

Gestione dei processi

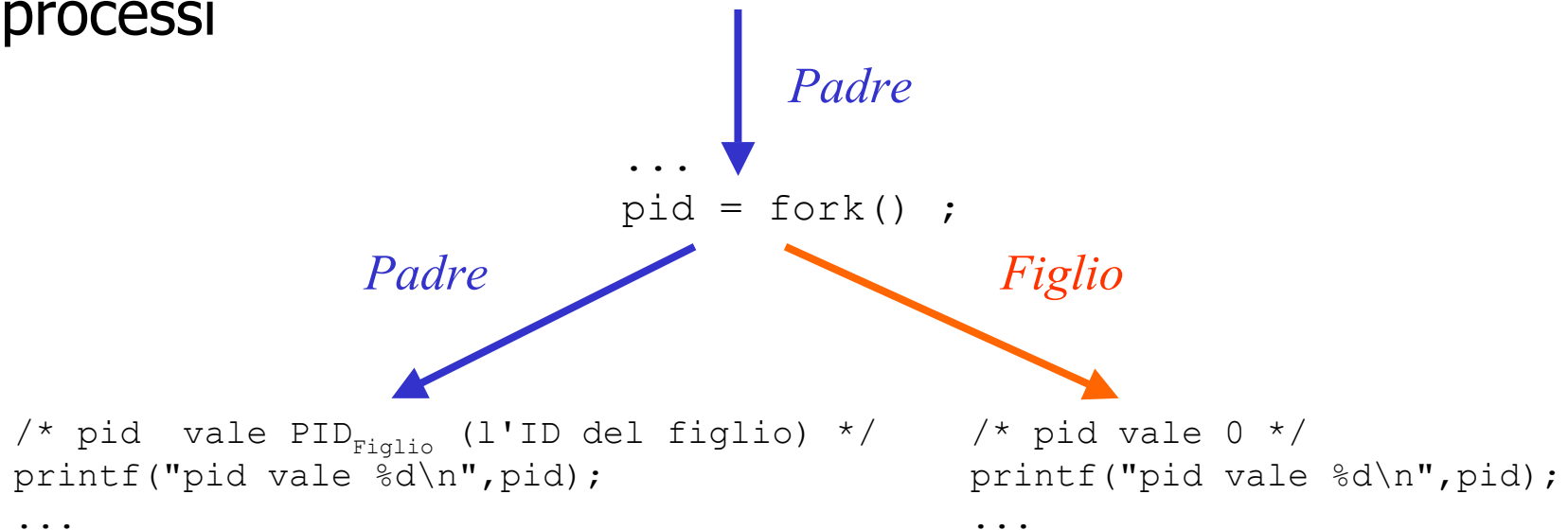
La creazione di un nuovo processo in UNIX

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

```
int fork(void);
```

Viene creato un **nuovo** processo (*figlio*) identico (stesso codice, area dati copiata) al processo (*padre*) che ha invocato la `fork`

Solo il valore di ritorno dalla `fork` è diverso per i due processi



Gestione dei processi

Schema di generazione

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

```
if (fork () ==0)
```

```
    { /* Codice eseguito dal figlio */
```

```
    ...
```

```
    }
```

```
else
```

```
    { /* Codice eseguito dal padre */
```

```
    ...
```

```
    }
```

I ruoli di padre e figlio sono relativi ad una specifica fork : il figlio può a sua volta generare altri processi

Dopo la generazione del nuovo processo, padre e figlio sono processi indipendenti

Il padre può decidere:

- se continuare la propria esecuzione concorrentemente a quella del figlio
- se attendere che il figlio termini (primitiva `wait`)

Gestione dei processi

Il processo figlio :

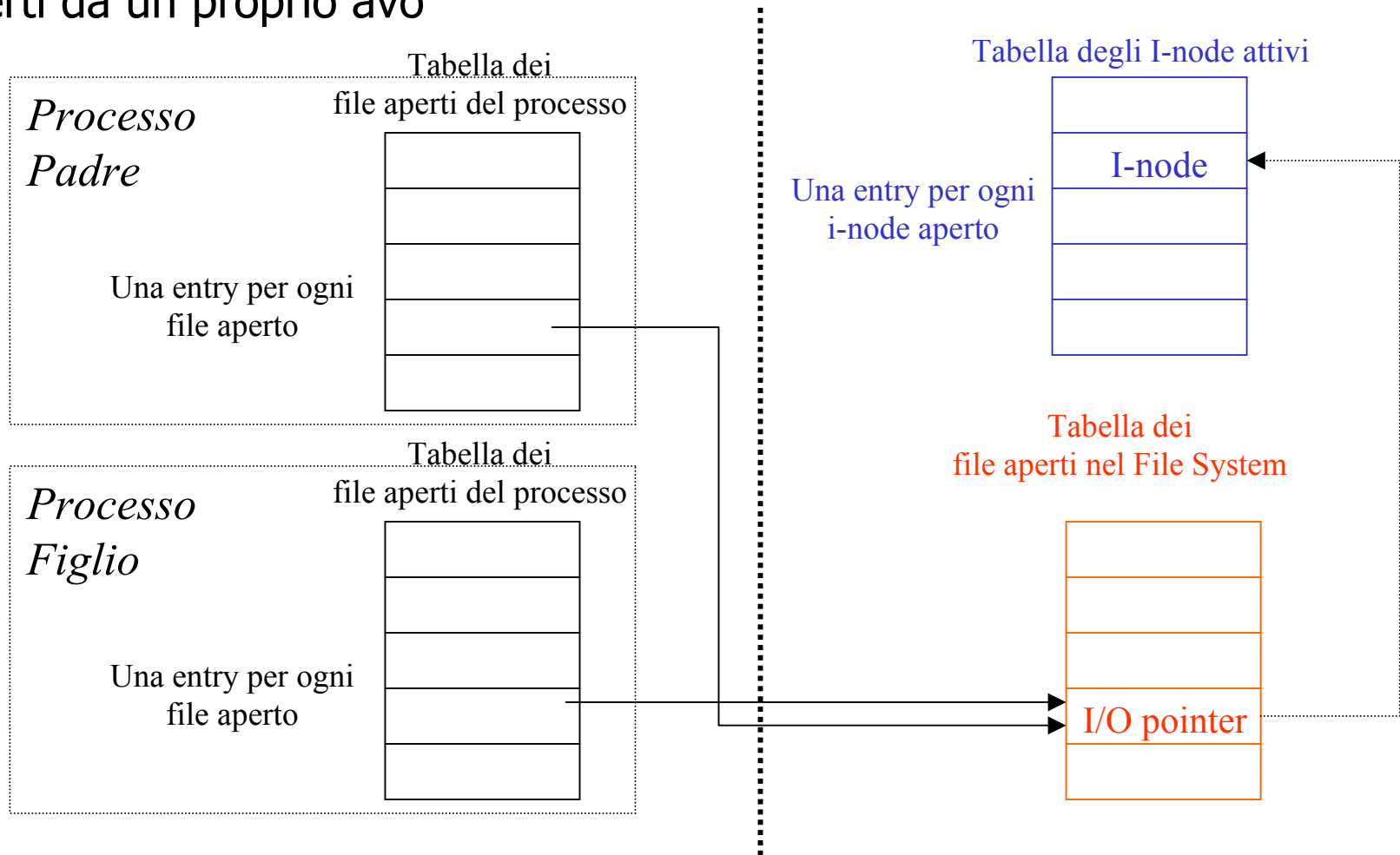
- utilizza lo stesso codice che sta eseguendo il padre ;
- dispone di una area dati (utente e kernel) che è una copia (spesso *on-write*) di quella che ha il padre all'atto della fork;
 - ⇒ non vi è nessuna condivisione di memoria tra i processi padre e figlio (MOLTO IMPORTANTE)

La duplicazione dell'area di kernel ha come effetto la duplicazione della tabella dei file aperti:

- i file aperti dal padre risultano aperti anche dal figlio con una condivisione degli I/O pointer
 - ⇒ le operazioni su uno stesso file condotte da una famiglia di processi modificano l'I/O pointer comune

Operazioni sui file

I processi di una stessa famiglia non hanno una visione indipendente dei file aperti da un proprio avo



È possibile chiudere il file descriptor "ereditato" e riaprire di nuovo il file per avere un accesso indipendente

Gestione dei processi

Identificazione dei processi

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

La `getpid` ritorna al processo il suo identificatore di processo (PID)

La `getppid` ritorna al processo chiamante l'identificatore di processo del suo processo padre (PID del padre)

Dalla `fork` il padre riceve il PID del processo figlio che deve invece invocare `getpid` per conoscere il proprio PID

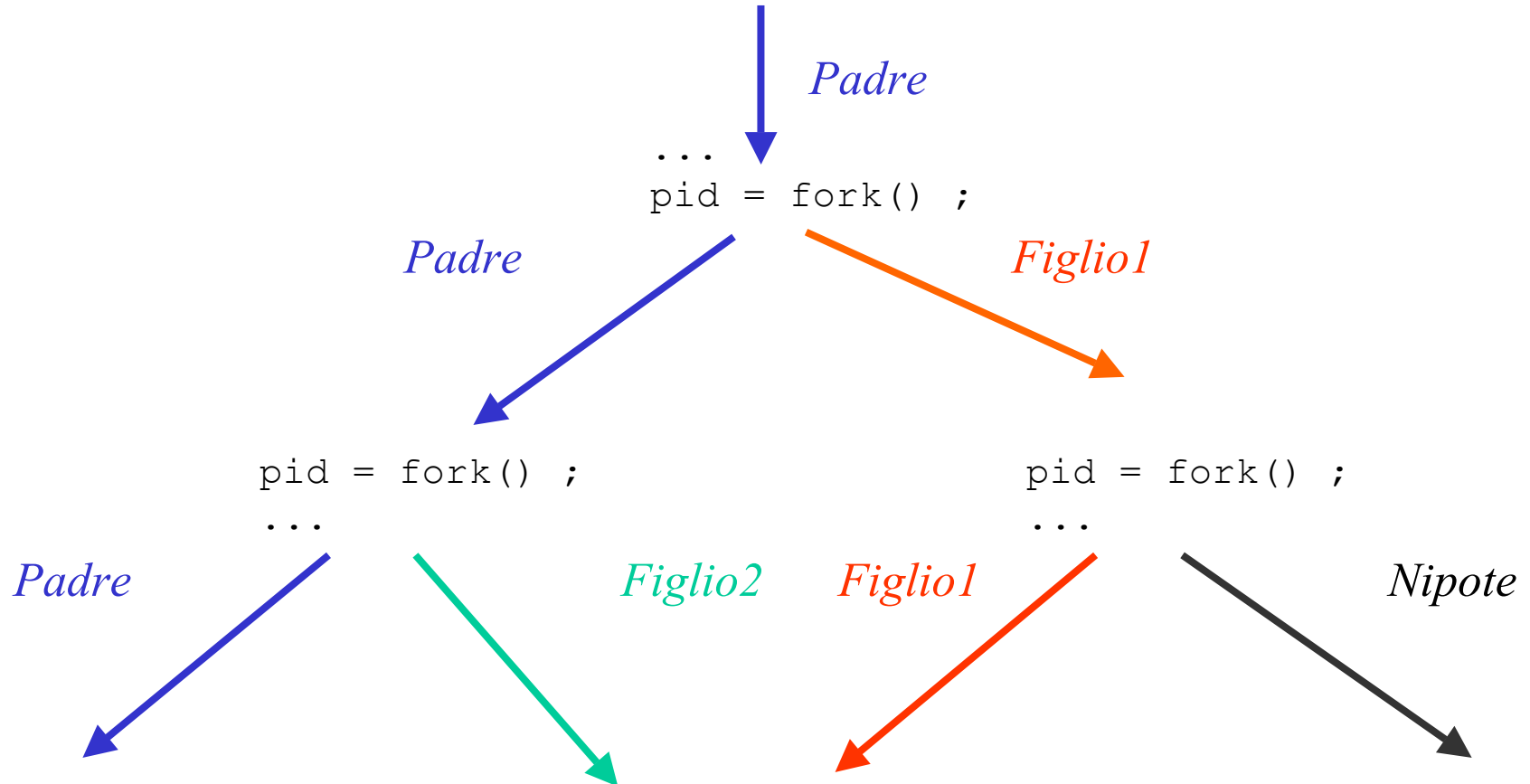
Gestione dei processi

Attenzione alla fork

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

Vengono creati 3 processi

- 2 dal padre
- 1 da un figlio



Gestione dei processi

Sincronizzazione tra padre e figlio

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

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

Il processo chiamante rimane bloccato in attesa della terminazione di uno tra i suoi figli (se i processi figli sono già terminati `wait` ritorna immediatamente)

- `status` contiene il valore di uscita del processo figlio: per ottenerlo (`status >> 8`) oppure `WEXITSTATUS(status)`
- se la `wait` ha successo il valore di ritorno è il pid del processo figlio che è terminato

Con la variante `waitpid` è possibile ottenere informazioni sullo stato dei processi figli anche senza bloccare il processo chiamante

Gestione dei processi

Uso della wait

```
int status;

if (fork () ==0)
    { /* Codice eseguito dal figlio */
    ...
    }
else
    { /* Codice eseguito dal padre */
    ...
    /* Attende che il figlio termini */
    wait(&status);
    ...
    }
```

Nel caso di più figli può essere necessario attendere la terminazione di uno specifico figlio (pid):

```
while ((rid = wait(&status)) != pid);
```

Gestione dei processi

Terminazione volontaria di un processo

```
#include <stdlib.h>
void exit(int status);
```

```
#include <unistd.h>
void _exit(int status)
```

Un processo termina volontariamente invocando la primitiva `_exit` oppure la funzione `exit` della libreria standard I/O del C oppure alla conclusione della funzione `main` (dove viene invocata automaticamente `_exit`);

`status` è il valore di uscita che viene reso disponibile al padre attraverso la `wait` o le altre primitive (`waitpid`, ...)

`_exit` e `exit` chiudono tutti i file aperti del processo

Gestione dei processi

Terminazione involontaria di un processo

Un processo termina involontariamente a seguito di:

- azioni non consentite:
 - riferimenti ad indirizzi di memoria non assegnati al processo (SIGSEGV)
 - esecuzione di codici di istruzioni non definite (SIGILL)
- segnali generati dall'utente da tastiera
 - ^C (SIGINT)
 - ^\ (SIGQUIT) con generazione del *corefile*
- segnali inviati da un altro processo
 - mediante la primitiva kill (che vedremo più avanti)
 - mediante il comando kill (ad es. `kill -9 pidprocesso`)

Gestione dei processi

Esempio di uso della wait

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

int main (int argc, char **argv)
{
    int fd,pid, pid_processo;
    int nread,nwrite,status;
    char st1[256];
    char st2[256];

    if (argc != 2) {fprintf(stderr,"Uso: %s nomefile\n",argv[0]); exit(-1); }

    /* Apertura del file in lettura e scrittura */
    if ((fd=open(argv[1], O_RDWR| O_CREAT|O_TRUNC,0644))<0)
        {perror("opening argv[1]"); exit(-2);}

    if((pid=fork())<0)
        {
            {perror("fork"); exit(-3);}
        }
    else
        if (pid==0) /* Processo figlio */
            {
                /* Continua nella trasparenza successiva ... */
            }
        }
}
```

Gestione dei processi

Esempio

```
    /* ... continua processo figlio */
    printf("Introduci una stringa e premi [Enter]\n");
    scanf("%s",st1);
    /* Il figlio eredita il descrittore fd dal padre */
    nwrite= write(fd,st1,strlen(st1)+1);/* per scrivere anche il '\0' */
    /* L' I/O pointer si e' spostato alla fine del file */
    exit(nwrite);
}
else
{ /* pid > 0 : Processo padre */

    /* Attesa della terminazione del figlio */
    pid_processo = wait(&status);
    /* Con un solo figlio generato, pid_processo è uguale a pid */
    /* Riposizionamento all'inizio del file */
    lseek(fd,0,SEEK_SET);

    if( (nread = read(fd,st2,256)) <0)
        {perror("read "); exit(-4);}

    printf("Il figlio ha letto la stringa %s\n",st2);
    close(fd);
    return(0);
}
exit(0);
}
```

Gestione dei processi

Esecuzione di un programma

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

```
int execve (const char *filename, char *const argv[],  
char *const envp[]);
```

Il processo chiamante passa ad eseguire il programma `filename` (file eseguibile o *script*): non è previsto il ritorno dalla "chiamata" a meno che non vi sia un errore (-1)

Con `argv` e `envp` è possibile specificare gli argomenti e le variabili di ambiente che il programma riceve

L'ambiente di esecuzione (codice e dati) del processo corrente viene modificato in quello del programma (senza che venga creato un nuovo processo - il PID rimane invariato):

- la `fork` crea un nuovo processo identico al padre
- la `exec` permette di modificare l'ambiente di esecuzione di un processo (in modo da renderlo diverso da quello del padre)

Gestione dei processi

Varianti della execve

e: environment

p: path

v: params specificati come vettore

l: params specificati come lista

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

```
int execl( const char *path, const char *arg, ... );
```

```
int execlp( const char *file, const char *arg, ... );
```

```
int execl( const char *path, const char *arg , ...,  
char * const envp[] );
```

```
int execv( const char *path, char *const argv[] );
```

```
int execvp( const char *file, char *const argv[] );
```

La `execlp` e la `execvp` ricercano il `file` nei direttori indicati nella variabile di ambiente `$PATH` (è un comportamento analogo a quello dello shell: non è necessario il path completo)

Gestione dei processi

Effetti delle primitive exec

La gestione dei segnali (vista più avanti) viene alterata:

- se essi venivano **ignorati** rimangono tali
- se erano collegati a **funzioni** (handler) vengono riportati alla gestione di default

Se il file eseguito ha il bit SUID attivo, gli identificatori **effettivi** del processo vengono modificati in quelli del **proprietario del file eseguito**, mentre quelli **reali** rimangono **inalterati**

Si ereditano:

direttorio corrente, maschera dei segnali, terminale di controllo ed altre informazioni

Gestione dei processi

Esempio di uso della execve

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

int main() {
    int status;
    pid_t pid;
    char *env[] = { "TERM=vt100",
                   "PATH=/bin:/usr/bin",
                   (char *) 0 };

    char *args[] = {"cat",
                   "f1",
                   "f2",
                   (char *) 0 };

    if ((pid=fork())==0) {
        /* Codice eseguito dal figlio */
        execve("/bin/cat",args,env);
        /* Si torna qui solo in caso di errore/fallimento della execve*/
        perror("exec");
        exit(-1);
    }
    else {
        /* Codice eseguito dal padre */
        wait(&status);
        printf("Il processo %d e' terminato con %d\n",pid,WEXITSTATUS(status));
    }

    exit(0);
}
```

Gestione dei processi

Stato di zombie

Quando un processo termina (o ucciso da un segnale di *kill* oppure avendo chiamato la `exit()`) e il padre non ha ancora ricevuto la notifica della terminazione del figlio, il processo terminato assume lo stato di *zombie*.

Un processo in stato <zombie> esiste solo nella tabella dei processi e non consuma risorse.

Rimane in modo che il processo padre possa accedere al codice di uscita

Stato di orfano

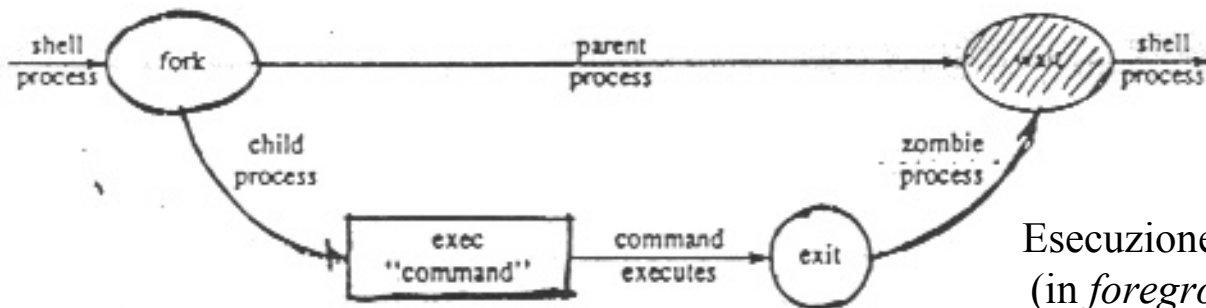
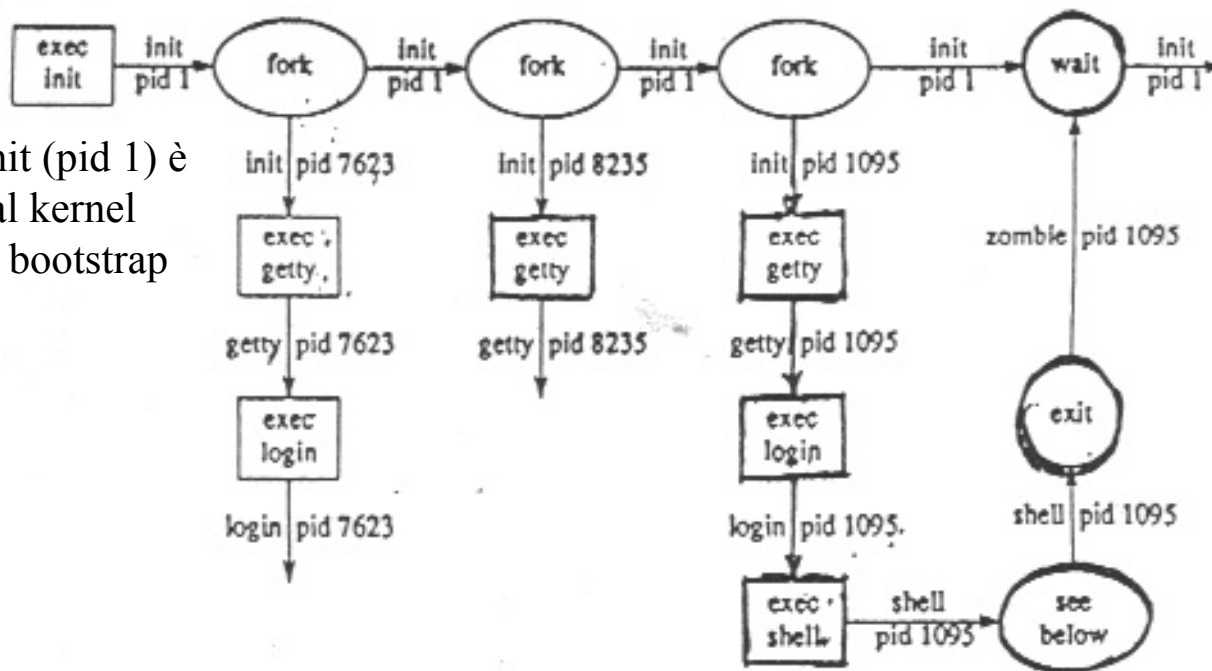
Un processo si dice orfano se il suo processo padre è terminato; diventa quindi figlio di `init(1)`;

Gestione dei processi

La gestione delle sessioni utente

Un processo per ogni dispositivo di linea (terminale) attraverso il quale un utente può fare un login (vedi /etc/inittab)

Il processo init (pid 1) è attivato dal kernel alla fine del bootstrap



Esecuzione di un comando (in foreground) nello shell