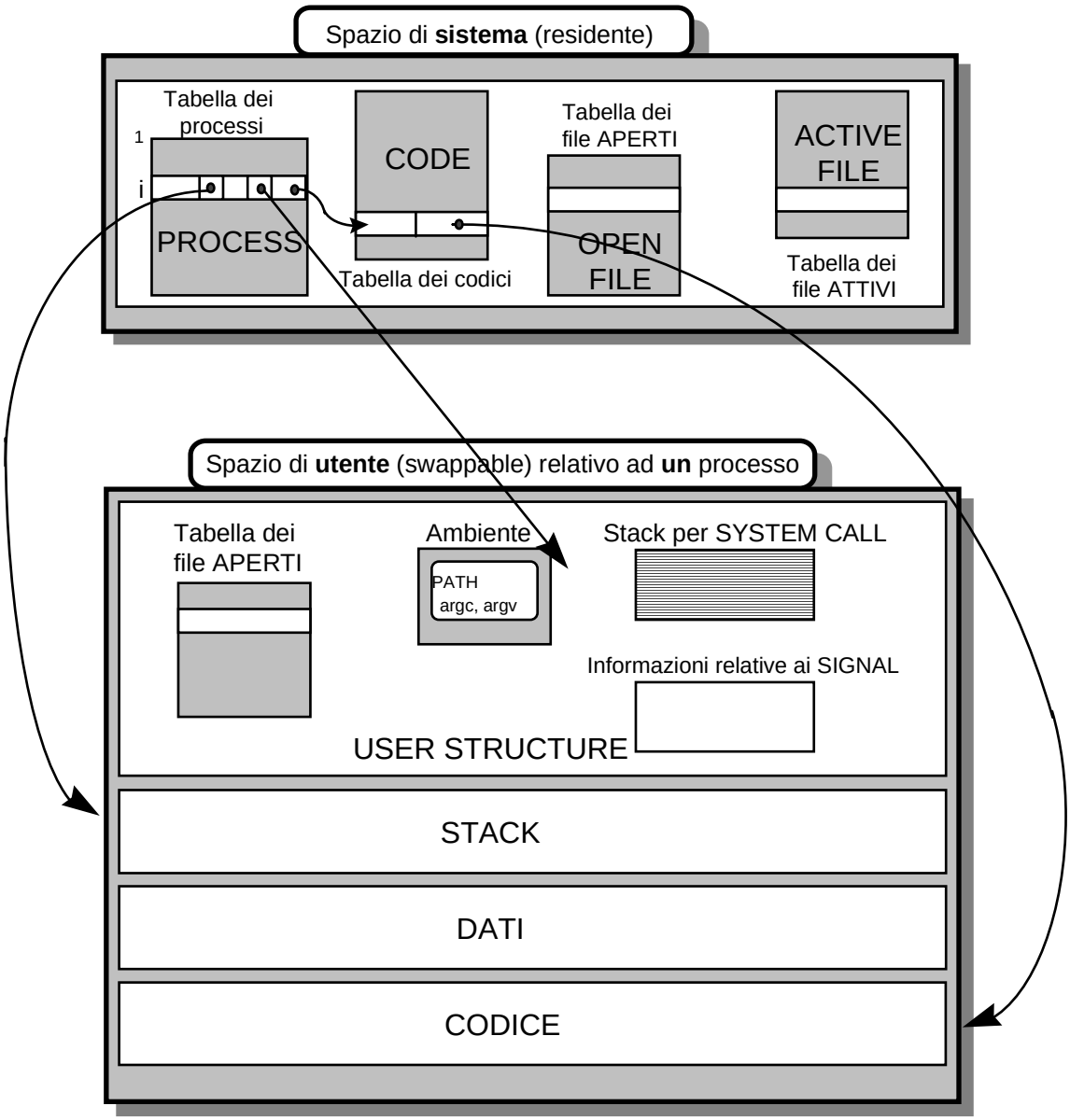
Processi in UNIX



Le tabelle sono vettori di dimensioni fissate.

processo di utente (**user process**)
processo di sistema (**system process**)
Un processo può trovarsi in **due modi** (stati) di esecuzione:
user, **kernel**

La esecuzione delle primitive avviene in stato kernel

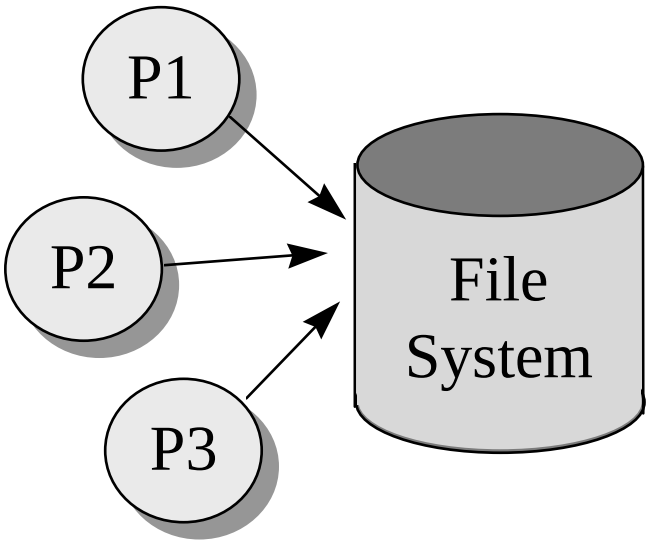
Modello di Processo in UNIX

Ogni processo ha un proprio spazio di indirizzamento completamente locale e non condiviso

➔ Modello ad Ambiente Locale

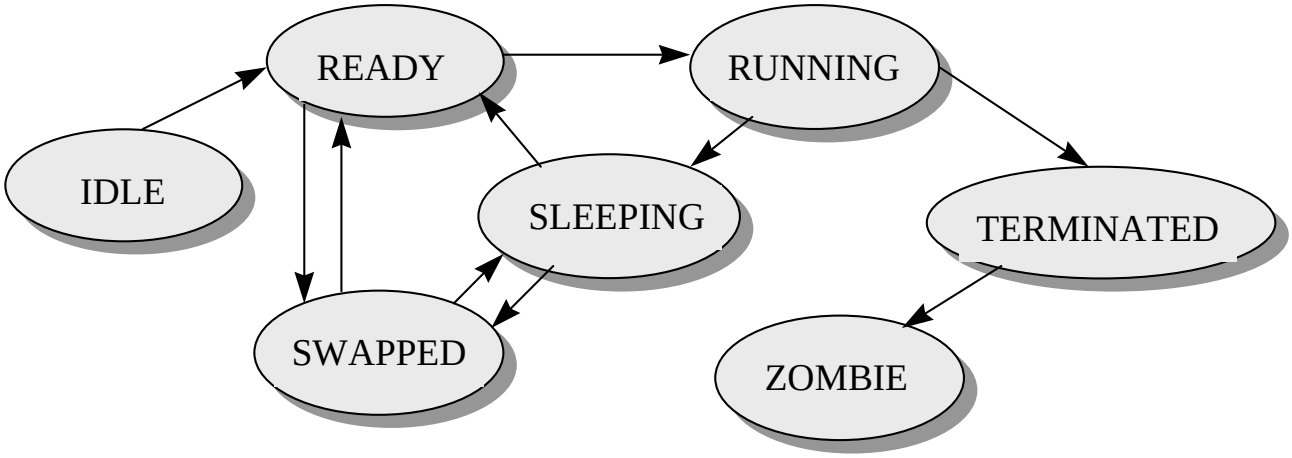
Eccezioni:

- il codice può essere condiviso
- il file system rappresenta un ambiente condiviso



Stati interni possibili di un processo UNIX

IDLE	stato iniziale
READY	pronto per l'esecuzione
RUNNING	in esecuzione
SWAPPED	immagine copiata su disco



SLEEPING	attesa di un evento per proseguire
TERMINATED	terminato
ZOMBIE	terminato ma ancora presente

Attributi di un Processo UNIX

- **pid** (**p**rocess **id**entifier)
- **ppid** (**p**arent **p**rocess **id**entifier)
- **pgid** (**p**rocess **g**roup **id**)
- **sid** (**s**ession **id**)

Un processo è lanciato da un utente, informazione di cui si tiene traccia in:

- **real uid** (real user identifier)
- **real user gid** (real user group identifier)

che corrispondono allo uid e gid dell'utente che ha lanciato il processo.
Attenzione che l'accesso ai file viene determinato sulla base di:

- **effective uid**
- **effective user gid**

Altre informazioni:

- **environment** (stringhe nome=valore)
- **current working directory**
- **root directory**
- **file mode creation mask**
- **dimensione massima** dei file creabili
- **maschera dei segnali**
- **controlling terminal**
- **priorità processo** (nice)

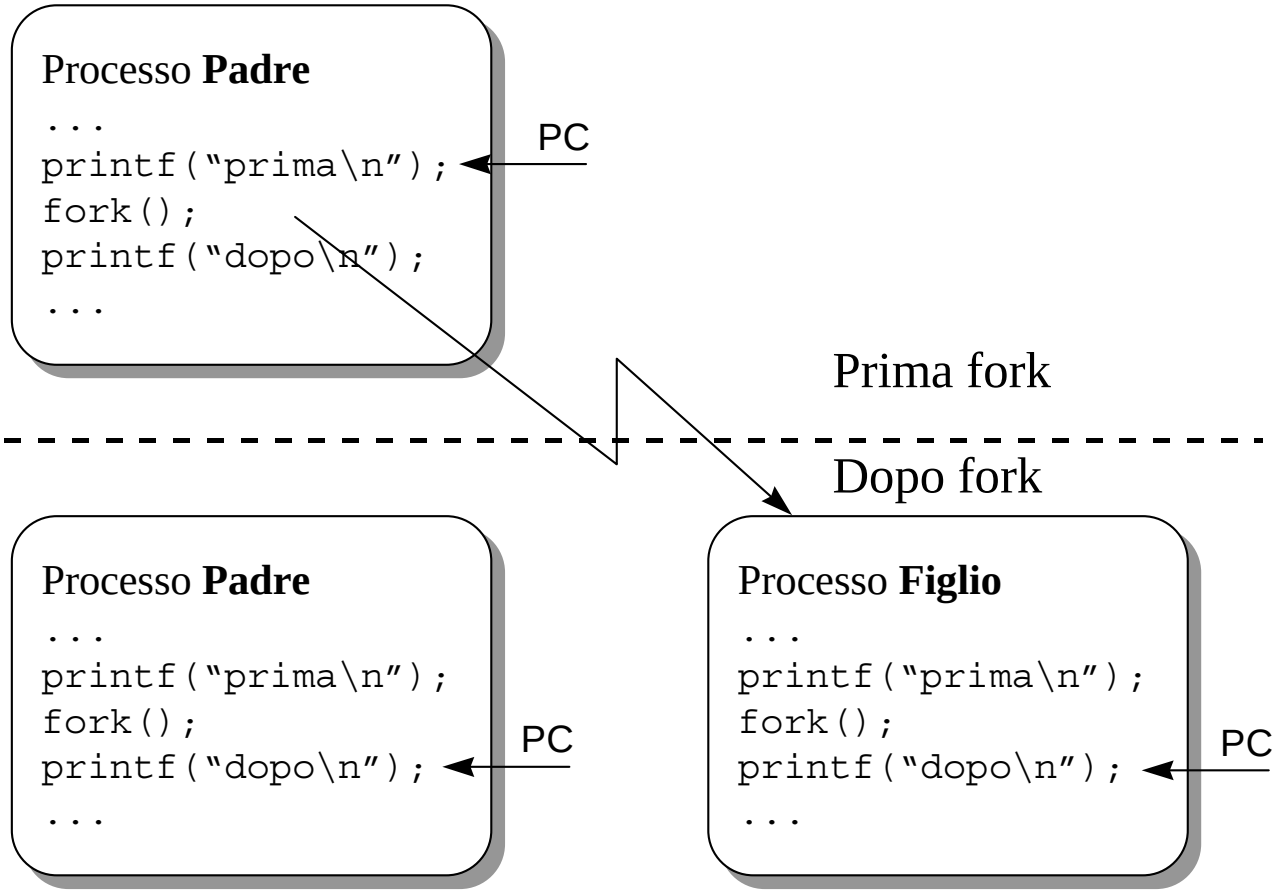
Primitive per la Gestione dei Processi

Creazione

FORK pid = **fork** ( );
pid_t pid;

Un processo ne genera un altro ➔ 2 processi concorrenti e indipendenti:

- il **parent** (processo padre), quello originario
- il **child** (processo figlio), quello generato.



Effetti della FORK

1. crea un nuovo **processo**
2. duplica i **dati** e **stack** sia **parte utente** sia **parte kernel;**
3. **stesso codice** per padre e figlio

fork restituisce l'**identificatore** del **processo creato (PID)** al padre
in caso di errore la fork restituisce al parent il valore **-1**
(limite al numero max di processi per utente e per sistema)

*Figlio eredita **tutti** attributi del processo padre, uniche **differenze** tra i due processi:*

- fork restituisce zero nel *child*, il pid del figlio nel *parent*
- il pid del child è diverso da quello del parent
- il parent pid è diverso

Schema di Generazione

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

Dopo la generazione del child il parent può decidere
se operare **contemporaneamente** ad esso
oppure
se **attendere** la sua terminazione (wait)

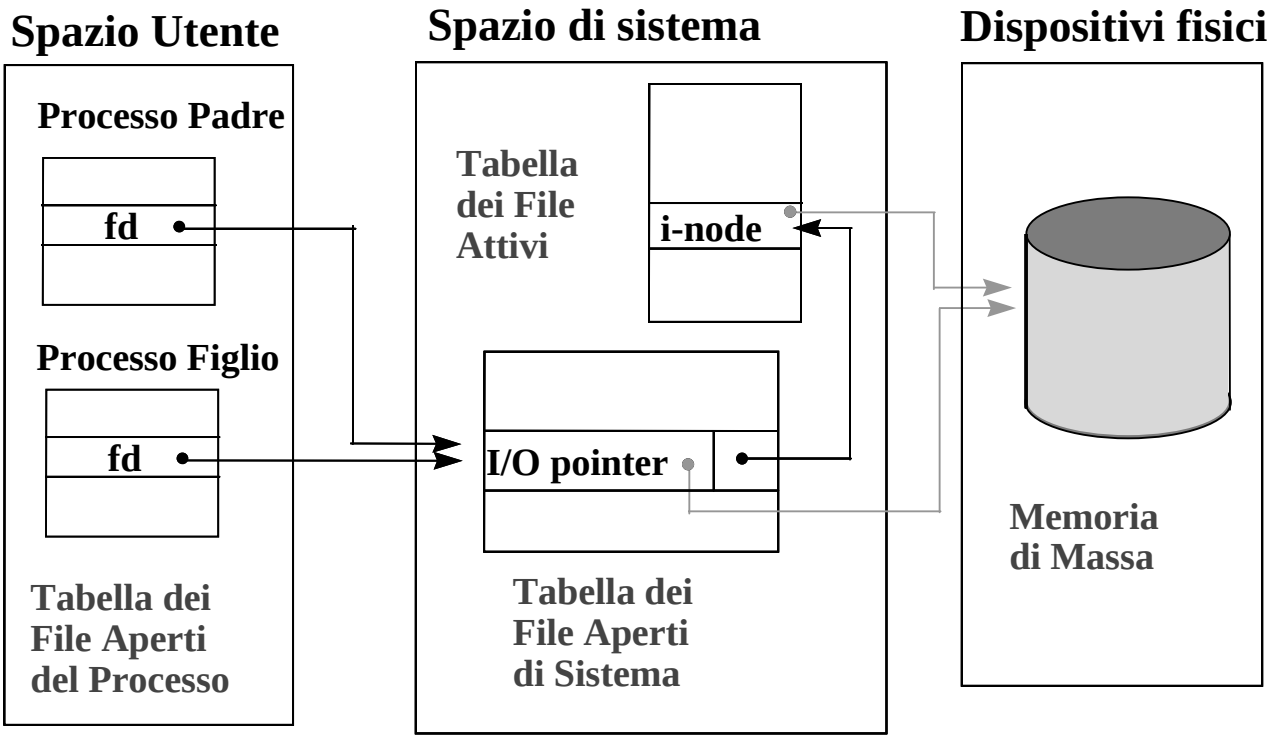
FORK: condivisione file

- **variabili e puntatori** del parent sono *copiati* e ***non* vengono *condivisi*** da parent e child ma *duplicati*.

- I fd vengono **dup**licati ➔ stessa entry nella tabella dei file aperti ➔ I **file aperti** dal processo sono ***condivisi***

  condivisi anche i **puntatori ai file** usati per I/O

  I/O pointer ***si sposta*** per entrambi in seguito a letture o scritture eseguite da una famiglia di processi.



Buffering e generazione processi

```
char buf[15]="Padre write";

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

write(1,buf, 15);

printf("Padre printf: prima della fork");

if(fork()==0){
        /* codice eseguito dal child */
        exit(0);
} else {
        /* codice eseguito dal parent */
         exit(0);
}
}
```

Quante printf vengono eseguite?

Quante stringhe `"Padre printf: prima della fork"` compaiono a video?

Provare ad aggiungere un newline nella printf ...

... e anche a ridirezionare lo standard output su file.

setbuf(stdout,NULL)

Esecuzioni differenziate del padre e del figlio
(uso di exec)

```
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);
}
```

Il figlio esegue immediatamente una **exec** e passa ad **eseguire** un altro programma

si carica il nuovo codice, i nuovi dati e tutto lo stato del nuovo programma

Si noti che il figlio **non** ritorna al suo stato precedente (GO TO)

Esecuzione di un Programma
(primitiva EXEC)

exec trasforma il processo chiamante caricando un nuovo programma nel suo spazio di memoria

- **NON** si prevede di tornare al **programma chiamante**.
- **exec non produce nuovi processi** ma solo il cambiamento dell'ambiente del processo interessato, sia come codice, sia come dati.

Sono disponibili molte funzioni della famiglia **exec**:

exec**l**, exec**le**, exec**lp**, exec**v**, exec**ve**, exec**vp**

p ➔ la funzione prende un nome di file come argomento e lo cerca nei direttori specificati in PATH;
l ➔ la funzione riceve una lista di argomenti (NULL terminata);
v ➔ la funzione riceve un vettore argv[];
e ➔ la funzione riceve anche un vettore envp[] invece di utlizzare l'environment corrente.

Alcuni esempi di exec

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

```
char  *path,   *arg0, *arg1, ..., *argn;
/* path è un pathname, assoluto o relativo*/
/* arg0 è il nome del file, seguono argomenti*/
```

EXECVE **execve** (path, argv, envp);

```
char   *path,   *argv[],   *envp[];
```

Il nuovo file può essere un file eseguibile o un file di dati per un interprete shell.

Nota: In alcuni sistemi operativi, solo una exec (execve) è una system call, le altre sono chiamate di libreria che invocano la execve.

Caratteristiche del processo dopo exec

Ci sono alcuni attributi che il processo che esegue l'exec mantiene:

- pid
- parent pid
- process gid
- session id

- real uid
- real user gid

- file mode creation mask
- current working directory
- root directory

- maschera dei segnali
- controlling terminal

- fd:
  i **file descriptor** sono ***conservati*** **MA**
  ogni descrittore di file aperto ha un flag close-on-exec,
  il default è tenerlo aperto (cambiabile con fcntl())

- Cosa succede all'effective uid ?

Sincronizzazione tra padre e figlio

WAIT pid= **wait** (&status);
int status;

```
...

if ((pid = fork()) == 0) {
   ...   /* codice eseguito dal child */
   ...
} else {
   ...   /* codice eseguito dal parent */
   ...
   wait(&status);
```

In caso di più figli **while (rid = wait (&status) != pid);**

Operazione di wait

Quando un processo termina, il kernel notifica la terminazione al processo padre (mandandogli un segnale).

Il padre riceve lo stato di uscita del figlio invocando la wait.

Il processo padre che esegue la **wait:**

- ***si sospende*** se nessun processo figlio è terminato
- ***non si sospende*** se almeno un processo figlio è terminato (zombie)

Esempio di uso della WAIT

```
/*  il figlio scrive su un file;  il padre torna all'inizio e legge */
#include <stdio.h>
#include <fcntl.h>

int procfile (f1)
char  *f1;    /* file di comunicazione */
{int nread, nwrite = 0, atteso, status, fd, pid;
 char *st1 = "              ",  st2 [80];

if (fd = open (f1, O_RDWR | O_CREAT, 0644)<0) {
     perror("open"); exit(1); }

if ((pid = fork()) < 0)  {perror("fork"); exit(1);}

if (pid == 0) {  /*  FIGLIO */
     scanf  ("%s", st1);
     nwrite = write  (fd, st1, strlen(st1));
     exit (0);
}

else {    /* PADRE */
     atteso = wait (&status);     /* attesa del figlio */
     lseek (fd, 0L, 0);
     nread = read (fd, st2, 80);
     printf ("Il padre ha letto la stringa  %s\n", st2);
     close (fd);
     return (0); }
}

main  (argc, argv)  ... {  int integi;
   integi = procfile (file1);
   exit (integi);
                              }
```

Terminazione di un processo

modo *volontario*
modo *forzato*

VOLONTARIO
- operazione primitiva **"exit"** o
- alla conclusione del programma main;

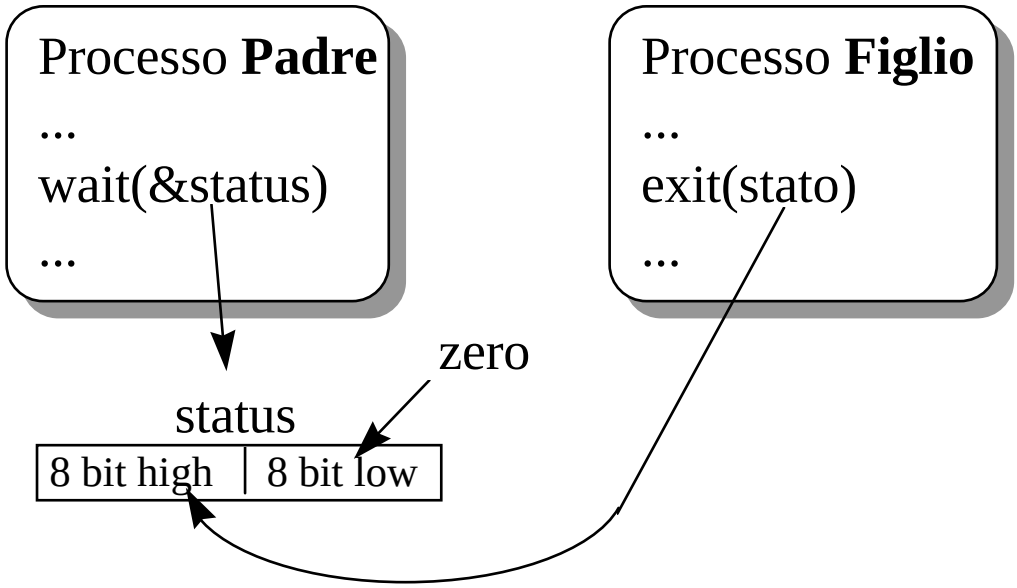
EXIT

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

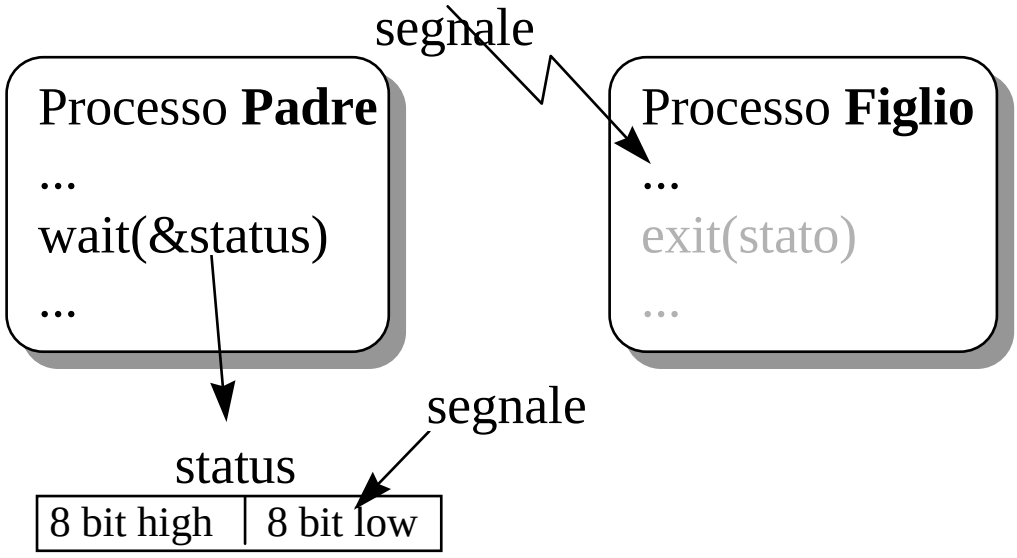
- **chiude tutti i file aperti** del processo che termina
- altre azioni tipo **scarico buffer** standard I/O library
- **status** viene passato al **processo padre**, se questo attende il processo che termina

FORZATA
a seguito di:
- **azioni non consentite** (come riferimenti a indirizzi scorretti o tentativi di eseguire codici di operazioni non definite, che generano segnali “sincroni”)
- **segnali generati dall'utente** da tastiera e ricevuti dal processo, oppure
- **segnali spediti da un altro processo** tramite la system call **kill**.

Caso di terminazione **volontaria** del processo figlio



Caso di terminazione **forzata** del processo figlio



Esempio: sincronizzazione padre-figlio

Processo Figlio:

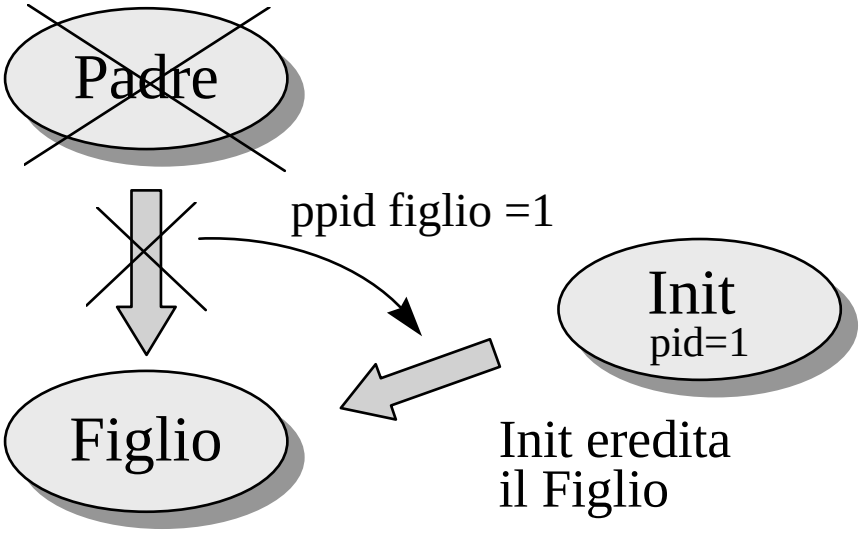
```
...
exit(status);
...
```

Processo Padre:

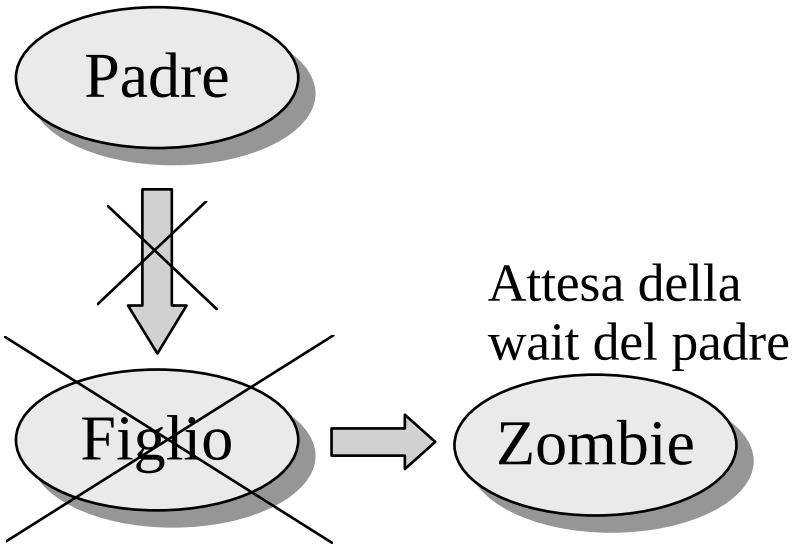
```
...
pid = wait(&stato);
printf("Padre:stato uscita figlio %d=%d\n",pid,stato);


if ((stato & 0xFF)==0) {
    /* byte meno significativo di stato e' zero");
    printf("=>Figlio terminato causa exit\n");
    stato_uscita=stato>>8;
    stato_uscita=stato_uscita&0xFF;
    printf("stato uscita di figlio %d e' %d\n",
                                pid, stato_uscita);
}
if ((stato & 0xFF)!=0) {
    /* byte meno significativo di stato
                              diverso da zero */
    printf("Figlio terminato causa segnale\n");
    stato_uscita=stato & 0xFF;
    printf("stato uscita figlio %d e' %d
          (sig ric)\n", pid, stato_uscita);
}

...
```

Parentela Processi e Terminazione

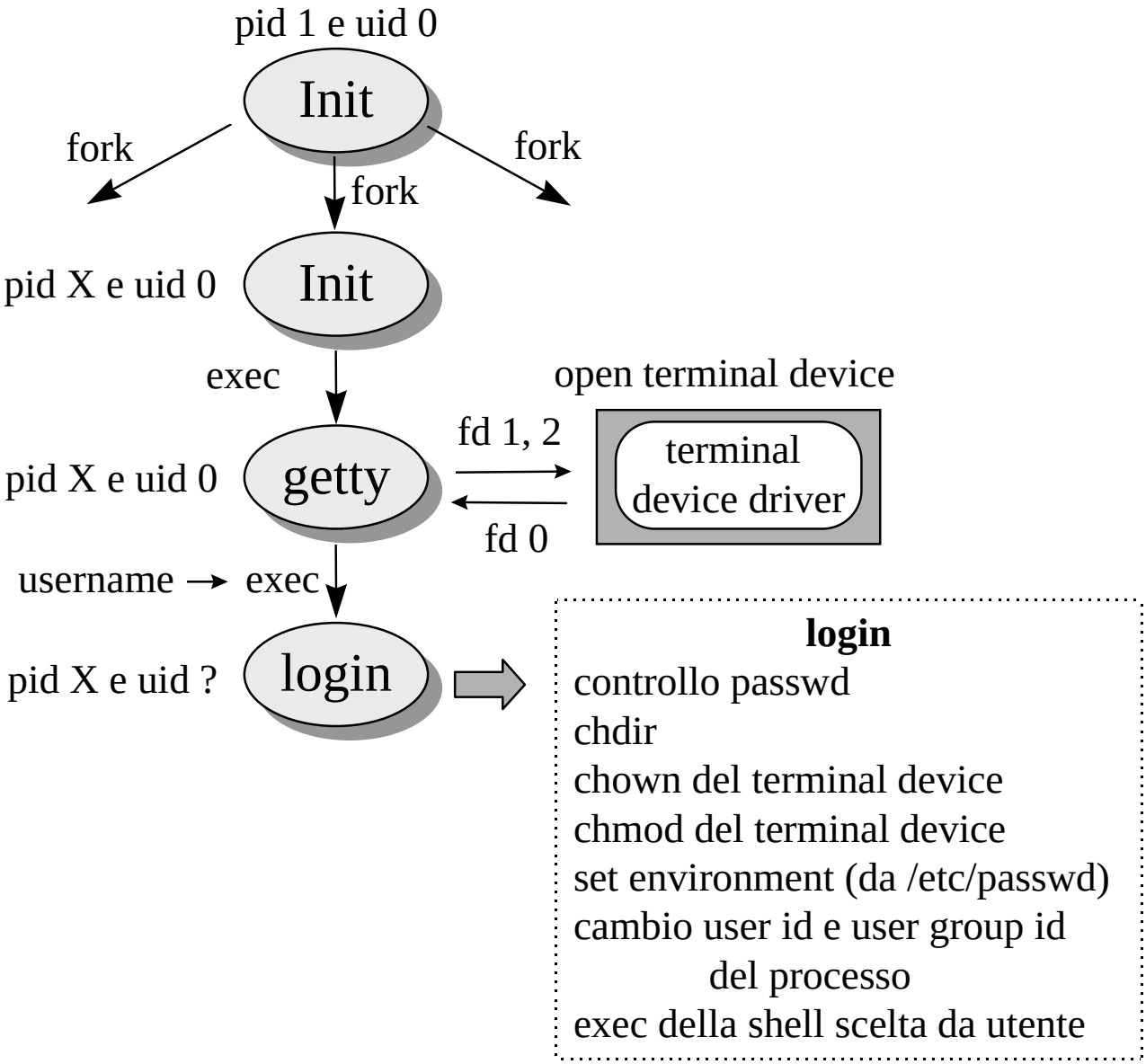
Terminazione del Padre



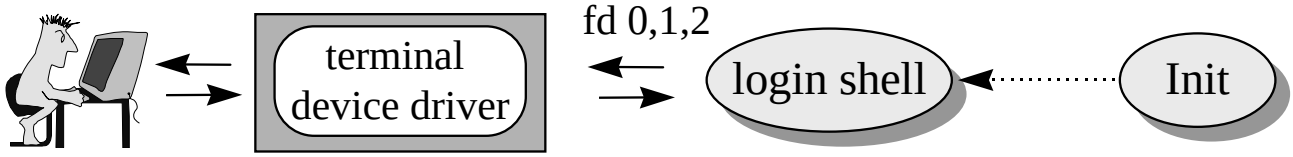
Terminazione del Figlio: Processi Zombie



Il terminal login



Situazione dei processi dopo la procedura di login



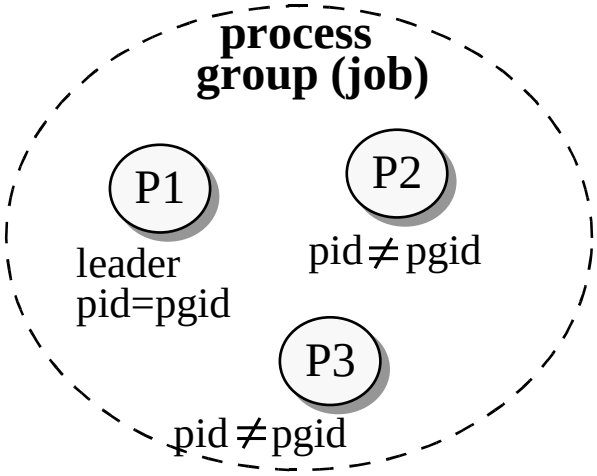
I gruppi di processi

Tutti i processi UNIX sono identificati dal **pid** e
appartengono a un **gruppo** (identificato da **pgid**)

L'insieme di processi appartenenti a uno stesso gruppo è detto anche **job**

esempio:

```
cesare lia00 ~/ 10>P1 | P2 | P3
```



In un gruppo di processi c'è un solo **group leader**,
che ha *pid* = *pgid*

Uso dei gruppi di processi :

- **segnali** mandati a tutti i processi in un gruppo
  (es. segnali da terminale)
- shell con **job control**

Primitive sui gruppi di processi

GRUPPI

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

pid_t    getpgid(pid_t pid);
```

restituisce il gruppo a cui appartiene il processo *pid* (quello del chiamante se pid=0)

```
int   setpgid(pid_t pid, pid_t pgid);
```

inserisce il processo *pid* nel gruppo *pgid*

casi particolari:

- *pid*=0 azione sul processo chiamante
- *pgid*=0 processo *pid* diventa un group leader

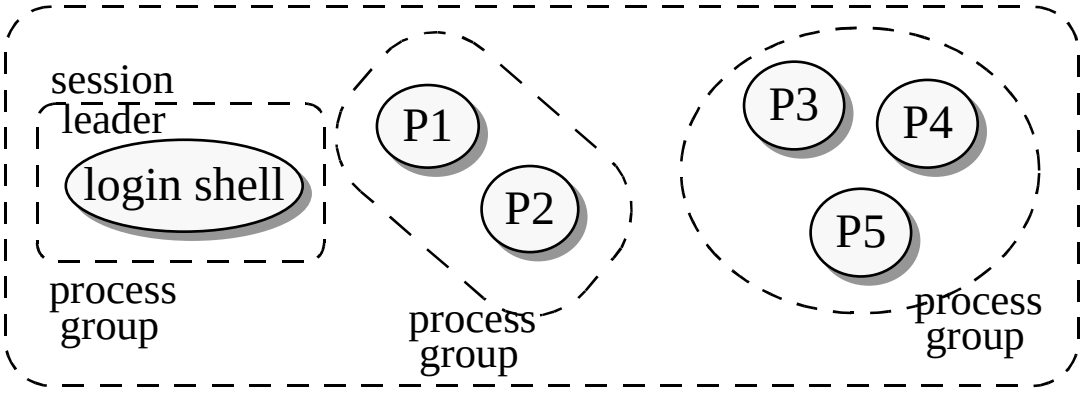
In un gruppo di processi c'è un solo **group leader**, che ha *pid* = *pgid*

Sessione

Una sessione è un insieme di più gruppi di processi

esempio:

```
cesare lia00 ~/ 10>P1 | P2 &
cesare lia00 ~/ 11>P3 | P4 | P5
```



Sessione

In una sessione ci possono essere più gruppi di processi ma un solo processo “session leader”

SESSIONE

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

pid_t    setsid();
```

trasforma il processo chiamante in un session leader e anche nel leader di un nuovo gruppo di processi. Rompe il legame con il terminale di controllo.

NB: setsid() restituisce un errore se il processo chiamante è il leader di un gruppo di processi.

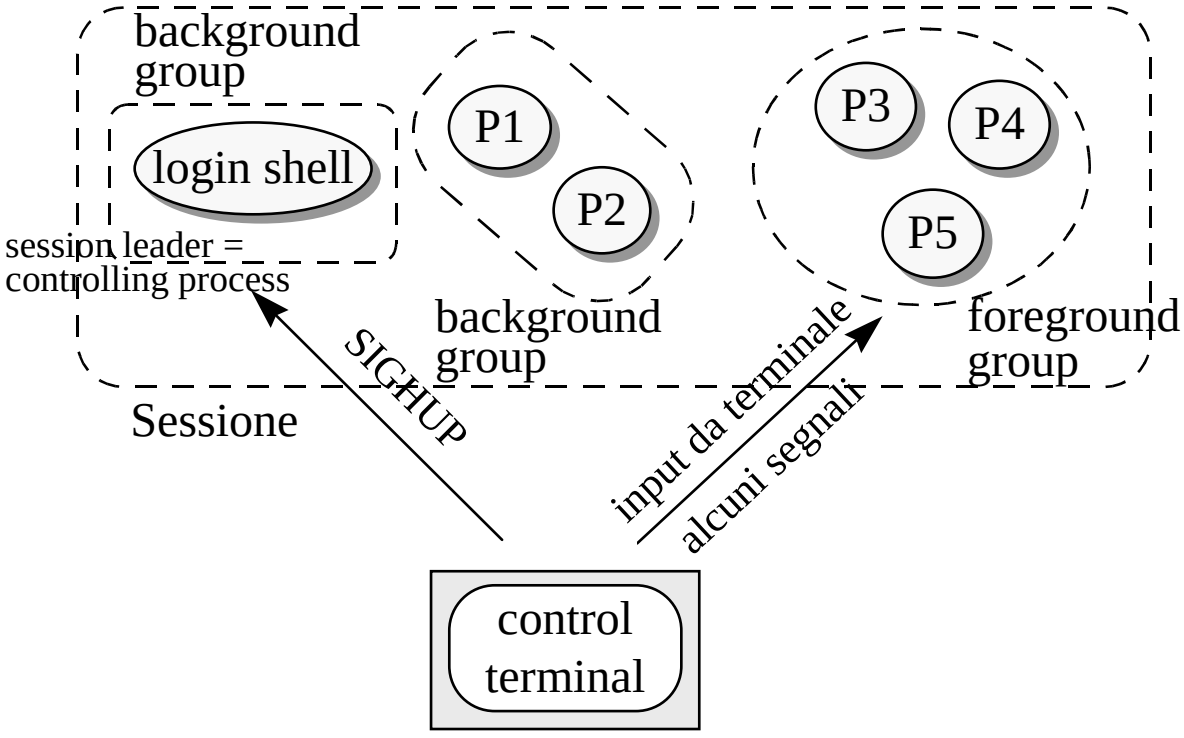
Sessione e terminale di controllo

Una sessione **può** avere
un solo terminale di controllo (terminal device)

Il terminale di controllo è associato al processo leader della sessione (detto anche *controlling process*)

In una sessione ci sono:

- 1 solo gruppo di processi in *foreground*
- più gruppi di processi in *background*



Shell con Job Control (csh)

I figli della shell di login diventano **leader** del proprio **gruppo,** ma rimangono nella stessa sessione

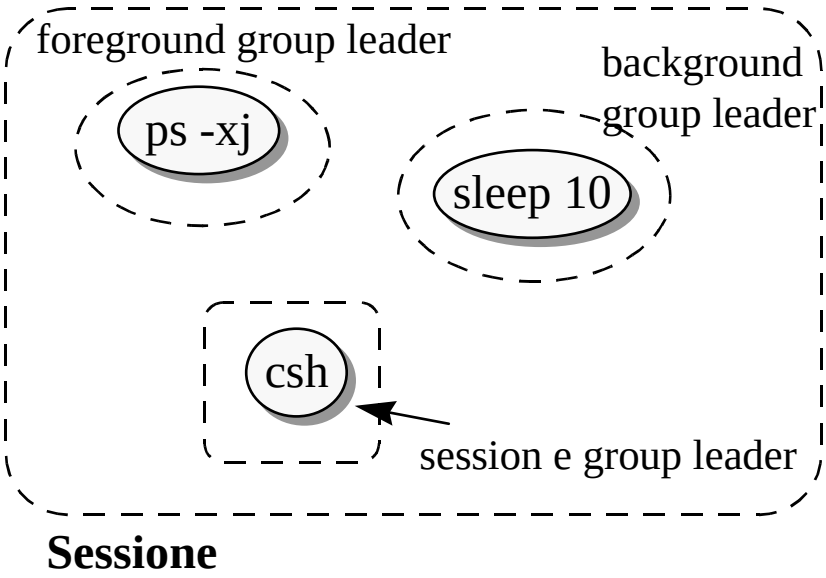
Caso BSD:

cesare deis33 ~ 34 > sleep 10 &
cesare deis33 ~ 34 > ps -xj

PPID	PID	PGID	SID	TPGID	COMMAND
1	25724	25724	25724	**29097**	csh (csh)
25724	29095	29095	25724	**29097**	sleep 10
25724	**29097**	**29097**	25724	**29097**	ps -xj

TPGID: pid processo leader gruppo foreground

SID: pid processo leader sessione



Shell senza Job Control (sh)

La shell di login è il **leader** del **gruppo** e di **sessione**
pid = pgid = sid

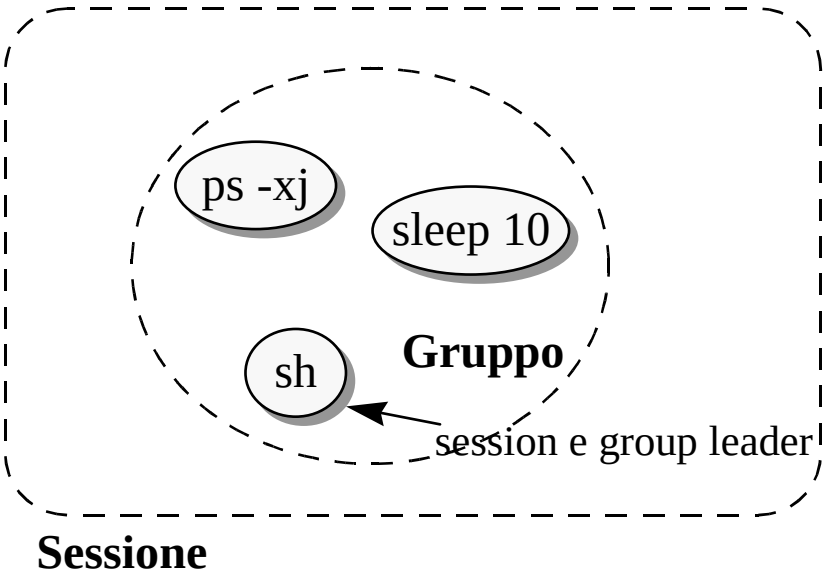
Tutti i figli appartengono a questo gruppo e hanno lo stesso pgid e sid

Caso BSD:
$ sleep 10 &
$ ps -xj

PPID	PID	PGID	SID	TPGID	COMMAND
29117	29118	**29117**	**29117**	**29117**	sleep 10
1	**29117**	**29117**	**29117**	**29117**	sh
29117	29119	**29117**	**29117**	**29117**	ps-xj

TPGID: pid processo leader gruppo foreground

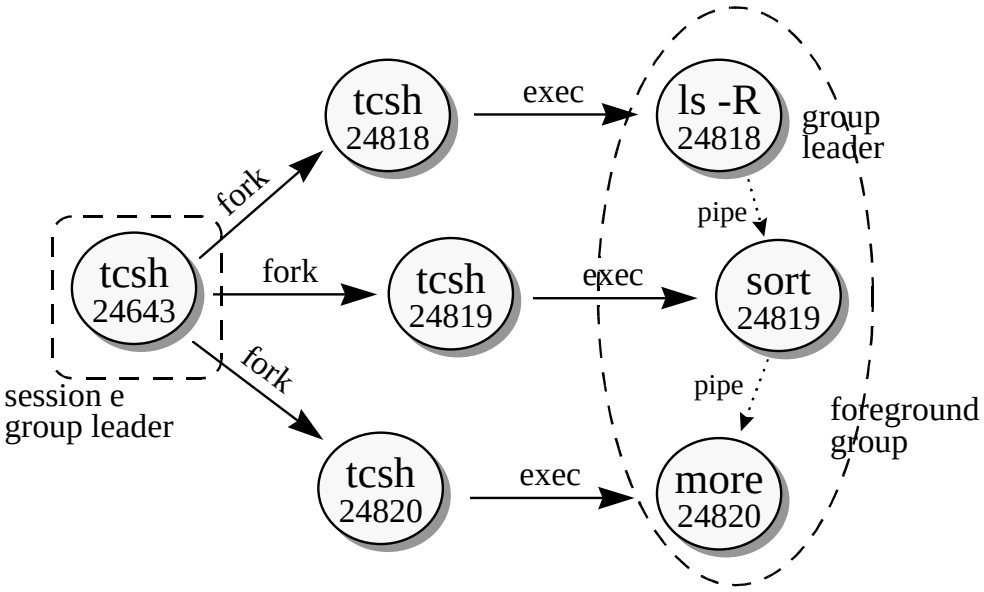
SID: pid processo leader sessione



caso tcsh:

cesare deis33 ~ 34 > **ls -R | sort |more**

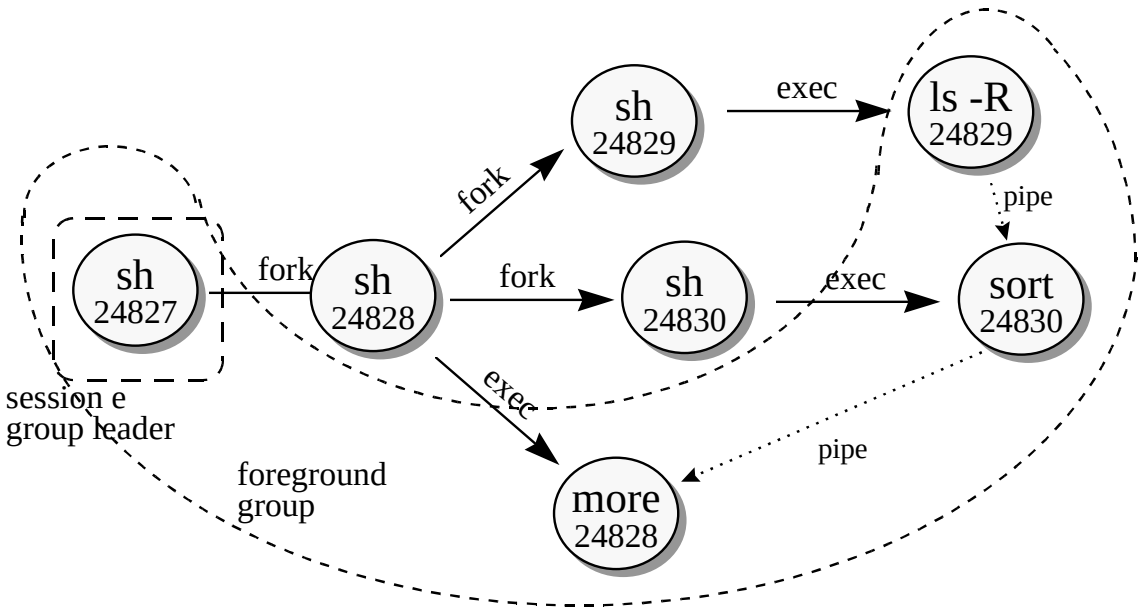
PPID	PID	PGID	SID	TPGID	COMND
24642	24643	24643	24643	24818	-tcsh
24643	24818	24818	24643	24818	ls-R
24643	24819	24818	24643	24818	sort
24643	24820	24818	24643	24818	more



caso sh:

$ ls -R | sort |more

PPID	PID	PGID	SID	TPGID	COMND
24643	24827	24827	24827	24827	sh
24827	24828	24827	24827	24827	more
24828	24829	24827	24827	24827	ls-R
24828	24830	24827	24827	24827	sort



Terminale di controllo, gruppi e sessione: strutture dati