

Gestione dei processi

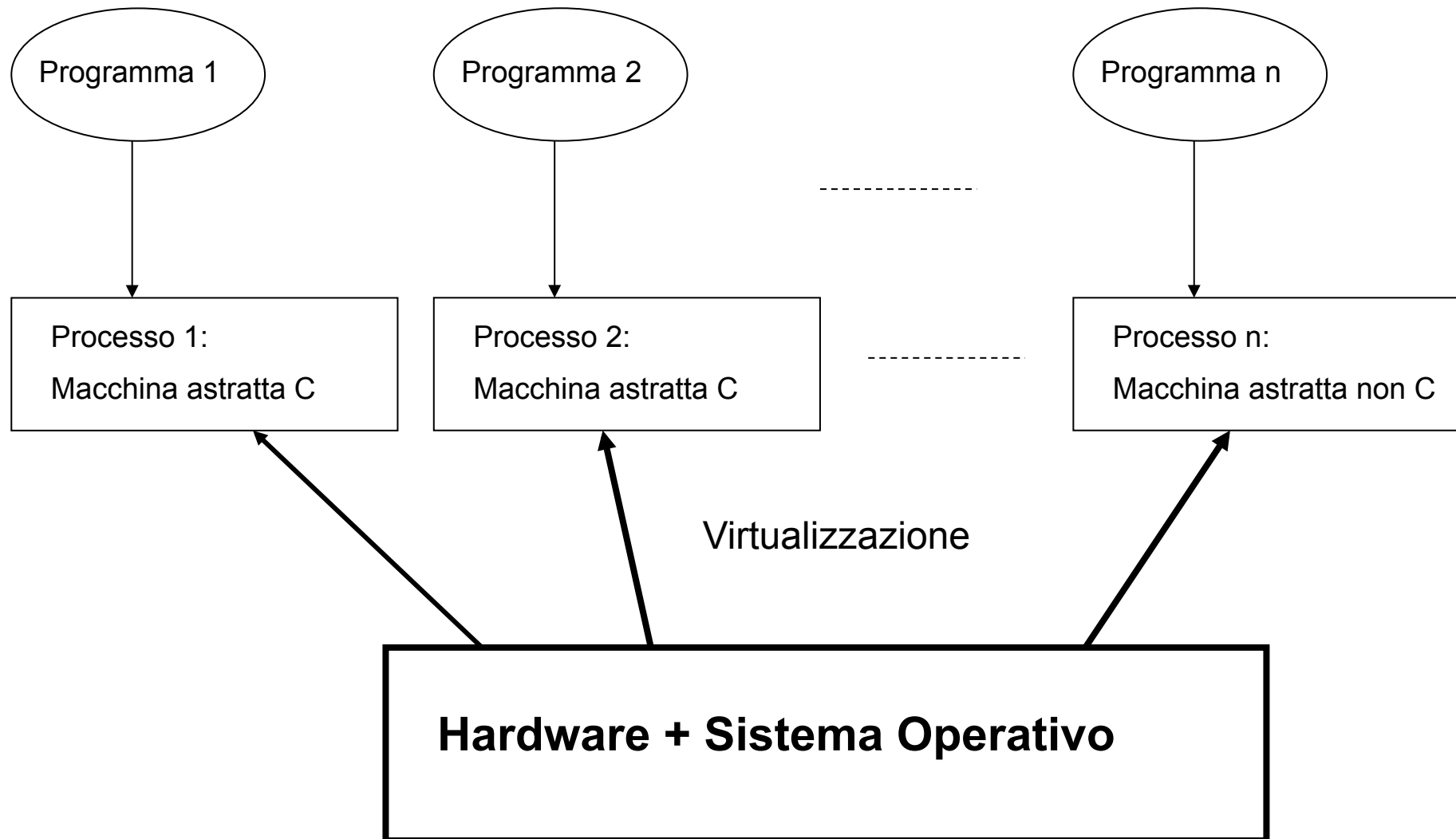
Introduzione

- Dalla macchina astratta sequenziale C
- Alla macchina “parallela”
 - Processi e programmi
- Le primitive per la gestione dei processi
 - Creazione, sospensione, morte dei processi
 - Esecuzione dei programmi
- Semplici(ssime) applicazioni

Processo

- Una singola macchina astratta cui viene assegnato un compito
 - Sistema:
 - Diversi processi
 - Che cooperano (o si coordinano, o competono) procedendo in parallelo spesso in modo asincrono
 - Qui ci interessano processi realizzati mediante macchine astratte informatiche

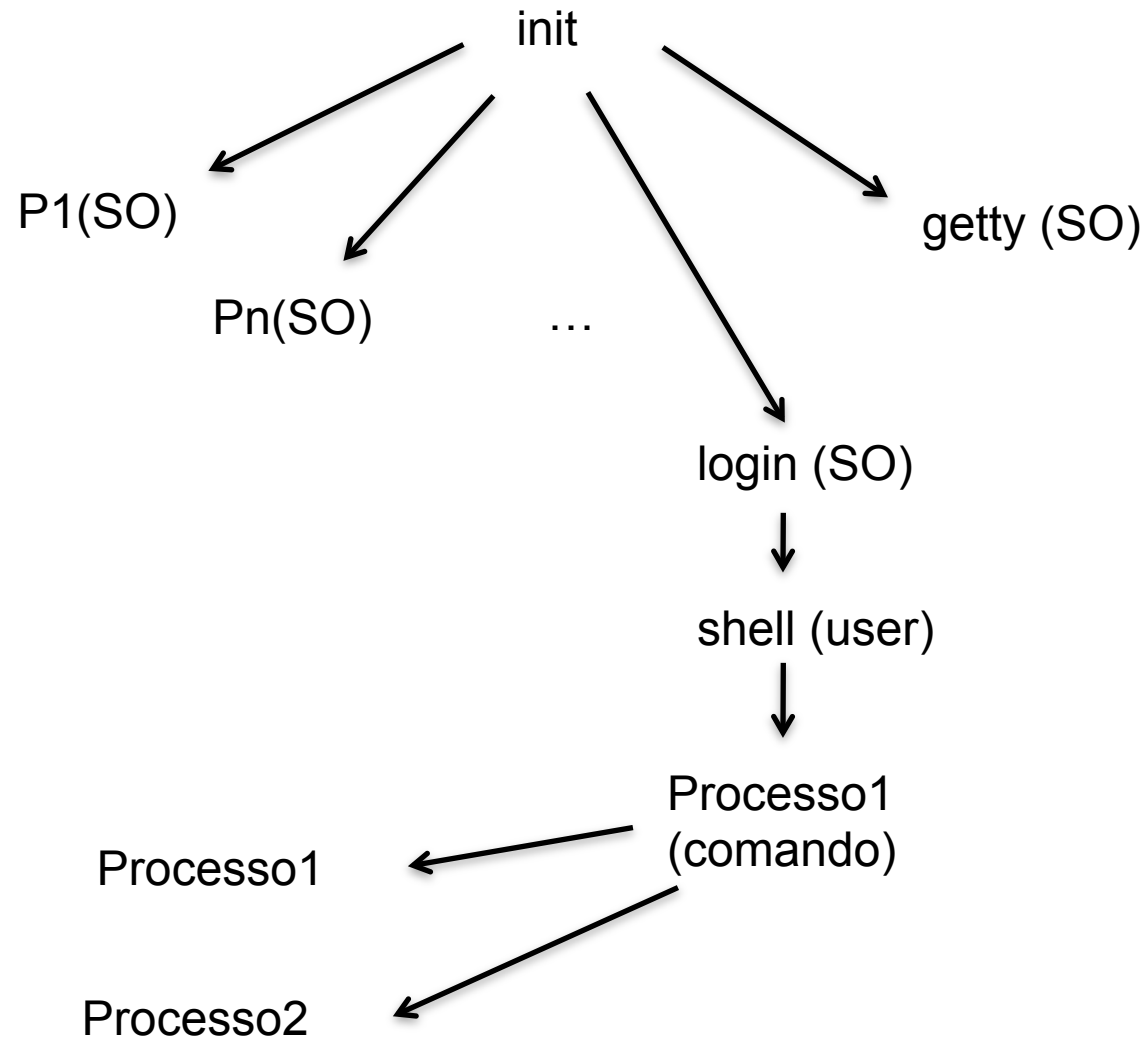
Sistema di processi su un calcolatore



Aspetti generali relativi ai processi

- Ogni processo è identificato da un **PID (Process Identifier)**
- Tutti i processi sono creati da altri processi e quindi hanno un processo padre
 - Unica eccezione: il processo "init"
- Memoria di lavoro associata ad un processo:
 - Segmento **codice** contiene l'eseguibile del programma
 - Segmento **dati** contiene tutte le variabili del programma
 - Segmento **di sistema** contiene i dati non gestiti esplicitamente dal programma in esecuzione, ma dal SO

La gerarchia dei processi



Primitive per la gestione di processi

- Generare un processo figlio
- Attendere la terminazione di un processo figlio
- Terminare (“uccidere”) un processo figlio
- Sostituire il programma (segmento codice) eseguito da un processo

Generazione

- `pid_t fork (void)`
- Biforca il processo in padre e figlio
 - al figlio restituisce sempre 0
 - al padre restituisce il `pid > 0` del figlio biforcato
 - restituisce -1 se la biforcazione del processo fallisce
 - restituisce -1 solo al padre, ovviamente, visto che il figlio non è stato biforcato

Processo figlio

- Tutti i segmenti del padre sono duplicati nel figlio
- La duplicazione del segmento di sistema prodotto dalla fork copia la tabella dei file aperti

Terminazione

- void **exit** (int stato)
 - Per il momento senza parametro, quindi exit(0)
- Termina il processo corrente
- Simile alla return: può essere superflua, ad esempio, se il programma giunge alla fine del codice

Esempio

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

Necessario per
importare il tipo pid_t.

```
void main(){
    pid_t pid;
```

```
    pid = fork();
    if (pid==0){
        printf ("sono il processo figlio\n");
        exit(0);
```

Solo il processo figlio eseguirà
questa parte di codice.

```
    }
    else {
        printf ("sono il processo padre\n");
        exit(0);
```

Solo il processo padre eseguirà
questa parte di codice.

```
    }
}
```

getpid()

- Consente ad un processo di conoscere il valore del proprio pid
- Prototipo
 - pid_t **getpid**(void)

Esempio

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

```
void main( )
{   pid_t pid, miopid;
```

```
    pid=fork();
    if (pid==0) {
        miopid=getpid( );
        printf("sono il processo figlio con pid: %i\n\n",miopid);
        exit(0);
    }
```

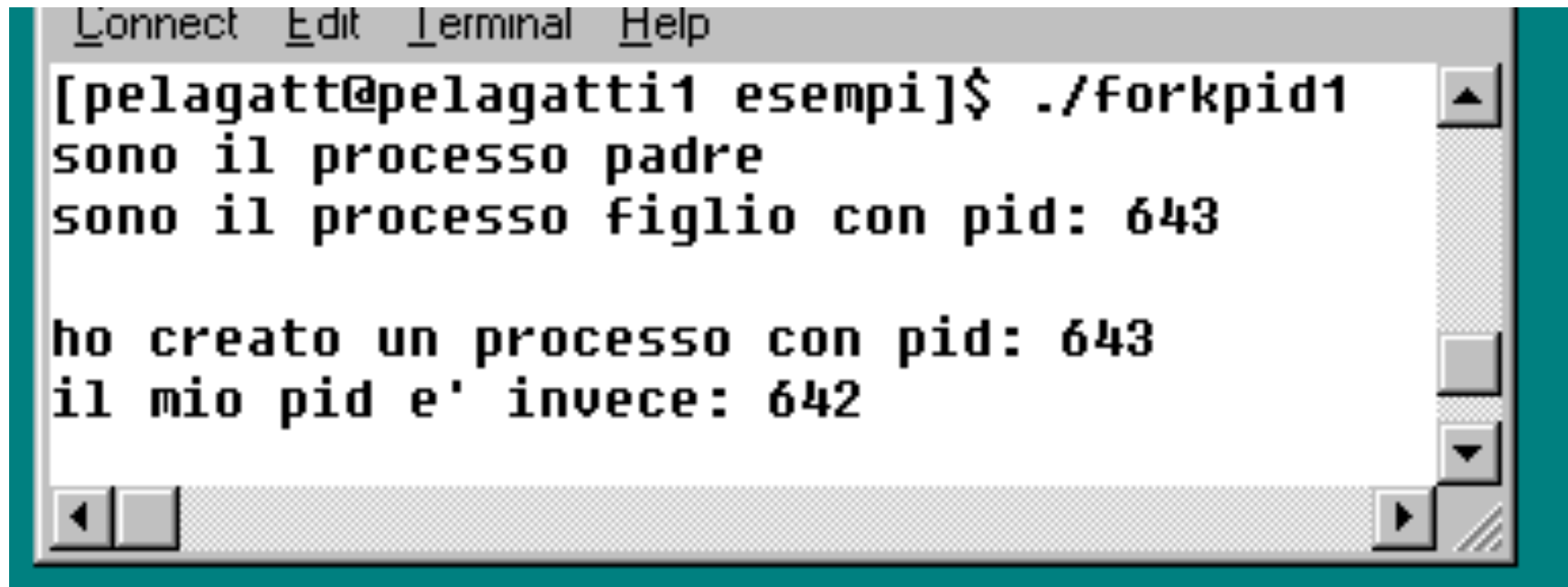
```
    else {
        printf("sono il processo padre\n");
        printf("ho creato un processo con pid: %i\n", pid);
        miopid=getpid( );
        printf("il mio pid e' invece: %i\n\n", miopid);
        exit(0);
    }
```

```
}
```

Solo il processo figlio eseguirà questa parte di codice.

Solo il processo padre eseguirà questa parte di codice.

Possibile esecuzione



```
Connect Edit Terminal Help
[pe1agatt@pe1agatti1 esempi]$ ./forkpid1
sono il processo padre
sono il processo figlio con pid: 643

ho creato un processo con pid: 643
il mio pid e' invece: 642
```

The image shows a terminal window with a menu bar containing 'Connect', 'Edit', 'Terminal', and 'Help'. The prompt is '[pe1agatt@pe1agatti1 esempi]\$'. The user has executed the command './forkpid1'. The output consists of two lines of text: 'sono il processo padre' and 'sono il processo figlio con pid: 643'. Below this, there is a blank line, followed by two more lines of text: 'ho creato un processo con pid: 643' and 'il mio pid e' invece: 642'. The terminal window has a standard Linux-style scrollbar on the right side.

Esempio

```
void main( )
{
    pid_t pid;

    pid=fork( );
    if (pid==0) {
        printf("1)sono il primo processo figlio con pid: %i\n", getpid( ));
        exit( );
    } else {
        printf("2)sono il processo padre\n");
        printf("3)ho creato un processo con pid: %i\n", pid);
        printf("4)il mio pid e' invece: %i\n", getpid( ));
        pid=fork( );
        if (pid==0) {
            printf("5)sono il secondo processo figlio con pid: %i\n", getpid( ));
        } else {
            printf("6)sono il processo padre\n");
            printf("7)ho creato un secondo processo con pid: %i\n", pid);
        }
    }
}
```

Solo il primo processo figlio
eseguirà questa parte di codice.

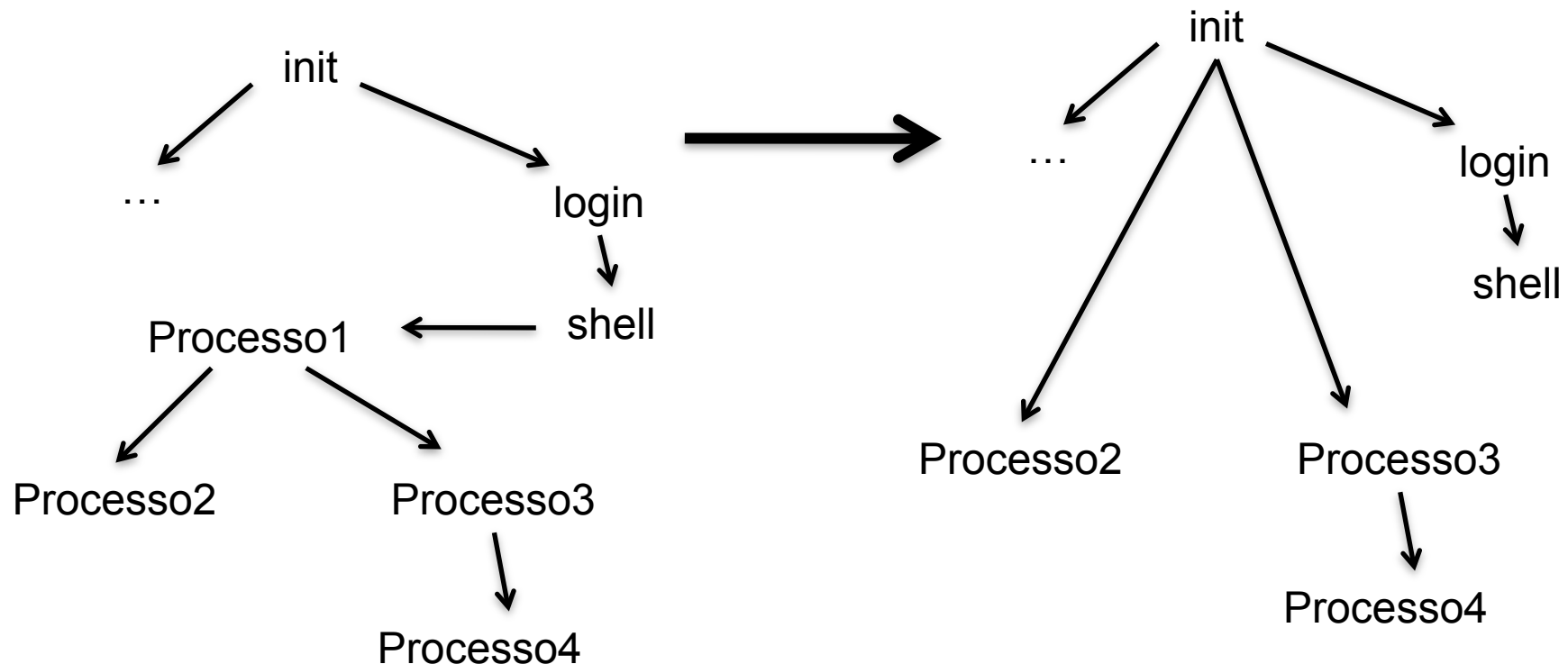
Secondo
figlio.

Possibile esecuzione

```
Connect Edit Terminal Help
[pelegatt@pelegatti1 esempi]$ ./forkpid2
2)sono il processo padre
1)sono il primo processo figlio con pid: 695
3)ho creato un processo con pid: 695
4)il mio pid e' invece: 694
6)sono il processo padre
5)sono il secondo processo figlio con pid: 696
7)ho creato un secondo processo con pid: 696
[pelegatt@pelegatti1 esempi]$
```


Che accade se un padre termina prima del(i) figli(o)?

La convenzione adottata da Linux è di far adottare i processi figli rimasti orfani (processi 2 e 3) e tutta la loro discendenza (processo 4) al processo init del sistema operativo



Sincronizzare processi asincroni

- Elemento fondamentale della gestione del parallelismo: processi indipendenti fino a ...
- `pid_t wait (int *)`
 - Sospende l'esecuzione del processo che la esegue e attende la terminazione di un **qualsiasi** processo figlio
 - Se un figlio è già terminato la wait del padre si sblocca immediatamente (nessun effetto)
 - Ritorna il pid del processo figlio terminato

Possibile utilizzo

- `pid_t pid;`
- `int stato;`
- `pid = wait (&stato);`
- La variabile `stato` è parametro passato per indirizzo:
 - Codice di terminazione del processo
 - Gli 8 bit superiori possono essere assegnati esplicitamente come parametro di `exit` del processo figlio che termina
 - Altri bit di `stato` assegnati dal S.O. per indicare condizioni di terminazione (e.g., errore)

exit()

- Esempio: exit(5)
 - Termina il processo e restituisce il valore 5 al padre
 - Se il padre è già terminato lo stato viene restituito all'interprete comandi
 - Se il padre è bloccato su una wait(), la wait() si sblocca e gli 8 bit superiori prendono il valore 5
 - Il valore restituito è costituito dagli 8 bit superiori di stato → lo stato ricevuto da wait è il parametro di exit moltiplicato per 256 (1280 in questo caso)

Esempio

```
#include <stdio.h>
#include <sys/types.h>
void main( )
{   pid_t pid;
    int stato_wait;

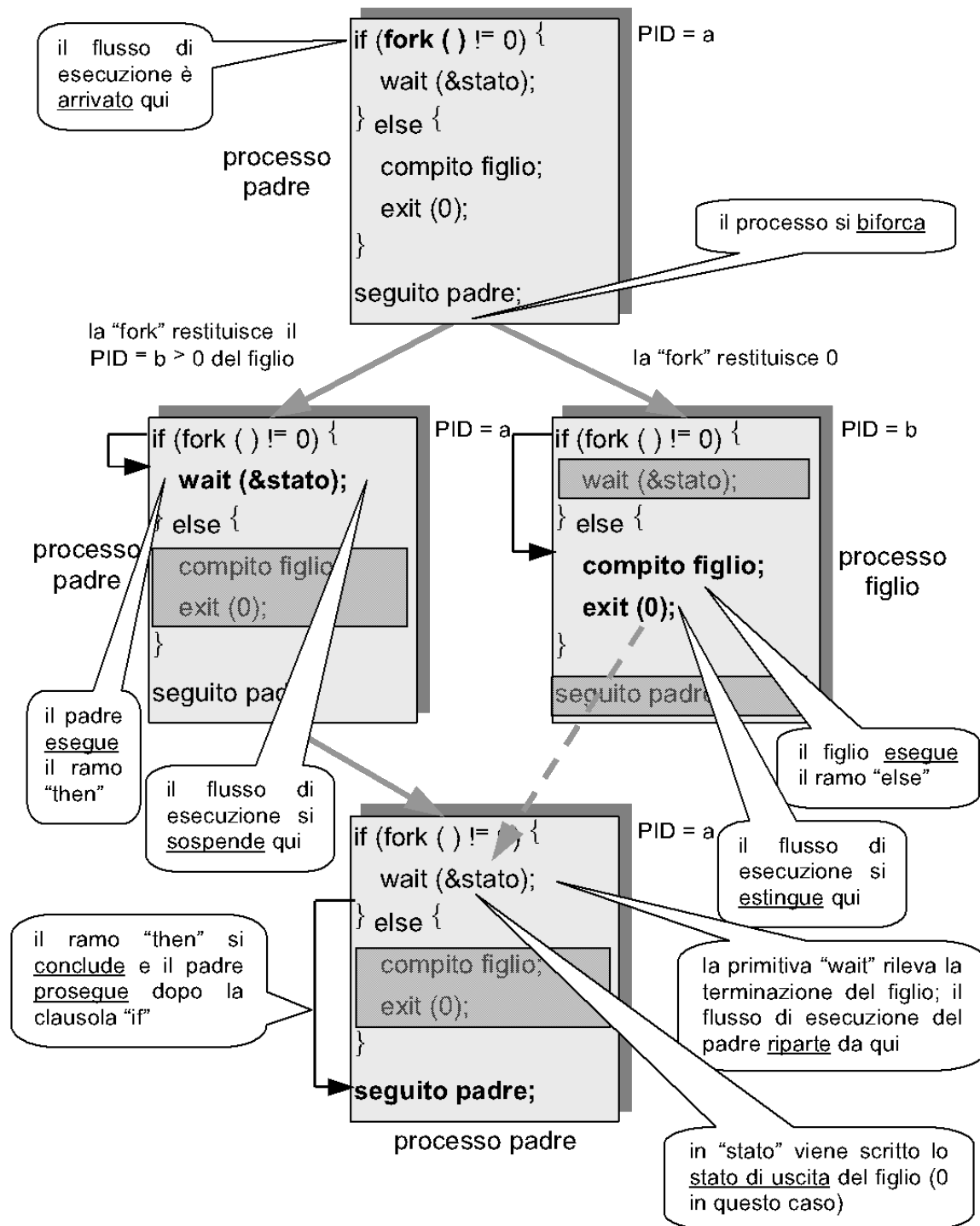
    pid=fork( );
    if (pid==0) {
        printf("sono il processo figlio con pid %i \n", getpid( ));
        printf("termino \n\n");
        exit(5);
    } else {
        printf("ho creato un processo figlio \n\n");
        pid=wait (&stato_wait);
        printf("terminato il processo figlio \n");
        printf("il pid del figlio era %i, lo stato con cui ha terminato è %i\n",pid,stato_wait/256);
    }
}
```

Il valore 5 passa dal figlio al padre tramite exit() e wait().

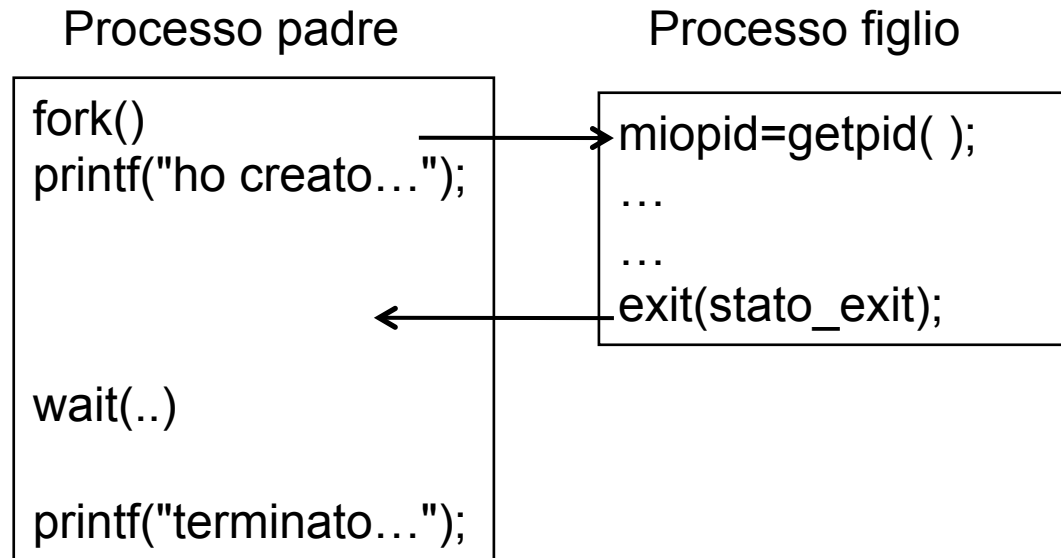
Risultato

```
Connect Edit Terminal Help
[pelegatt@pelegatti1 esempi]$ ./forkwait1
ho creato un processo figlio
sono il processo figlio con pid 769
termino

terminato il processo figlio
il pid del figlio e' 769, lo stato e' 5
[pelegatt@pelegatti1 esempi]$
```



Attenzione



- Se il figlio è già terminato la wait() del padre si sblocca immediatamente, e riceve comunque le informazioni di stato
- A tal fine, il SO procede come segue:
 - Memorizza il valore di stato nella parte di sistema operativo dedicata al processo
 - Forza la chiusura di tutti i file aperti presenti nella tabella dei file aperti del segmento di sistema del processo
 - Passa il processo dallo stato di “attivo” allo stato di “zombie”
- Il processo figlio dopo l’exit rimane quindi “in vita”, ma solo per aspettare che il processo padre possa recuperare lo stato

Altri casi rilevanti

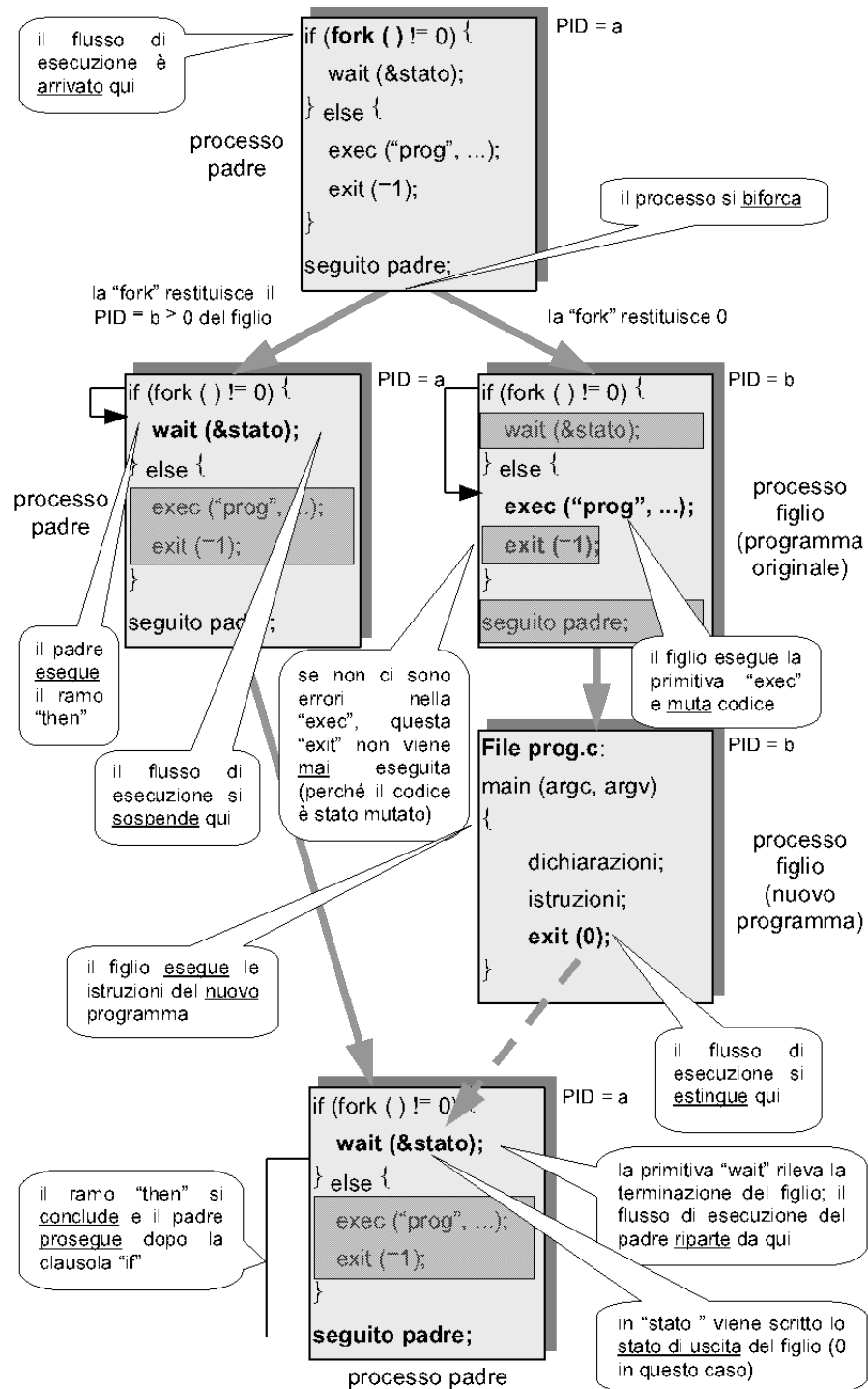
- Un processo padre che non ha generato processi figli esegue una `wait()`
 - Il sistema operativo restituisce il codice di errore -1 e non pone in attesa il processo padre
- Un processo padre esegue una `wait()` in presenza di un processo figlio che non esegue mai una `exit` (ad esempio per un ciclo infinito)
 - Il processo padre rimane sospeso all'infinito
 - Questa situazione richiede un intervento esterno per forzare la terminazione di entrambi i processi
- Un processo padre termina l'esecuzione del proprio programma, provocando la propria distruzione senza eseguire una `wait`, in presenza di uno o più processi figli attivi
 - Il sistema operativo prende tutti i processi rimasti orfani dalla morte del processo padre e li fa adottare al processo `init`
 - Quando questi processi figli eseguono l'`exit` passano allo stato zombie senza avere più un padre che li aspetti
 - Periodicamente, il processo `init` esegue una `wait` proprio al fine di eliminare i processi zombie inutilmente presenti nel sistema

waitpid()

- `pid_t waitpid (pid_t pid, int * stato, int opzioni)`
- Esempio: `waitpid (10, &stato, opzioni)`
- Mette un processo in stato di attesa dell'evento di terminazione di un processo figlio con **un determinato pid** "pid" (10 in questo caso) e ne restituisce il pid (10 in questo caso)
- La variabile "stato" assume il valore di "exit" del processo figlio terminato
- Il parametro "opzioni" specializza la funzione "waitpid"

Sostituzione del programma in esecuzione

- **exec()** e le sue varianti
 - Sostituisce i segmenti codice e dati del processo in esecuzione con codice e dati di un programma contenuto in un file eseguibile specificato
 - Attenzione:
 - I file precedentemente aperti rimangono aperti!
 - Il processo rimane lo stesso (stesso pid)
 - Programma \neq processo!
 - Può passare parametri al nuovo programma
 - main è semplicemente una particolare funzione!!
 - Esistono diverse varianti di exec



exec()

- **exec()** (char *nome_programma, char *arg0, char *arg1, ..., NULL);
 - accetta una **lista** di parametri (da cui il nome)
 - nome_programma: stringa che identifica completamente (pathname) il file eseguibile contenente il programma da lanciare
 - arg0, arg1, ...: puntatori a stringhe da passare come parametri al main da lanciare; l'ultimo è NULL perché il numero di arg è variabile
- Il main, che è una particolare funzione, può avere a sua volta dei parametri!

main

- `void main (int argc, char *argv[])`
 - `argc`: numero di parametri ricevuti
 - `argv[]`: vettore di puntatori a stringhe;
 - ogni stringa è un parametro
 - `argv[0]` contiene sempre il nome del programma
- `execl` provoca quindi l'esecuzione del ("chiama" il) programma il cui eseguibile si trova nel file `nome_programma` e gli passa come parametri (per indirizzo: sono puntatori) `arg0`, `arg1`, ...)

Esempio

```
#include <stdio.h>
#include <sys/types.h>
void main( )
{
    char P0[ ]="main1";
    char P1[ ]="parametro 1";
    char P2[ ]="parametro 2";

    printf("sono il programma exec1\n");

    execl("/home/pelagatt/esempi/main1", P0, P1, P2, NULL);

    printf("errore di exec");
}
```



Normalmente non si
arriva mai qui!

Possibile esecuzione

```
Connect Edit Terminal Help
[pelegatt@pelegatti1 esempi]$ ./exec1
sono il programma exec1

sono il programma main1
ho ricevuto 3 parametri
il parametro 0 è: main1
il parametro 1 è: parametro 1
il parametro 2 è: parametro 2
[pelegatt@pelegatti1 esempi]$
```


Altre versioni di exec

- **execv()**: sostituisce alla lista di stringhe di `exec1` un puntatore a un vettore di stringhe
`char ** argv`
- **execlp()** e **execvp()** permettono di sostituire il pathname completo con il solo nome del file nel direttorio di default
- **execle()** e **execve()** hanno un parametro in più che specifica l'ambiente di esecuzione del processo

Exec e fork in combinazione

- il padre crea uno o più figli e assegna loro un compito
- attende i loro risultati
- quando hanno finito e prodotto i risultati li raccoglie e li gestisce
 - Interprete comandi

Esempio

```
#include <stdio.h>
#include <sys/types.h>
void main( ) {
    pid_t pid;
    int stato_wait;
    char P0[ ]="main1";
    char P1[ ]="parametro 1";
    char P2[ ]="parametro 2";

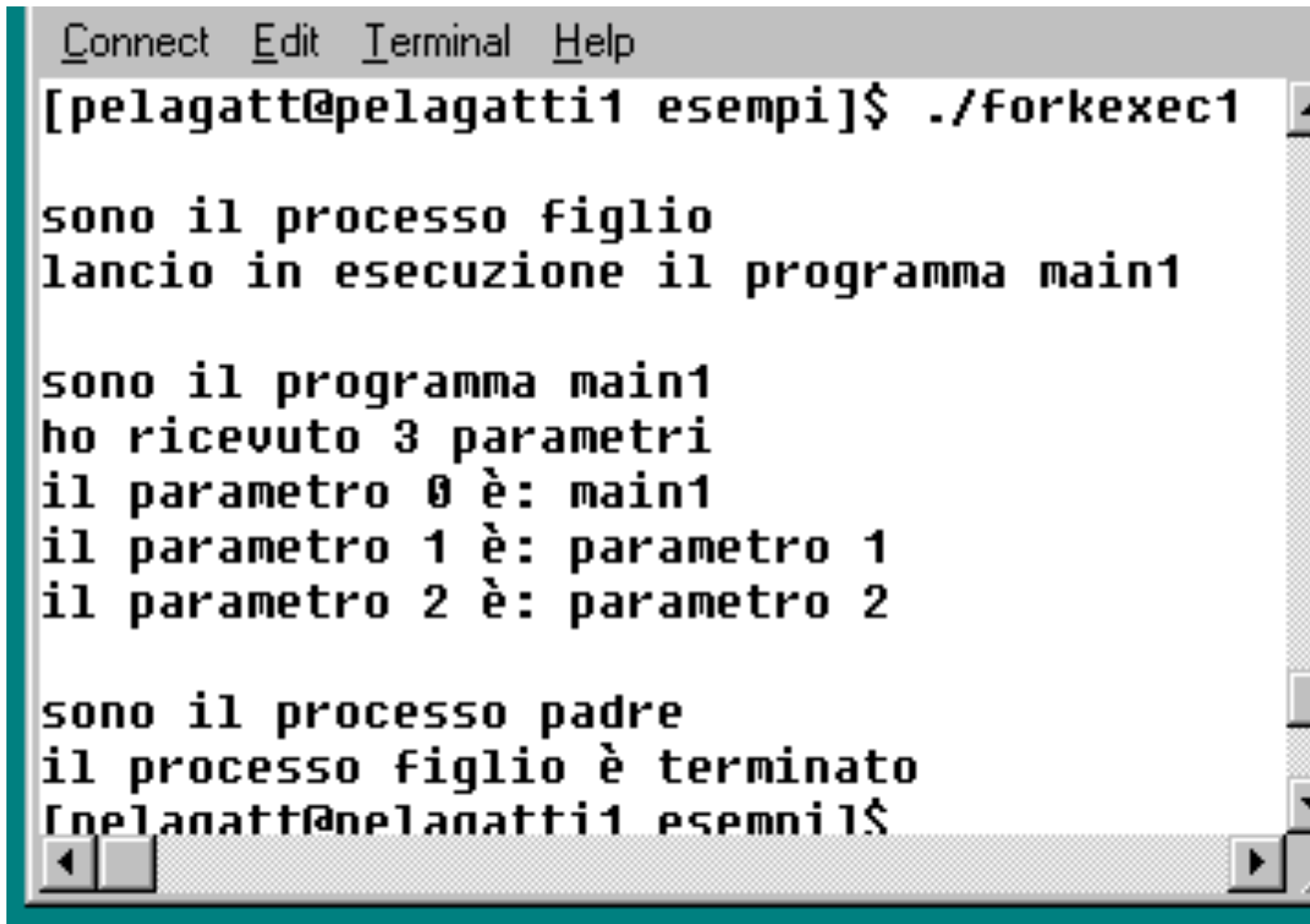
    pid=fork( );

    if (pid==0) {
        printf("\n sono il processo figlio \n");
        printf("lancio in esecuzione il programma main1\n");
        execl("/home/pelagatt/esempi/main1", P0, P1, P2, NULL);
        printf("errore di exec");
        exit(0);
    } else {
        wait(&stato_wait );
        printf("\nsono il processo padre\n");
        printf("il processo figlio è terminato\n");
        exit(0);
    }
}
```



Normalmente non si
arriva mai qui!

Possibile esecuzione



```
Connect Edit Terminal Help
[pe1agatt@pe1agatti1 esempi]$ ./forkexec1

sono il processo figlio
lancio in esecuzione il programma main1

sono il programma main1
ho ricevuto 3 parametri
il parametro 0 è: main1
il parametro 1 è: parametro 1
il parametro 2 è: parametro 2

sono il processo padre
il processo figlio è terminato
[pe1agatt@pe1agatti1 esempi]$
```

The image shows a terminal window with a menu bar (Connect, Edit, Terminal, Help) and a title bar. The prompt is [pe1agatt@pe1agatti1 esempi]\$. The user runs ./forkexec1. The output shows the execution flow: the parent process forks a child process, the child process runs main1 with three arguments, and then the parent process continues, indicating the child has terminated.


Ottavo esempio

- Pseudocodice di un interprete comandi semplificato (programma simpleshell) che legge da terminale un comando, procede a creare un processo figlio dedicato all'esecuzione del comando, mentre il processo padre ne attende la sua terminazione prima di ripetere la richiesta di un altro comando

Pseudocodice

```
#include <stdio.h>
#include <sys/types.h>
#define fine "logout"
#define prompt "simpleshell: "

void main( ){
    pid_t  pid; int stato_wait;
    ....
    while (! logout dell' utente) {
        printf ("%s",prompt);
        [ lettura riga di comando e identificazione componenti del comando ]
        pid=fork( );
        if (pid==0) {
            execl(comando, arg0, arg1, ... argn, NULL);
            printf("errore di exec");
            exit(0);
        } else wait(&stato_wait );
    }
    exit(0);
}
```



Il figlio viene sostituito
dal comando da eseguire

Ricapitoliamo...

- Si devono riempire tre tabelle in base all'esecuzione di 3 processi
 - Un padre, un figlio, e un nipote
- Ogni tabella deve indicare:
 - Il valore delle variabili i,j,k, pid1, pid2 in specifici punti (linee di codice) dell'esecuzione
 - Se nel momento indicato la variabile non esiste (perché non esiste il processo) la tabella deve riportare NE
 - Se la variabile esiste ma non se ne conosce il valore con certezza (perché non si sa a che punto si sia dell'esecuzione del singolo processo) la tabella deve riportare U
- Si suppone che tutte le fork abbiano successo e che il S.O. assegni ai figli creati i valori di pid a partire da 500

Codice di esempio...

```
01: main()
02: {
03:     int i, j, k, stato;
04:     pid_t pid1, pid2;
05:     i=10; j=20; k=30;
06:     pid1 = fork();
07:     if (pid1 == 0) {
08:         j=j+1;
09:         pid2 = fork();
10:         if (pid2 == 0) {
11:             k=k+1;
12:             exit();
13:         } else {
14:             wait(&stato);
15:             exit();
16:         }
17:     } else {
18:         i=i+1;
19:         wait(&stato);
20:         exit(); }
21: }
```



Primo figlio.



Secondo figlio
(nipote).

Struttura delle 3 tabelle da compilare

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
Dopo l'istruzione 6					
dopo l'istruzione 9					
dopo l'istruzione 11					
dopo l'istruzione 19					

Variabili nel processo padre

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
Dopo l'istruzione 6	500	U	10	20	30
dopo l'istruzione 9	500	U	U	20	30
dopo l'istruzione 11	500	U	U	20	30
dopo l'istruzione 19	500	U	11	20	30

Il padre non esegue queste istruzioni, e non possiamo sapere se avrà già eseguito la 18 quando i figli eseguono 9 o 11.

Variabili nel primo processo figlio

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
Dopo l'istruzione 6	0	U	10	20	30
dopo l'istruzione 9	0	501	10	21	30
dopo l'istruzione 11	0	501	10	21	30
dopo l'istruzione 19	NE	NE	NE	NE	NE

Variabili nel secondo processo figlio (nipote)

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
Dopo l'istruzione 6	NE	NE	NE	NE	NE
dopo l'istruzione 9	0	0	10	21	30
dopo l'istruzione 11	0	0	10	21	31
dopo l'istruzione 19	NE	NE	NE	NE	NE

In conclusione...

- Ovviamente da un parallelismo puramente logico di questo tipo non ci si possono attendere grandi risultati pratici in termini di efficienza, anzi ...
- Però il parallelismo logico diventa tanto più utile quanto più “indipendenti” diventano i singoli compiti dei vari processi
- Quando poi il parallelismo logico diventa anche fisico (coprocessori, multi-core) ...