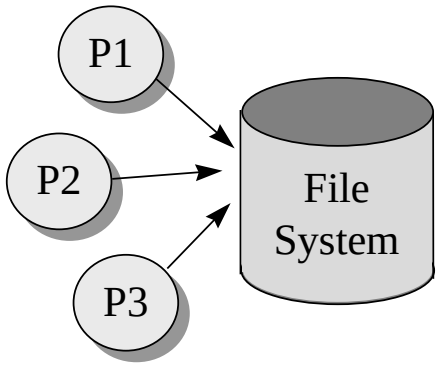
Modello di Processo in UNIX

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



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)

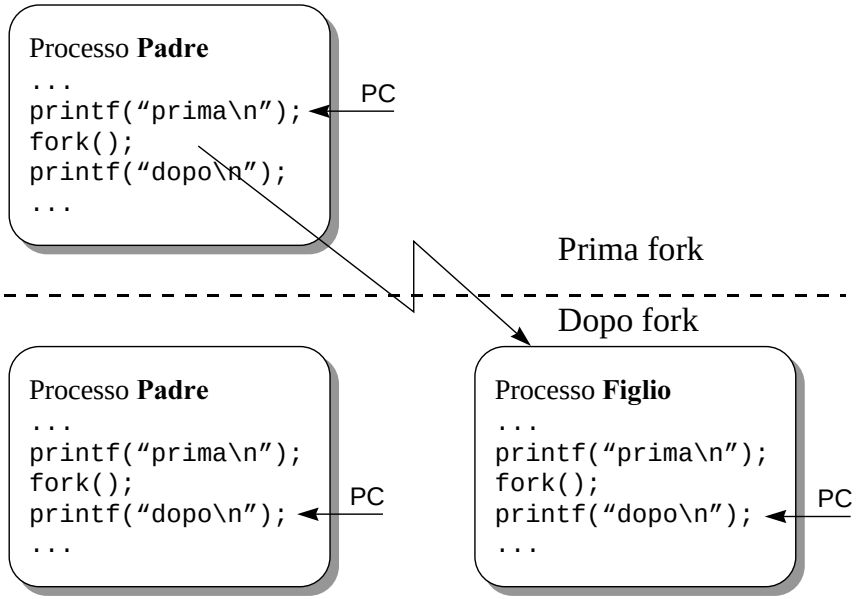
Primitive per la Gestione dei Processi

Creazione: FORK

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

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)

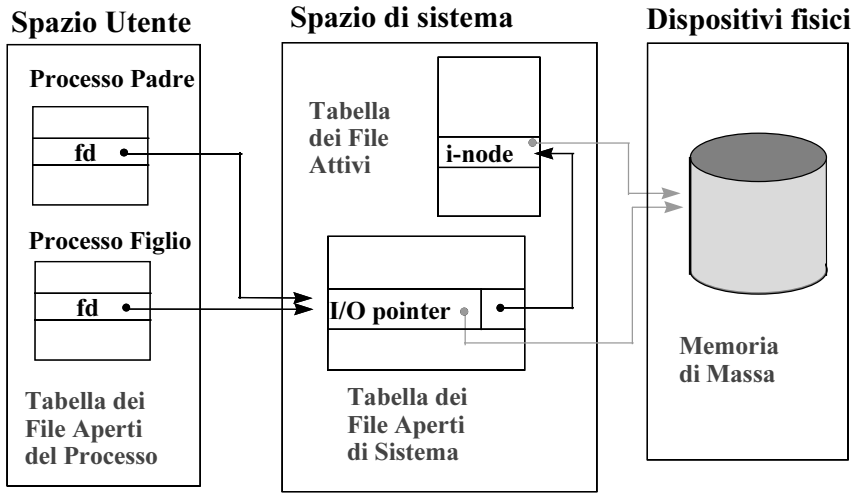
FORK: condivisione file

- **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.



Sincronizzazione tra padre e figlio
WAIT

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

Terminazione di un processo

❖ modo *volontario*
❖ modo *forzato*

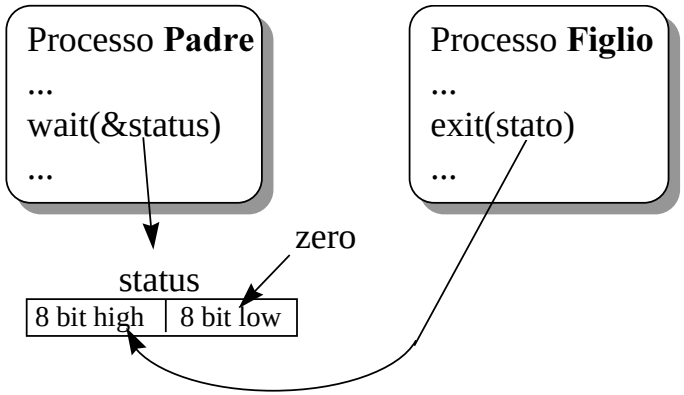
VOLONTARIO
✓ operazione primitiva **"exit"** o
✓ alla conclusione del programma main;

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

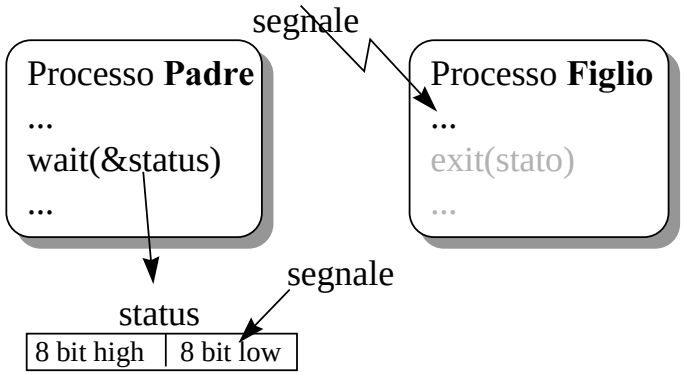
- **chiude tutti i file aperti** del processo che termina
- altre azioni tipo **scarico buffer** standard I/O library
- **status** viene passato al **processo padre**, se questo attende il processo che termina

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
✓ **segnali spediti da un altro processo** tramite la system call **kill**.

Caso di terminazione **volontaria** del processo figlio

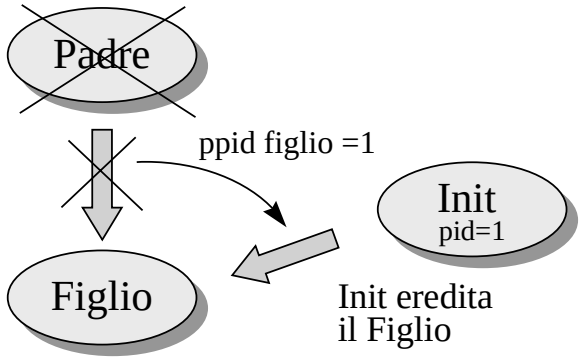


Caso di terminazione **forzata** del processo figlio

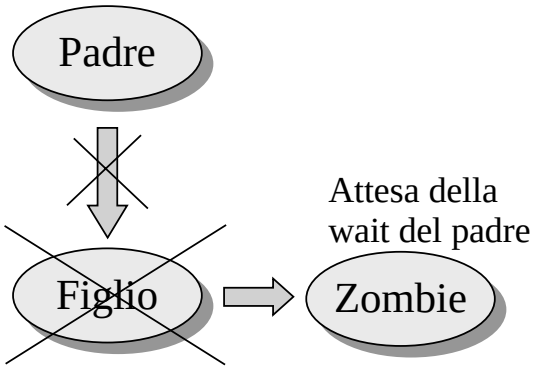


Parentela Processi e Terminazione

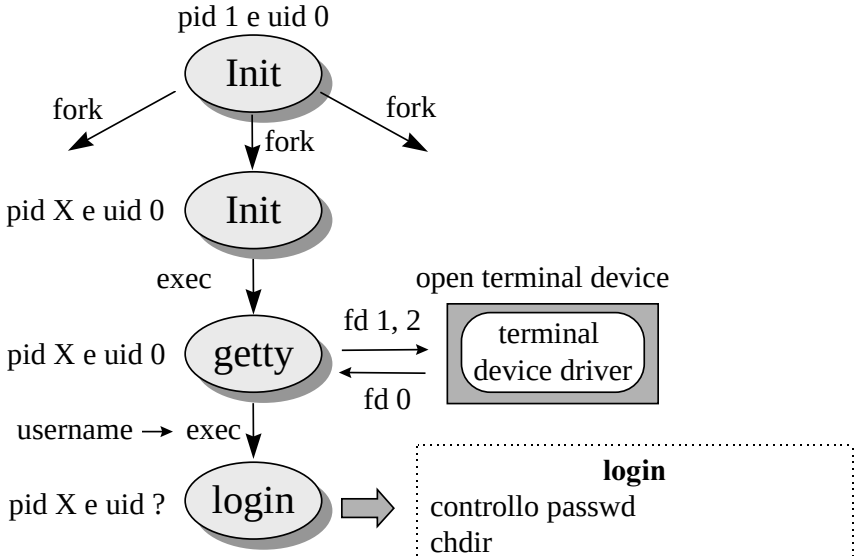
Terminazione del Padre



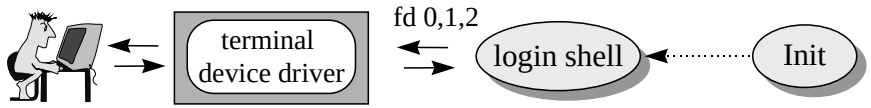
Terminazione del Figlio: Processi Zombie



Il terminal login



Situazione dei processi dopo la procedura di login



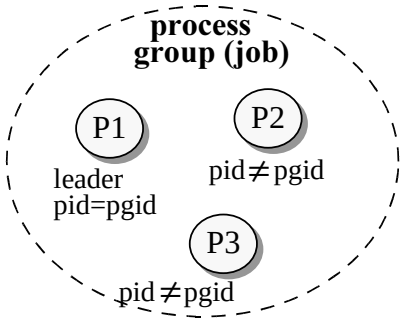
I gruppi di processi (Job)

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:

```
paolo lia00 ~/>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>
#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

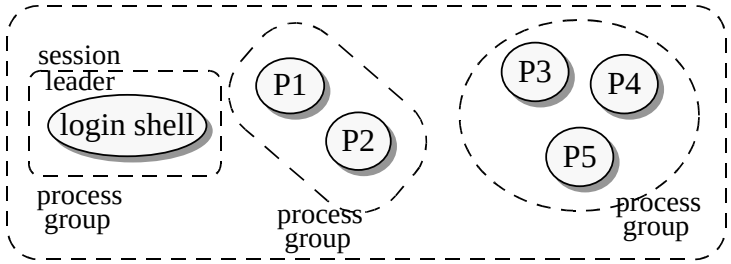
In un gruppo di processi c'è sempre un solo **group leader**, che ha *pid = pgid*

Sessione

Una sessione è un insieme di più gruppi di processi

esempio:

```
paolo lia00 ~/>P1 | P2 &
paolo lia00 ~/>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.

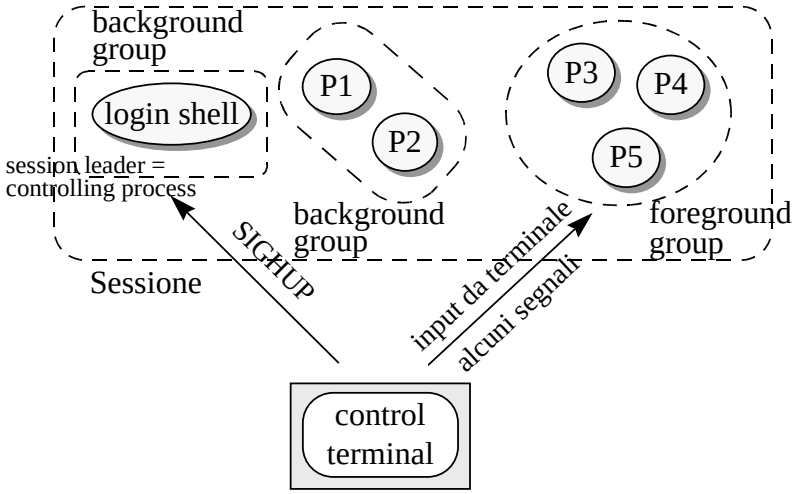
Sessione e terminale di controllo

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

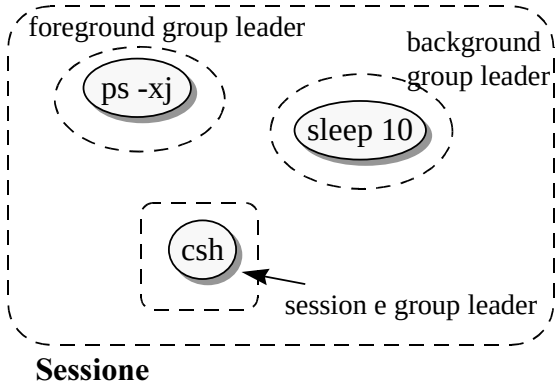
Caso BSD:

```
paolo lia00 ~ > sleep 10 &
paolo lia00 ~ > 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

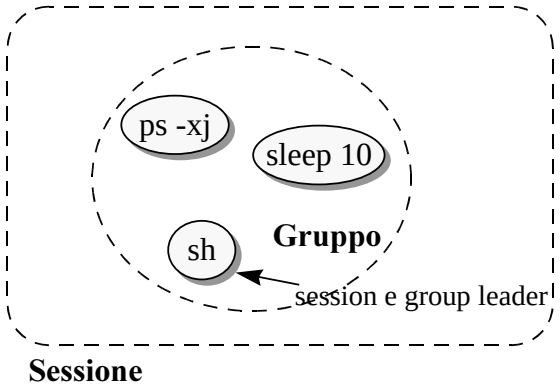
Caso BSD:

```
$ 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

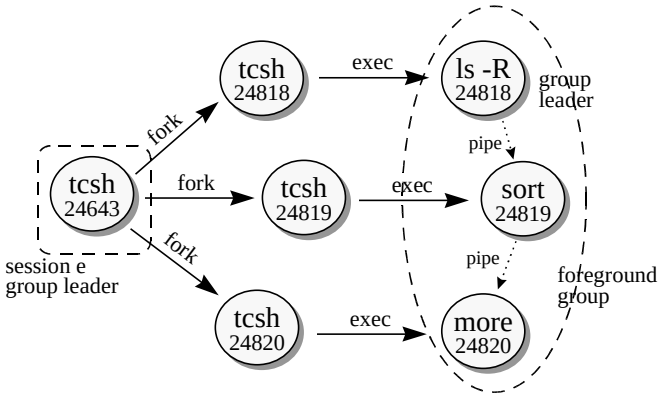
TPGID: pid processo leader gruppo foreground

SID: pid processo leader sessione



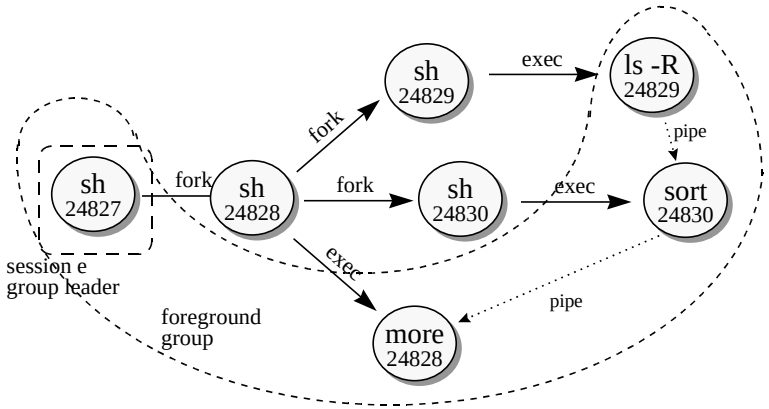
caso tcsh:

```
paolo lia00 ~ > 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
```

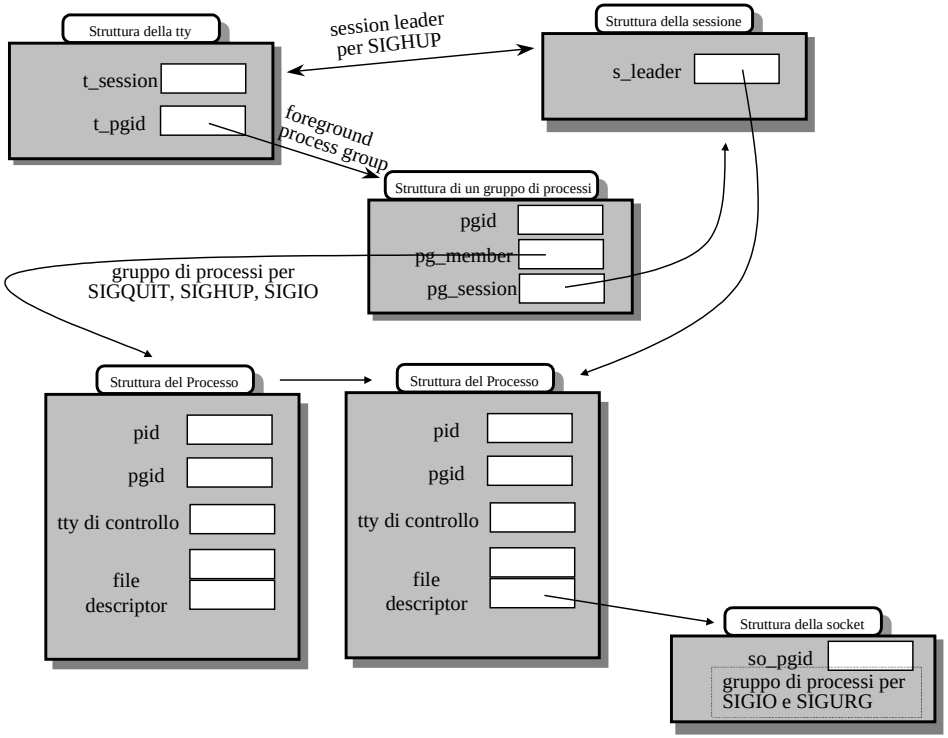


caso sh:

```
$  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



Sincronizzazione tra processi:

I Segnali

I segnali sono interrupt software, che segnalano il verificarsi di **eventi asincroni**:

- *generati da terminale* (combinazioni di tasti, es. ctrl-C)
- *eccezioni hardware* (divisioni per 0, accessi non autorizzati in memoria, ecc.)
- *spedizione esplicita di segnali* (tramite kill)
- *condizioni software* (scrittura su pipe chiusa, alarm)

Quando un processo riceve un segnale può:

1. *ignorarlo*
2. eseguire una qualche *azione* definita dall'utente
3. mettere in atto una azione di *default*

Attenzione: i segnali hanno tutti la **stessa priorità**

Esempi

1) # programma
   .....
   <CTRL>-C
   segnale SIGINT al processo
   azione default ➔ terminazione

2) # programma
   .....
   Illegal instruction - core dumped
   programma (corrotto) ha eseguito una istruzione illegale
   segnale SIGILL al processo
   azione default ➔ terminazione
   con produzione di un file core
   core ➔ immagine del processo

3) $ programma &
   <PID>
   $ ...
   $ kill <PID>
   segnale SIGTERM al processo
   azione default ➔ terminazione
   <PID> Terminated

4) $ programma &
   <PID>
   $ kill -9 <PID>
   segnale SIGKILL al processo
   (non ignorabile, né gestibile)
   azione default ➔ terminazione
   <PID> Terminated

Elenco dei Segnali (file /usr/include/sys/signal.h)

Name	Value	Default	Event
SIGHUP	**1**	**Exit**	**Hangup**
SIGINT	**2**	**Exit**	**Interrupt**
SIGQUIT	**3**	**Core**	**Quit**
SIGILL	**4**	**Core**	**Illegal Instruction**
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	**9**	**Exit**	**Killed**
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	**12**	**Core**	**Bad System Call**
SIGPIPE	**13**	**Exit**	**Broken Pipe**
SIGALRM	**14**	**Exit**	**Alarm Clock**
SIGTERM	**15**	**Exit**	**Terminated**
SIGUSR1	**16**	**Exit**	**User Signal1**
SIGUSR2	**17**	**Exit**	**User Signal2**
SIGCHLD	**18**	**Ignore**	**Child Status Changed**
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	**21**	**Ignore**	**Urgent Socket Condition**
SIGIO	22	Ignore	I/O Event
SIGSTOP	**23**	**Stop**	**Stopped**
SIGTSTP	24	Stop	Stopped
SIGCONT	25	Ignore	Continued
SIGTTIN	**26**	**Stop**	**Stopped(tty input)**
SIGTTOU	**27**	**Stop**	**Stopped(tty output)**
SIGPROF	29	Exit	Profiling
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGLWP	33	Ignore	Inter-LWP signal reserved

Attenzione: dipende dalle macchine (vedere diff. BSD System V)

Azioni al ricevimento di un segnale

Quando riceve un segnale, un processo può avere diversi comportamenti:

1. *ignorare* il segnale (solo 2 segnali NON possono essere ignorati, il SIGKILL e il SIGSTOP)
2. *gestire* il segnale (il processo può definire una funzione da mettere in esecuzione nel momento che riceve un determinato segnale)
3. mettere in atto l'azione di *default* (ogni segnale ha un'azione di default, es. la terminazione del processo)

Come specificare l'azione?

SIGNAL

#include <signal.h>

void (***signal** (int sig, void (*func)(int)))(int);

definisce il comportamento del processo al ricevimento del segnale. Specifica il segnale (**sig**) e il suo trattamento (**func**). Restituisce il trattamento precedente del segnale sig oppure -1.

1. per *ignorare* il segnale ➔ SIG_IGN
2. per *gestire* segnale attraverso handler ➔ void (*func)()
3. per riportare all'azione di *default* ➔ SIG_DFL

Notare che:

```
#define   SIG_DFL     void (*)() 0
#define   SIG_IGN     void (*)() 1
#define   SIG_ERR     void (*)() -1
```

Segnali e fork()

Un processo figlio eredita tutta la disposizione dei segnali del padre:

- *ignora* gli stessi segnali ignorati dal padre
- *gestisce* nello stesso modo i segnali gestiti dal padre
- stessa azione di *default* del padre

Segnali e exec()

exec cambia la disposizione dei segnali:

- segnali *gestiti* prima di exec ritornano alla condizione di *default*
- processo dopo exec *ignora* gli stessi segnali ignorati prima

Perché è stata fatta questa scelta?

Segnali unreliable

Molte implementazioni UNIX hanno segnali non affidabili (*unreliable*)

Problemi:

⇒ **segnale può essere perso** e non ricevuto da un processo

⇒ se più occorrenze "contemporanee" di uno **stesso** segnale sono dirette a **un** processo, tale processo ne riceve solo **una**, le altre NON sono registrate

⇒ **scarso controllo sulla gestione** dei segnali, esempio: gestore segnale non rimane installato

⇒ possibile **innestamento** della routine di gestione del segnale (stack)

⇒ possibile **interruzione delle system call**

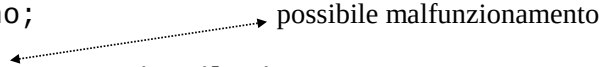
Caso System V

- gestore segnale non rimane installato (riportato a default)
- anche nel caso si riinstalli possibile malfunzionamento
- non blocca il segnale durante la sua gestione
- possibile innestamento multiplo delle routine di gestione

Esempio:

```
handler(signo) /* routine gestione segnale*/
int signo;
{
 signal(SIGINT, handler);
 printf ("handler: signo=%d\n", signo);
}


main()
...
signal (SIGINT, handler); /*aggancia segnale*/
...
```



Caso BSD

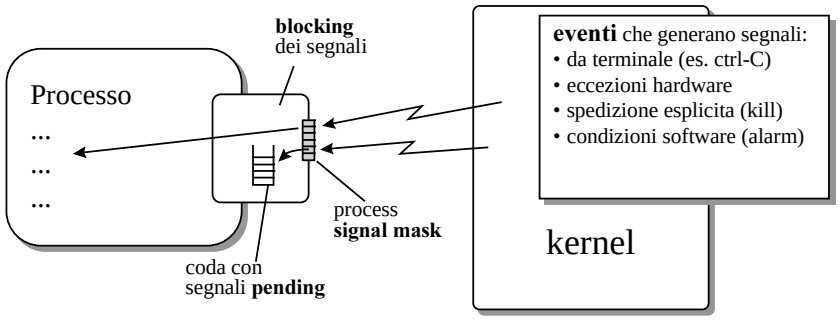
- gestore segnale rimane installato
- durante la routine di gestione del segnale, il segnale stesso è bloccato (cioè bufferizzato)
- occorrenze "contemporanee" di uno **stesso** segnale ➔ processo ne riceve solo **una**

Esempio:

```
handler(signo) /* routine gestione segnale*/
int signo;
{
printf ("handler: signo=%d\n", signo);
}


main()
...
signal (SIGINT, handler); /*aggancia segnale*/
...
```

Modello reliable segnali



- occorrenze "contemporanee" di uno **stesso** segnale sono memorizzate:
  - ➔ processo le riceve **tutte** (1 contatore per ogni segnale)
  - ➔ non si perdono segnali
- è possibile **bloccare** un segnale diretto a un processo P, e gestirlo successivamente ➔ regioni critiche

I segnali in attesa sono detti **pending**

La **process signal mask** definisce quali segnali bloccare per un processo P (i segnali sono messi in attesa)

Interrupted System Call

Alcune system call possono essere molto lente o bloccarsi (I/O su terminali o via rete, non su file).
È possibile interrompere le system call.

System call interrotte da un segnale ritornano **-1**
ed **errno** vale **EINTR**

Comportamento delle system call interrotte varia a seconda della implementazione:

- applicazione deve esplicitamente trattare il caso

```
do {
   read(0,buf,20)
} while (errno==EINTR)
```

- *automatic restart* delle system call
- scelta esplicita del comportamento per ogni segnale

Altre Primitive

Sospensione temporizzata
SLEEP
#include <unistd.h>
unsigned int **sleep** (numsecondi);
unsigned int numsecondi;

Sospende il processo per numsecondi.
Il processo può **ripartire** prima di numsecondi se riceve un **segnale**.

Installazione di un allarme
ALARM
#include <unistd.h>
unsigned int alarm (numsecondi);
unsigned int numsecondi;

Fa partire un **timer**. Dopo numsecondi, il kernel notifica al processo chiamante un segnale **SIGALRM**.
Un solo timer per processo: nel caso di due chiamate successive il valore della seconda sovrascrive il primo.

Sospensione in attesa di un qualunque segnale
PAUSE
#include <unistd.h>
int pause (void);

Sospende il processo chiamante fino al ricevimento di un qualunque segnale ➔ ritorna **sempre** -1 ed errno=**EINTR**

Altre Primitive

Invio di segnali
KILL
```
#include <signal.h>
#include <sys/types.h>
int kill (pid,signo);
pid_t pid;
int signo;
```

`kill` invia il segnale `signo` ad un processo oppure ad un gruppo di processi.

Quattro casi a seconda del valore di pid:

- **pid == 0** il segnale viene mandato a tutti i processi che appartengono allo stesso gruppo del processo che manda il segnale.
- **pid == -1** e **l'effettivo user-id** del processo non è super-user, il segnale è mandato a tutti i processi con user-id reale uguale all'effettivo user-id del processo che invia il segnale.
- **pid == -1** e **l'effettivo user-id** del processo è super-user, il segnale è mandato a tutti i processi (escluso alcuni processi di sistema).
- **pid < -1** il segnale è inviato a tutti i processi il cui **p**rocess **g**roup-**id** è uguale in valore assoluto a pid (eventualmente incluso il processo che invia il segnale).

Corse Critiche

Corsa critica: situazione in cui più processi modificano dei dati condivisi. ma l'esito finale dipende dall'**ordine relativo non predicibile** con cui sono eseguite le operazioni dai processi.

Molte situazioni di corse critiche si possono risolvere utilizzando dei meccanismi di sincronizzazione e comunicazione tra i processi tipo segnali, pipe, etc.

Corse critiche anche nell'uso di **segnali**!!

Esempio: alarm e pause (sleep)

```
....
alarm(nsec);
pause();
....
```

Corsa critica: il SIGALRM potrebbe arrivare al processo PRIMA della `pause ()` ➔ blocco del processo
(in dipendenza dal carico della macchina, dal numero `nsec` di secondi, etc.)

Segnali in Solaris 2.5 (System V)

Per **bloccare** i segnali:
SIGHOLD
#include <signal.h>
int **sighold** (int sig);

Aggiunge il segnale `sig` alla process signal mask.
Il segnale `sig` diretto al processo è **bloccato**, cioè bufferizzato.
Un `sig` bloccato raggiungerà il processo quando sbloccato tramite `sigrelse()` oppure `sigpause()`

Per **sbloccare** i segnali:
SIGRELSE
#include <signal.h>
int **sigrelse** (int sig);

Rimuove il segnale `sig` dalla *process signal mask*

SIGPAUSE
#include <signal.h>
int **sigpause** (int sig);

Rimuove il segnale `sig` dalla process signal mask e sospende il processo (equivale a sigrelse + pause)

Esempio di corsa critica

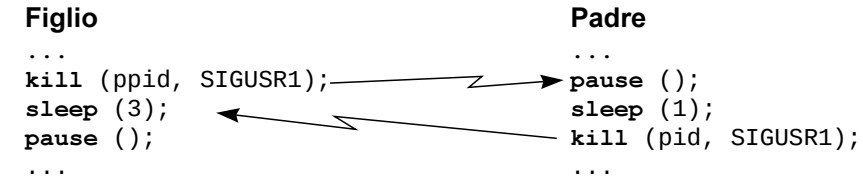
```
....
alarm(nsec);
pause();
....
```

il SIGALRM potrebbe arrivare al processo

Soluzione senza corsa critica

```
....
sighold(SIGALRM);
alarm(secs);
sigpause(SIGALRM);
....
```


Esempio: padre e figlio si scambiano il controllo alternativamente, utilizzando kill e pause



La sleep(3) nel figlio simula una condizione in cui si verifica una corsa critica, figlio riceve SIGUSR1 dal padre e DOPO va in pause (blocco)

Soluzione che evita la corsa critica (Solaris 2.5)

```
sigset(SIGUSR1,handler); /*aggancio di SIGUSR1 */
if ((pid=fork())==0) { /* figlio */
      sighold(SIGUSR1);
      .............
      printf("Figlio \n");
      kill (ppid, SIGUSR1); /* invio al padre */
      sleep(3);
      sigpause(SIGUSR1); /*attesa del segnale*/

} else { /* Padre*/
      sighold(SIGUSR1);
      ..............
      printf ("Padre\n");
      sigpause(SIGUSR1); /*attesa del segnale*/
      kill (pid, SIGUSR1); /* invio al figlio */
      ...........
}
```

Per qualunque numero di secondi di sleep non si verificano corse

Comunicazione tra Processi in UNIX

Processi UNIX:

- definiscono un ambiente ***locale***
- ***nessuna risorsa condivisa*** a parte file e codice

Motivazioni:
sistemi multi-utente e multi-processo
- non interferenza tra utenti
- non interferenza tra processi

sistema più "**safe**"

Può essere importante far "interagire" processi diversi
- comunicazione
- risorse condivise
- sincronizzazione

Ad esempio, molti processi "**demoni**" devono poter interagire con altri processi del sistema

☑ **IPC** (inter-process communication)

Modalità di IPC in UNIX

✓ ***Segnali:*** sincronizzazione

✓ ***File System:*** comunicazione, condivisione di risorse

✓ ***Pipe:*** comunicazione

✓ ***FIFOs:*** comunicazione

✓ ***Spazio dei nomi:*** comunicazione, condivisione di risorse

FILE SYSTEM

Il file system in UNIX è unico per tutti i processi, quindi prevede uno **spazio dei nomi unico**

possibilità di comunicare leggendo/scrivendo sullo stesso file

Problemi:

- sincronizzazione (accessi ai file non sincronizzati)
- costo (di accesso al file system)

PIPE

Le pipe sono **canali di comunicazione unidirezionale** tra processi

```
int P[2];

pipe(P)
pid=fork();
```

```
/* padre */
write(P[1], buffer, nw);
```

```
/* figlio */
read(P[0], buffer, nr);
```

Problemi delle pipe come canale di comunicazione:

- solo tra processi imparentati
- meccanismo poco flessibile

FIFO (named pipe)

Meccanismo di comunicazione che segue il modello delle pipe, ma è associato ad un **nome unico (pathname) nel file system**

Comunicazione e sincronizzazione tra
processi non imparentati

```
/* creazione */
mknod("pathname", PERMS | S_IFIFO);

dopo di che si usa esattamente come un file:

                open, read, write, close

/* distruzione */
unlink(nomefifo);
```

Note:

- scrittura bloccata in assenza di lettore
- lettura bloccata in caso di canale vuoto

Problemi:

- come pipe, meccanismo poco flessibile

PIPE/FIFO vs FILE

✓ al file descriptor delle pipe/FIFO non corrisponde un oggetto residente nel file system

➡ la pipe/FIFO è una struttura che **non permane** alla terminazione del processo

✓ alla pipe/FIFO è associato un buffer di **dimensioni fisse**

➡ l'atomicità delle write su una pipe/FIFO è garantita solo per operazioni che non eccedono la dimensione di questo buffer

✓ la pipe/FIFO usa una gestione first-in-first-out

➡ non c'è un IO pointer (no lseek)

Spazio dei Nomi: KEY

Le keys sono numeri usati per identificare una risorsa rendendola condivisibile tra vari processi

Tipo di dato di una chiave: **key_t**

- definito nel file "types.h"
- in Unix System V equivale a un long int

Si genera a partire da un

- pathname (char *)
- progetto (int)

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

key_t keyval, ftok();
char *pathname, progetto;
…
keyval = ftok(pathname, progetto);
```

Per ogni pathname, si possono generare una serie di canali di IPC, ognuno con associato il suo nome.

Deve esistere un ***accordo preventivo*** fra i processi sul ***pathname***

System V IPC (1)

Usando le chiavi i processi possono condividere

- **code di messaggi**
- **semafori**
- **segmenti di memoria**

In generale, i passi d'uso sono:

1. Inclusione file:
   - ✓ *sys/types.h e sys/ipc.h* (in ogni caso)
   - ✓ *sys/msg.h, sys/sem.h o sys/shm.h*
     (secondo la risorsa che si vuole condividere)

2. Ottenimento della chiave:
   - ✓ *key_t ftok(...)*

3. Chiamata per creare (server) o aprire (client) l'oggetto condiviso:
   - ✓ *msgget(...), semget(...), shmget(...)*
     (secondo il tipo di risorsa condivisa, restituiscono un identificatore intero che si usa come un file descriptor)

System V IPC (2)

4. Eventuale chiamata per operazione di controllo (inizializzazione e setting particolare):
   - ✓ *msgctl( ), semctl( ), shmctl( )*

5. Accesso all'oggetto
   - ✓ *msgsnd( ), msgrcv( )*
     (classiche send e receive con diverse opzioni)
   - ✓ *semop( )*
     (funzionamento poco intuitivo; punto di partenza di basso livello per realizzare wait() e signal())
   - ✓ *shma( )t, shmdt( )*
     (localmente corrispondono circa alla malloc() e alla free())

6. Chiamata di controllo di epilogo (operazioni di chiusura)

Esempio: Shared Memory

Lato Server	Lato Client
#include <sys/shm.h> #include <sys/ipc.h>	#include <sys/shm.h> #include <sys/ipc.h>
long key; int shid; char *ptr;	long key; int shid; char *ptr;
key = ftok(“pp”, pro1);	key = ftok(“pp”, pro1);
/* entrambi i processi entrano in possesso del nome unico*/	
shid = shmget(key, SIZE, PERMS|IPC_CREAT); /* creata la shared memory */	/* prima deve essere creata dal server poi il cliente la può aprire */ shid=shmget(key, SIZE, 0);
ptr=shmat(shid, 0, flag);	/* aperta shared memory */ ptr=shmat(shid, 0, flag);
/* puntatore alla memoria accesso con i puntatori */ /* NOTA: no sincronizzazione accessi. USARE SEMAFORI */	

shmdt(ptr);
/* free locale, non chiusura e deallocazione globale: serve la chiamata di controllo*/
shmctl(...);
	shmdt(prt);

I Processi Demone

Un processo **demone** è un processo che **non** è associato a un terminale di controllo e fornisce un servizio:

- ad intervalli di tempo prefissati
- in risposta ad un evento

Hanno in generale un tempo di vita molto lungo, spesso coincidente col tempo di funzionamento del sistema

Esempio: demone di stampa, inetd, telnetd, sendmail, cron...

ps -ef, caso BSD:

```
PPID  PID   PGID  SID   TT  TPGID  UID  COMMAND
0     0     0     0     ?   -1     0    swapper
0     1     0     0     ?   -1     0    /sbin/init
0     2     0     0     ?   -1     0    pagedaemon
1     55    55    55    ?   -1     0    portmap
1     60    41    41    ?   -1     0    keyserv
1     78    74    74    ?   -1     0    (biod)
1     89    89    89    ?   -1     0    syslogd
1     97    97    97    ?   -1     0    /usr/lib/sendmail
1     103   103   103   ?   -1     0    (nfsd)
1     104   104   104   ?   -1     0    rpc.mountd
1     115   115   115   ?   -1     0    rpc.statd
1     117   117   117   ?   -1     0    rpc.lockd
1     131   131   131   ?   -1     0    cron
1     136   136   136   ?   -1     0    inetd
1     3016  3016  3016  ?   -1     0    /usr/lib/lpd
136   5147  5147  5147  ?   -1     0    in.rlogind
136   6909  6909  6909  ?   -1     0    rpc.rusersd
136   7455  7455  7455  ?   -1     0    in.telnetd
136   8859  8859  8859  ?   -1     65   in.fingerd
1     7033  7033  7033  co  7033   0    console(getty)
7341  7799  7799  7341  p2  7799   230  ps-ef
```

I Processi Demone

Un processo demone può essere messo in **esecuzione**:

- al ***boot*** del sistema (file `/etc/rc` eseguito da processo init)
- dal demone ***cron***, che consulta lo script
  - ✓ `/usr/lib/crontab` di sistema
  - ✓ `crontab` di utente (solo System V)
- dal comando ***at***
- dal ***terminale utente***

Il progetto di un demone deve portare a un funzionamento corretto qualunque sia il modo scelto per la messa in esecuzione del demone, per cui bisogna considerare l'interazione del processo demone con:

- i segnali
- il terminale
- il gruppo di processi

Progetto di un demone

- esecuzione di una **fork** con terminazione del padre, per:
  - mettere in ***background*** il figlio restituendo il controllo alla shell (considerare il caso in cui il demone venga lanciato da uno script `/etc/rc`)
  - rendere il figlio ***non leader*** di un gruppo (prerequisito per `setsid()`)

- esecuzione di **setsid**() per **sganciare il processo dal terminale di controllo** e **staccarsi dal gruppo del processo padre** (il processo diventa leader di una nuova sessione non collegata a nessun terminale)

- **chdir("/")** per cambiare la directory ed andare in / (non si deve bloccare un eventuale filesystem montato), oppure **chdir("/var/spool")** nel caso di demoni che lavorino in specifiche directory del filesystem

- file mode creation mask a 0 (**umask**(0)), per impedire la modifica dei bit di accesso di qualsiasi file creato dal demone

- chiusura di tutti i file descriptor

- installare un **gestore** del SIGTERM per una terminazione graceful e controllata delle sue operazioni (termine processi: SIGTERM e dopo alcuni secondi SIGKILL)

- non lasciare figli zombie:
  - raccogliere lo stato di terminazione oppure
  - generare figli segnalando al kernel di non essere interessati allo stato di ritorno dei figli stessi

Processi e terminale di controllo

```
.....
pid=fork();
if(pid>0){  /* padre */
   ...
   exit(0);
} else       /* figlio */
```

ps -ej quando il padre **e** il figlio sono vivi (caso BSD)

```
 PPID   PID  PGID   SID TPGID UID COMMAND
 9371  9372  9372  9372 9539 230  -tcsh
(tcsh)
 9372  9539  9539  9372 9539 230  provaback
 9539  9540  9539  9372 9539 230  provaback
```

ps -ej dopo il termine del padre

```
 PPID   PID  PGID   SID TPGID UID COMMAND
 9371  9372  9372  9372 9372 230  -tcsh
(tcsh)
    1  9540  9539  9372 9372 230  provaback
```


Progetto di un demone

```
#include <stdio.h>
#include <signal.h>
#include <sys/param.h>
#include <errno.h>
#include <sys/types.h>
extern int errno;

#ifdef SIGTSTP                  /* Unix BSD */
#include <sys/file.h>
#include <sys/ioctl.h>
#endif

/* Routine per la gestione di SIGCLD in Unix BSD */
sig_child()
{#ifdef BSD
    int pid;
    union wait status;
    while ((pid = wait3(&status, WNOHANG,
                  (struct rusage *) 0))>0);
 #endif
}

daemon_init(void)
{
 int fd, childpid;

 #ifdef SIGTTOU                 /* Unix BSD */
    signal(SIGTTOU, SIG_IGN);
 #endif
 #ifdef SIGTTIN
    signal(SIGTTIN, SIG_IGN);
 #endif
 #ifdef SIGTSTP
    signal(SIGTSTP, SIG_IGN);
 #endif
```

```
if ( (childpid=fork()) < 0)
    err_sys("impossibile fork del I figlio");
else if(childpid > 0)
        exit(0);

/* Si dissocia dal terminale di controllo e dal gruppo di processi */
#ifdef SIGTSTP              /* Unix BSD */
 if (setpgrp(0, getpid()) == -1)
    err_sys("impossibile cambiare il gruppo");
 if ((fd = open("/dev/tty", O_RDWR)) >0 0)
    { ioctl (fd, TIOCNOTTY, (char *) 0);
      close(fd);
    }
#else                       /* System V */
    setsid();
#endif

/* Chiude i descrittori di file aperto */
for (fd = 0; fd < NOFILE; fd++)
    close(fd);

/* Cambia la directory corrente per assicurarsi di non essere
*   su un mounted filesystem */
chdir("/");

/* Cancella la maschera di creazione file ereditata */
umask(0);

/* Evita che i figli diventino zombie */
#ifdef SIGTSTP              /* Unix BSD */
    int sig_child();
    signal(SIGCLD, sig_child);
#else                       /* System V */
    signal (SIGCLD, SIG_IGN);
#endif
}
```