Gestione processi e segnali

funzioni e librerie

**NOME**

fork() - crea un processo figlio

**SINOPSI**

#include <unistd.h>

pid\_t fork(void);

**DESCRIZIONE**

La funzione **fork()** crea un processo figlio che differisce dal genitore solamente per il suo PID (Process IDentifier) e PPID (Parent Process ID).  
Non sono ereditati i lock sui files, mentre la ricezione dei segnali provoca i comportamenti standard.

**VALORE DI RITORNO**

La funzione **fork()** ritorna in caso di successo:

* nel thread del padre, il PID del figlio
* nel thread del figlio, 0

Se la funzione **fork()** fallisce, ritorna -1 e non viene generato nessun processo figlio. In questa situazione la variabile **errno** e' settata con il codice di errore appropriato.

**CONFORME A**

SVr4, SVID, POSIX, X/OPEN, BSD 4.3.

**ESEMPIO**

#include <unistd.h>

...

pid\_t pid;

...

pid = fork();

if (pid < 0) {

/\* gestione errore \*/

}

else if (pid == 0) {

/\* processo figlio \*/

}

else {

/\* processo padre.

\* la variabile pid contiene il pid del processo figlio

\*/

}

...

**NOME**

\_exit() - termina il processo corrente

**SINOPSI**

#include <unistd.h>

void \_exit(int status);

**DESCRIZIONE**

La funzione **\_exit()** termina il processo che ha invocato la funzione e provoca i seguenti effetti.

* Ogni descrittore di file aperto dal processo chiamante, viene chiuso.  
  I files condivisi con altri processi non vengono chiusi.
* Ciascun figlio del processo corrente viene ereditato dal processo 1 (init, il padre di tutti i processi), sopravvivendo alla scomparsa del processo parent.
* Al padre del processo corrente viene inviato il segnale SIGCHLD, per indicare il termine del processo in esecuzione.
* Il valore di **status** e' ritornato al padre del processo corrente che lo puo' rilevare tramite la funzione **[wait()](https://digilander.libero.it/uzappi/C/librerie/funzioni/wait.html)**.  
  Per convenzione un valore 0 di **status** indica una terminazione senza errori, mentre un valore diverso da 0 indica la presenza di una condizione di errore.

La funzione **\_exit()**, a differenza della funzione [**exit()**](https://digilander.libero.it/uzappi/C/librerie/funzioni/exit.html) non chiama la funzione **[fflush()](https://digilander.libero.it/uzappi/C/librerie/funzioni/fflush.html)** per lo svuotamento dei buffer associati all'I/O bufferizzato, e non effettua le chiamate alle funzioni specificate con la funzione **[atexit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/atexit.html)** e **[on\_exit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/on_exit.html)**.

**VALORE DI RITORNO**

La funzione **\_exit()** non ritorna. Termina l'esecuzione del processo.

**CONFORME A**

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

### NOME

execve() - esegue un programma

### SINOPSI

#include <unistd.h>

int execve (const char \*filename, char \*const argv[], char \*const envp[]);

### DESCRIZIONE

La funzione **execve()** esegue il programma puntato da **filename**.  
Il programma deve essere o un eseguibile (binario) o uno script iniziante con la linea *#! interprete [arg]* e lunga al massimo 127 caratteri; in questo caso *interprete* deve essere una pathname valida che indichi un eseguibile (binario) il quale possa essere eseguito come interprete dello script tramite l'invocazione *interprete [arg]****filename***.  
Per esempio:

#! /bin/sh

**argv** e' un array di stringhe che costituiscono gli argomenti del programma **filename**. Deve sempre essere passato almeno un argomento **argv[0]** con il nome del programma. Per indicare il termine dell'array, l'ultimo argomento significativo deve essere seguito da un puntatore a *NULL*.

**envp**e' un array di stringhe che costituiscono l'ambiente (environment) da passare al programma.  
Ciascuna stringa deve essere della forma *parametro=valore*; per esempio: "TERM=vt100" oppure: "PATH=/bin:/usr/bin:/usr/local/bin:~/bin:."  
Anche in questo caso l'ultimo termine significativo dell'array deve essere seguito da un puntatore a *NULL*.

Il programma chiamato puo' accedere agli argomenti ed all'ambiente passati dal chiamante tramite la funzione *main* cosi' definita:

int main(int argc, char \*argv[], char \*envp[])

**execve()** sostituisce il programma chiamato al processo corrente, sovrascrivendone le aree testo (codice), dati, heap e stack.  
Il nuovo programma eredita dal processo chiamante il PID e ciascun file aperto che non abbia il flag "close on exec" abilitato.  
I segnali gestiti tramite la funzione [signal()](https://digilander.libero.it/uzappi/C/librerie/funzioni/signal.html) sono ripristinati al loro comportamento originale. Se il programma **filename** ha impostato il flag set-uid (in tal caso il nuovo programma puo' impiegare i diritti del proprietario del programma, anziche' di chi lo invoca), l'effettivo user ID del processo chiamante e' cambiato con il proprietario del programma **filename**.  
In modo analogo, se e' il programma **filename** ha settato il flag set-gid, l'effettivo group ID del processo chiamante e' sostituito con il gruppo del programma **filename**.  
**Nota:** Linux ignora i flag set-uid e set-gid impostati sugli script.

### VALORE DI RITORNO

In caso di successo, la funzione **execve()** non ritorna al chiamante.  
In caso di errore la funzione **execve()** ritorna -1 ed **errno** e' settato in modo opportuno. In tal caso vengono eseguite le istruzioni seguenti la chiamata di **execve()**.

### CONFORME A

SVr4, SVID, X/OPEN, BSD 4.3, POSIX

### NOME

execl(), execlp(), execle(), execv(), execvp() - esegue un programma

### SINOPSI

#include <unistd.h>

int execl(const char \*pathname, const char \*arg, ...);

int execlp(const char \*pathname, const char \*arg, ...);

int execle(const char \*pathname, const char \*arg , ..., char \*const envp[]);

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

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

### DESCRIZIONE

Le funzioni **execl(), execlp(), execle(), execvl(), execvp()** sostituiscono il processo corrente con un nuovo processo. In realta' le funzioni sono una interfaccia per la funzione [execve()](https://digilander.libero.it/uzappi/C/librerie/funzioni/execve.html) a cui si rimanda per ulteriori approfondimenti.

**pathname** e' un programma eseguibile (binario) o uno script che dovra' sostituire il processo corrente.

|  |  |
| --- | --- |
| Nel nome della della funzione chiamata sono contenuti dei caratteri che indicano sia il tipo di argomenti accettati e sia il tipo di comportamento dalla funzione stessa. | |
| **l** | La funzione accetta una *lista* di argomenti. Il primo argomento e' convenzionalmente il nome del programma che verra' eseguito. All'ultimo argomento valido deve seguire un puntatore a NULL. |
| **v** | La funzione accetta un *vettore* di argomenti. Il primo elemento (**argv[0]**) deve contenere il nome del programma da invocare. L'ultimo argomento valido deve essere seguito da un elemento contenente un puntatore a NULL. |
| **e** | La funzione accetta fra i suoi parametri un array di stringhe dell'ambiente (*environment*) che deve essere passato al **pathname** da eseguire. Ogni stringa di ambiente e' del formato parametro=valore (es: TERM=vt100). L'array **envp[]** e' terminato da un elemento a NULL. Le funzioni con il nome non comprendenti la lettera **e**, fanno ereditare l'ambiente del processo attuale il quale e' contenuto nella variabile esterna **environ**, al programma individuato da **pathname**. |
| **p** | In questo caso, se il **pathname** non contiene il carattere slash **/**, **pathname** viene ricercato in base alla variabile di ambiente *PATH*. Se la variabile PATH non e' specificata, **pathname** viene per default ricercato nelle directory /bin e /usr/bin |

### VALORE DI RITORNO

Le funzioni **execl(), execlp(), execle(), execvl(), execvp()** se hanno successo non effettuano ritorno al chiamante. In caso contrario il valore -1 viene tornato e la variabile **errno** viene settata con il valore appropriato.

### NOME

pause() - attende un segnale

### SINOPSI

#include <unistd.h>

int pause(void);

### DESCRIZIONE

La system call **pause()** pone in attesa il processo corrente, finche' non viene ricevuto un segnale.

### VALORE DI RITORNO

La system call **pause()** ritorna sempre -1 e setta la variabile **errno**.

### CONFORME A

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

**NOME**

signal() - gestione dei segnali

**SINOPSI**

#include <signal.h>

void (\*signal(int signum, void (\*sighandler)(int)))(int);

**DESCRIZIONE**

La funzione **signal()** stabilisce che quando il processo corrente riceve il segnale **signum**, il controllo deve passare come e' specificato dal *signal handler* **sighandler**.  
Per specificare il segnale **signum** puo' essere impiegato uno dei valori costanti definiti nell'header standard (o in un altro header da questo richiamato) oppure un valore numerico.  
**sighandler** puo' essere o una funzione utente oppure uno dei valori **SIG\_IGN** o **SIG\_DFL**.  
L'uso di una funzione di *signal handler* per un segnale e' detto "***catching the signal***" (cattura del segnale).

Quando il processo riceve il segnale **signum**, il processo puo' comportarsi in 3 modi differenti a seconda che **sighandler** sia stato impostato come:

**SIG\_IGN**

Il segnale viene ignorato.

**SIG\_DFL**

Viene eseguita l'azione di default prevista per quel segnale.

**Funzione utente**

Viene chiamata la funzione utente, avente come argomento il numero del segnale (**signum**) che ne ha provocato la chiamata.

**N.B.** - I segnali SIGKILL e SIGSTOP non possono essere ne' ignorati e ne' catturati.

**VALORE DI RITORNO**

In caso di successo la funzione **signal()** ritorna il precedente valore della funzione *signal handler*.  
Viene ritornato **SIG\_ERR** in caso di errore.

**NOTA**  
Il prototype indicato nella sezione [SINOPSI](https://digilander.libero.it/uzappi/C/librerie/funzioni/signal.html" \l "SINOPSI) puo' essere semplificato nella seguente forma:

#include <signal.h>

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

Ora appare piu' leggibile quali siano il tipo degli argomenti passati alla funzione e quale sia il tipo tornato da **signal()**.  
La funzione accetta due argomenti: uno di tipo *int* mentre l'altro e' di tipo puntatore a funzione (la quale a sua volta accetta un argomento intero e torna void).  
Il tipo ritornato da **signal** e' un puntatore a funzione che che accetta un argomento intero e torna void.

**CONFORME A**

Ansi C

**PORTABILITA'**

|  |  |
| --- | --- |
| **System V** | Alla ricezione del segnale **signum** la funzione di *signal handler* viene ripristinata al valore di default **SIG\_DFL** prima che venga eseguita la funzione **sighandler** associata con **signal()**. In risposta alle successive ricezioni del segnale **signum** viene attuato il comportamento standard previsto per il segnale. |
| **BSD** | Con questa versione di Unix, quando il processo riceve il segnale **signum**, prima che venga passato il controllo alla funzione di signal handler, vengono bloccati ulteriori riconoscimenti del segnale. Il blocco ha termine all'uscita della funzione **sighandler**. |

Pertanto perche' un programma che gestisce i segnali sia portabile tra le diverse versioni di Unix, occorre scrivere la funzione **sighandler** affinche':

1. All'inizio della funzione **sighandler** venga subito chiamata la funzione **signal()** per il segnale **signum** associandolo al valore costante **SIG\_IGN**  
   Cosi' facendo i successivi arrivi del segnale **signum** vengono ignorati, evitando corse critiche.
2. Implementazione del corpo della funzione **sighandler**
3. Al termine della funzione **sighandler** deve essere chiamata ancora la funzione **signal()** per associare il segnale **signum** nuovamente a **sighandler**  
   A questo punto successivi arrivi del segnale **signum** vengono gestiti correttamente dalla funzione **sighandler**

**ESEMPIO**

/\* Schema di funzionamento della funzione 'signal handler' \*/

#include <signal.h>

...

void sighandler(int segnale\_ricevuto);

...

int main(int argc, char \*argv[])

{

void (\*prev\_sighandler)(int);

...

prev\_sighandler = signal(SIGINT, sighandler);

...

}

void sighandler(int segnale\_ricevuto) /\* gestione del segnale \*/

{

/\* dichiarazioni/definizioni \*/

...

/\* vengono ignorati altri arrivi del segnale \*/

signal(segnale\_ricevuto, SIG\_IGN);

/\* gestione del segnale \*/

...

/\* ripristino del signal handler \*/

signal(segnale\_ricevuto, sighandler);

}

### NOME

sleep - sospende temporaneamente il processo

### SINOPSI

#include <unistd.h>

unsigned int sleep(unsigned int seconds);

### DESCRIZIONE

La funzione **sleep()** sospende il processo corrente per **seconds** secondi o fino all'arrivo di un segnale che non sia ignorato.

**sleep()** puo' essere basata sull'arrivo del segnale **SIGALRM**, pertanto e' da evitare un uso contemporaneo di **sleep()** e della system call [alarm()](https://digilander.libero.it/uzappi/C/librerie/funzioni/alarm.html).

### VALORE DI RITORNO

La funzione **sleep()** ritorna il numero di secondi che il processo deve sospendersi.

### CONFORME A

POSIX.1

### NOME

wait(), waitpid() - attende il termine di un processo

### SINOPSI

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status)

pid\_t waitpid(pid\_t pid, int \*status, int options);

### DESCRIZIONE

La funzione **wait()** sospende il processo corrente finche' un figlio (child) termina o finche' il processo corrente riceve un segnale di terminazione o un segnale che sia gestito da una funzione.  
Quando un child termina il processo, senza che il parent abbia atteso la sua terminazione attraverso la funzione di **wait()**, allora il child assume lo stato di "zombie" ossia di processo "defunto".  
Se il processo corrente esegue la funzione di **wait()**, mentre ci sono uno o piu' child in stato di zombie, allora la funzione ritorna immediatamente e ciascuna risorsa del child viene liberata.

La funzione **waitpid()** sospende il processo corrente finche' il figlio (child) corrispobndente al **pid** passato in argomento termina o finche' il processo corrente riceve un segnale di terminazione o un segnale che sia gestito da una funzione.  
Se il processo corrente esegue la funzione di **waitpid()** e il child identificato dal **pid** e' in stato di zombie, allora la funzione ritorna immediatamente e ciascuna risorsa del child viene liberata.

Il valore del **pid** puo' essere uno dei seguenti:

|  |  |
| --- | --- |
| **< -1** | la funzione attende ciascun child avente il *process group ID* uguale al valore assoluto del **pid**. |
| **-1** | la funzione attende ciascun child analogamente alla funzione **wait()**. |
| **0** | la funzione attende ciascun child avente il *process group ID* uguale a quello del processo corrente. |
| **> 0** | la funzione attende il child avente il *process ID* corrispondente al valore del **pid**. |

Se **status** non e' NULL, le funzioni **wait()** e **waitpid()** memorizzano l'informazione dello stato nell'area di memoria puntata da questo argomento.  
Sono disponibili alcune macro per la valutazione dello stato **status**:

**WIFEXITED(status)**

risulta vera (diversa da zero) se il child e' uscito normalmente.

**WEXITSTATUS(status)**

riporta gli 8 bit meno significativi del codice di ritorno del child.  
Il child puo' comunicare il codice di ritorno al parent, con l'argomento passato alla funzione [\_exit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/_exit.html) o [exit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/exit.html) o nello statement **return** della funzione main().  
Questa macro puo' essere valutata solamente se la macro **WIFEXITED** e' risultata vera (diversa da zero).

**WIFSIGNALED(status)**

risulta vera (diversa da zero) se il child e' uscito per mezzo di un segnale non gestito.

**WTERMSIG(status)**

riporta il segnale (il suo numero) che ha causato il termine del processo figlio.  
Questa macro puo' essere valutata solamente se la macro **WIFSIGNALED** e' risultata vera (diversa da zero).

**WIFSTOPPED(status)**

risulta vera (diversa da zero) se il child e' fermo (stop).  
Questa possibilita' e' condizionata dall'impiego del flag **WUNTRACED** nell'argomento delle **options**.

**WSTOPSIG(status)**

ritorna il numero del segnale che ha causato il child a fermarsi (stop). Questa macro puo' essere valutata solamente se la macro **WIFSTOPPED** e' risultata vera (diversa da zero).

Il paramentro **options** puo' valere zero o puo' essere in OR con i seguenti valori costanti:

|  |  |
| --- | --- |
| **WNOHANG** | specifica il ritorno immediato se i child non sono usciti. |
| **WUNTRACED** | specifica di ritornare anche se i child sono fermati (stop), e lo stato non deve venire riportato. |

### VALORE DI RITORNO

In caso di successo le funzioni **wait()** e **waitpid()** ritornano il process ID del child che termina.  
Tornano -1 in caso di errore e **errno** viene settato in modo appropriato.  
**waitpid()** torna 0 se e' stato impiegato WNOHANG e non ci sono figli che hanno terminato.

### CONFORME A

SVr4, POSIX.1

### NOME

kill() - invia un segnale ad un processo

### SINOPSI

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig);

### DESCRIZIONE

Tramite la system call **kill()** e' possibile inviare un segnale ad un process group o ad un processo.

|  |  |
| --- | --- |
| Se il **pid** e': | |
| **> 0** | il segnale **sig** e' inviato al processo avente il pid uguale a **pid**. |
| **0** | il segnale **sig** e' inviato ad ogni processo nel process group del processo corrente. |
| **-1** | il segnale **sig** e' inviato ad ogni processo eccetto per il primo. L'ordine di invio del segnale va dal processo che ha il pid piu' grande nella tabella dei processi, a quello avente il pid piu' piccolo. |
| **< -1** | il segnale e' inviato a ogni processo che ha il process group uguale al valore assoluto del **pid**. |

Se **sig** e' 0, il segnale non viene inviato, ma viene fatta ugualmente una ricerca degli errori. Cio' e' utile per verificare se un **pid** e' associato a qualche processo.

Il processo che invia il segnale e i processi che lo ricevono, devono avere lo stesso user ID effettivo, a meno che a inviare il segnale sia il super-utente.

### VALORE DI RITORNO

In caso di successo, **kill()** ritorna 0.  
In caso di errore, viene ritornato -1 e la variabile **errno** e' settata con il codice di errore.

### NOTA

Non e' possibile inviare segnali al processo avente **pid** numero 1. Questo processo *speciale* e' il processo **init** chiamato all'inizializzazione del sistema ed e' il padre di tutti i processi.  
Questo garantisce che **init** non venga terminato accidentalmente a causa dell'invio di un segnale, fermando l'intero sistema.

### CONFORME A

SVr4, SVID, POSIX.1, X/OPEN, BSD 4.3

### NOME

alarm() - imposta un allarme temporizzato per l'invio di un certo segnale

### SINOPSI

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

### DESCRIZIONE

La funzione **alarm()** invia al processo corrente il segnale **SIGALRM** dopo che siano trascorsi **seconds** secondi.

Una successiva chiamata ad **alarm()** reimposta il tempo di invio del segnale **SIGALRM**.

Se **seconds** e' 0, non viene attivato nessun allarme e viene disabilitato l'eventuale allarme precedentemente impostato.

### VALORE DI RITORNO

La funzione **alarm()** ritorna il numero di secondi rimasti all'invio del segnale **SIGALRM** predisposto da una precedente chiamata alla funzione **alarm()**.  
La funzione **alarm()** ritorna 0 se non e' atteso nessun allarme.

### CONFORME A

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

### ESEMPIO

/\* Prova la funzione alarm() \*/

#include <stdlib.h>

#include <stdio.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <unistd.h>

#define MAX\_LOOP 80000L /\* ciclo di loop (dipendente dal sistema) \*/

void wait\_alarm(int isignal);

void traffic(void);

int main(int argc, char \*argv[])

{

unsigned int sec;

signal(SIGALRM, wait\_alarm); /\* gestione del segnale SIGALRM \*/

/\* Attivazione del segnale di allarme fra 2 secondi \*/

sec = alarm(2);

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

printf(" ALLARME precedente impostato a %u secondi\n", sec);

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

/\* Attivazione del segnale di allarme fra 8 secondi (ci ho ripensato...)\*/

sec = alarm(8);

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

printf(" ALLARME precedente impostato a %u secondi\n", sec);

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

traffic(); /\* traffic (indica che il programma sta' girando) \*/

return 0;

}

void wait\_alarm(int isignal)

{

puts("");

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

printf("\n\n\t\tRicevuto il segnale %d - Termine programma\n\n", isignal);

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

puts("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

exit(0);

}

void traffic(void)

{

long j;

while(1) { /\* loop infinito \*/

printf("\r-");

fflush(stdout);

for (j=MAX\_LOOP; j > 0; j--) /\* perdi tempo \*/

;

printf("\r\\");

fflush(stdout);

for (j=MAX\_LOOP; j > 0; j--) /\* perdi tempo \*/

;

printf("\r|");

fflush(stdout);

for (j=MAX\_LOOP; j > 0; j--) /\* perdi tempo \*/

;

printf("\r/");

fflush(stdout);

for (j=MAX\_LOOP; j > 0; j--) /\* perdi tempo \*/

;

}

}

### NOME

system() - esegue un comando di shell

### SINOPSI

#include <stdlib.h>

int system(const char \*string);

### DESCRIZIONE

La system call **system()** esegue il comando specificato nella stringa **string**, chiamando la shell con la sintassi:  
/bin/sh -c *string* (versione UNIX)

Ritorna dopo che il comando sia stato completato.

Durante l'esecuzione del comando **SIGCHLD** (segnale di terminazione del processo child) puo' essere bloccato, mentre i segnali **SIGINT** (segnale di interrupt da tastiera ^C) e **SIGQUIT** (segnale di uscita da tastiera ^\) potrebbero essere ignorati.

**NOTA:**  
**system()** non influisce lo stato di attesa (wait()) su altri processi figli.

### VALORE DI RITORNO

Se la stringa **string** e' **NULL**, la funzione **system()** ritorna un valore non nullo se esite un command processor (shell), altrimenti torna zero.

Se la stringa **string** non e' **NULL**, la funzione **system()** ritorna un valore che dipende dall'implementazione del sistema operativo.  
Nel caso di Linux, **system()** ritorna 127 se fallisce la chiamata a [execve()](https://digilander.libero.it/uzappi/C/librerie/funzioni/execve.html) per la shell "/bin/sh", altrimenti torna -1 se avviene un altro errore, oppure ritorna il return code del comando.

### CONFORME A

ANSI C, POSIX.2, BSD 4.3

Librerie

## Libreria sys/types.h

* o [closedir() - chiude una directory](https://digilander.libero.it/uzappi/C/librerie/funzioni/closedir.html)
* o [kill() - invia un segnale ad un processo](https://digilander.libero.it/uzappi/C/librerie/funzioni/kill.html)
* o [lseek() - riposiziona l'offset per la successiva operazione di read/write](https://digilander.libero.it/uzappi/C/librerie/funzioni/lseek.html)
* o [mkdir() - crea una directory](https://digilander.libero.it/uzappi/C/librerie/funzioni/mkdir.html)
* o [open(), creat() - apre e possibilmente crea un file o device](https://digilander.libero.it/uzappi/C/librerie/funzioni/open.html)
* o [opendir() - apre una directory](https://digilander.libero.it/uzappi/C/librerie/funzioni/opendir.html)
* o [readdir() - legge l'entry della directory](https://digilander.libero.it/uzappi/C/librerie/funzioni/readdir.html)
* o [rewinddir() - resetta il directory stream](https://digilander.libero.it/uzappi/C/librerie/funzioni/rewinddir.html)
* o [telldir() - ritorna la locazione corrente del directory stream](https://digilander.libero.it/uzappi/C/librerie/funzioni/telldir.html)
* o [wait(), waitpid() - attende il termine di un processo](https://digilander.libero.it/uzappi/C/librerie/funzioni/wait.html)

## Libreria sys/wait.h

* o [wait(), waitpid() - attende il termine di un processo](https://digilander.libero.it/uzappi/C/librerie/funzioni/wait.html)

## Libreria unistd.h

* o [\_exit() - termina il processo corrente](https://digilander.libero.it/uzappi/C/librerie/funzioni/_exit.html)
* o [access() - controlla i permessi dello user per un file](https://digilander.libero.it/uzappi/C/librerie/funzioni/access.html)
* o [alarm() - imposta un allarme temporizzato per l'invio di un certo segnale](https://digilander.libero.it/uzappi/C/librerie/funzioni/alarm.html)
* o [chdir(), fchdir() - cambia la directory di lavoro](https://digilander.libero.it/uzappi/C/librerie/funzioni/chdir.html)
* o [close() - chiude un file descriptor](https://digilander.libero.it/uzappi/C/librerie/funzioni/close.html)
* o [dup(), dup2() - duplica un file descriptor](https://digilander.libero.it/uzappi/C/librerie/funzioni/dup.html)
* o [execl(), execlp(), execle(), execv(), execvp() - esegue un programma](https://digilander.libero.it/uzappi/C/librerie/funzioni/execl.html)
* o [execve() - esegue un programma](https://digilander.libero.it/uzappi/C/librerie/funzioni/execve.html)
* o [fcntl() - manipola il file descriptor](https://digilander.libero.it/uzappi/C/librerie/funzioni/fcntl.html)
* o [fork() - crea un processo figlio](https://digilander.libero.it/uzappi/C/librerie/funzioni/fork.html)
* o [getcwd(), get\_current\_dir\_name(), getwd() - acquisisce la directory corrente di lavoro](https://digilander.libero.it/uzappi/C/librerie/funzioni/getcwd.html)
* o [link() - crea un nuovo nome per un file](https://digilander.libero.it/uzappi/C/librerie/funzioni/link.html)
* o [lseek() - riposiziona l'offset per la successiva operazione di read/write](https://digilander.libero.it/uzappi/C/librerie/funzioni/lseek.html)
* o [mkdir() - crea una directory](https://digilander.libero.it/uzappi/C/librerie/funzioni/mkdir.html)
* o [pause() - attende un segnale](https://digilander.libero.it/uzappi/C/librerie/funzioni/pause.html)
* o [pipe - crea una 'pipe'](https://digilander.libero.it/uzappi/C/librerie/funzioni/pipe.html)
* o [read() - legge da un file descriptor](https://digilander.libero.it/uzappi/C/librerie/funzioni/read.html)
* o [rmdir() - elimina una directory](https://digilander.libero.it/uzappi/C/librerie/funzioni/rmdir.html)
* o [sleep - sospende temporaneamente il processo](https://digilander.libero.it/uzappi/C/librerie/funzioni/sleep.html)
* o [stat(), lstat(), fstat() - acquisisce informazioni su un file.](https://digilander.libero.it/uzappi/C/librerie/funzioni/stat.html)
* o [symlink() - crea un nuovo nome per un file](https://digilander.libero.it/uzappi/C/librerie/funzioni/symlink.html)
* o [unlink() - elimina un nome e se possibile il file a cui si riferisce](https://digilander.libero.it/uzappi/C/librerie/funzioni/unlink.html)
* o [write() - scrive su un file descriptor](https://digilander.libero.it/uzappi/C/librerie/funzioni/write.html)

# Linguaggio C - signal.h (libreria standard gestione segnali)

L'inclusione della libreria standard **signal.h** consente l'utilizzo delle funzioni per la gestione dei segnali fra processi. Un processo puo' ricevere un segnale sia al verificarsi di un evento hardware che software.  
I segnali derivanti dagli eventi hardware sono generati internamente dal sistema come ad esempio avviene per una eccezione sul calcolo in floating-point, oppure per un intervento dell'operatore quando preme i tasti *<ctrl>C* per interrompere un programma in esecuzione.  
I segnali derivanti dagli eventi software sono generati dai processi per mezzo di funzioni specifiche a tale scopo, come ad esempio le funzioni **kill()**, **killpg()**, **alarm()**. Inoltre, sfruttando tali funzioni, sono disponibili dei comandi che introdotti da tastiera inviano dei segnali a dei processi individuati (tipicamente per Unix un tale comando e' **kill**). Anche la terminazione di un processo genera un segnale: infatti il processo che termina invia al padre il segnale **SIGCHLD**. Un processo puo' intercettare e gestire un segnale tramite la funzione **signal()**.Di seguito sono riportati i principali segnali presenti nella versione POSIX di Unix.

|  |  |  |
| --- | --- | --- |
| **Segnale** | **Azione di default** | **Note** |
| **SIGHUP** | Termina il processo | Hangup del terminale di controllo. Spesso viene effettuata la reinizializzazione del processo |
| **SIGINT** | Termina il processo | Interruzione da tastieara (^C) |
| **SIGQUIT** | Termina il processo | Uscita da tastiera (^\) |
| **SIGILL** | Termina il processo | Istruzione illegale |
| **SIGABRT** | Termina il processo effettuando la core dump | Segnale di abort generato dalla funzione **abort()** |
| **SIGFPE** | Termina il processo effettuando la core dump | Eccezione floating point |
| **SIGKILL** | Termina il processo | Segnale di Kill. Il segnale non puo' essere ne' catturato ne' ignorato |
| **SIGSEGV** | Termina il processo effettuando la core dump | Riferimento di memoria invalido |
| **SIGPIPE** | Termina il processo | Broken pipe: scrittura alla pipe senza processi consumatori |
| **SIGALRM** | Termina il processo | Segnale generato dalla funzione **alarm()** |
| **SIGTERM** | Termina il processo | Segnale di terminazione |
| **SIGUSR1** | Termina il processo | Segnale 'user-defined 1' |
| **SIGUSR2** | Termina il processo | Segnale 'user-defined 2' |
| **SIGCHLD** | Ignora il segnale | Il child process e' terminato o e' fermo |
| **SIGCONT** |  | Se il processo e' fermo, il processo riprende |
| **SIGSTOP** | Ferma il processo | Ferma il processo. Il segnale non puo' essere ne' catturato ne' ignorato |
| **SIGTSTP** | Ferma il processo | Stop dato da una tty (^Z) |
| **SIGTTIN** | Ferma il processo | tty input per un processo in background |
| **SIGTTOU** | Ferma il processo | tty output per un processo in background |

Per utilizzare la libreria standard **signal.h** e' necessario [includere l'header signal.h](https://digilander.libero.it/uzappi/C/C-preprocessor.html#include) nel file sorgente:

#include <signal.h>

## Funzioni riguardanti i segnali

o [kill()](https://digilander.libero.it/uzappi/C/librerie/funzioni/kill.html" \t "funzioni)  
o [killpg()](https://digilander.libero.it/uzappi/C/librerie/funzioni/killpg.html" \t "funzioni)  
o [signal()](https://digilander.libero.it/uzappi/C/librerie/funzioni/signal.html" \t "funzioni)

## Riferimenti ad altre librerie

Ulteriori funzioni di gestione dei segnali sono disponibili nelle librerie [unistd](https://digilander.libero.it/uzappi/C/librerie/C-unistd.html) (interfaccia con il sistema Unix) e *wait* (attesa della terminazione di un processo).  
Per utilizzare le funzioni elencate di seguito, e' necessario includere gli header appropriati, come mostrato nella sinopsi di ogni funzione. Es.:

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

### Funzioni della libreria unistd.h

o [alarm()](https://digilander.libero.it/uzappi/C/librerie/funzioni/alarm.html" \t "funzioni)  
o [pause()](https://digilander.libero.it/uzappi/C/librerie/funzioni/pause.html)  
o [sleep()](https://digilander.libero.it/uzappi/C/librerie/funzioni/sleep.html" \t "funzioni)

### Funzioni della libreria sys/wait.h

o [wait(), waitpid()](https://digilander.libero.it/uzappi/C/librerie/funzioni/wait.html" \t "funzioni)

# Linguaggio C - librerie standard per la gestione dei processi

Fra le funzioni disponibili nelle librerie del C, ce ne sono alcune che consentono la gestione dei processi, permettendo:

|  |  |
| --- | --- |
| o | la nascita di un nuovo processo ([fork()](https://digilander.libero.it/uzappi/C/librerie/C-processi.html" \l "fork)) |
| o | l'attesa di un processo figlio ([wait()](https://digilander.libero.it/uzappi/C/librerie/C-processi.html" \l "wait)) |
| o | la trasformazione di un processo corrente (funzioni del gruppo [exec()](https://digilander.libero.it/uzappi/C/librerie/C-processi.html" \l "exec)) |
| o | l'invio di segnali fra processi (libreria [signal.h](https://digilander.libero.it/uzappi/C/librerie/C-signal.html)) |
| o | la condivisione di file a scopo di comunicazione o sincronizzazione fra processi ([pipe](https://digilander.libero.it/uzappi/C/librerie/C-processi.html#pipe)) |

### Generalita' sui processi (Unix)

Con il termine *processo* si intende il programma in esecuzione con l'impegno di piu' risorse (CPU, memoria, file system, ...) che deve condividere con gli altri processi presenti sul medesimo calcolatore.  
L'esecuzione di un programma puo' generare piu' di un processo (vedi [fork()](https://digilander.libero.it/uzappi/C/librerie/C-processi.html" \l "fork)).

Ogni processo Unix e' contraddistinto da un identificatore di processo (*process ID*) indicato con **pid** e da un identificatore del processo padre che viene indicato con **ppid** (*parent process ID*).

Il *processo padre* (*parent process*) e' il processo che genera un altro processo, detto *processo figlio* (*child process*).  
Piu' propriamente il processo padre dovrebbe essere chiamato *processo genitore* dall'inglese (*parent*=*genitore*), ma il termine *padre* si adatta ugualmente bene allo scopo.

Il **pid** e' un numero univoco in un determinato istante, fornito dal sistema operativo e necessario ad identificare un processo in "corso" sul sistema.

Il **ppid** e' l'identificativo del padre del processo che si sta' esaminando. In altri termini il **ppid** e' il **pid** del processo che ha generato il processo considerato.

Un processo si puo' trovare in uno dei seguenti stati:

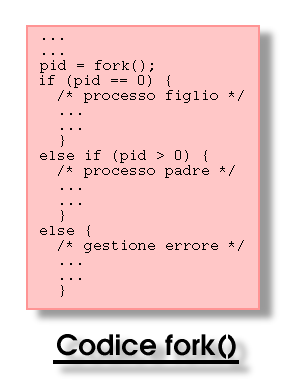
|  |  |  |
| --- | --- | --- |
| o | **Ready** | E' pronto per essere eseguito. |
| o | **Running** | E' in esecuzione su una CPU del sistema. |
| o | **Sleeping** | Attende un evento. |
| o | **Swapped** | Parte del processo e' stato trasferito su disco, per liberare la memoria per altri processi. |
| o | **Terminated** | Il processo e' terminato. Invio del segnale SIGCHLD al parent. |
| o | **Zombie** | Il processo ha terminato la sua esecuzione, ma il parent non ha raccolto il segnale SIGCHLD. Il processo mantiene ancora allocate delle risorse. |

### Fork di un processo

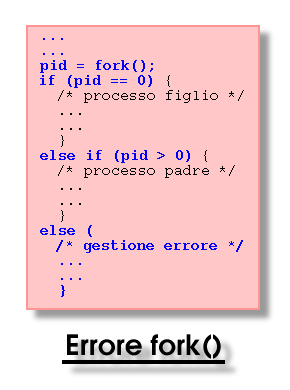
Un processo, durante la sua esecuzione, puo' creare un nuovo processo tramite la system call **fork()**.  
Il processo corrente e' detto *processo padre* o *parent process*, mentre il nuovo processo e' detto *processo figlio* o *child process*. Entrambi i processi condividono lo stesso codice programma ma vengono eseguiti in concorrenza fra loro assieme al resto dei processi elaborati sul sistema in oggetto. Normalmente il padre e il figlio eseguono delle istruzioni differenti.  
Il child process eredita i file aperti dal parent process. Questi possono costituire un mezzo di interazione fra i 2 processi.  
Diversamente i dati (le variabili del programma), lo stack e l'ambiente (environment) del parent process vengono duplicati per il nuovo processo e posti in un'area di memoria a lui riservata e non visibile dagli altri processi, parent compreso.

La **fork()** restituisce al parent o un valore negativo in caso di errore (il child process non viene generato), o un valore positivo corrispondente al **pid** del child.  
La **fork()** ritorna al child un valore sempre nullo.

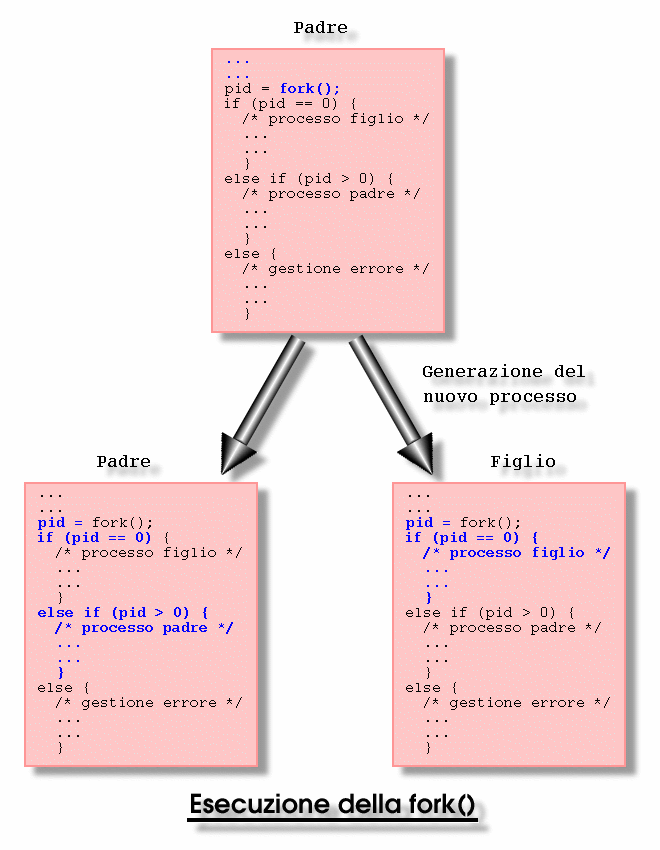
A titolo di esempio riporto un breve spezzone di codice:



In caso che venga rilevato un errore durante la creazione del nuovo processo, il codice deve essere in grado di gestirlo correttamente.  
Nella figura sottostante, ho indicato in blu le istruzioni eseguite dal programma in caso di errore ritornato dalla system call **fork()**.



Normalmente il processo padre esegue le istruzioni del programma, fino ad incontrare la chiamata alla **fork()** (istruzioni in blu). A questo punto il processo padre genera un nuovo processo. Da questo momento i 2 processi hanno vita indipendente e il risultato tornato dalla **fork()** e' differente per ciascun processo.  
E' normale che i 2 processi eseguano percorsi differenti all'interno dello stesso programma (istruzioni in blu).

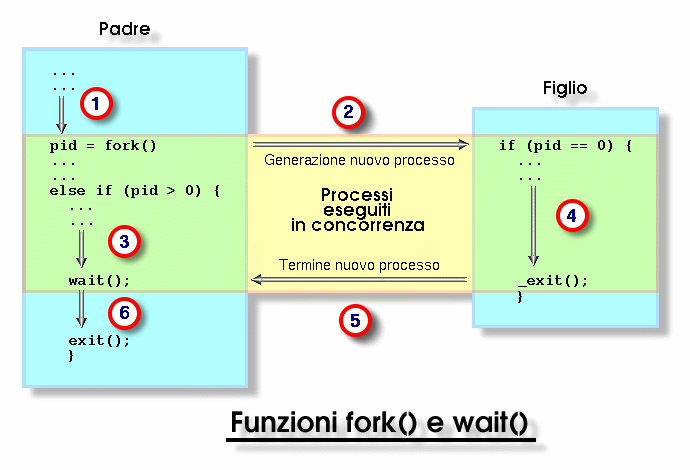


### Attesa di un processo

Per i processi generati con una **[fork()](https://digilander.libero.it/uzappi/C/librerie/C-processi.html" \l "fork)** si possono creare uno dei seguenti casi:

|  |  |
| --- | --- |
| o | Il processo padre termina prima del processo figlio. In questa situazione il child process rimane **orfano** del padre e viene *adottato* dal processo **init** che per definizione e' *il padre di tutti i processi*. |
| o | Il processo figlio termina, ma il padre non rileva il suo termine. Quando cio' si verifica, il processo figlio e' definito **defunto** oppure **zombie** e rimane in tale stato finche' o il padre non ha rilevato la sua terminazione, oppure fino a quando anche il padre termina; al termine del padre il processo figlio viene ereditato dal processo **init** che ne rileva la sua terminazione. Un processo zombie mantiene allocate le risorse fino quando non sia stato rilevato il suo stato di terminazione o dal processo padre o dal processo init. |

Nella situazione di normalita', il padre deve quindi rilevare la terminazione del figlio tramite la system call **wait()**. Il figlio puo' cosi' rilasciare ogni risorsa impegnata.



Legenda della numerazione posta a fianco del flusso di processo:

1. E' in esecuzione il solo processo padre.
2. Il processo padre invoca la funzione **fork()**. Viene generato il processo figlio.
3. Il processo padre e' eseguito in concorrenza con il processo figlio (oltre agli altri processi *running* sul sistema).  
   Se il padre invoca la funzione **wait()**, il padre attende la terminazione del figlio.
4. Il processo figlio e' eseguito in concorrenza con il processo padre (oltre agli altri processi *running* sul sistema).
5. Il figlio ha eseguito la terminazione del programma invocando la funzione **\_exit()** o **exit()**.  
   Al padre e' inviato il segnale **SIGCHLD**.
6. Il padre ha rilevato la terminazione del figlio tramite la funzione **wait()** e le risorse impegnate dal figlio vengono liberate.  
   Il processo padre riprende l'esecuzione fino alla sua terminazione.

**Funzioni del gruppo exec**

Le funzioni del gruppo exec sono in grado di **sostituire** il processo corrente con un altro processo. Il **pid** e il **ppid** rimangono invariati. Praticamente si ha una **trasformazione** del processo.

Quando una funzione del gruppo exec viene eseguita con successo, il processo carica il programma o lo script indicato fra gli argomenti della funzione chiamata e lo manda in esecuzione in sostituzione del processo attuale. Non e' previsto nessun tipo di ritorno al vecchio processo se non nel caso che non sia possibile avviare il nuovo processo.

Le funzioni per la gestione dei processi non sono definite da una libreria specifica, ma fanno riferimento a piu' librerie standard; pertanto per il loro utilizzo e' necessario [includere gli headers](https://digilander.libero.it/uzappi/C/C-preprocessor.html#include) appropriati come descritto nella sinopsi di ciascuna funzione. Es.:

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

### Funzioni della libreria stdlib.h

o [atexit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/atexit.html" \t "funzioni)  
o [exit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/exit.html)  
o [on\_exit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/on_exit.html" \t "funzioni)  
o [system()](https://digilander.libero.it/uzappi/C/librerie/funzioni/system.html)

### Funzioni della libreria stdio.h

o [popen(), pclose()](https://digilander.libero.it/uzappi/C/librerie/funzioni/popen.html" \t "funzioni)

### Funzioni della libreria unistd.h

o [\_exit()](https://digilander.libero.it/uzappi/C/librerie/funzioni/_exit.html)  
o [alarm()](https://digilander.libero.it/uzappi/C/librerie/funzioni/alarm.html)  
o [execl(), execv(), execle(), execlp(), execvp()](https://digilander.libero.it/uzappi/C/librerie/funzioni/execl.html" \t "funzioni)  
o [execve()](https://digilander.libero.it/uzappi/C/librerie/funzioni/execve.html" \t "funzioni)  
o [fork()](https://digilander.libero.it/uzappi/C/librerie/funzioni/fork.html)  
o [pause()](https://digilander.libero.it/uzappi/C/librerie/funzioni/pause.html)  
o [pipe()](https://digilander.libero.it/uzappi/C/librerie/funzioni/pipe.html)  
o [sleep()](https://digilander.libero.it/uzappi/C/librerie/funzioni/sleep.html)

### Funzioni della libreria sys/wait.h

o [wait(), waitpid()](https://digilander.libero.it/uzappi/C/librerie/funzioni/wait.html" \t "funzioni)

### Funzioni della libreria signal.h

o [kill()](https://digilander.libero.it/uzappi/C/librerie/funzioni/kill.html" \t "funzioni)  
o [killpg()](https://digilander.libero.it/uzappi/C/librerie/funzioni/killpg.html" \t "funzioni)  
o [signal()](https://digilander.libero.it/uzappi/C/librerie/funzioni/signal.html" \t "funzioni)