Esercitazione:
System call per la gestione dei processi

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2023/2024, Canale San Giovanni

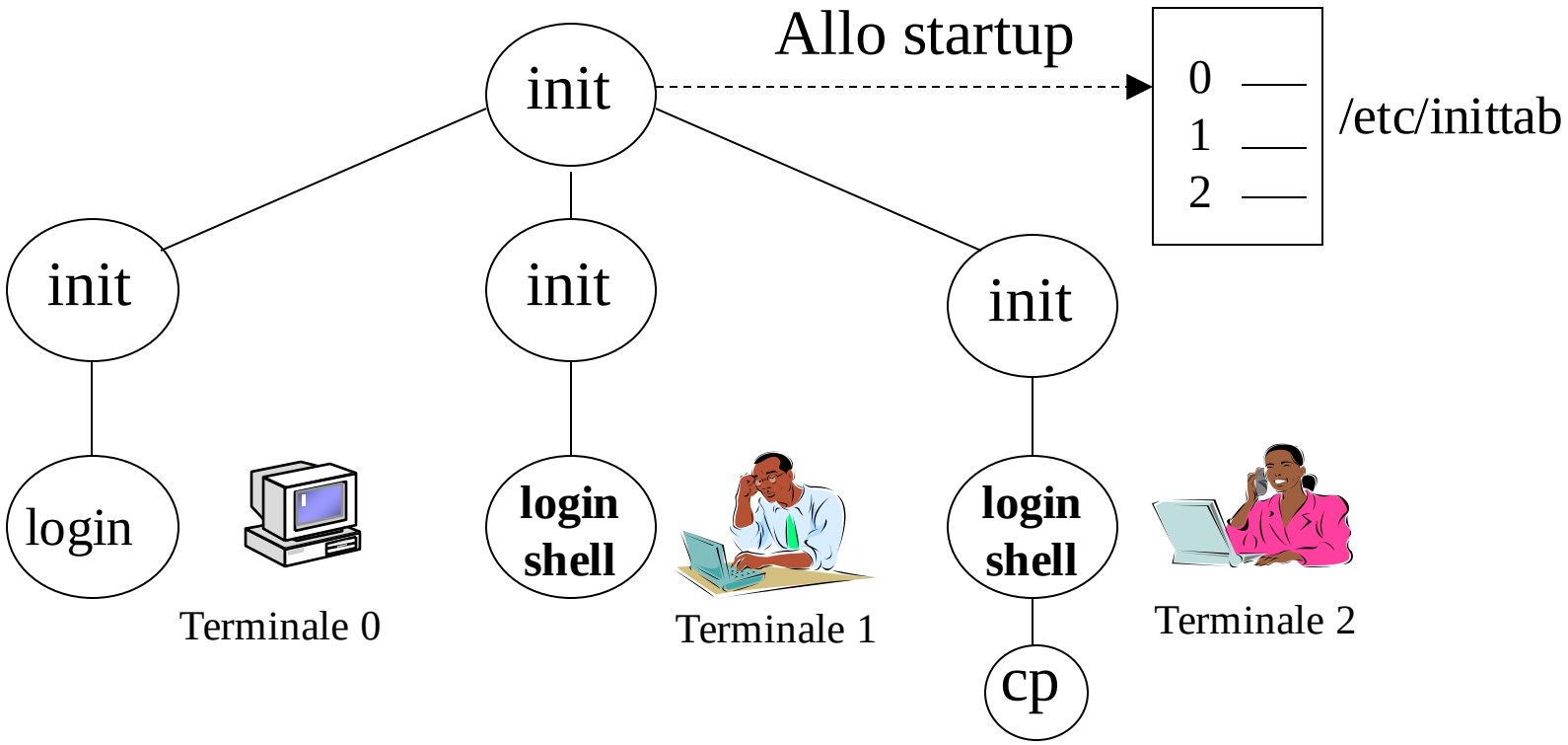
Sommario

- Chiamate di sistema UNIX per la gestione dei processi:
  - getpid()
  - sleep()
  - fork()
  - exec()
  - wait()
  - exit()
- Riferimenti
  - Ancillotti, Boari, Ciampolini, Lipari, "*Sistemi Operativi*", 2a ed., Capitolo 8
  - www.ostep.org, Cap. 5 ("Process API")

I processi in Unix

- I processi in Unix sono organizzati gerarchicamente
- Un processo può solo essere creato da altri processi (chiamata di sistema *fork()*)
- Fa eccezione il processo *init* , che viene creato all'avvio dal kernel

Genealogia dei processi



NB:

- al terminale 0 non è presente nessun utente
- al terminale 1, un utente ha appena terminato la fase di login
- al terminale 2, un utente è già entrato nel sistema e sta eseguendo una copia di file

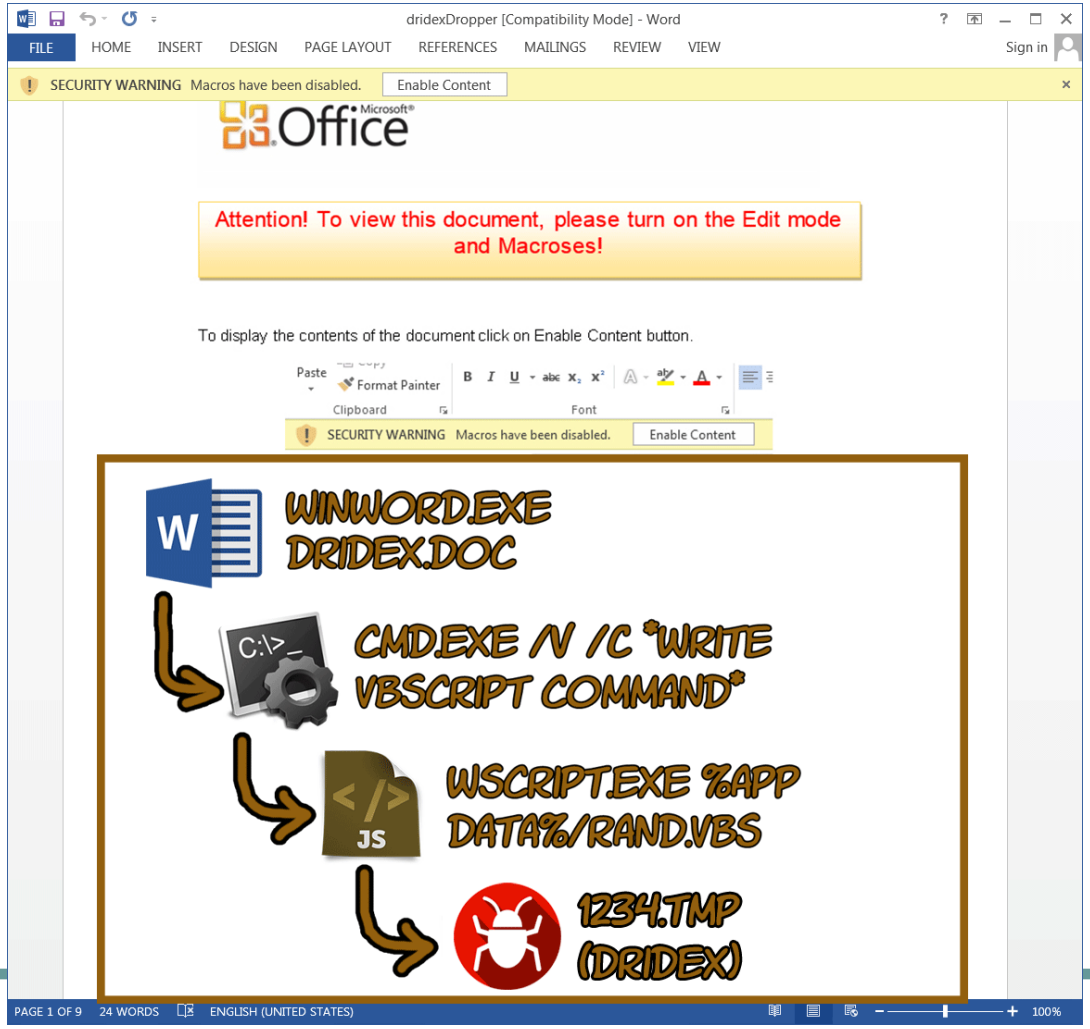
Genealogia dei processi

- *init* legge il file */etc/inittab*, per determinare il numero di terminali del sistema da attivare
- Per ogni terminale, *init* genera un processo figlio che esegue il programma */bin/login*
- Quando l'utente inserisce il suo nome e password, il programma *login* li verifica consultando il file */etc/passwd*
- In caso affermativo, esegue il programma *shell*, ossia l'interprete dei comandi

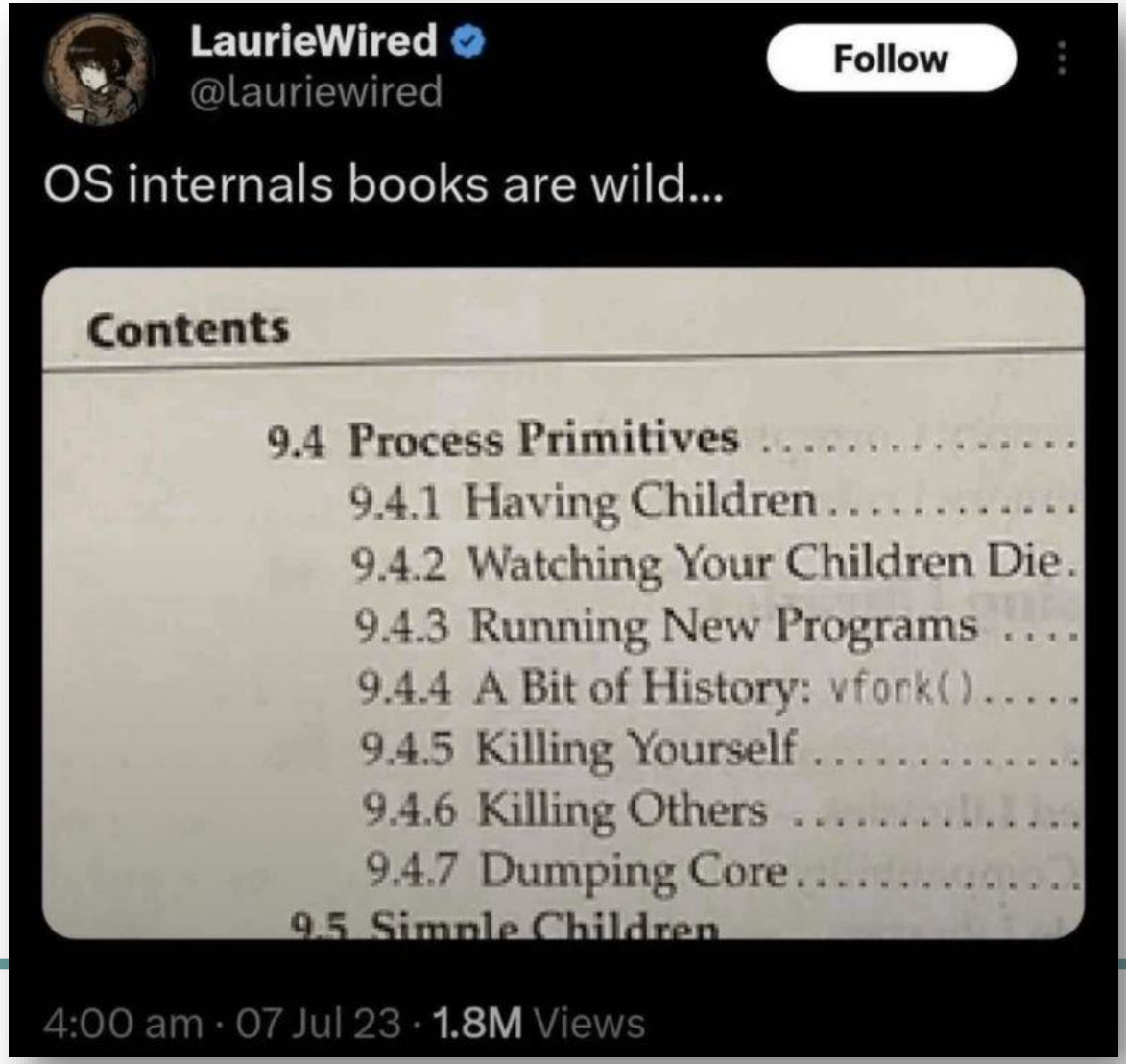
pstree

```
$ pstree

systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
                  ...
        ├─sshd
        ├─systemd─┬─(sd-pam)
        |         ├─at-spi2-registr───2*[{at-spi2-registr}]
        |         ├─dbus-daemon
        |         ├─dconf-service───2*[{dconf-service}]
       ...       ...
        |         ├─gnome-terminal-─┬─bash───pstree
        |         |                 ├─bash───firefox─┬─Privileged Cont───19*[{Privileged Cont}]
        |         |                 |                ├─Socket Process───5*[{Socket Process}]
        |         |                 |                ├─3*[Web Content───12*[{Web Content}]]
        |         |                 |                ├─WebExtensions───18*[{WebExtensions}]
        |         |                 |                ├─xdg-settings─┬─cut
        |         |                 |                |              └─dbus-send
        |         |                 |                └─86*[{firefox}]
        |         |                 └─3*[{gnome-terminal-}]
       ...       ...
```

Genealogia dei processi nel malware



Primitive per gestione processi



ID del processo (PID)

- Ogni processo ha un identificatore univoco (Process Identifier, PID)
  - intero tra 0 e 32768 (nei sistemi a 32bit)
  - assegnato dal kernel al momento della sua creazione

- Un processo può consultare il proprio pid attraverso la chiamata di sistema:

```
int getpid();
```

nella documentazione e negli header di sistema UNIX, il tipo "`int`" è sostituito da typedef "`pid_t`", "`size_t`", etc., per portabilità tra diverse CPU

Organizzazione Gerarchica

- Ogni processo ha un processo padre (eccetto il processo *init*) con il relativo pid
- Un processo ottiene il pid del padre attraverso la chiamata di sistema :

```
int getppid();
```

getpid() e getppid() - Esempio

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

int main(void)
{
	pid_t pid, ppid;

	pid = getpid();
	printf("Sono il processo pid = %d\n", pid);

	ppid = getppid();
	printf("Il mio processo genitore ha pid = %d\n", ppid);

	return 0;
}
```

Funzione: sleep

- Sospende (transizione stato *Blocked*) un processo per un certo numero di secondi

```
unsigned int sleep(int sec);
```

System Call: *fork*

- La creazione di nuovi processi avviene mediante:

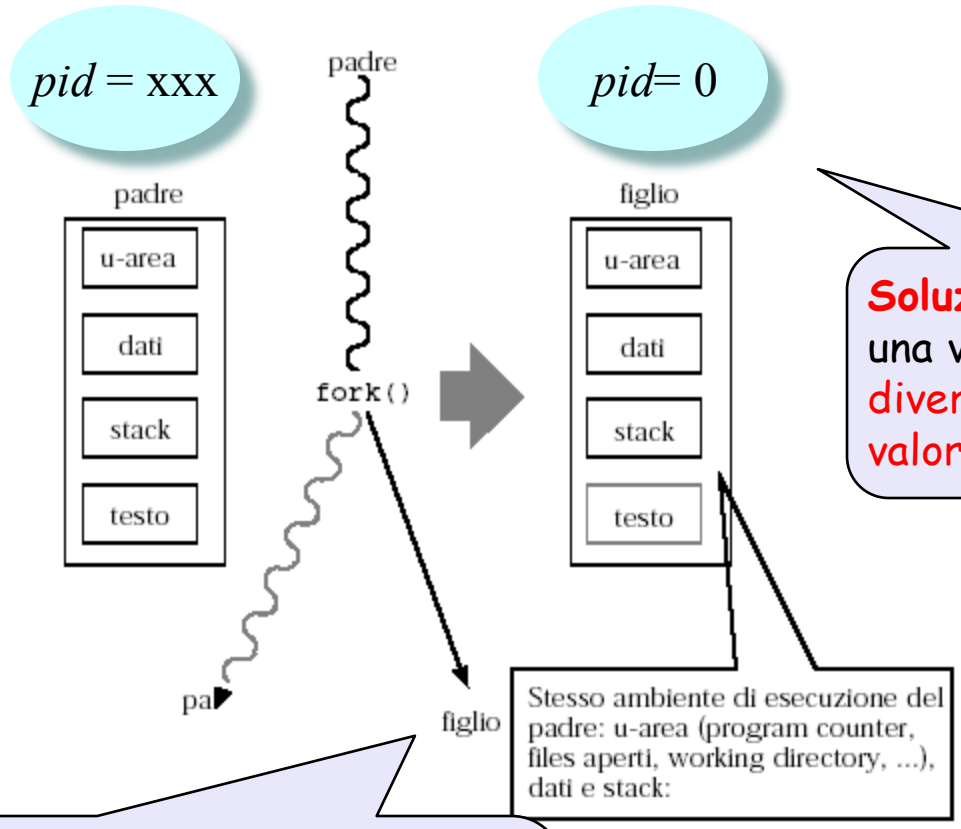
```
int fork(void);
```

- Crea una **copia esatta** (***figlio***) del processo chiamante (***padre***)
- Stesso codice, stessi dati
  - inclusi i registri (es. Program Counter), stack, heap, dati globali

I processi padre e figlio sono identici, ma le loro risorse sono delle **copie distinte (non sono condivise)**.

Le modifiche ai dati in uno dei due processi non sono visibili all'altro processo.

System Call: *fork*



Soluzione: il SO predispone una variabile che ha valore diverso nei due processi (il valore di ritorno di fork())

Problema: tipicamente, in un programma concorrente vogliamo che il processo figlio esegua una attività diversa dal padre.

Se il figlio è un "clone" identico, come differenziare?

System Call: *fork*

Entrambi i processi hanno una variabile "pid", ma con valori diversi

Sia il padre sia figlio eseguono a partire da questo punto

```
pid = fork();

if (pid > 0) {
        /* codice eseguito
           dal padre */

        wait();
}
else if (pid == 0) {
        /* codice eseguito
           dal figlio */
}
```

- Il Program Counter (PC) è uguale per padre e figlio
- Entrambi eseguono dallo stesso punto in cui è stata chiamata fork()
- fork() ritorna un valore (intero) che differenzia il padre dal figlio

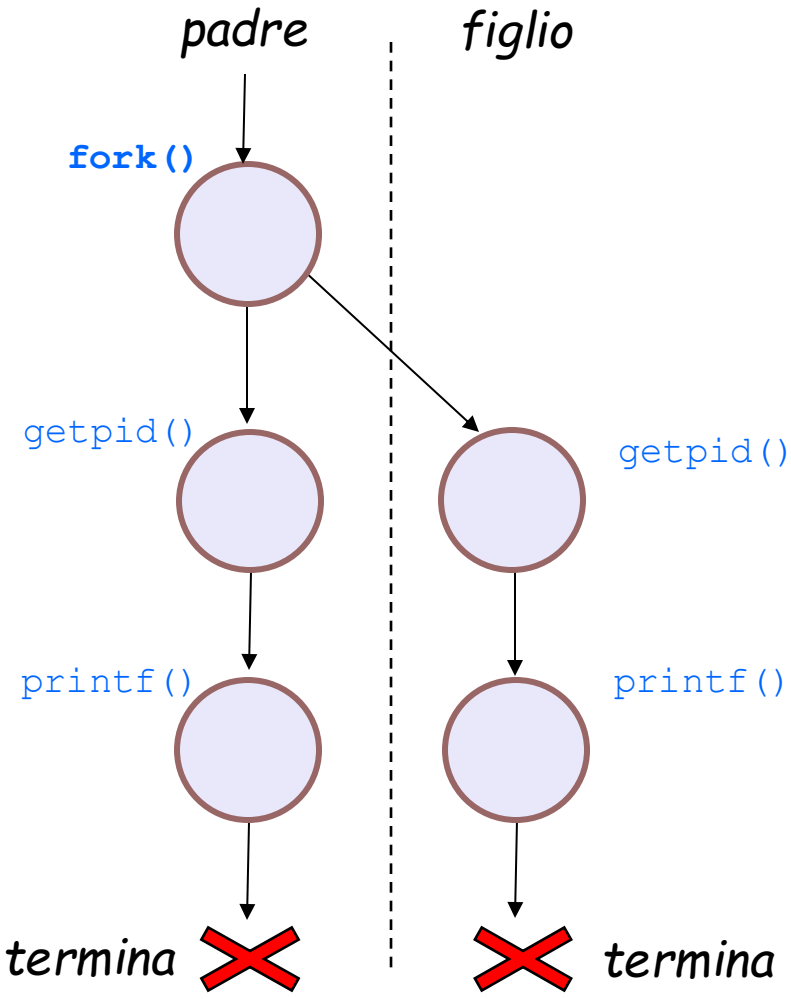
System Call: *fork*

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

int main(void) {

        pid_t pid;

        pid = fork();

        if(pid > 0) {

                pid_t mypid = getpid();
                printf("Sono il padre, il mio PID è %d\n", mypid);

        } else if(pid == 0) {

                pid_t mypid = getpid();
                printf("Sono il figlio, il mio PID è %d\n", mypid);
        }

        return 0;
}
```

System Call: *fork*

Nota: si può verificare che il **padre stampi il messaggio prima del figlio**, o **viceversa**.

Dipende dallo **scheduler del SO**.



System Call: *fork*

- fork() è una *system call potenzialmente molto onerosa* !
  - Allocazione memoria per il processo figlio
  - Copia della memoria e dei registri del processo padre su quello figlio

Gestione degli errori

- In caso di errori, le chiamate di sistema possono terminare con un insuccesso
  - Memoria esaurita
  - Raggiunti i limiti di sistema (es. n° max di processi)
  - Mancanza di permessi
  - Risorsa non trovata
  - ...

Gestione degli errori

```
pid = fork();

if (pid > 0) {
        /* codice eseguito
           dal padre */

        wait();
}
else if (pid == 0) {
        /* codice eseguito
           dal figlio */
}
else if (pid < 0) {
        /* chiamata fallita */
        perror("FORK FALLITA");
        exit(1);
}
```

- Per gestire gli errori, si controlla il valore di ritorno (se negativo, indica un fallimento)
- Usare perror("...") per stampare un messaggio
- perror() aggiunge informazioni sull'errore (in base alla variabile globale di sistema errno)

Gestione degli errori

dal manuale della system call fork()

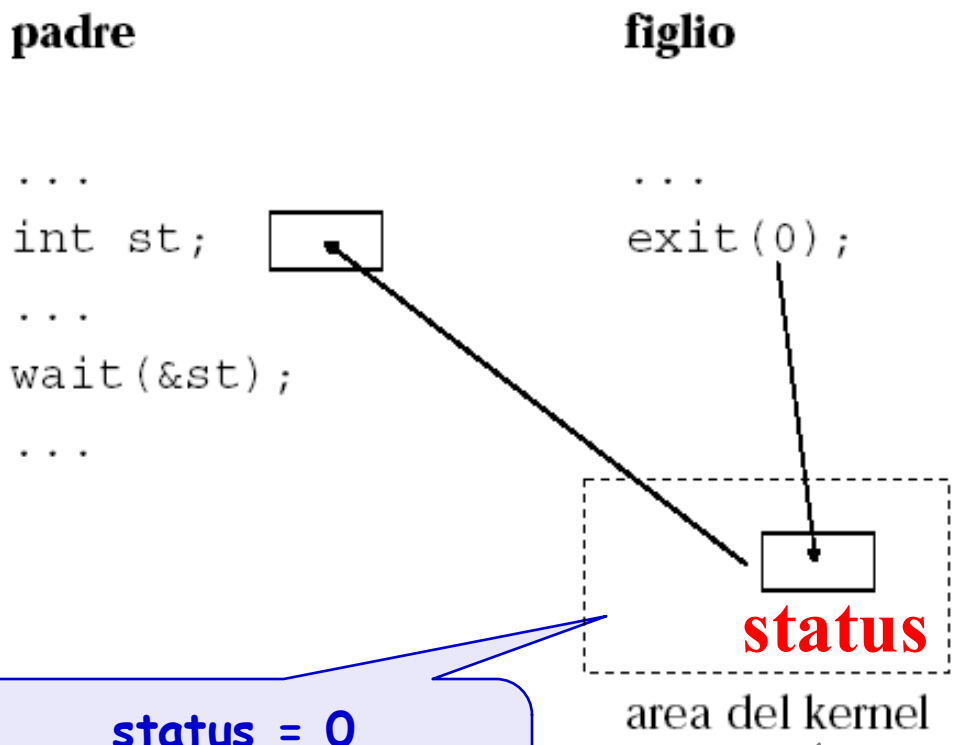
```
RETURN VALUES
     Upon successful completion, fork() returns a value of 0 to the child process and returns the
     process ID of the child process to the parent process.  Otherwise, a value of -1 is returned
     to the parent process, no child process is created, and the global variable errno is set to
     indicate the error.

ERRORS
     fork() will fail and no child process will be created if:

     [EAGAIN]           The system-imposed limit on the total number of processes under execution
                        would be exceeded.  This limit is configuration-dependent.

     [EAGAIN]           The system-imposed limit MAXUPRC (<sys/param.h>) on the total number of
                        processes under execution by a single user would be exceeded.

     [ENOMEM]           There is insufficient swap space for the new process.
:
```

```
int main() {
 while(1){
   pid_t pid = fork();
   if(pid < 0) {
     perror("Fork fallita");
     exit(1);
   }
 }
}
```

fork bomb

```
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
Fork fallita: Resource temporarily unavailable
```

Wait ed Exit



- Il processo padre attende la terminazione del figlio (operazione di "join") tramite la system call wait()
- Il figlio tramite exit() deposita un valore numerico "status", che indica l'esito della sua esecuzione

status = 0
l'esecuzione è avvenuta correttamente

status ≠ 0
indica una anomalia

System Call wait

- Consente al padre di raccogliere lo stato di terminazione di un processo figlio

```
int wait(int *stato)
```

- Se nessun figlio è ancora terminato, il kernel **sospende il processo padre**
- La variabile ***stato*** (in uscita) conterrà il valore passato dal processo figlio alla system call ***exit***
- Se vi sono più figli, il processo si sblocca al primo figlio che termina (qualunque esso sia)

System Call exit

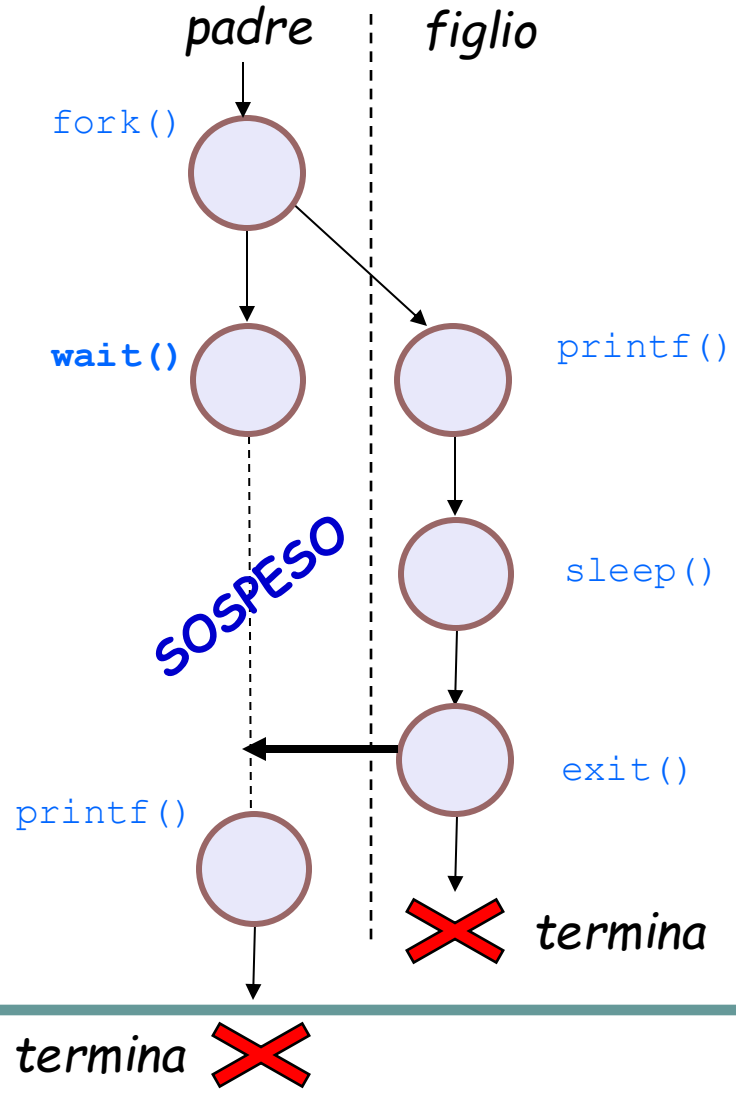
- Un processo termina con la chiamata di sistema *exit*
- Quando *exit* viene invocata, viene passato uno stato di uscita numerico intero dal processo al kernel
- Tale valore è disponibile al processo padre attraverso la chiamata di sistema *wait*
- Un processo che termina normalmente restituisce uno stato di uscita 0

Wait e Exit: esempio

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

int main() {

    pid_t pid;

    pid = fork();

    if (pid == 0) {
      printf("Figlio\n");
      sleep(10);

      exit(0);
    }
    else if (pid > 0) {

      wait(NULL);  // NULL "ignora" lo status di terminazione

      printf("Il figlio ha terminato.\n");
    }
}
```

System Call: *wait*

Con wait(), il padre **termina sempre dopo il figlio**, indipendentemente dallo scheduler del SO.



Wait e Exit: esempio

```
...
else if (pid > 0) {
  int status;
  pid_t pid = wait(&status);

  if ( WIFEXITED(status) ) {
      printf("Terminazione volontaria di %d\n", pid);
      printf("Stato di uscita: %d\n", WEXITSTATUS(status) );
  }
  else if ( WIFSIGNALED(status) ) {
      printf("Terminazione involontaria\n");
      printf("Segnale: %d\n", WTERMSIG(status) );
  }
}
```

Il processo padre può utilizzare le seguenti MACRO:

- ***WIFEXITED(status)*** restituisce vero se il figlio è terminato volontariamente: in questo caso, ***WEXITSTATUS(status)*** restituisce lo stato di terminazione.
- ***WIFSIGNALED(status)*** restituisce vero, se il figlio è terminato involontariamente: in questo caso, ***WTERMSIG(status)*** restituisce il numero del segnale UNIX che ha causato la terminazione.

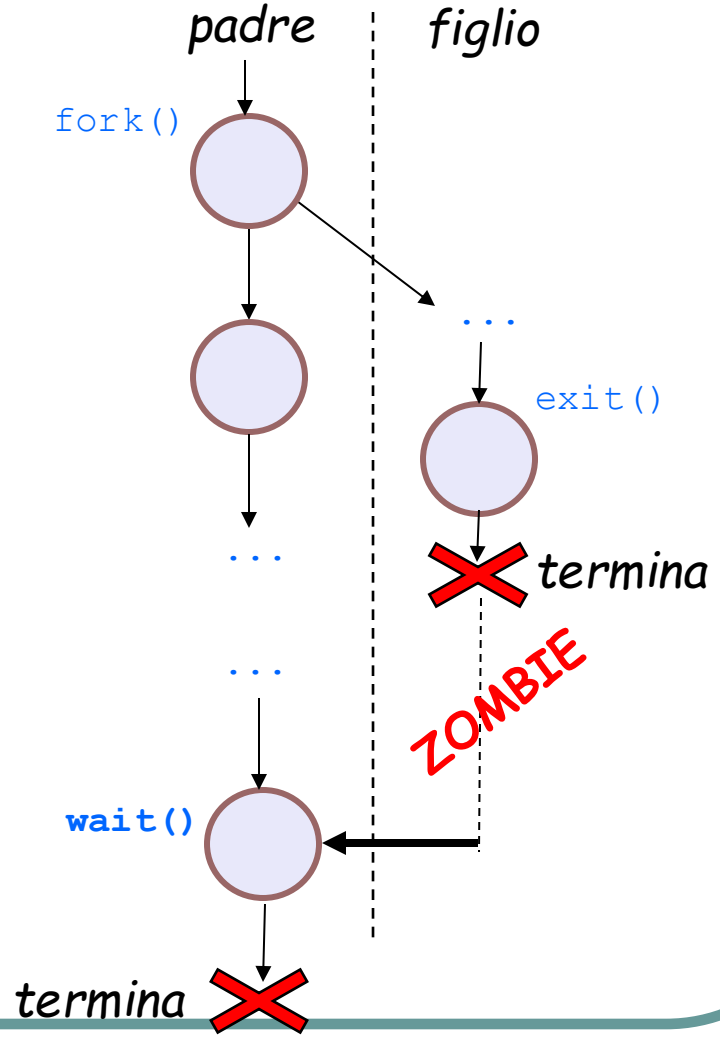
System Call waitpid

- Variante della wait, consente al programmatore di indicare il PID di uno specifico processo figlio di cui attendere la terminazione

```
int waitpid ( pid_t pid, int *stato, int options )
```

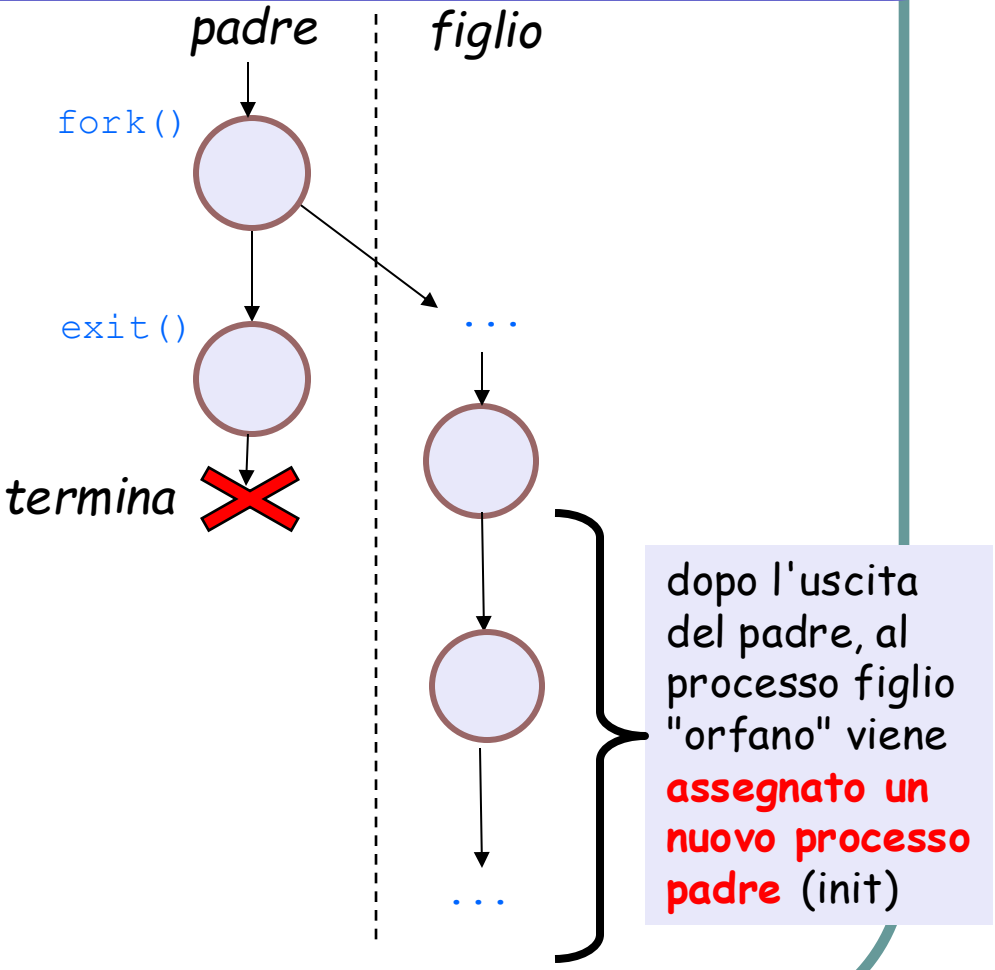
Casi particolari: Processi Zombie

- Il processo figlio termina **prima che il padre si metta in attesa** (*wait*) della sua terminazione
  - Il processo figlio terminato è detto ***processo zombie***
  - Il kernel rilascia tutte le risorse tranne il suo stato di terminazione, per dare la possibilità di ricongiungersi con il padre



Casi particolari: Processi Orfani

- Il processo genitore termina prima dei suoi processi figli (*attivi* o *zombie*) tali processi sono detti ***orfani***
  - Il kernel assegna come parent ID il valore 1, cioè diventano figli del processo *init* (non termina mai)
  - I figli non sono consapevoli della terminazione del processo padre



System Call *fork* – Creazione di *n* Processi

Supponiamo di voler creare 5 processi

Soluzione 1:

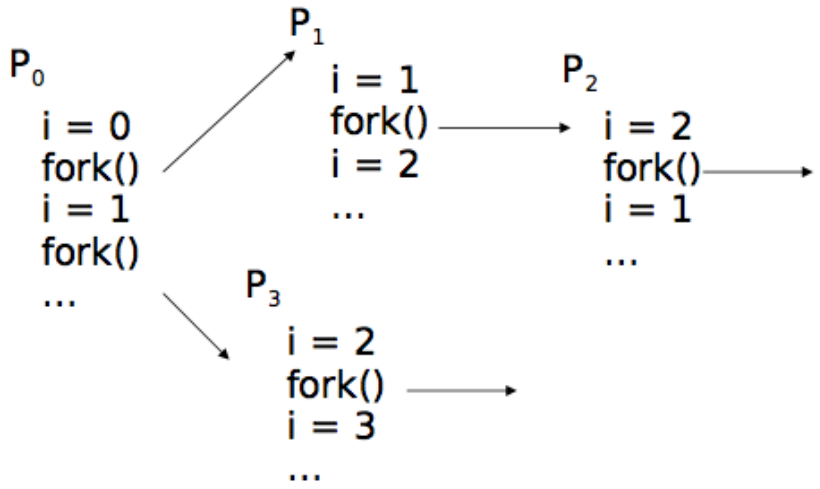
```
for (int i=0; i<5; i++) {
    pid = fork();

    if(pid == 0) {
       /* codice del figlio */
    }
}
```

System Call *fork* – Creazione di *n* Processi

Supponiamo di voler creare 5 processi

Soluzione 1:

```
for (int i=0; i<5; i++) {
    pid = fork();

    if(pid == 0) {
       /* codice del figlio */
    }
}
```



Numero totale di processi:

1+2+4+8+16 = 31

$$\sum_{i=0}^{4} 2^i$$

System Call *fork* – Creazione di *n* Processi

Supponiamo di voler creare 5 processi

Soluzione 2 (corretta):

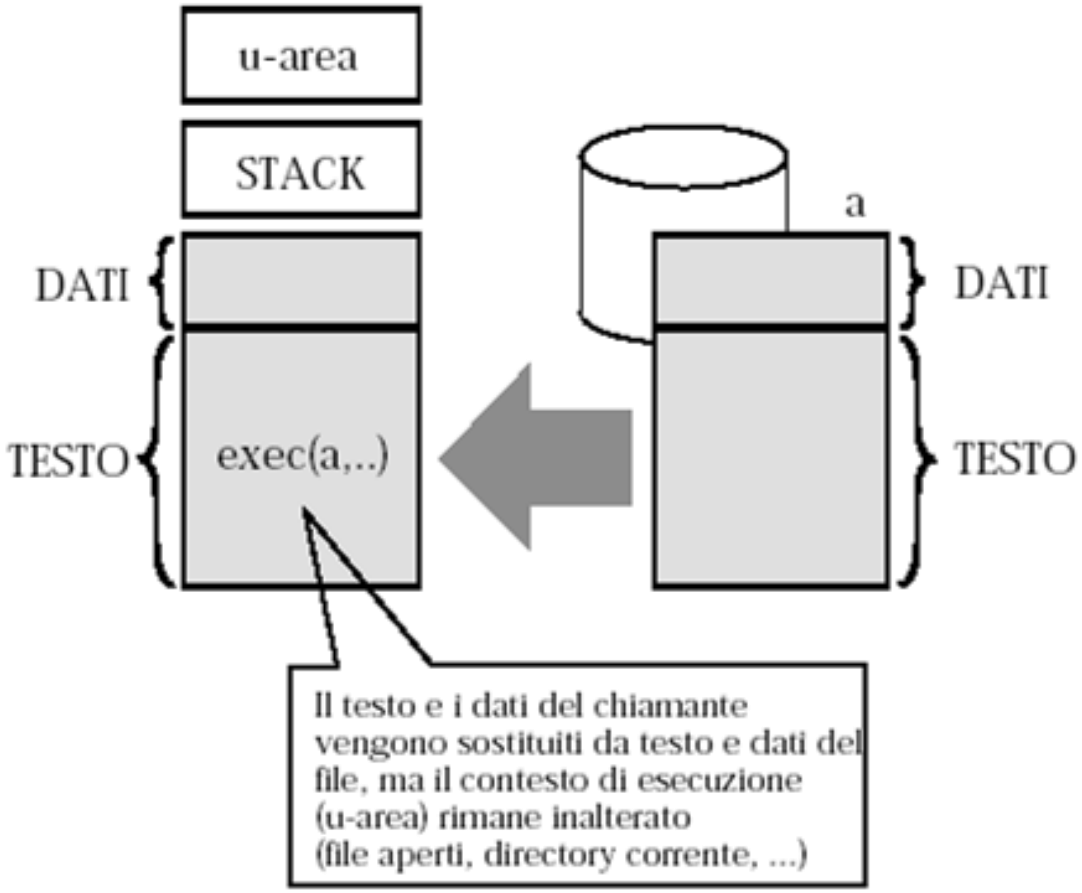
```
for (int i=0; i<5; i++) {

    pid = fork();

    if(pid == 0) {

       /* codice del figlio */
       exit(0);
    }
}
```

System Call: *exec*

fork() permette di creare nuovi processi che eseguono lo **stesso programma del chiamante**. Da sola, questa system call **non consente** di eseguire altri programmi!

- Per eseguire un nuovo programma, occorre:
  - che un processo (es. la shell) crei un nuovo processo, tramite fork()
  - che il nuovo processo esegua il programma desiderato, chiamando la system call **exec()**
  - Il controllo passa al nuovo programma, senza mai tornare al programma chiamante (eccetto in caso di errore)

System Call: *exec*



Il nuovo programma viene eseguito nel contesto del processo che chiama exec() (cioè il pid non cambia)

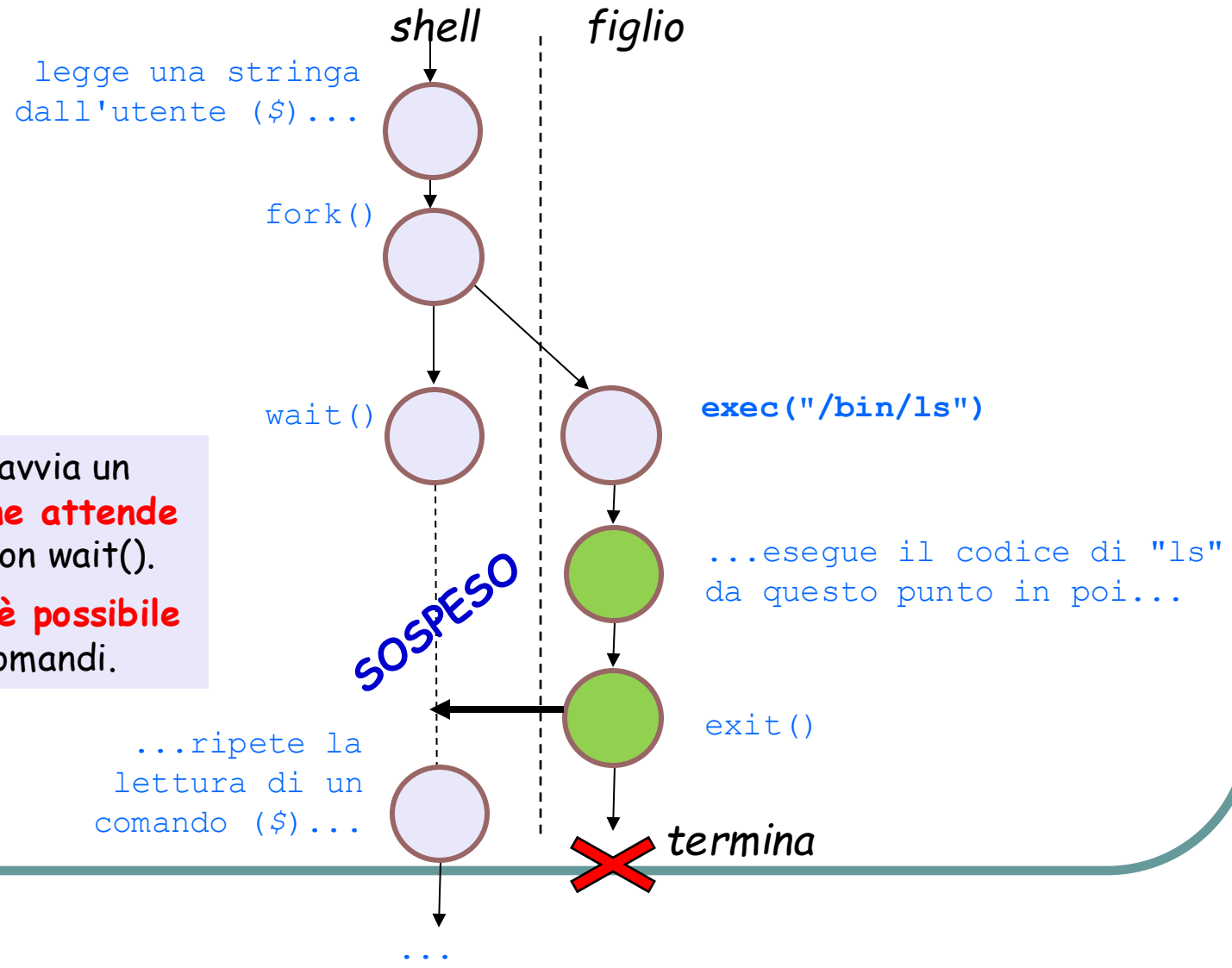
System Call: *exec*

- Il processo dopo l'exec:
  - mantiene lo stesso process control block
  - mantiene la stessa user area (a parte PC e informazioni legate al programma)
  - mantiene lo stack nel kernel
  - ha codice, dati globali, stack e heap nuovi
  - riferisce una nuova area codice (text)

System Call: *exec*

```
pid = fork();

if (pid == 0) {

        // codice figlio
        …
        if (execlp("program",…)<0){
            perror("exec fallita");
            exit(1);
        }

        // il processo figlio non
        // esegue mai in questo punto
}
// Il padre continua da questo
// punto in poi…
```

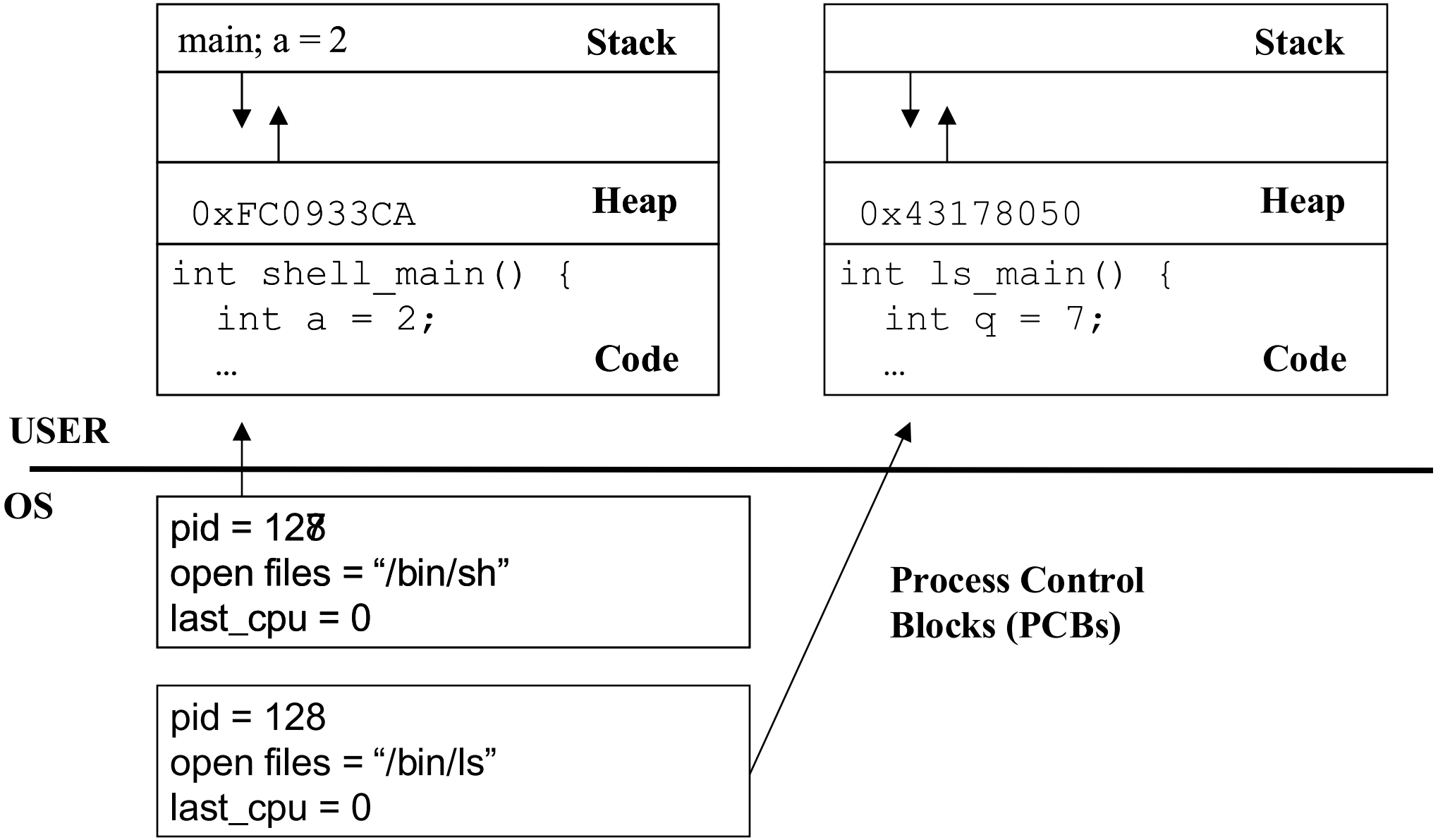
Esempio: uso di fork ed exec nella shell

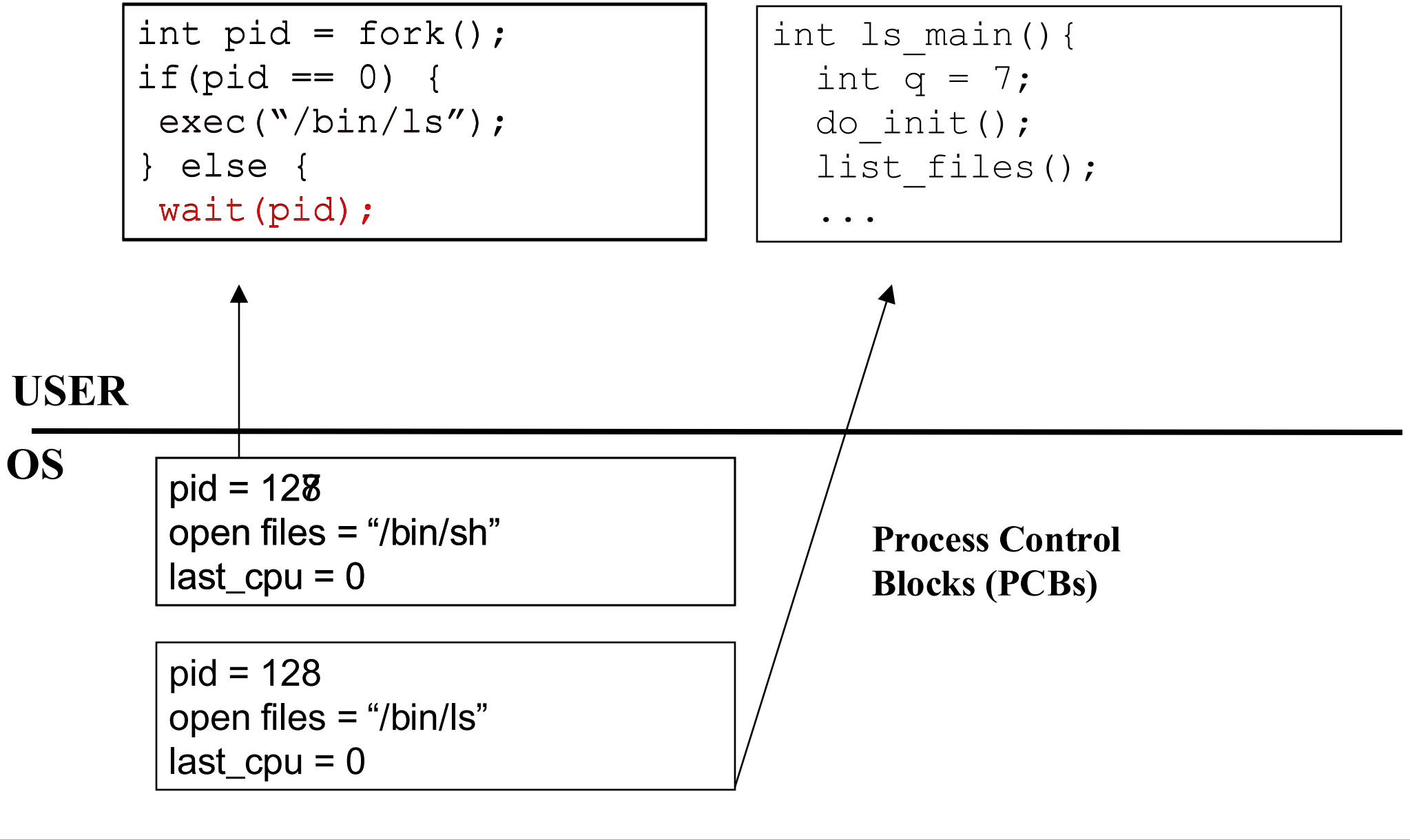


Nota: quando si avvia un comando, **la shell ne attende la terminazione** con wait().

Fino ad allora, **non è possibile digitare** altri comandi.

Una shell fa la “fork” ed esegue **ls** con “exec”




int pid = fork();
if(pid == 0) {
exec("/bin/ls");
} else {
wait(pid);
int ls_main(){
int q = 7;
do_init();
list_files();
...
USER
OS
pid = 128
open files = "/bin/sh"
last_cpu = 0
pid = 128
open files = "/bin/ls"
last_cpu = 0
Process Control Blocks (PCBs)

System Call: *exec*

Esistono più versioni della *exec* in Linux:

Percorso completo dell'eseguibile, parametri tramite lista

*int execl(char *path, char *arg0, char *arg1, ..., (char *) 0);*

Nome dell'eseguibile (da cercare nelle cartelle di sistema), parametri tramite lista

*int execlp(char *nomefile, char *arg0, char *arg1, ..., (char *) 0);*

Percorso completo dell'eseguibile, parametri tramite array

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

Nome dell'eseguibile (da cercare nelle cartelle di sistema), parametri tramite array

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

Exec: esempio (execl)

```
int main() {

        pid_t pid = fork();

        if (pid == 0) {

                execl("/bin/ls", "ls", "-l", NULL);

                /* Se exec() va a buon fine, il codice da questo
                 * punto non verrà mai eseguito */

                perror("Exec fallita!!\n");
                exit(1);
        }
        else if (pid > 0) {

                wait(NULL);
                printf("ls completato\n");
        }
}
```

Eseguibile

In UNIX, è convenzione che il 1° parametro coincida con il nome del programma

Exec: esempio (execv)

```
int main() {

        pid_t pid = fork();

        if (pid == 0) {

                char * parametri[3];

                parametri[0] = "ls";
                parametri[1] = "-l";
                parametri[2] = NULL;   // indica la fine del vettore

                execv("/bin/ls", parametri);

                perror("Exec fallita!!\n");
                exit(1);
        }
        else if (pid > 0) {

                wait(NULL);
                printf("ls completato\n");
        }
}
```

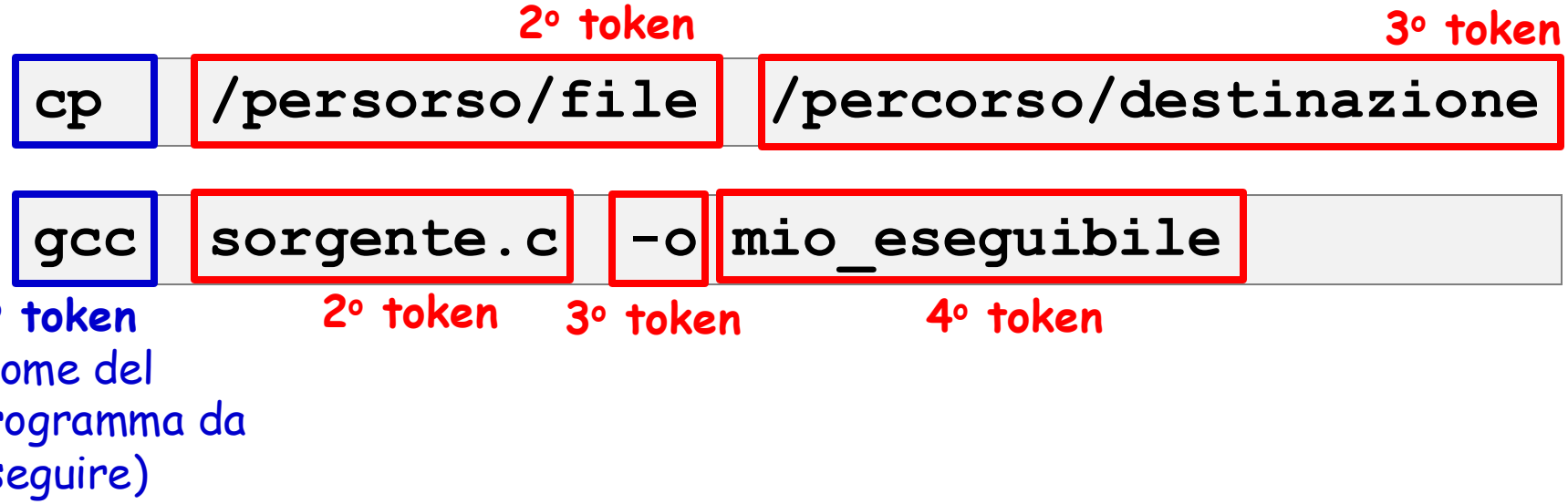
argc/argv

- Un programma riceve i valori di ingresso dalla exec() tramite argc e argv nel main
  - argc: il numero di parametri di ingresso
  - argv: vettore di stringhe

```
int main(int argc, char * argv[]) {

        for(int i = 0; i<argc; i++) {
                printf("Parametro [%d] = %s\n", i, argv[i]);
        }
}
```

```
$ ./argv -a -b -c

Parametro [0] = ./argv
Parametro [1] = -a
Parametro [2] = -b
Parametro [3] = -c
```

Esempio: uso di fork ed exec nella shell



- L'interprete dei comandi (*shell*) estrae i token dalla linea di comando (parole separate da spazi)
- crea un nuovo processo (*fork*), esegue il programma e gli passa i parametri (*exec*)
- attende che il figlio termini, poi mostra di nuovo il *prompt* dei comandi

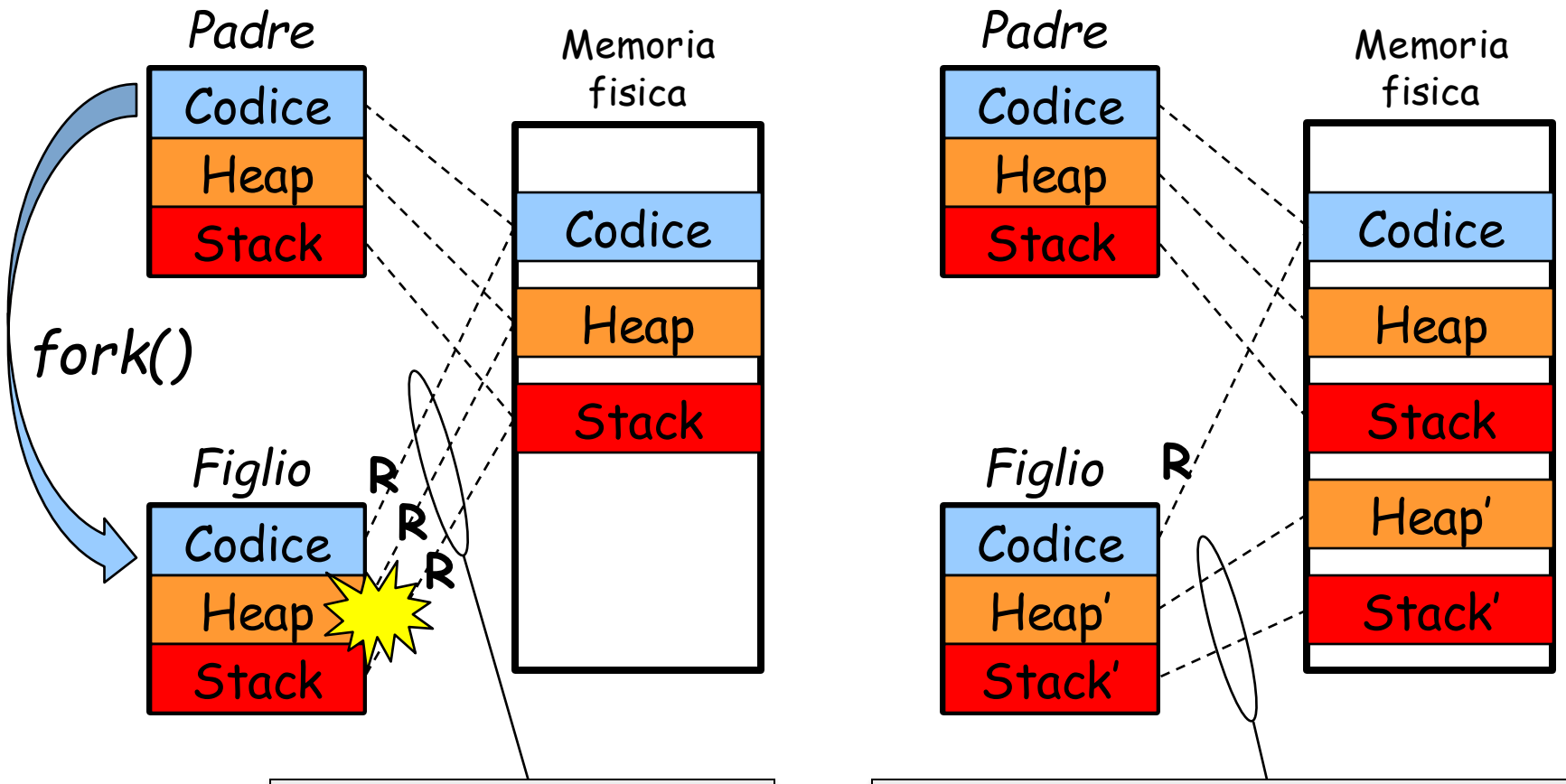
Copy-On-Write (COW)

- Nel 99% dei casi, dopo una fork() viene eseguita una exec
  - L'operazione di copia della memoria tra padre e figlio è nella maggior parte dei casi sprecata
  - L'overhead dunque è consistente
  - Alcuni SO combinano fork ed exec in un'unica system call (es. Windows)

copy-on-write

- Il figlio condivide inizialmente la memoria con il padre, in modalità read-only
- Quando il figlio modifica una porzione di memoria, il kernel crea una nuova copia di quella sola porzione di memoria nel processo figlio ("copy-on-write")

Copy-On-Write (COW)



Pagine **read-only**, causano eccezioni in caso di modifica

Il SO gestisce l'eccezione, creando una copia delle pagine che il figlio tenta di **modificare**

Perchè due primitive diverse ?

- Separare fork da exec permette di configurare il processo figlio prima di eseguire un programma

```
int pid = fork();
if(pid == 0) {
  // Il processo figlio esegue qui operazioni di gestione
  // (es., si riducono i permessi, si chiudono risorse, …)

  // si avvia qui il nuovo programma
   execl("program", arg0, arg1, arg2, …);
}
```

Esercizio: The Unix Shell

```
while(1) {
   (1) Legge il nome del programma (arg0) da input
   (2) Legge gli argomenti passati al programma (arg1 … argN)
   (3) Esegue il programma in un processo figlio (fork+exec)

   int pid = fork();                    // crea il figlio

   if (pid == 0) {                      // il figlio continua qui
        exec("programma", ... argomenti ...);
   }
   else {                               // il padre continua qui
        …
   }
}
```

Esercizio: The Unix Shell

- Alcuni suggerimenti:
  - È possibile utilizzare scanf("%[^\n]", ...) per leggere una linea di comando dall'utente (o altra funzione analoga, come getline() oppure fgets())
  - Usare la funzione strtok(), in un ciclo, per estrarre le singole parole ("token"), usando il carattere spazio come separatore
  - È consentito raggruppare i token in un array di stringhe, di capacità massima prefissata (eventuali token oltre la capacità sono ignorati)
  - Utilizzare il passaggio di parametri ad exec() tramite vettore

Header files

- #include <unistd.h> // fork(), exec*(), // sleep(), getpid()
- #include <sys/wait.h> // wait()
- #include <stdlib.h> // exit()
- #include <sys/types.h> // pid_t, size_t, ...
- #include <string.h> // strtok(), strcmp(), ...
- #include <stdio.h> // perror()

Man pages

- man 2 fork
- man 3 exec
- man 2 wait
- man 3 exit
- man 3 sleep

Il numero indica la sezione della documentazione, per evitare le ambiguità (es., "sleep" è sia una chiamata di libreria C, sia un comando built-in della shell)