I Processi nel Sistema Operativo Unix

Processi Unix

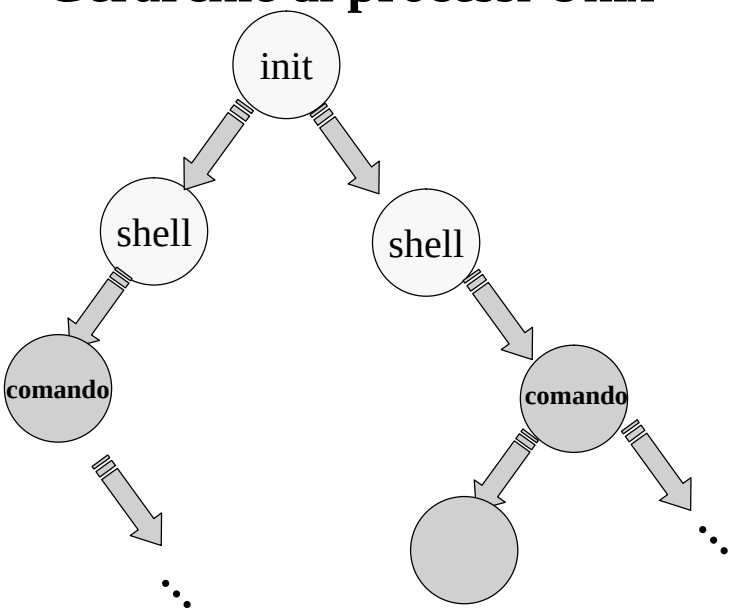
Unix è un sistema operativo *multiprogrammato* a *divisione di tempo*: l'unità di computazione è

il processo.

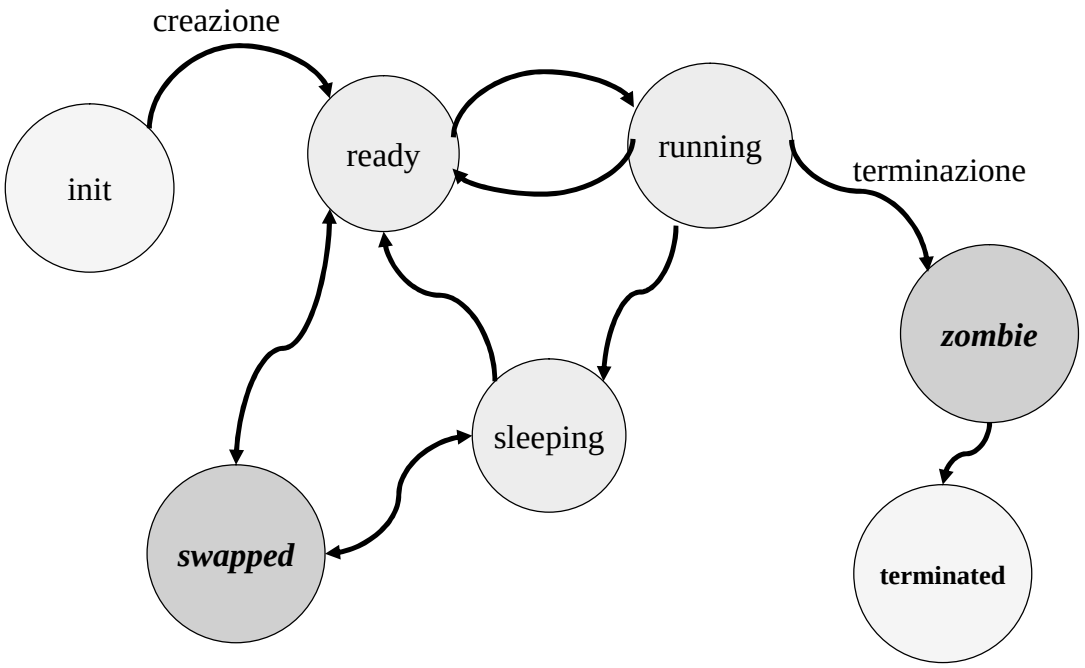
Caratteristiche del processo Unix:

- **processo pesante con codice *rientrante*:**
  - ✓ **dati non condivisi**
  - ✓ **codice condivisibile con altri processi**
- **funzionamento dual mode:**
  - ✓ **processi di utente (modo *user*)**
  - ✓ **processi di sistema (modo *kernel*)**
  - ☞ **diverse potenzialità e, in particolare, diversa visibilità della memoria.**

Gerarchie di processi Unix



Stati di un processo Unix



Stati di un processo Unix

Come nel caso generale:

- Init: caricamento in memoria del processo e inizializzazione delle strutture dati del S.O.
- Ready: processo pronto
- Running: il processo usa la CPU
- *Sleeping*: il processo è sospeso in attesa di un evento
- Terminated: deallocazione del processo dalla memoria.

In aggiunta:

- ***Zombie***: il processo è terminato, ma è in attesa che il padre ne rilevi lo stato di terminazione.
- ***Swapped***: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria.

Processi Swapped

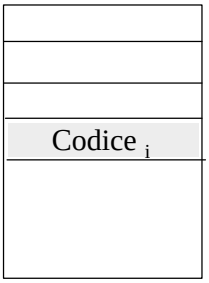
Lo scheduler a medio termine (swapper) gestisce i trasferimenti dei processi:

- **da memoria centrale a secondaria (dispositivo di swap): *swap out***
  - **si applica preferibilmente ai processi bloccati (sleeping), prendendo in considerazione tempo di attesa, di permanenza in memoria e dimensione del processo (preferibilmente i processi più lunghi)**
- **da memoria secondaria a centrale : *swap in***
  - **si applica preferibilmente ai processi più corti**

Rappresentazione dei processi Unix

Il codice dei processi è rientrante: più processi possono condividere lo stesso codice (*text*):

- ✓ **codice e dati sono separati (modello a *codice puro*)**
- ✓ **il S.O. gestisce una struttura dati globale in cui sono contenuti i puntatori ai codici utilizzati ed eventualmente condivisi) dai processi: *text table*.**
- ✓ **L'elemento della text table si chiama *text structure* e contiene, ad esempio:**
  - ❒ ***puntatore* al codice** (se il processo è swappato, riferimento a memoria secondaria)
  - ❒ **numero dei processi che lo condividono**
  - ❒ **...**



Text table: 1 elemento ∀ segmento di codice utilizzato

Rappresentazione dei processi Unix

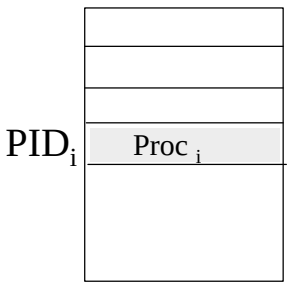
- **Process Control Block**: il descrittore del processo in Unix e` rappresentato da 2 strutture dati:
  - ***Process Structure:*** **informazioni necessarie al sistema per la gestione del processo (a prescindere dallo stato del processo)**
  - ***User Structure:*** **informazioni necessarie solo se il processo e` residente in memoria centrale**

Process Structure

- ***Process Structure:*** contiene, tra l'altro, le seguenti informazioni:
  - ✓ process identifier (**PID**): è un intero positivo che individua univocamente il processo nel sistema
  - ✓ stato del processo
  - ✓ puntatori alle varie **aree dati e stack** associati al processo
  - ✓ riferimento indiretto al **codice**: la process structure contiene il riferimento all'elemento della text table associato al codice del processo
  - ✓ informazioni di **scheduling** (es: priorità, tempo di CPU,etc)
  - ✓ Riferimento al **processo padre** (PID del padre)
  - ✓ Informazioni relative alla **gestione di segnali** (segnali inviati ma non ancora gestiti, maschere)
  - ✓ Puntatori a processi successivi in **code** di scheduling (ad esempio, ready queue)
  - ✓ Puntatore alla ***User Structure***

Rappresentazione dei processi Unix

- **Process Structure:** sono organizzate in un vettore: ***Process Table***



Process table: 1 elemento per ogni processo

User Structure

Contiene le informazioni necessarie al S.O. per la gestione del processo, **quando è residente**:

- copia dei **registri** di CPU
- informazioni sulle risorse allocate (ad es. **file aperti**)
- informazioni sulla gestione di **segnali** (puntatori a *handler*, etc.)
- **ambiente** del processo: direttorio corrente, utente, gruppo, argc/argv, path, etc.

Immagine di un processo Unix

L'immagine di un processo è l'insieme delle aree di memoria e delle strutture dati associate al processo.

- Non tutta l'immagine è accessibile in modo user:
  - parte di **kernel**
  - parte di **utente**
- Ogni processo può essere soggetto a swapping: non tutta l'immagine può essere trasferita in memoria:
  - parte ***swappable***
  - parte residente o ***non swappable***

Immagine di un processo Unix

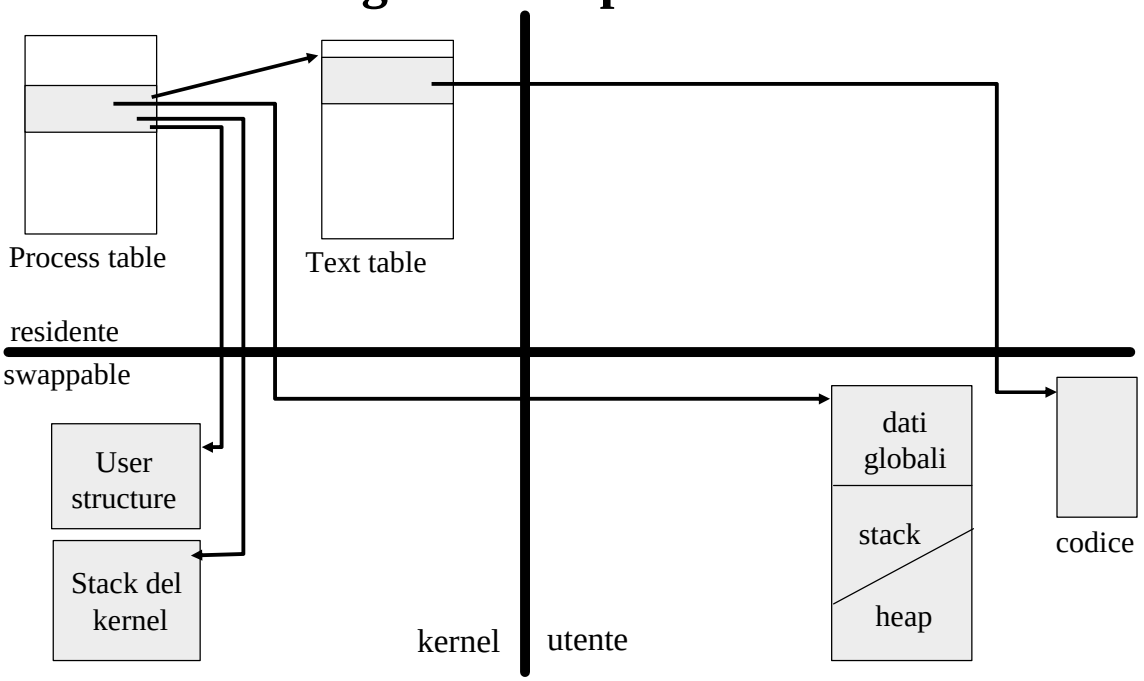


Immagine di un processo Unix

Componenti:

- **process structure**: è l'elemento della process table associato al processo (kernel, residente)
- **text structure**: elemento della text table associato al codice del processo (kernel, residente)
- area **dati globali di utente**: contiene le variabili globali del programma eseguito dal processo (user, swappable)
- **stack, heap** di utente: aree dinamiche associate al programma eseguito (user, swappable)
- **stack del kernel**: stack di sistema associato al processo per le chiamate a system call (kernel, swappable)
- **user structure**: struttura dati contenente i dati necessari al kernel per la gestione del processo quando è residente (kernel, swappable).

PCB= Process Structure + User Structure

- **Process Structure**: mantiene le informazioni necessarie per la gestione del processo, anche se questo è ***swappato*** in memoria secondaria.
- **User Structure**: il suo contenuto è necessario **solo in caso di esecuzione** del processo (stato *running*); se il processo è soggetto a swapping, anche la user structure può essere trasferita in memoria secondaria.

☞**Process structure:** contiene il riferimento a **user structure** (in memoria centrale o secondaria)

System Call per la gestione di Processi

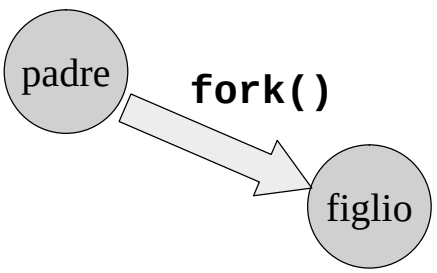
Chiamate di sistema per:

- creazione di processi: **`fork()`**
- sostituzione di codice e dati: **`exec..()`**
- terminazione: **`exit()`**
- sospensione in attesa della terminazione di figli: **`wait()`**

N.B. Le system call di Unix sono attivabili attraverso funzioni di librerie C standard: fork(), exec(), etc. sono quindi funzioni di libreria che chiamano le system call corrispondenti.

Creazione di processi: fork()

- La funzione **`fork()`** consente a un processo di generare un processo figlio:
  - padre e figlio **condividono** lo stesso **codice**
  - il figlio *eredita* una **copia dei dati** (di utente e di kernel) del padre



fork()

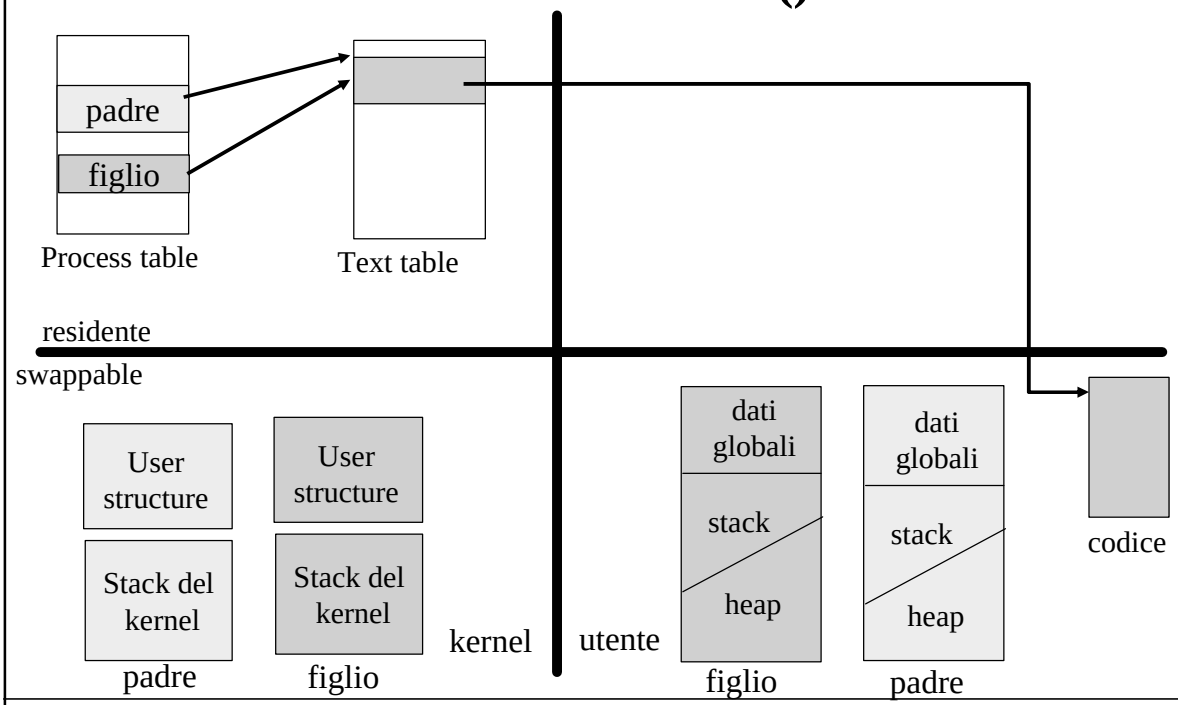
```
int fork(void);
```

- la fork non richiede parametri
- restituisce un intero che:
  - ✓ per il processo creato vale **0**
  - ✓ per il processo padre è un valore **positivo** che rappresenta il PID del processo figlio
  - ✓ è un valore **negativo** in caso di errore (la creazione non è andata a buon fine)

Effetti della fork()

- Allocazione di una **nuova process structure** nella process table associata al processo figlio e sua inizializzazione
- Allocazione di una **nuova user structure** nella quale viene copiata la user structure del padre
- Allocazione dei **segmenti di dati e stack** del figlio nei quali vengono copiati dati e stack del padre
- Aggiornamento della **text structure** del codice eseguito (condiviso col padre): incremento del contatore dei processi, etc.

Effetti della fork()



fork(): esempio

```
#include <stdio.h>
main()
{ int pid;
    pid=fork();
    if (pid==0)
    { /* codice figlio */
   printf("Sono il figlio ! (pid: %d)\n", getpid());
   }
    else if (pid>0)
      { /* codice padre */
     printf("Sono il padre: pid di mio figlio: %d\n", pid);
    ....
      }
   else printf("Creazione fallita!");
 }
```

NB: la system call **`getpid`** ritorna il pid del processo che la chiama.

Relazione Padre-Figlio in Unix

Dopo una `fork()`:

- **concorrenza:**
  - ✓ padre e figlio procedono in parallelo
- **lo spazio degli indirizzi è duplicato :**
  - ✓ ogni variabile del figlio è inizializzata con il valore assegnatole dal padre prima della fork()
- la **user structure** è duplicata **:**
  - ✓ le risorse allocate al padre (ad esempio, i file aperti) prima della generazione sono condivise con i figli
  - ✓ le informazioni per la gestione dei segnali sono le stesse per padre e figlio (associazioni segnali-handler)
  - ✓ il figlio nasce con lo stesso Program Counter del padre: la prima istruzione eseguita dal figlio è quella che segue immediatamente la fork().

Terminazione di processi

Un processo può terminare:

- **involontariamente**:
  - ✓ tentativi di azioni illegali
  - ✓ interruzione mediante segnale

  ☞ salvataggio dell'immagine nel file **`core`**
- **volontariamente**:
  - ✓ chiamata alla funzione **`exit()`**
  - ✓ esecuzione dell'ultima istruzione

exit( )

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

- la funzione **`exit`** prevede un parametro (**`status`**) mediante il quale il processo che termina può comunicare al padre informazioni sul suo stato di terminazione (ad es., l'esito della sua esecuzione).
- è sempre una chiamata senza ritorno

exit()

Effetti di una exit():

- chiusura dei file aperti non condivisi
- terminazione del processo:
  - ✓ se il processo che termina ha **figli in esecuzione**, il processo **`init`** adotta i figli dopo la terminazione del padre (nella process structure di ogni figlio al pid del processo padre viene assegnato il valore 1 (pid di init)
  - ✓ se il processo **termina prima che il padre ne rilevi lo stato di terminazione** (con la system call **`wait),`** il processo passa nello stato **zombie**.

NB. Quando termina un processo adottato da **`init`**, il processo init rileva automaticamente il suo stato di terminazione -> i processi figli di **`init`** non diventano mai zombie !

`wait`

- Lo stato di terminazione può essere rilevato dal processo padre, mediante la system call **`wait()`:**

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

  - **il parametro `status` è l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio**
  - **il risultato prodotto dalla `wait` è il pid del processo terminato, oppure un codice di errore (<0)**

`wait`

Effetti della system call `wait(&status)`:

- il processo che la chiama può avere figli in esecuzione:
  - ✓ se tutti i figli non sono ancora terminati, il processo si sospende in attesa della terminazione del primo di essi
  - ✓ se almeno un figlio è già terminato ed il suo stato non è stato ancora rilevato (cioè è in stato ***zombie***), la **`wait`** ritorna immediatamente con il suo stato di terminazione (nella variabile **`status`**)
  - ✓ se non esiste neanche un figlio, la **`wait`** non è sospensiva e ritorna un codice di errore (valore ritornato < 0).

`wait`

Rilevazione dello stato: in caso di terminazione di un figlio, la variabile status raccoglie il suo stato di terminazione; nell'ipotesi che lo stato sia un intero a 16 bit:

- ✓ se il byte meno significativo di status è zero, il più significativo rappresenta lo stato di terminazione (terminazione volontaria, ad esempio con **`exit`**)
- ✓ in caso contrario, il byte meno significativo di status descrive il segnale che ha terminato il figlio (terminazione involontaria).

wait & exit: esempio

```
main()
{int pid, status;
pid=fork();
if (pid==0)
   {printf("figlio");
   exit(0);
}
else{ pid=wait(&status);
       printf("terminato processo figlio n.%d", pid);
       if ((char)status==0)
       printf("term. volontaria con stato %d", status>>8);
       else printf("terminazione involontaria  per segnale
                    %d\n", (char)status);
   }
}
```

wait

- **Rilevazione dello stato:** e` necessario conoscere la rappresentazione di **`status:`**
  - lo standard Posix.1 prevede delle macro (definite nell'header file <sys/wait.h> per l'analisi dello stato di terminazione. In particolare:
    - **`WIFEXITED(status):`** restituisce vero, se il processo figlio è terminato volontariamente: in questo caso la macro **`WEXITSTATUS(status)`** restituisce lo stato di terminazione.
    - **`WIFSIGNALED(status):`** restituisce vero, se il processo figlio è terminato involontariamente
    - in questo caso la macro **`WTERMSIG(status)`** restituisce il numero dell'interruzione SW che ha causato la terminazione.

wait & exit: esempio

```
#include <sys/wait.h>
main()
{int pid, status;
pid=fork();
if (pid==0)
   {printf("figlio");
   exit(0);
}
else{ pid=wait(&status);
       if (WIFEXITED(status))
              printf("Terminazione volontaria di %d con
                     stato %d\n", pid, WEXITSTATUS(status));
       else if (WIFSIGNALED(status))
              printf("terminazione involontaria  per segnale
                     %d\n", WTERMSIG(status));  }}
```

Esempio con più figli:

```
#include <sys/wait.h>
#define N 100
main()
{int pid[N], status, i, k;
for (i=0; i<N; i++)
{ pid[i]=fork();
  if (pid[i]==0)
      {printf("figlio: il mio pid è: %d", getpid());
      ....
      exit(0);
      }
}
```

```
/* continua (codice padre).. */

for (i=0; i<N; i++) /* attesa di tutti i figli */
{ k=wait(&status);
      if (WIFEXITED(status))
            printf("Term. volontaria di %d con
                  stato %d\n", k,
                  WEXITSTATUS(status));
      else if (WIFSIGNALED(status))
            printf("term. Involontaria di %d  per
                  segnale %d\n",k, WTERMSIG(status));
  }
```

System call `exec`

- Mediante la fork i processi padre e figlio condividono il codice e lavorano su aree dati duplicate: in Unix è possibile differenziare il codice dei due processi mediante una system call della famiglia **`exec`**:

  `execl(), execle(), execlp(), execv(), execve(), execvp()..`

Effetto principale di una exec:

- vengono sostituiti **codice** e **dati** del processo che chiama la system call, con codice e dati di un programma specificato come parametro della system call

execl()

```
int execl(char *pathname, char *arg0, ..
          char *argN, (char*)0);
```

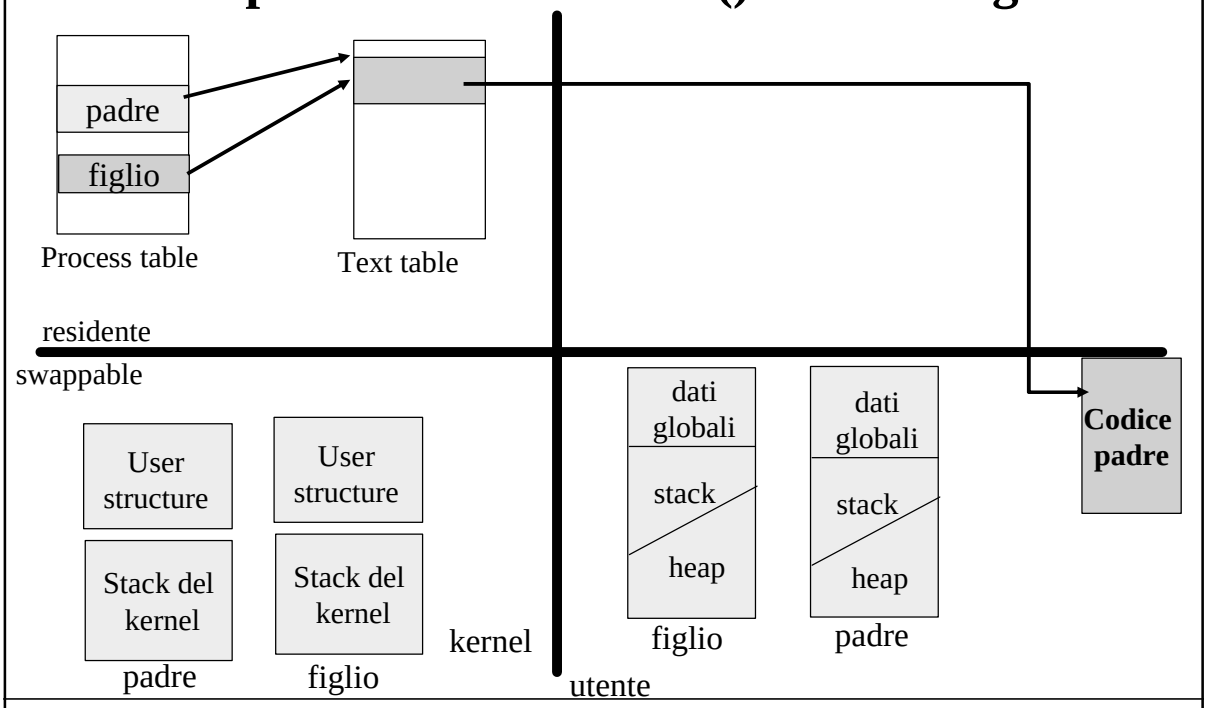
- **pathname** è il nome (assoluto o relativo) dell'eseguibile da caricare
- **arg0** è il nome del programma (argv[0])
- **arg1,..argN** sono gli argomenti da passare al programma
- **(char *)0** è il puntatore nullo che termina la lista.

Esempio: execl()

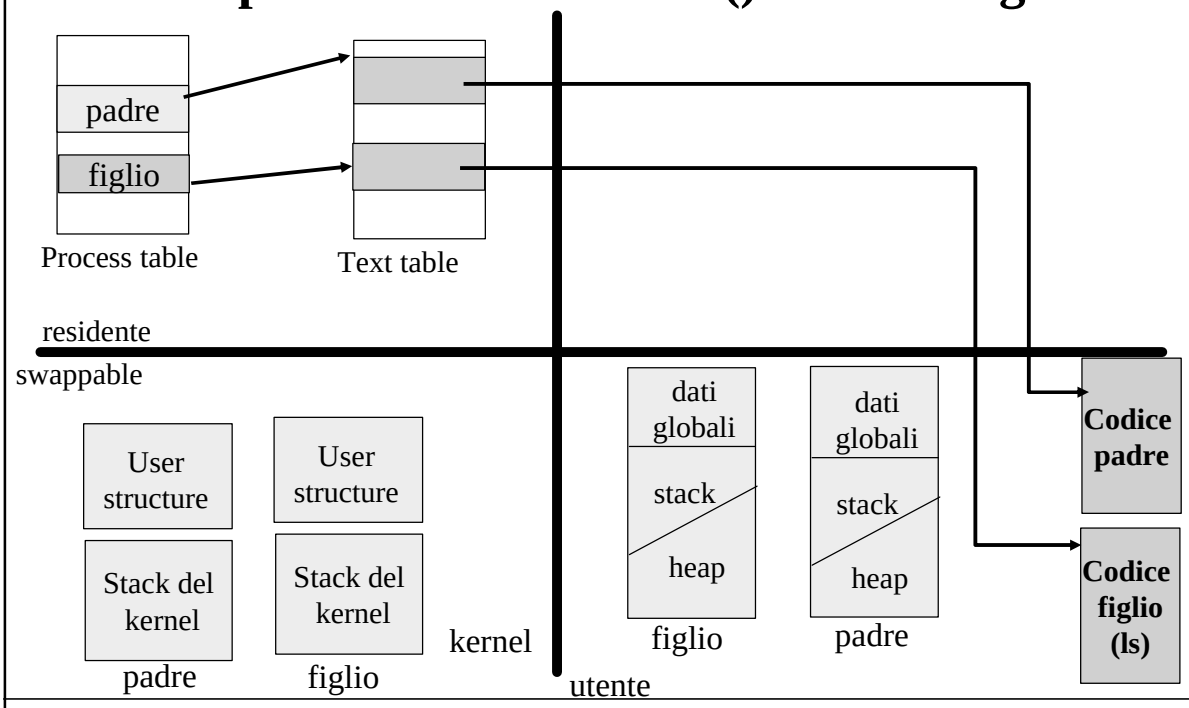
```
main()
{int pid, status;
pid=fork();
if (pid==0)
   {execl("/bin/ls", "ls","-l","pippo",(char *)0);
    printf("exec fallita!\n");
    exit(1);
}
else if (pid >0)
   { pid=wait(&status);
     /* gestione dello stato.. */
   }
else printf("fork fallita!);
}
```

NB: in caso di successo, l' exec è una chiamata senza ritorno!

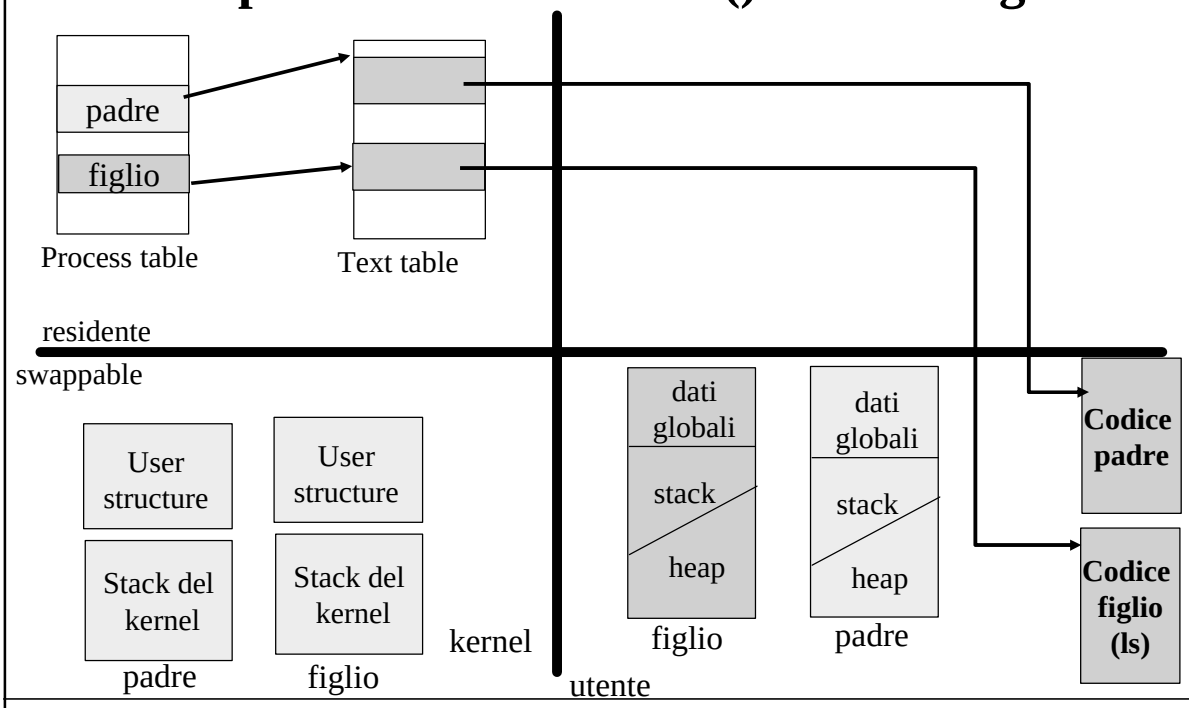
Esempio: effetti della exec() sull'immagine



Esempio: effetti della execl() sull'immagine



Esempio: effetti della execl() sull'immagine



Effetti dell'exec

Il processo dopo l'exec:

- mantiene la stessa **process structure (salvo le informazioni relative al codice)**:
  - ✓ stesso pid
  - ✓ stesso pid del padre
  - ✓ ...
- ha **codice**, **dati globali**, **stack** e **heap** nuovi
- riferisce una nuova **text structure**
- mantiene **user area** (**a parte PC e informazioni legate al codice**) e stack del kernel:
  - ✓ mantiene le stesse risorse (es: file aperti)
  - ✓ mantiene lo stesso *environment* (a meno che non sia `execle` o `execve`)

System call exec . .

Varianti dell'exec: a seconda del suffisso

- **l : gli argomenti da passare al programma da caricare vengono specificati mediante una LISTA di parametri (terminata da NULL) (es. execl())**
- **p : il nome del file eseguibile specificato come argomento della system call viene ricercato nel PATH contenuto nell'ambiente del processo (es. execlp())**
- **v : gli argomenti da passare al programma da caricare vengono specificati mediante un VETTORE di parametri(es. execv())**
- **e: la system call riceve anche un vettore (envp[]) che rimpiazza l'environment (path, direttorio corrente, etc.) del processo chiamante (es. execle())**

Esempio: execve()

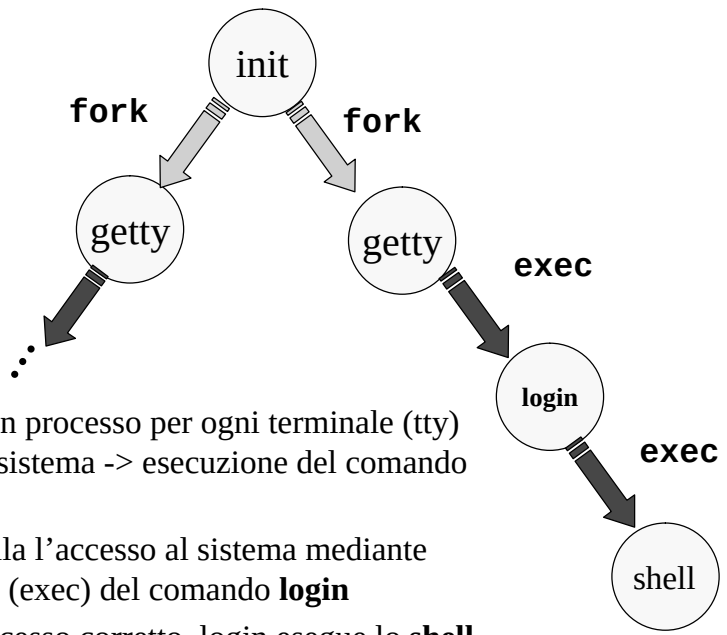
```
int execve(char *pathname, char *argV[], ..
           char * env[]);
```

- pathname è il nome (assoluto o relativo) dell'eseguibile da caricare
- argV è il vettore degli argomenti del programma da eseguire
- env è il vettore delle variabili di ambiente da sostituire all'ambiente del processo (contiene stringhe del tipo "VARIABILE=valore")

Esempio: execve()

```
char *env[]={“USER=anna”, “PATH=/home/anna/d1”, (char *)0};
char *argv[]={“ls”,”-l”, “pippo”, (char *)0};
main()
{int pid, status;
pid=fork();
if (pid==0)
   {execve(“/bin/ls”, argv, env);
    printf(“exec fallita!\n”);
    exit(1);
}
else if (pid >0)
   { pid=wait(&status);  /* gestione dello stato.. */
   }
else printf(“fork fallita!);
}
```

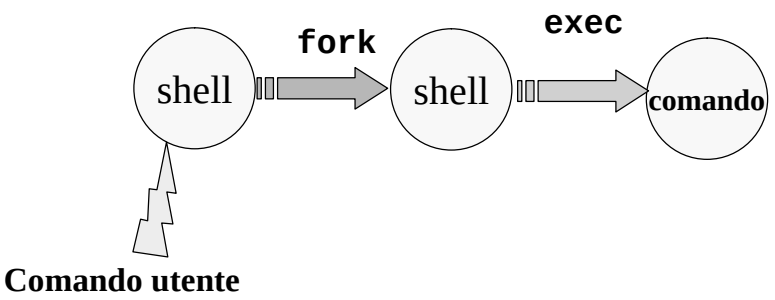
Inizializzazione dei processi Unix



- init genera un processo per ogni terminale (tty) collegato al sistema -> esecuzione del comando **getty**
- getty controlla l'accesso al sistema mediante l'esecuzione (exec) del comando **login**
- in caso di accesso corretto, login esegue lo **shell** (specificato dall'utente in /etc/passwd)

Interazione con l’utente tramite shell

- Ogni utente può interagire con lo shell mediante la specifica di comandi.
- Per ogni comando, lo shell genera un processo figlio dedicato all’esecuzione del comando:



Relazione shell padre-shell figlio

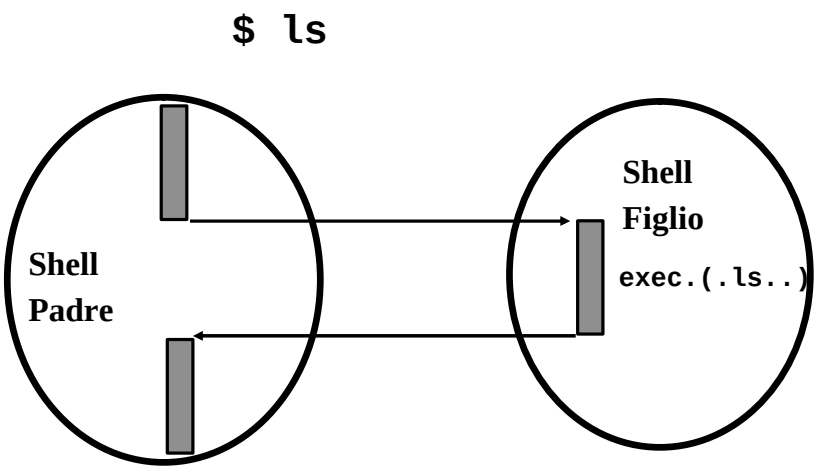
- Per ogni comando, lo shell genera un figlio; possibilità di **due diversi comportamenti**:
  - il padre si pone in attesa della terminazione del figlio (esecuzione in ***foreground***); es:

    ```
    ls - l pippo
    ```

  - il padre continua l'esecuzione concorrentemente con il figlio (esecuzione in ***background***):

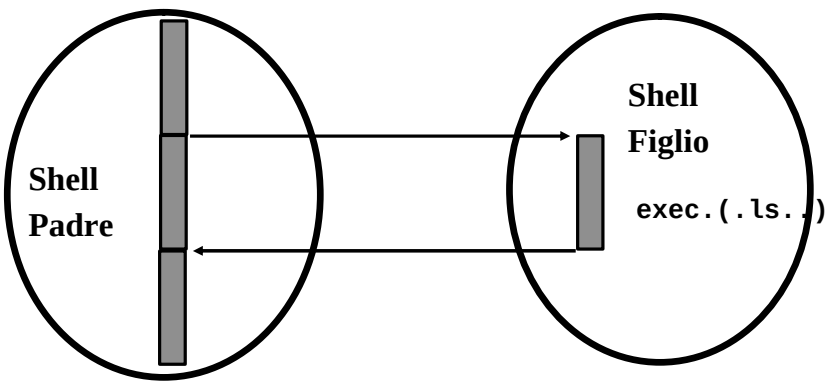
    ```
    ls - l pippo &
    ```

foreground



foreground

```
$ ls&
```



Gestione degli errori: perror()

Convenzione:

- in caso di fallimento, ogni system call ritorna un valore negativo (tipicamente, -1)
- in aggiunta, Unix prevede la variabile globale di sistema **`errno`**, alla quale il kernel assegna il codice di errore generato dall'ultima system call eseguita; per interpretarne il valore è possibile usare la funzione **`perror()`**:
  - perror("stringa") stampa "stringa" seguita dalla descrizione del codice di errore contenuto in errno
  - la corrispondenza tra codici e descrizioni è contenuta in **`<sys/errno.h>`**

perror()

```
main()
{int pid, status;
pid=fork();
if (pid==0)
   {execl("/home/anna/prova", "prova",(char *)0);
    perror("exec fallita a causa dell'errore:");
    exit(1);
}
...
```

Esempio di output:

```
exec fallita a causa dell'errore: No such file or directory
```