System call per la gestione di processi

Chiamate di sistema per

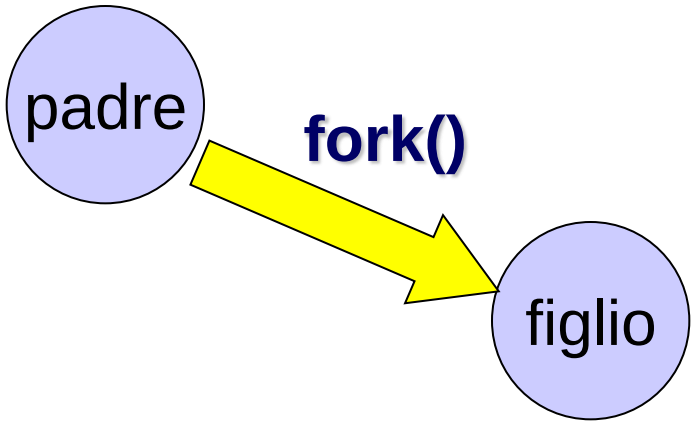
- ***creazione di processi***: **`fork()`**
- ***sostituzione di codice e dati***: **`exec...()`**
- ***terminazione***: **`exit()`**
- ***sospensione*** in attesa della ***terminazione di figli***: **`wait()`**

N.B. System call di UNIX sono attivabili attraverso chiamate a funzioni di librerie C standard: **`fork(), exec(), ...`** 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()

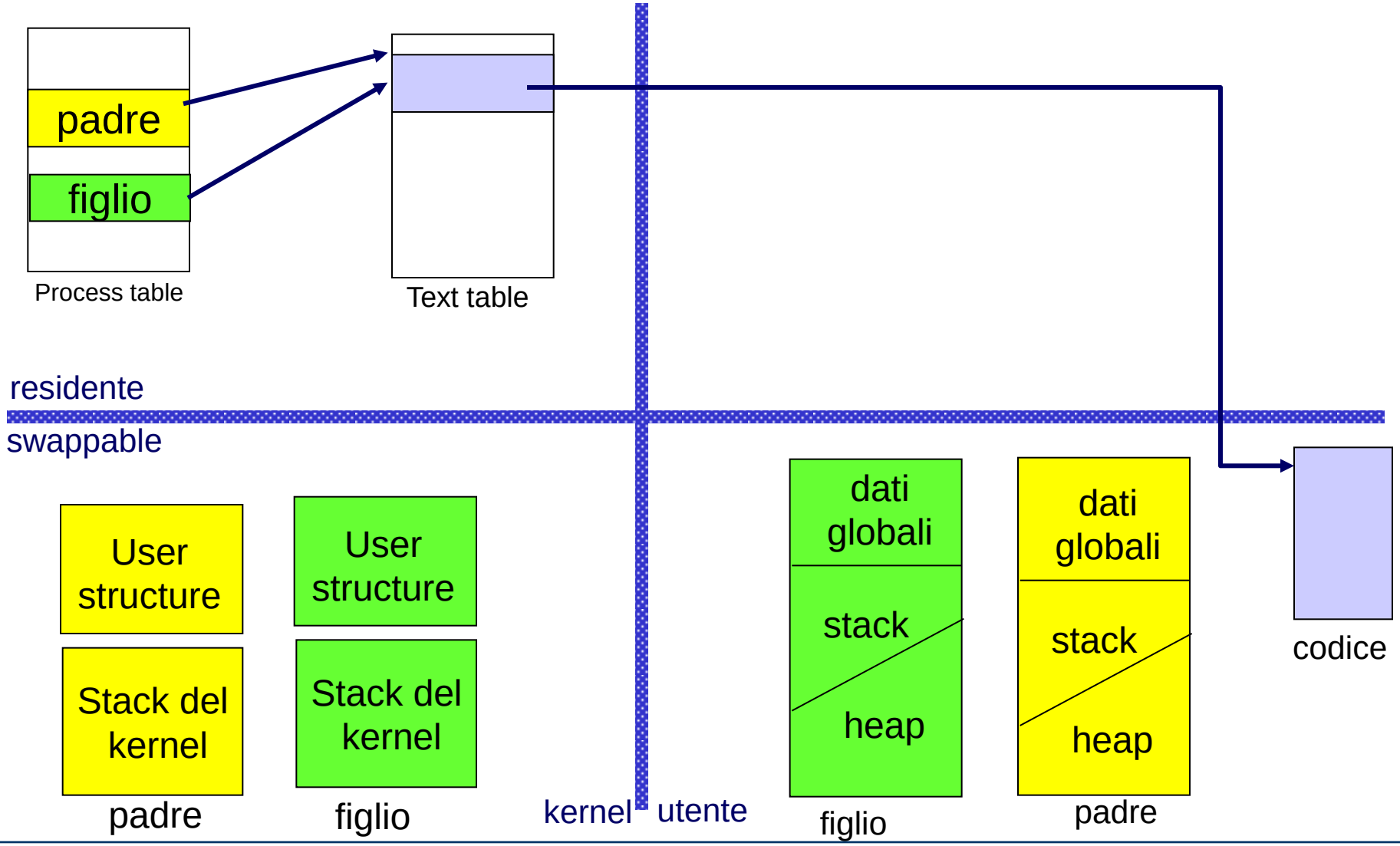
int fork(void);

- **`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 Control Block (PCB)*** nella quale viene ***copiata la PCB del padre***
- Allocazione dei ***segmenti di dati e stack*** del figlio nei quali vengono ***copiati dati e stack del padre***
- Aggiornamento del riferimento al codice eseguito (condiviso col padre)

Effetti (reali) della fork()



Esecuzioni differenziate del padre e del figlio

```
...
if (fork()==0) {
   ...      /* codice eseguito dal figlio */
      ...
} else {
   .../* codice eseguito dal padre */
      ...
}
```

Dopo la generazione del figlio ***il padre può decidere***

se operare ***contemporaneamente*** ad esso

oppure

se ***attendere la sua terminazione*** (system call **`wait()`**)

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: 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 **`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 esempio esito dell'esecuzione)
- è ***sempre una chiamata senza ritorno***

exit()

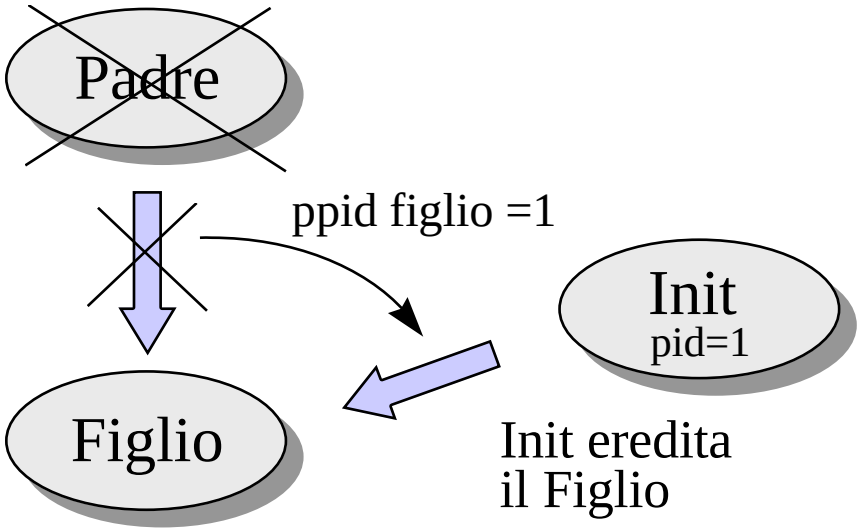
Effetti di una exit():

- ***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)
  - ✓ 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***

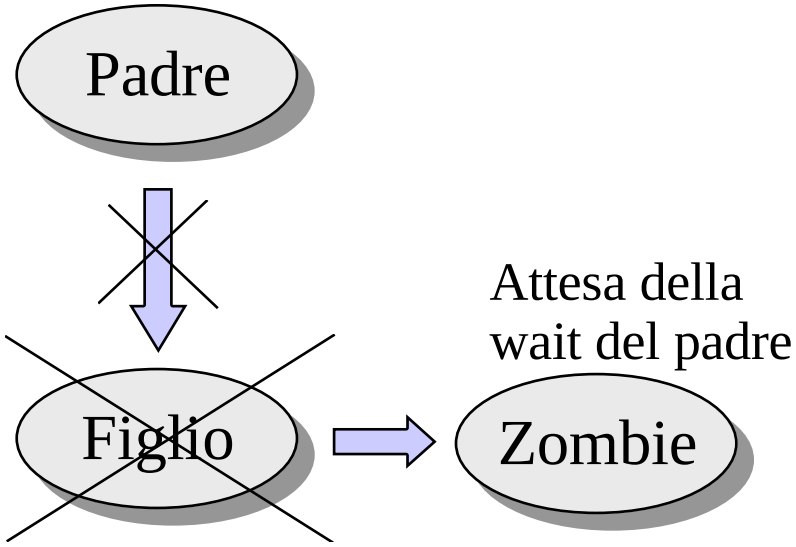
NB: Quando termina un processo adottato dal processo **init**, **init** rileva automaticamente il suo stato di terminazione -> i processi figli di **init** non permangono nello stato di zombie

Parentela processi e terminazione

Terminazione del padre



Terminazione del figlio: processi zombie



wait()

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

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

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

wait()

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

- 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***), **`wait()`** ***ritorna immediatamente con il suo stato di terminazione*** (nella variabile **status**)
  - se non esiste neanche un figlio, **`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 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***)

System call exec()

Mediante **`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 system call famiglia exec:

– vengono ***sostituiti codice ed eventuali argomenti di invocazione*** del processo che chiama la system call, ***con codice e argomenti di un programma specificato come parametro*** della system call

NO generazione di nuovi processi

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

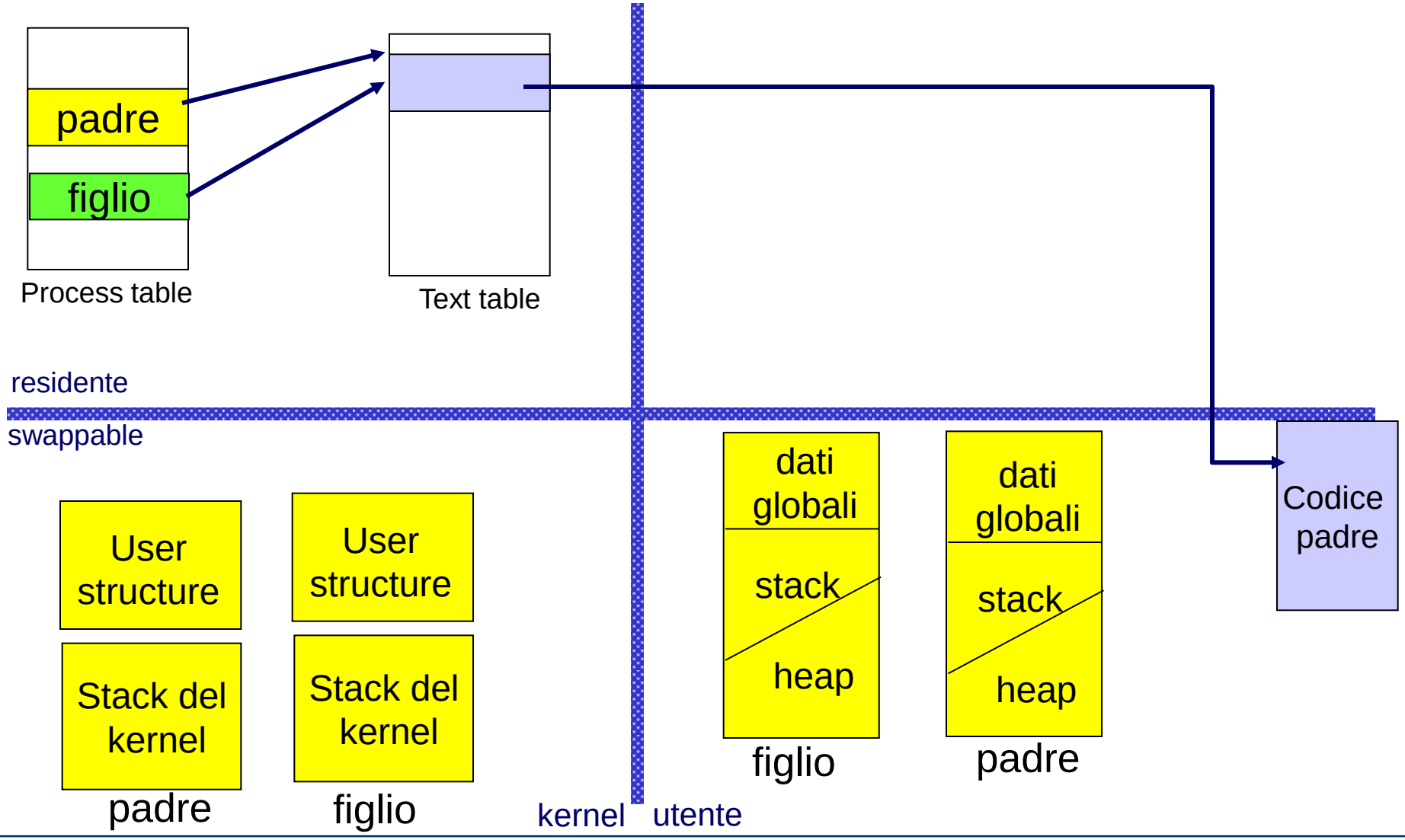
Utilizzo system call exec()
(differenziare comportamento del padre da quello del figlio)

```
pid = fork();
if (pid == 0){ /* figlio */
      printf("Figlio: esecuzione di ls\n");
      execl("/bin/ls", "ls", "-l", (char *)0);
      perror("Errore in execl\n");
      exit(1); }
if (pid > 0){ /* padre */
            ...
            printf("Padre ....\n");
            exit(0); }
if (pid < 0){ /* fork fallita */
            perror("Errore in fork\n");
            exit(1); }
```

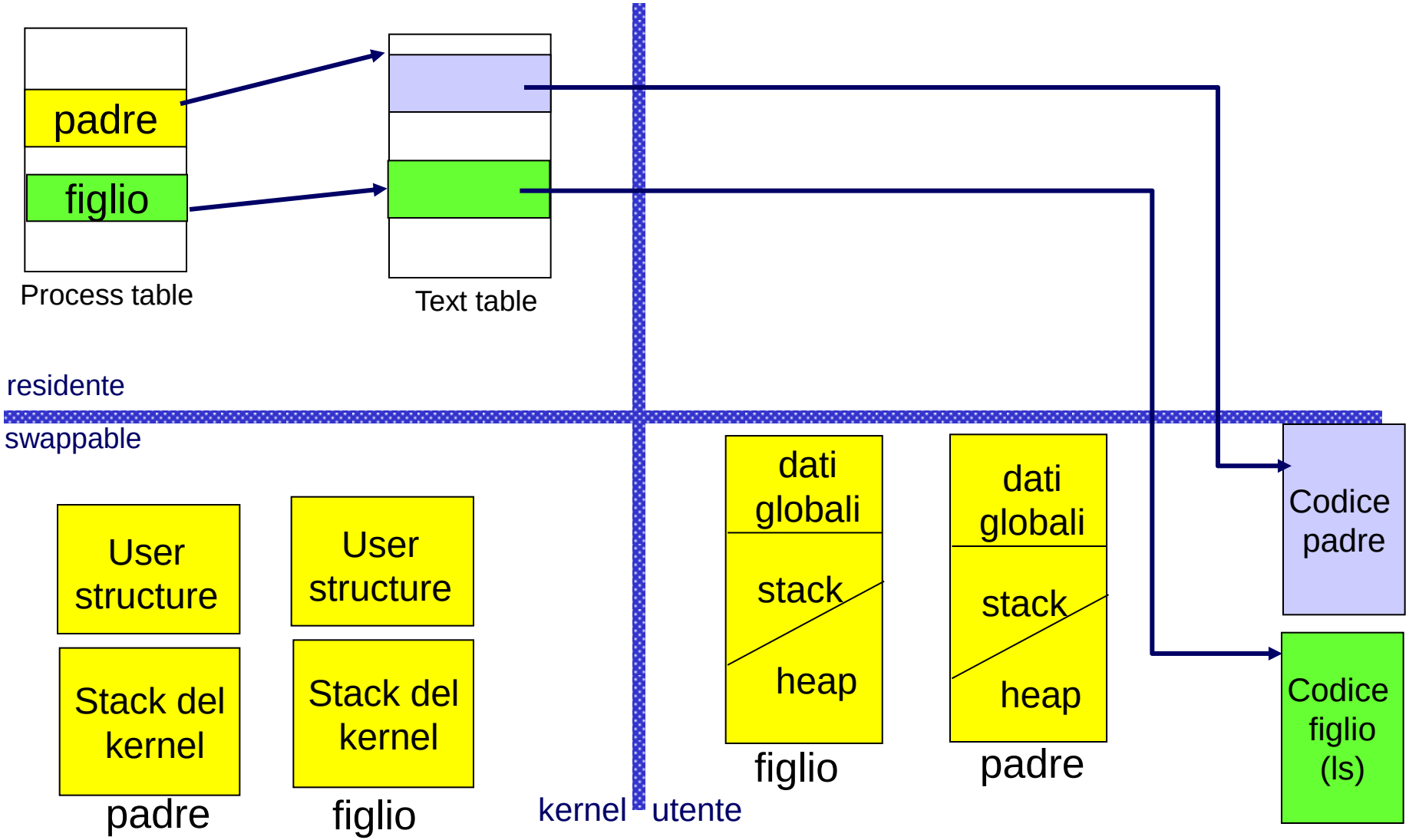
Figlio passa a **eseguire** un altro programma: *si caricano il nuovo codice e gli argomenti per il nuovo programma*

Si noti che exec è operazione senza ritorno

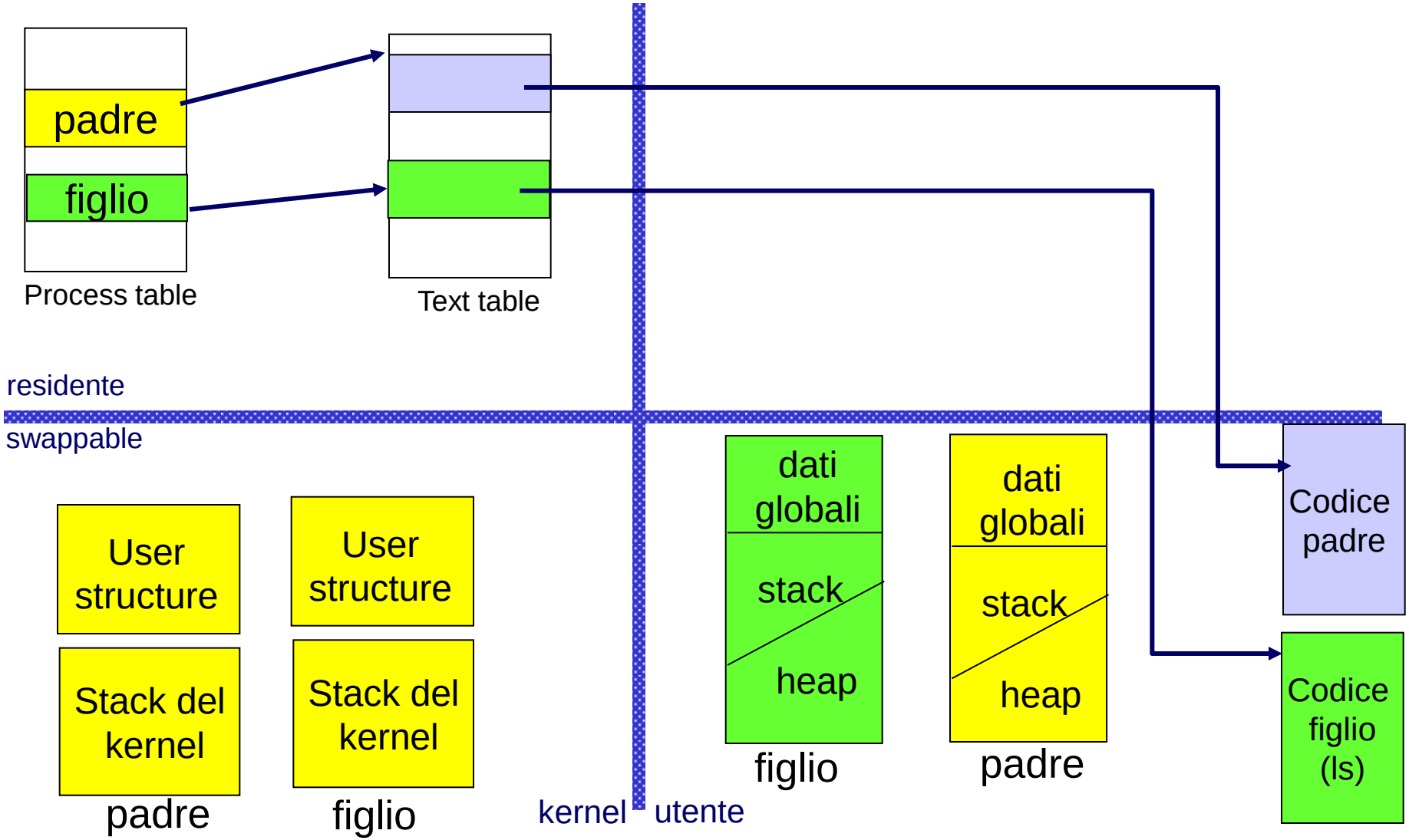
Esempio: effetti della exec() sull'immagine



Esempio: effetti della execl() sull'immagine

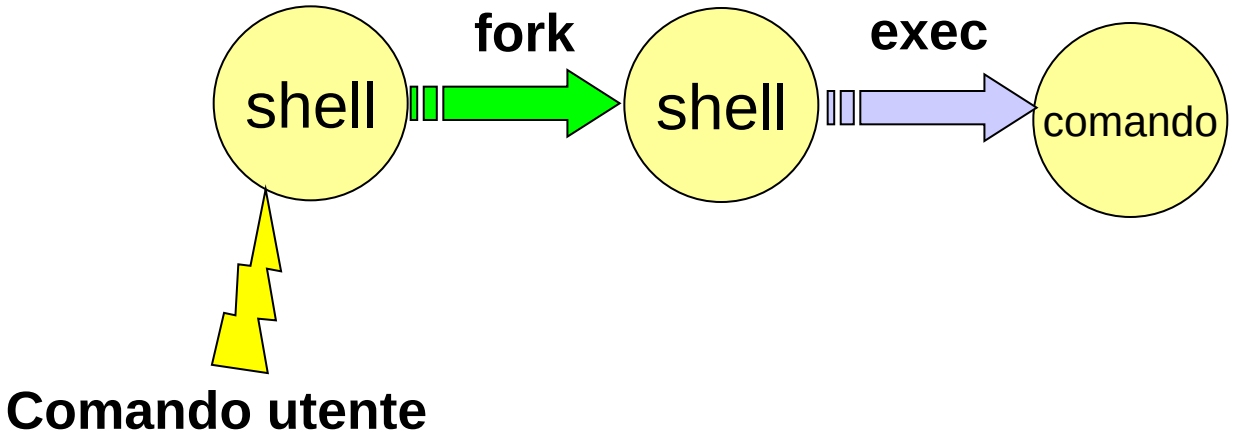


Esempio: effetti della execl() sull'immagine



Interazione con l'utente tramite shell

- Ogni utente può interagire con lo ***shell*** mediante la ***specifica di comandi***
- Ogni ***comando*** è presente nel file system come ***file eseguibile*** (direttorio ***/*bin**)
- Per ogni comando, ***shell genera un processo figlio*** dedicato all'esecuzione del comando:



Relazione shell padre-shell figlio

Per ogni comando, shell genera un figlio; possibilità di **due diversi comportamenti**:

- il padre si pone in attesa della terminazione del figlio (esecuzione in ***foreground***); es:

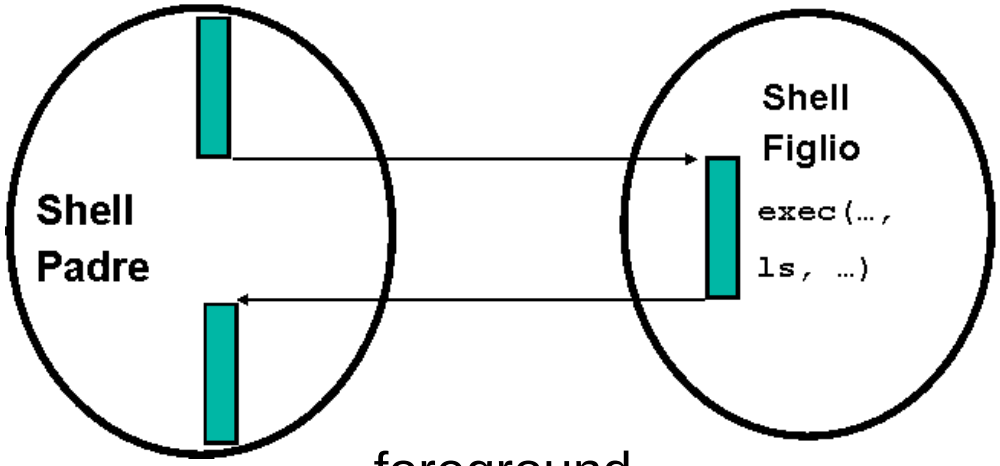
```
ls -l pippo
```

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

```
ls -l pippo &
```

foreground vs background

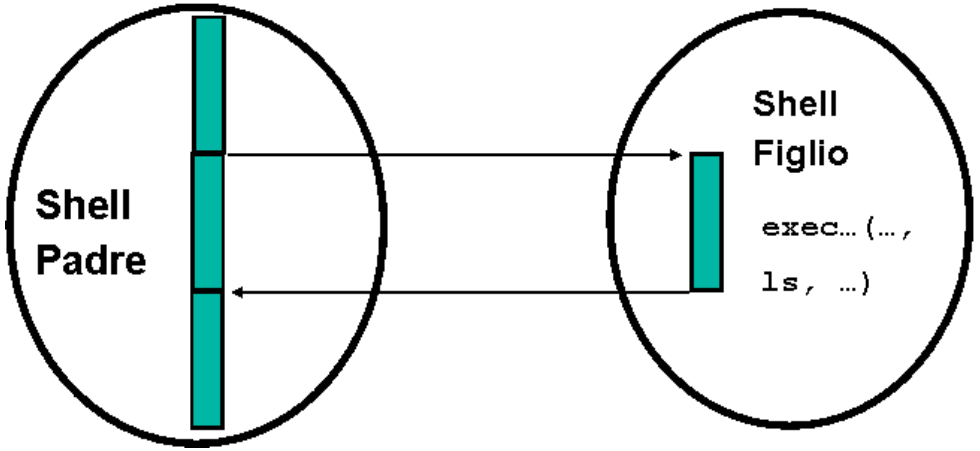
```
$ ls
```



foreground

```
$ ls&
```

background



Il mio primo filtro

Proviamo a realizzare un filtro a linee che stampi sull'output tutte le linee che contengono la lettera 'a'.

Come organizzereste
il codice del programma?