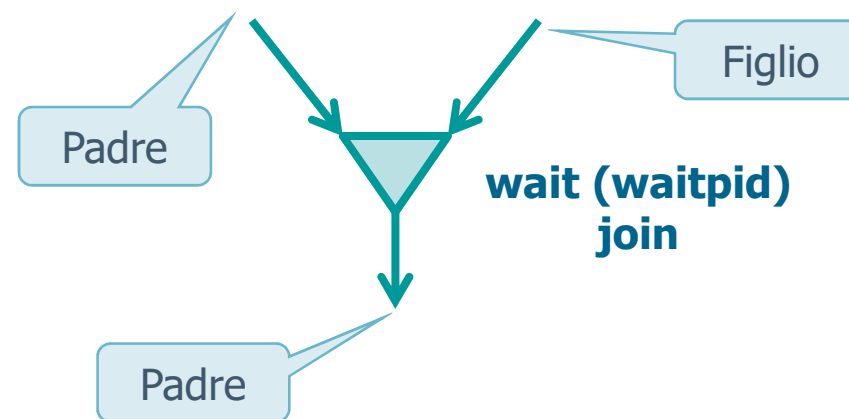
## System call wait () e waitpid ()

- ❖ Quando un processo termina tanto in maniera normale quanto anomala
  - Il kernel invia un segnale (**SIGCHLD**) al padre
  - La ricezione di un segnale da parte di un processo è un evento asincrono
  - Il processo padre può decidere di
    - Gestire la terminazione del figlio in maniera
      - **Asincrona**
      - **Sincrona**
    - Ignorare la terminazione del figlio

Il default è ignorare la terminazione del figlio

## System call `wait ()` e `waitpid ()`

- ❖ Se un processo decide di gestire la terminazione di un figlio occorre effettuare la gestione
  - Asincrona mediante un gestore del segnale **SIGCHLD**
    - Questa strategia verrà analizzata nella sezione relativa ai segnali
  - Sincrona mediante una chiamata alle system call
    - `wait`
    - `waitpid`



## System call wait ()

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

pid_t wait (int *statLoc);
```

- ❖ La chiamata alla system call **wait** da parte di un processo ha effetti diversi a seconda dello stato dei processi figli del processo stesso
  - Ritorna con un errore se il processo non ha figli
    - Si suppone che un processo senza figli non faccia una wait, ovvero non cerchi di attendere un figlio
    - In caso di errore il valore ritornato è -1



## System call wait ()

- Blocca il processo se tutti i figli del processo sono ancora in esecuzione
  - Il processo rimarrà bloccato sino a quando uno dei figli non terminerà
  - Alla terminazione di un figlio, la funzione wait ritornerà al chiamante lo stato del figlio terminato

```
pid_t wait (int *statLoc);
```

## System call wait ()

- Restituisce al processo (immediatamente) lo stato di terminazione di un figlio, se **almeno** uno dei figli è terminato ed è in attesa che il suo stato di tarminazione sia recuperato
  - Quando un processo termina e il padre non fa una wait il suo stato di terminazione rimane pendente
  - Alcune risorse associate al processo rimangono bloccate

```
pid_t wait (int *statLoc);
```

## System call wait ()

### ❖ Parametro

- **statLoc** indica lo stato di terminazione del processo figlio terminato
- È un puntatore a un intero
  - Se non è NULL specifica lo stato di uscita del processo figlio (valore restituito dal figlio)
- Le informazioni di stato sono
  - Implementation dependent
  - Interpretabili con delle macro presenti in **<sys/wait.h>**

```
pid_t wait (int *statLoc);
```

## System call wait ()

- Tra le macro applicabili a **statLoc** ricordiamo
  - WIFEXITED(statLoc) è vero se la terminazione è stata corretta
  - Se la terminazione è stata corretta, WEXITSTATUS(statLoc) cattura gli 8 LSBs del parametro passato a exit (oppure a \_exit o a \_Exit)
- ❖ Valore di ritorno
  - Il PID del processo figlio terminato

```
pid_t wait (int *statLoc);
```

## Esempio

- ❖ Si scriva un processo in grado di
  - Sganciare un figlio
  - Alla sua terminazione raccoglierne lo stato di terminazione

```
...
pid_t pid, childPid;
int statVal;
...
pid = fork();
if (pid==0) {
    // Child
    sleep (5);
    exit (6);
} else {
```

## Esempio

```
// Father
childPid = wait (&statVal);
printf("Figlio terminato: PID = %d\n", childPid);
if (WIFEXITED(statVal))
    printf ("Valore restituito: %d\n",
           WEXITSTATUS (statVal));
else
    printf ("Terminazione anormale\n");
}
exit(25);
}
...
```

Stampa 6

echo \$?  
(da shell) visualizza 25

## Processi Zombie

- ❖ Un processo terminato per il quale il padre non ha ancora eseguito una **wait** si dice **zombie**
  - Il segmento dati del processo **non** viene rimosso dalla process table per tenere traccia dello stato di uscita
  - L'entry viene rimossa solo dopo che il padre ha eseguito una **wait**
    - Se il processo padre non esegue le wait opportuno, il SO deve gestire i processi zombie
    - Un eccessivo numero di processi zombie, appesantisce e rallenta il SO

## Processi Orfani

- ❖ Se il padre termina prima di eseguire la **wait** il processo figlio
  - Diviene orfano
  - I processi orfani, per non rimanere tali, vengono ereditati dal processo **init** (quello con PID=1) oppure da un processo **init custom utente**
  - Processi orfani e ereditati dal processo init non diverranno più processi zombie



## System call `waitpid ( )`

- ❖ Se si desidera attendere un figlio specifico con una **wait** occorre
  - Controllare il PID del figlio terminato
  - Eventualmente memorizzare il PID del figlio terminato nella lista dei processi figlio terminati (per future verifiche/ricerche)
  - Effettuare un'altra wait sino a quando termina il figlio desiderato

## System call `waitpid ( )`

- ❖ La **waitpid** si differenzia dalla **wait** in quanto
  - Può essere non bloccante
    - Non blocca in padre in attesa della terminazione di un figlio se nessun figlio è terminato
  - Può attendere la terminazione di un figlio specifico

## System call `waitpid ( )`

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

pid_t waitpid (pid_t pid, int *statLoc, int options);
```

### ❖ Parametri

#### ➤ **pid** permette di attendere

- Un qualsiasi figlio (`waitpid==wait`) se `-1`
- Il figlio con quel `PID=pid` se `>0`
- Un qualsiasi figlio il cui `group ID` è uguale a quello del chiamante se `0`
- Il figlio il cui `group ID` è uguale a `abs(pid) <-1`

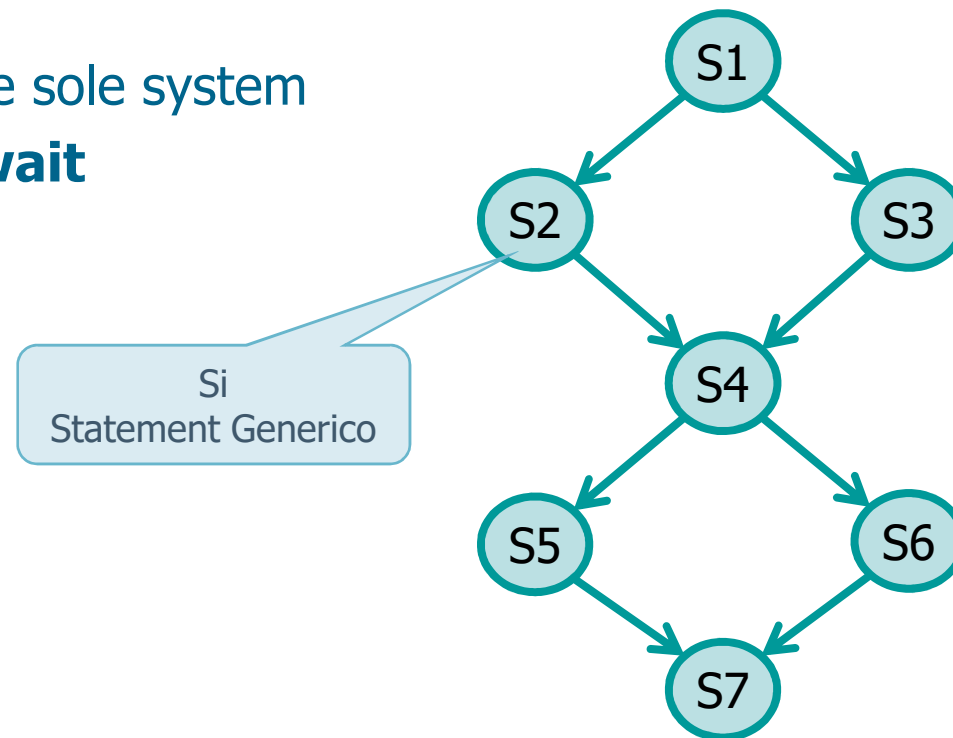
## System call `waitpid ( )`

- **statLoc** ha lo stesso significato che ha nella `wait`
- **options** permette controlli aggiuntivi
  - Assume valore 0 oppure è un OR bit a bit di costanti
  - Tra le costanti utilizzabili, ricordiamo
    - `WNOHANG`, se il figlio di PID specificato non è terminato il chiamante non si ferma (versione **non** bloccante della `wait`)
    - `WCONTINUED` e `WUNTRACED` permettono di conoscere lo stato di un figlio in condizioni particolari

```
pid_t waitpid (pid_t pid, int *statLoc, int options);
```

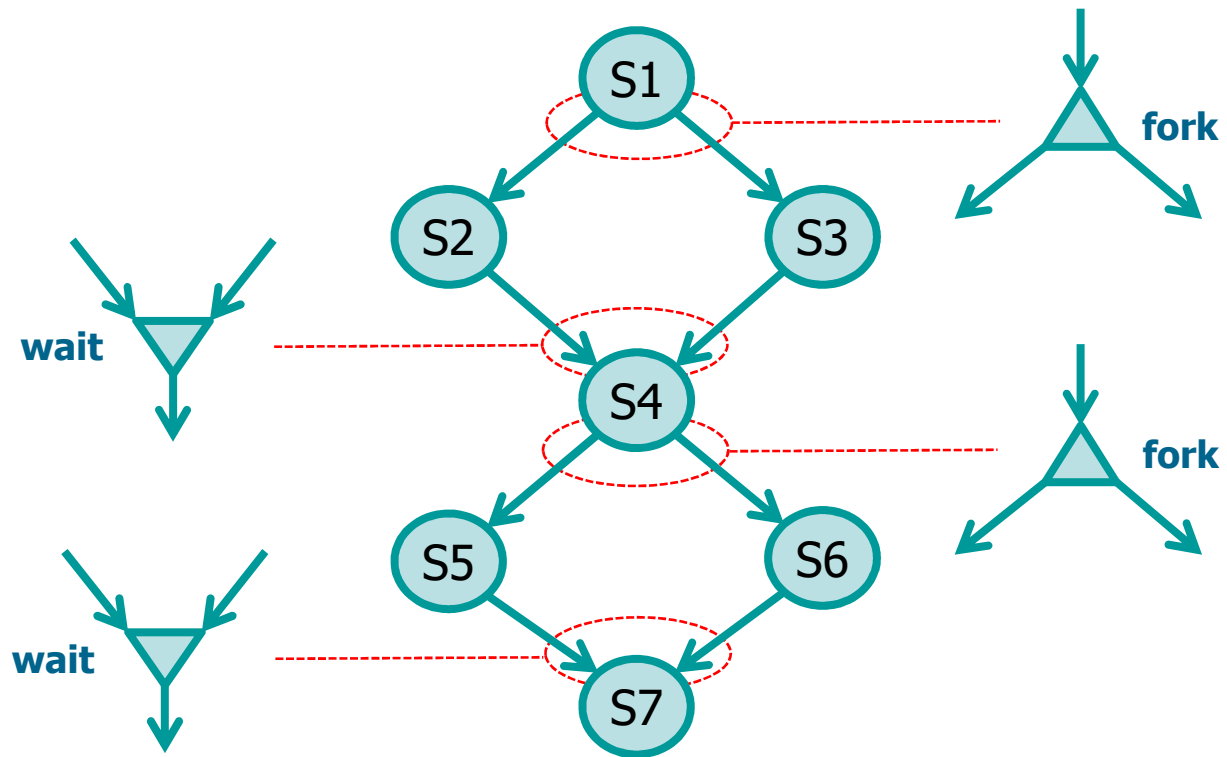
## Esercizio

- ❖ Realizzare un unico processo in cui blocchi di istruzioni rispettino il seguente Control Flow Graph (CFG)
  - Si utilizzino le sole system call **fork** e **wait**



# Esercizio

## ❖ Interpretazione



## Soluzione

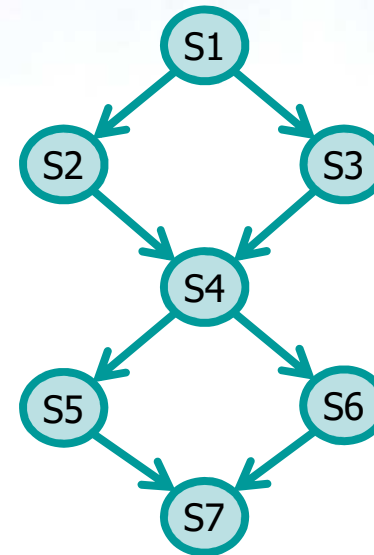
```
int main() {  
    pid_t pid;  
    printf ("S1\n");  
    pid = fork();  
    if (pid == 0) {  
        //sleep (2);  
        printf ("S3\n");  
        exit (0);  
    } else {  
        //sleep (2);  
        printf ("S2\n");  
        wait ((int *) 0);  
    }  
}
```

Debug ...

Figlio

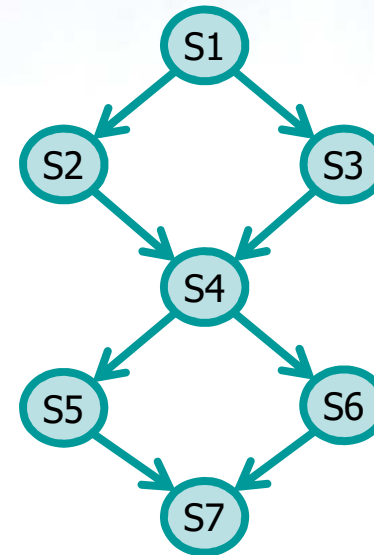
Padre

PID ritornato ignorato

Stato di terminazione  
trascurato

## Soluzione

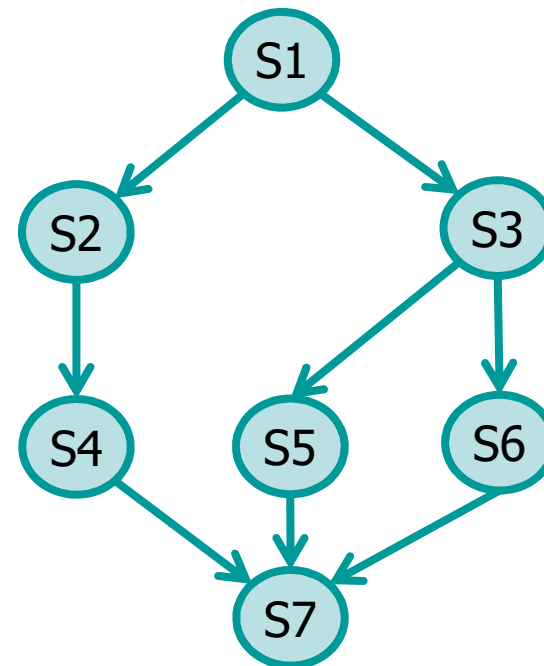
```
printf ("S4\n");  
pid = fork();  
if (pid == 0) {  
    //sleep (2);  
    printf ("S6\n");  
    exit (0);  
} else {  
    //sleep (2);  
    printf ("S5\n");  
    wait ((int *) 0);  
}  
printf ("S7\n");  
return (0);  
}
```





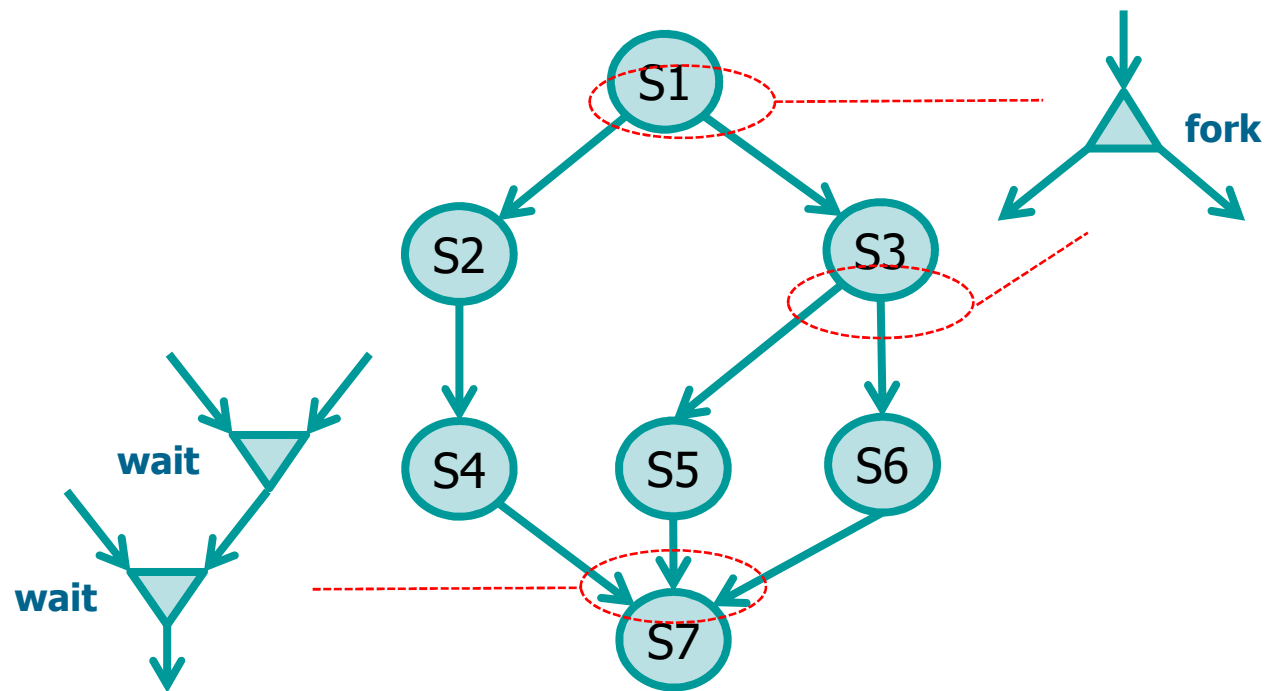
## Esercizio

- ❖ Realizzare un unico processo in cui blocchi di istruzioni rispettino il seguente Control Flow Graph (CFG)
  - Si utilizzino le sole system call **fork** e **wait**



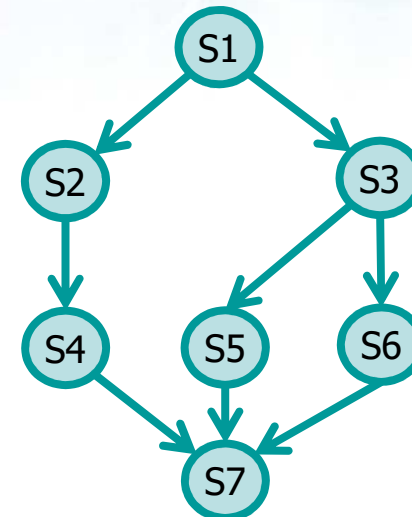
# Esercizio

## ❖ Interpretazione



# Soluzione

```
int main () {
    pid_t pid;
    printf ("S1\n");
    if ( (pid = fork())== -1 )
        err_sys( "can't fork" );
    if ( pid == 0 ){
        P356();
    } else {
        printf ("S2\n");
        printf ("S4\n");
        while (wait((int *)0) != pid);
        printf ("S7\n");
        exit (0);
    }
    return (1);
}
```

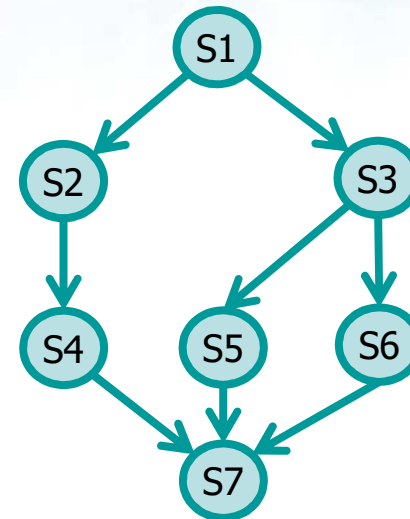


Controllo su diverse terminazioni (inutile in questo caso e sostituibile con waitpid)

# Soluzione

```

P356() {
    pid_t pid;
    printf ("S3\n");
    if ( ( pid = fork() ) == -1 )
        err_sys( "can't fork" );
    if (pid > 0 ) {
        printf ("S5\n");
        while (wait((int *)0)!=pid );
    } else {
        printf ("S6\n");
        exit (0);
    }
    exit (0);
}
    
```



## Esercizio

- ❖ Scrivere un programma in grado di
  - Ricevere sulla riga di comando un valore intero **n**
  - Allocare dinamicamente un vettore di interi di dimensione **n** e leggerlo da tastiera
  - Visualizzare (a video) gli elementi del vettore in ordine inverso (dall'elemento **n** all'elemento **0**) utilizzando **n-1** processi ciascuno dei quali visualizza un singolo elemento del vettore
  
- Suggerimento
  - Sincronizzare i processi mediante system call **wait** in modo da ottenere l'ordine di visualizzazione desiderato

## Soluzione

```
int main(int argc, char *argv[]) {
    int i, n, *vet;
    int retValue;
    pid_t pid;
    n = atoi (argv[1]);
    vet = (int *) malloc (n * sizeof (int));
    if (vet==NULL) {
        fprintf (stderr, "Allocation Error.\n");
        exit (1);
    }
    fprintf (stdout, "Input:\n");
    for (i=0; i<n; i++) {
        fprintf (stdout, "vet[%d]:", i);
        scanf ("%d", &vet[i]);
    }
}
```

## Soluzione

```
fprintf (stdout, "Output:\n");  
for (i=0; i<n-1; i++) {  
    pid = fork();  
    if (pid>0) {  
        pid = wait (&retValue);  
        break;  
    }  
    fprintf (stdout, "Run PID=%d\n", getpid());  
}  
  
fprintf (stdout, "vet[%d]:%d - ", i, vet[i]);  
fprintf (stdout, "End PID=%d\n", getpid());  
  
exit (0);  
}
```

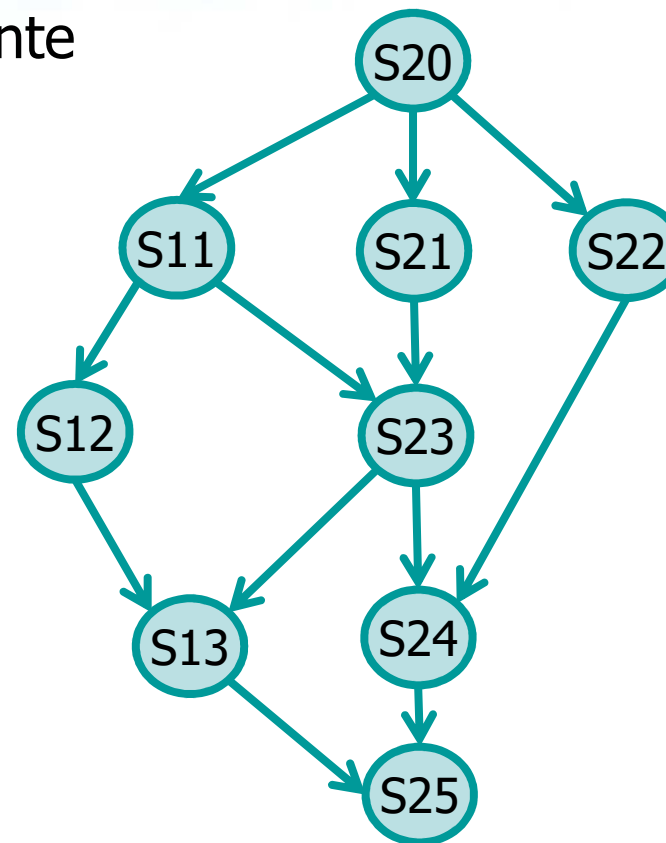
Il padre attende il figlio e quindi esce dal ciclo

Il figlio procede nell'iterazione

## Esercizio

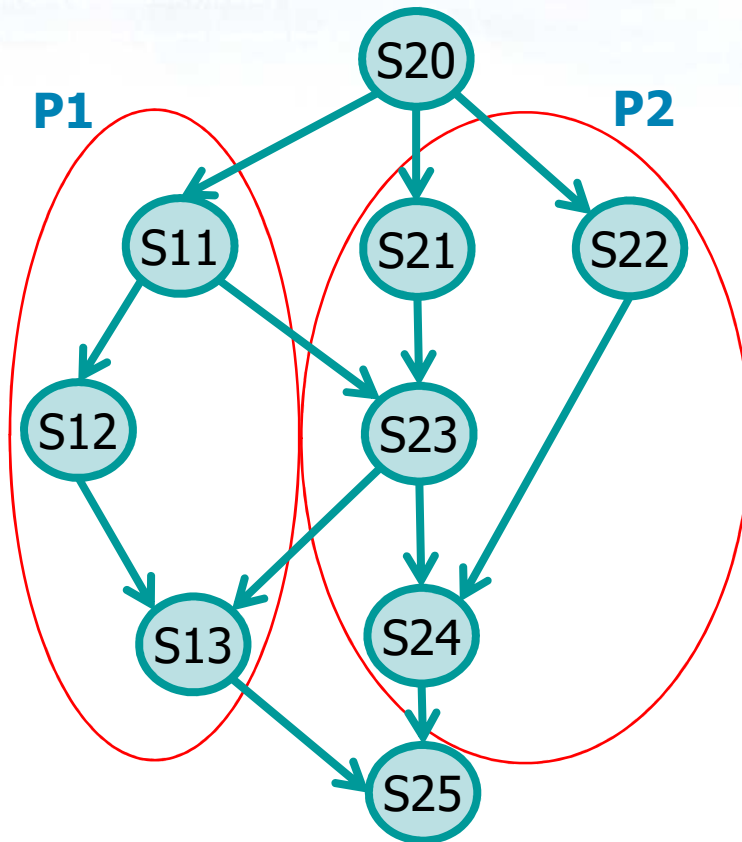
- ❖ Realizzare un unico processo in cui blocchi di istruzioni rispettino il seguente Control Flow Graph (CFG)

- Si utilizzino le sole system call **fork** e **wait**



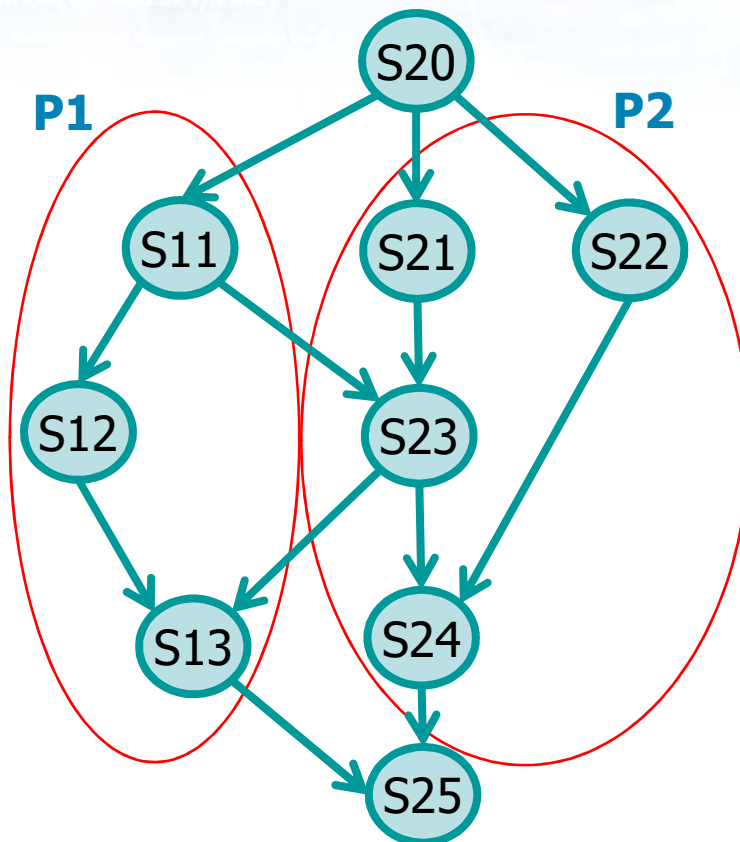


## Soluzione



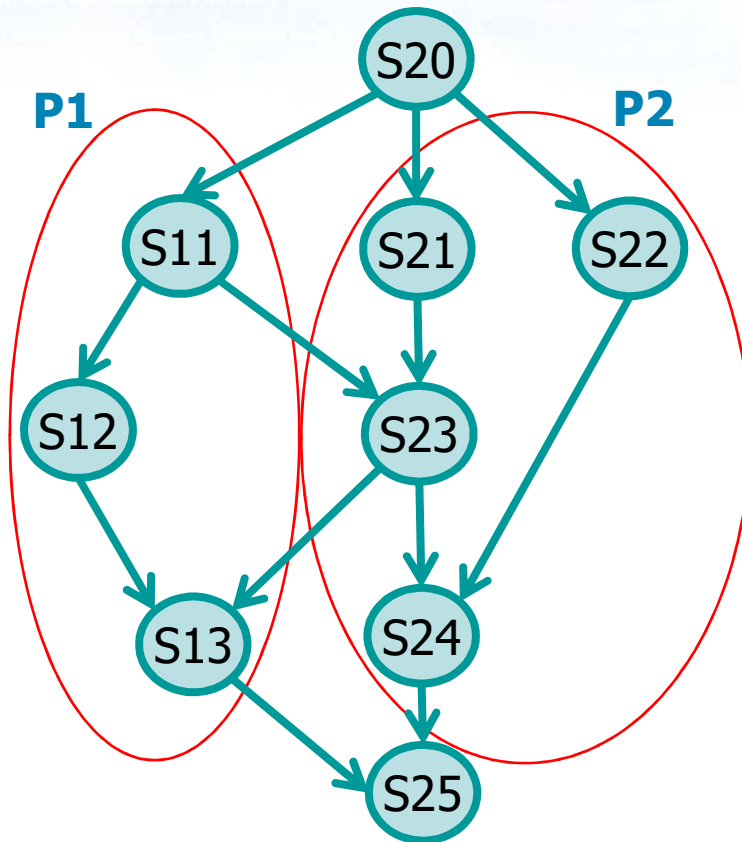
```
main () {
  S20 ();
  pid = fork ();
  if (pid>0) {
    P1 ();
    wait ((int *)0);
  } else {
    P2 ();
  }
  S25 ();
  return;
}
```

## Soluzione



```
P1() {  
    S11 ();  
    pid = fork ();  
    if (pid>0) {  
        S12 ();  
        wait((int *)0);  
    } else {  
        ??? To P2 ???;  
        exit(0);  
    }  
    S13 ();  
}
```

## Soluzione



```
P2() {  
  pid = fork ();  
  if (pid>0) {  
    S21 ();  
    ??? From S1 ???;  
    S23 ();  
    wait((int *)0);  
  } else {  
    S22 ();  
    exit(0);  
  }  
  S24 ();  
  exit (0);  
}
```

Soluzione

Grafo irrealizzabile

```

P1() {
  S11 ();
  pid = fork ();
  if (pid>0) {
    S12 ();
    wait((int *)0);
  } else {
    ??? To P2 ???;
    exit(0);
  }
  S13 ();
}
    
```

```

P2() {
  pid = fork ();
  if (pid>0) {
    S21 ();
    ??? From S1 ???;
    S23 ();
    wait((int *)0);
  } else {
    S22 ();
    exit(0);
  }
  S24 ();
  exit (0);
}
    
```

