Università degli Studi di Cagliari

Sistemi Operativi

Modulo di Laboratorio 10

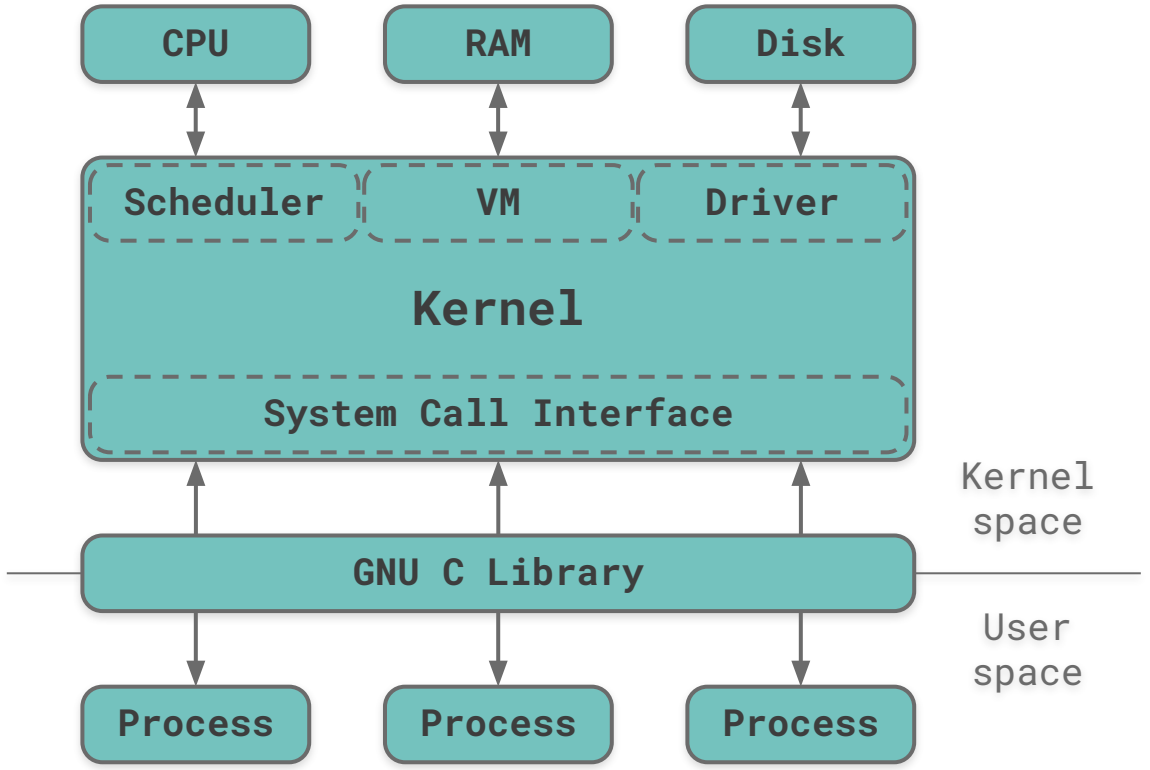
Anno Accademico 2023/2024

01

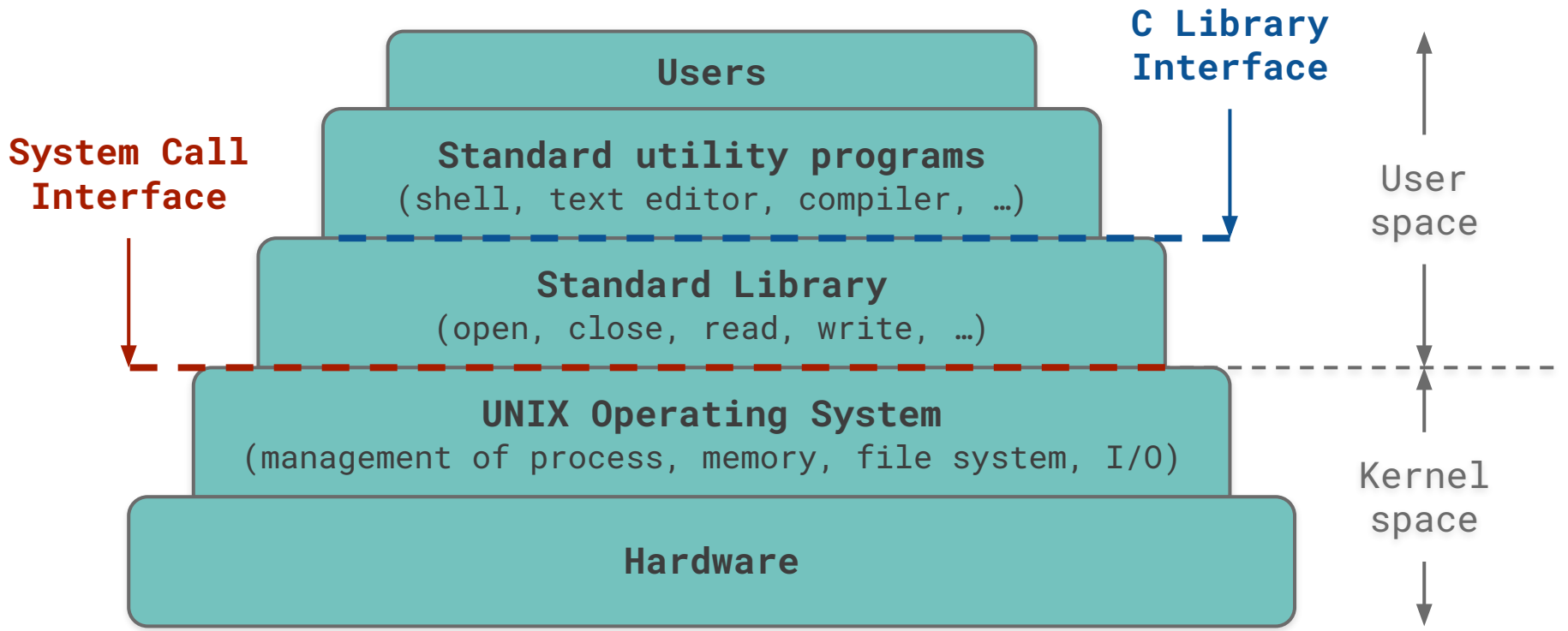
Introduzione alla Programmazione di Sistema

Programmazione di sistema

- Consiste nell'**utilizzo dell'interfaccia di system call** fra il kernel e le applicazioni che girano sotto Unix
- Il kernel è la parte di Linux che corrisponde al sistema operativo vero e proprio e gestisce sia i processi che l'I/O
- Una **system call** è una richiesta fatta da un programma verso il kernel (chiamata al sistema operativo) per richiedere l'esecuzione di un servizio eseguito nel kernel space



Architettura Linux



System calls

- Per un programma C le system call sono equivalenti a funzioni da invocare
  - Ogni system call ha un prototipo, ad esempio:

```c
pid_t fork(void);
```

  - Tipicamente restituiscono un valore negativo (come -1) per indicare un errore

```c
pid_t pid;
pid = fork();
```

- Vi sono system call relative a:
  - Gestione di file
  - Controllo di processi
  - Comunicazione tra processi
  - Invio e ricezione segnali

System calls

- Vediamo alcune chiamate di sistema specifiche per il controllo dei processi

Nome	Funzionalità
`getpid getppid getpgrp …`	Forniscono degli attributi dei processi (PID, PPID, gruppo, ...)
`fork`	Crea un processo figlio duplicando il chiamante
`exec`	Trasforma un processo sovrascrivendo lo spazio di memoria del chiamante con un nuovo programma
`wait`	Permette la sincronizzazione fra processi: il chiamante “attende” la terminazione di un processo figlio
`exit`	Termina un processo, restituendo uno specifico exit status

02

Creazione di processi

Creazione processi mediante fork

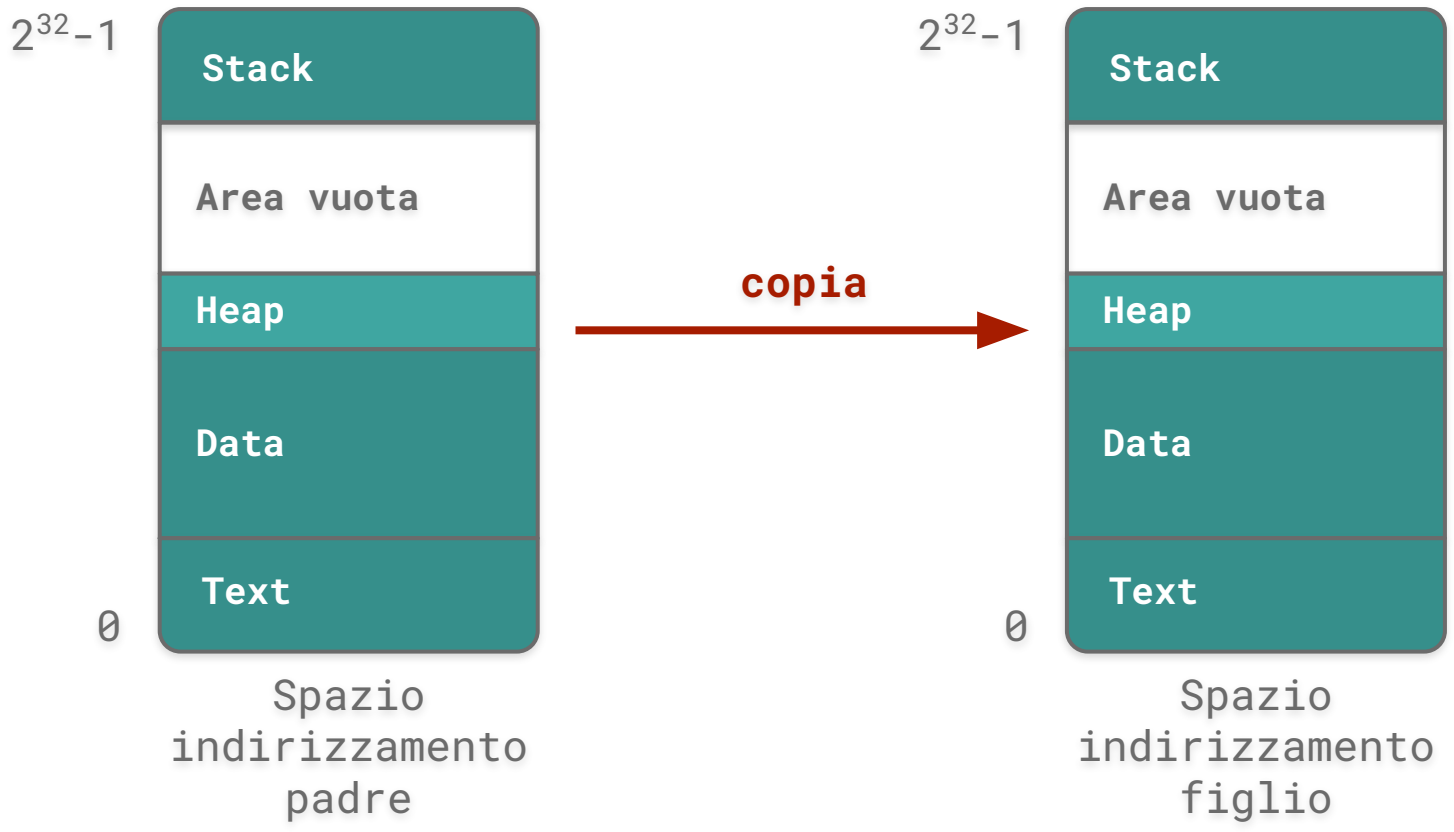
- La funzione **`fork` crea un nuovo processo duplicando il chiamante**
- Il suo prototipo è il seguente:

```
1  pid_t fork(void);
```

- Il tipo `pid_t` è definito negli "include" di sistema ed è tipico equivalga a un tipo intero
- Il valore `pid_t` restituito da `fork` viene usato per distinguere tra **processo padre/genitore** (chiamante) e **processo figlio** (generato dall'esecuzione di `fork`)
  - Al padre viene restituito il PID del figlio, mentre al figlio viene restituito `0`
  - In caso di errore restituisce `-1` al padre, per esempio per tabella dei processi piena
- Lo spazio di indirizzamento del nuovo processo figlio è un duplicato del padre
- Padre e figlio hanno due tabelle dei descrittori di file diverse (il figlio ha una copia di quella del padre), ma condividono la tabella dei file aperti, e quindi anche il puntatore alla locazione corrente di ogni file

Creazione processi mediante fork

- Spazio di indirizzamento di padre e figlio dopo una `fork` terminata con successo



Riepilogo fork

- Dopo l'invocazione di `fork()` il processo padre continua l'esecuzione normalmente, mentre il figlio "inizia" dall'istruzione successiva a `fork()`
  - Padre e figlio si distinguono per il valore restituito dalla `fork()` stessa

- Entrambi i processi hanno un **identico ma separato spazio di indirizzamento**

- Tutte le variabili inizializzate prima di invocare la `fork()` hanno lo stesso valore in entrambi gli spazi di indirizzamento

- Dato che ciascun spazio di indirizzamento è separato, ogni successiva modifica sarà indipendente per ciascun processo

Esempio di utilizzo di fork

```c
#include <stdio.h>
#include <unistd.h>  // Unix Standard Library
#include <sys/types.h>
int main() {
    pid_t pid;
    printf("Processo di partenza con PID %d\n", (int)getpid());
    pid=fork();  // Generazione nuovo processo con duplicazione chiamante
    // Controllo: quale processo sono?
    if(pid == 0)
        printf("Sono il processo figlio, il mio PID e' %d\n", (int)getpid());
    else
        if(pid > 0) {
            printf("Sono il processo genitore:\n");
            printf(" - il mio PID e' %d\n", (int)getpid());
            printf(" - il PID di mio figlio e' %d\n", (int)pid);
        }
        else
            printf("Si e' verificato un errore nella chiamata a fork\n");
}
```

03

Terminazione e attesa dei processi

Terminazione dei processi mediante exit

- La funzione **`exit` termina immediatamente il processo che la invoca**
- Il suo prototipo è il seguente:

```c
void exit(int status);
```

- La sua chiamata inoltre porta a:
  - Chiudere tutti i descrittori di file posseduti dal processo
  - Liberare lo spazio di indirizzamento
  - Inviare il segnale `SIGCHLD` al padre per avvisare della sua terminazione come figlio
  - Salvare il primo byte di status (`0-255`) nella tabella dei processi, in attesa che il padre lo accetti con una `wait()` o `waitpid()`
  - Far adottare a `init` o `systemd` ogni suo processo figlio "orfano" ancora in esecuzione, il cui PPID viene quindi impostato a 1
- Se eseguita nel `main` è equivalente ad una `return`

Attesa terminazione processo figlio

- Le funzioni **`wait`/`waitpid` permettono di attendere la terminazione di un figlio**
- I loro prototipi sono i seguenti:

```c
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- Il processo padre, eseguendo l'istruzione, viene sospeso in attesa della terminazione di uno dei processi figlio (per la `wait`) o di un figlio con dato PID (per la `waitpid`)
- Rendono il PID del figlio che si è atteso, o `-1` in caso di errore (per esempio, no figli)
- Lo stato di terminazione che il figlio ha passato alla `exit` è restituito in `stat_loc`
- È possibile interpretare lo stato di terminazione con alcune macro fornite:

`WIFEXITED(stat_val)`	Rende `True` se il figlio è terminato correttamente
`WEXITSTATUS(stat_val)`	Se `WIFEXITED` è `True`, da accesso al valore restituito da `exit`

Attesa terminazione con wait

- Più nel dettaglio, la **system call `wait`** consente ad un processo di attendere che termini uno dei suoi processi figli, recuperando lo stato passato alla sua `exit`
  - Se il processo chiamante non ha figli, la chiamata fallisce restituendo il codice di errore `-1`, mentre il contenuto di `stat_loc` è da considerare indefinito
  - Se il processo chiamante ha generato processi figli mediante `fork()`, ma non è terminato ancora nessuno di questi, la system call si blocca
  - Se il processo chiamante ha figli già terminati, viene reso il PID del primo dei processi figlio terminati
- I file header da includere ed il prototipo della system call `wait` sono:

```
1  #include <sys/types.h>
2  #include <sys/wait.h>
3  pid_t wait(int *stat_loc);
```

Attesa terminazione con waitpid

- Più nel dettaglio, la **system call `waitpid`** consente ad un processo di attendere che termini un determinato processo figlio, specificato mediante PID come argomento
- Se il processo figlio specificato tramite PID è già terminato (si trova nello stato di zombie) la chiamata a `waitpid` esegue la return immediatamente, liberando tutte le risorse del processo figlio
- I file header da includere ed il prototipo della system call `waitpid` sono:

```c
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Attesa terminazione con waitpid

- Il parametro `pid` di `waitpid` accetta i seguenti valori:

`pid`	Significato
`< -1`	Attende la terminazione dei figli con process group ID uguale al valore assoluto del PID specificato (indicando `-n` si attendono i figli con process group ID pari a `n`)
`-1`	Attende la terminazione di qualunque figlio, operando esattamente come una `wait`
`0`	Attende la terminazione dei figli con process group ID uguale a quello del processo corrente
`> 0`	Attende il processo figlio con il PID specificato

- Sulla base di quanto detto, queste due system call sono quindi equivalenti:

```
1  wait(&status)
2  waitpid(-1, &status, 0);
```

Esempio di attesa di processo figlio

- Consultare il manuale per ulteriori dettagli sulle funzioni `wait` e `waitpid`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(){
    pid_t pid; int status;
    pid = fork();
    if (pid > 0) {  // Siamo nel padre (0=figlio, >0=padre, <0=errore)
        sleep(20);  // Attendiamo 20 secondi
        pid = wait(&status);  // Restituisce il PID del processo completato
        if (WIFEXITED(status))  // !=0 se figlio termina normalmente
            printf("Stato del figlio %d\n", WEXITSTATUS(status));
    } else if (pid == 0) {  // Siamo nel figlio
        printf("Processo %d, figlio.\n", (int)getpid());
        _exit(17);  // Terminiamo con un valore di ritorno (exit status) di 17
    }
}
```

NOTA: la **`exit`** prima di uscire ripristina lo stato del processo (per esempio, chiude file e descrittori aperti), mentre la **`_exit`** esce senza effettuare queste operazioni, modalità da impiegare in un contesto operativo basato su impiego di **`fork`**

Processi zombie

- Un processo si trova nello **stato zombie** nel tempo che passa **tra la sua terminazione e l'invocazione di una una `wait`** che lo coinvolga da parte del padre
- Il programma dell'esempio precedente permette di osservare tale stato zombie
  - Eseguiamo il programma e, dopo la stampa del figlio, sospendiamo con `[CTRL+z]`
    - Il figlio avrà eseguito la `_exit`, ma il padre sarà bloccato dall'invocazione di `sleep`
  - Verifichiamo con il comando **`ps`** lo stato **zombie** (o **defunct**) del figlio

```
$  gcc esempio.c; ./a.out
   Processo 1449, figlio.
   ^Z
   [1]+  Stopped                 ./a.out
$  ps
    PID TTY          TIME CMD
    354 pts/0    00:00:00 bash
   1448 pts/0    00:00:00 a.out
   1449 pts/0    00:00:00 a.out <defunct>
   1450 pts/0    00:00:00 ps
```

Processi zombie

- La `wait` ha lo scopo di recuperare le informazioni sul processo figlio, e consente di terminare correttamente il processo figlio
  - Il figlio rimane nello stato zombie, finchè la `wait` non viene chiamata
- Un processo zombie non consuma memoria, ma un numero elevato di essi potrebbe portare all'impossibilità di assegnare nuovi PID, saturando la tabella dei processi, e quindi rendendo impossibile la creazione di nuovi processi
- Se il padre muore prima di poter invocare la `wait` sul processo figlio, quest'ultimo viene adottato dal processo `init` o `systemd`, che si occuperà di effettuare la `wait`

04

Differenziazione di processi

Differenziazione dei processi

- Se `fork` fosse l'unico modo per creare nuovi processi la programmazione in Linux risulterebbe alquanto ostica, potendo creare soltanto copie dello stesso processo
- La famiglia di primitive **exec fornisce varie system call che consentono di avviare un nuovo programma** sovrascrivendo la memoria di un processo già esistente
  - Alcune di queste sono `execl`, `execle`, `execlp`, `execv`, `execvp` e `execve`
    - Tutte richiamano in realtà `execve` (seppur in modo diverso) che rappresenta la principale system call della famiglia
  - Si distinguono per i parametri accettati, e ogni carattere dopo `exec` ha un significato:
    - `l` indica che accetta una lista di argomenti
    - `v` indica che accetta un array di puntatori di riferimento agli argomenti
    - `e` indica che accetta un array di puntatori a variabili di ambiente
    - `p` indica che accetta il solo nome del programma, cercandolo in automatico in PATH
  - In assenza di errori queste system call non restituiscono nulla, altrimenti rendono `-1`

Differenziazione dei processi: execl

- In quanto “funzionalmente equivalenti”, approfondiamo la sola primitiva `execl`
- L’invocazione di `execl` porta a sovrascrivere il programma originale con un nuovo programma di cui si specifica il path nel file system
- Le istruzioni che seguono l’invocazione di `execl` sono eseguite solo in caso di errore durante la chiamata ad `execl`, poiché altrimenti sovrascritte dal nuovo programma
- Il prototipo della funzione in questione è il seguente:

```
1  int execl(char *pathname, char *arg0, ...);
```

  - Il `pathname` è appunto il percorso assoluto del programma da eseguire
  - Segue una lista di variabili che contengono un puntatore ad una stringa di caratteri
    - Queste variabili sono i valori da passare ad `argv` nel nuovo programma eseguito
    - Tale lista deve terminare con un puntatore di tipo `NULL`

Esempio di utilizzo di execl

- Per esempio, il programma C seguente richiama l'esecuzione del comando `ls -l`

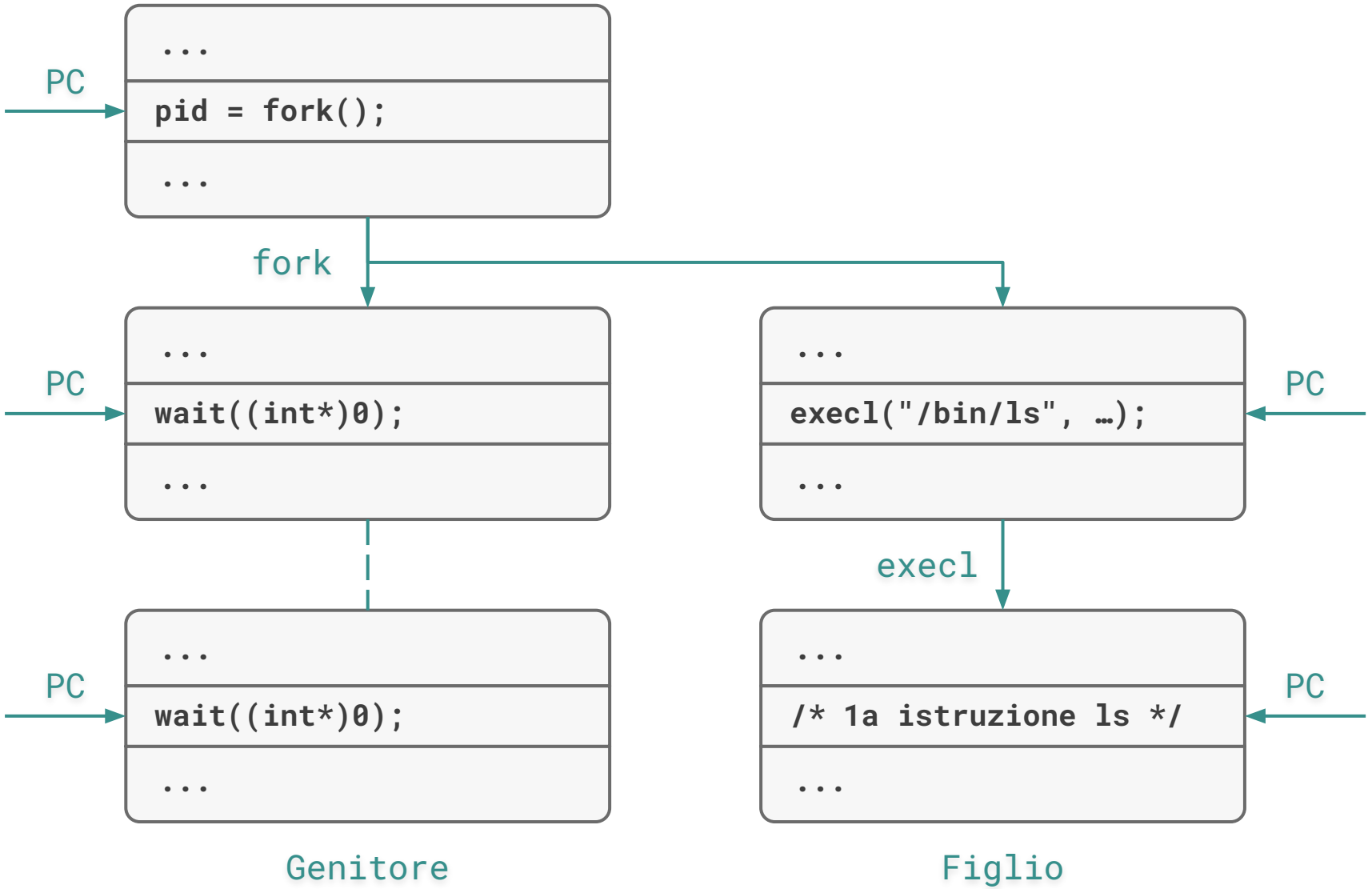
```c
#include <stdio.h>
#include <unistd.h>
int main(){
    printf("Esecuzione di ls -l\n");
    execl("/bin/ls", "-l", NULL);
    perror("execl ha generato un errore\n");
    _exit(1);
}
```

- L'esecuzione di `execl` modifica il programma in esecuzione e il suo stato:

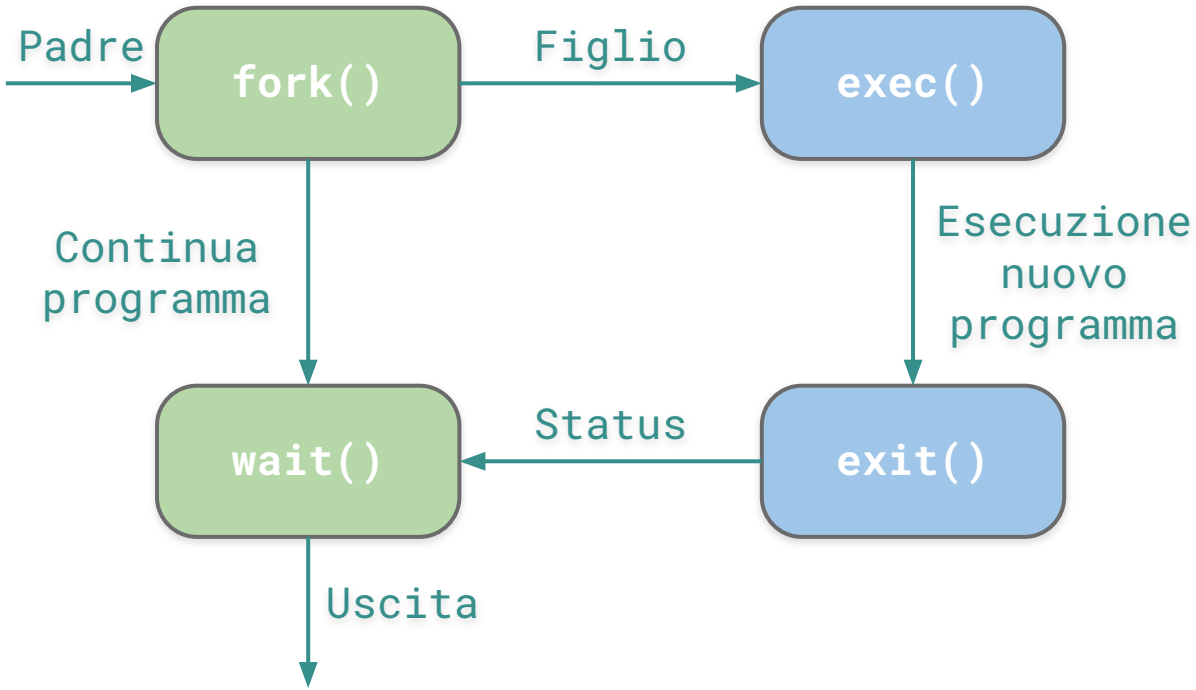
Creazione di nuovi processi con fork ed exec

- L'utilizzo **combinato** di `fork` (per creare un nuovo processo) e di `exec` (per far eseguire al figlio un nuovo programma) è un potente strumento in ambiente Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main(){
    pid_t pid;
    pid = fork();
    if (pid < 0)
        printf("fork failed");
    else if (pid == 0) {  // Figlio
        execl("/bin/ls", "-l", NULL);
        printf("exec failed");  // Di norma non eseguito
    } else {  // Padre
        wait((int *)0);
        fflush(stdin);
        printf("ls completed\n");
        exit(0);
    }
}
```


...
PC
pid = fork();
...
fork
...
PC
wait((int*)0);
...
...
PC
wait((int*)0);
...
...
execl("/bin/ls", …);
...
PC
execl
...
/* 1a istruzione ls */
...
PC
Genitore
Figlio

Creazione di nuovi processi con fork ed exec



Fine Modulo 10