Fork:

una system call che crea un nuovo processo uguale a quello del

padre in cui l’unica differenza tra processo padre e processo figlio

è il valore restituito dalla funzione fork:

= Pid del processo figlio nel padre,

= zero nel figlio.

wait

La funzione wait ha la seguente sintassi:

int wait(int \*status)

La funzione wait() sospende il processo corrente finché un figlio

(child) termina o finché 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 più child in stato di zombie, allora la funzione ritorna

immediatamente e ciascuna risorsa del child verrà liberata.

La funzione wait restituisce

il pid del figlio in caso di successo,

-1 in caso di errore.

Nel caso un processo abbia più figli il valore di ritorno (il pid del

figlio) permette di identificare quello che è terminato.

Se il parametro status non è NULL, la funzione memorizza

l'informazione dello stato nell'area di memoria puntata da status.

Ossia se il figlio ha eseguito un exit(9); l’indirizzo di memoria

puntata da status avrà valore 9.

waitpid

La funzione waitpid ha la seguente sintassi:

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

La funzione waitpid() sospende il processo corrente finchè il

figlio (child) corrispobndente al pid passato in argomento termina

o finché 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 è in stato di zombie, allora la funzione ritorna

immediatamente e ciascuna risorsa del child viene liberata.

Le seguenti macro sono utilizzate per valutare lo stato:

WIFEXITED(status)

risulta vera se il child è uscito normalmente

WEXITSTATUS(status)

riporta gli 8 bit meno significativi del codice di ritorno del

child. Il child può comunicare il codice di ritorno al parent,

attraverso l'argomento passato alla funzione exit()

o return della funzione main(). Questa macro può essere

valutata solamente se la macro WIFEXITED e' risultata vera.

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 può essere valutata solamente

se la macro WIFSIGNALED e' risultata.

WIFSTOPPED(status) risulta vera se il child è fermo (stop).

Questa possibilità è 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 può essere valutata solamente se la

macro WIFSTOPPED e' risultata vera

Il parametro options può essere zero o i seguenti valori costanti

anche messi in OR:

NOHANG ritorno immediato se i child non sono usciti.

WUNTRACED ritornare anche se i child sono fermati (stop),

e lo stato non deve venire riportato.

la funzioni waitpid ritorna:

il process ID del child che termina in caso di sucesso.

-1 in caso di errore e errno viene settato in modo

appropriato.

0 se e' stato impiegato WNOHANG e non ci sono figli che

hanno terminato.

pid\_t  
è un tipo speciale definito in <sys/types.h> e, solitamente, corrisponde ad un tipo intero.  
#include <unistd.h> /\* include la definizione di pid\_t \*/

system()

restituisce -1 in caso di errore, tuttavia la restituzione di 0 non implica che comando sia stato effettivamente eseguito, infatti system() esegue il comando ricevuto come parametro, esso non fa altro che lanciare una copia dell'interprete stesso, proprio come se fosse stato digitato il comando.

Quindi system() si limita a restituire 0 nel caso in cui sia riuscita a lanciare correttamente l'interprete, non solo non è possibile conoscere il valore restituito al sistema dal child, ma non è neppure possibile sapere se questo sia stato effettivamente eseguito.

|  |
| --- |
| Esempio di System  **#include <stdlib.h>**  **#include <stdio.h>**  **#include <errno.h>// per errno**  **int main()**  **{if(system("/bin/ls \*.c ") == -1)**  **fprintf(stderr,"errore %d in system()\n", errno);**  **return(0);**  **}**  spawn...()  Le funzioni della famiglia spawn...() ocome system(), consentono di lanciare programmi esterni come se fossero subroutine, tuttavia esse non fanno ricorso all'interprete dei comandi, in quanto si basano sul servizio 4Bh dell'interupt 21h.  Di conseguenza, non è possibile utilizzarle per invocare comandi interni né file batch, ma si ha un controllo sull'esito dell'operazione.  Esse infatti restituiscono -1 se l'esecuzione del child non è riuscita; in caso contrario restituiscono il valore che il programma child ha restituito a sua volta.  Tutte le funzioni spawn...() richiedono come primo parametro un intero, di solito dichiarato nei prototipi con il nome mode, che indica la modalità di esecuzione del programma child. |

In PROCESS.H sono definite le seguenti costanti dette manifest:

 P\_WAIT il child è eseguito come una subroutine, quindi il processo chiamante si interrompe e aspetta la terminazione del nuovo processo

 P\_OVERLAY il child è eseguito sostituendo in memoria il parent, proprio come se fosse chiamata la corrispondente funzione della famiglia exec...()

 P\_NOWAIT Continua l'esecuzione del processo chiamante (asynchronous **\_ spawn** )

 P\_DETACH Continua l'esecuzione del processo chiamante; il nuovo processo gira in background senza accesso alla console o alla keyboard.

Il secondo parametro, di tipo char \*, è il nome del programma da eseguire.

exec...()

Le funzioni della famiglia exec...(), a differenza delle spawn...(), non trattano il child come una subroutine del parent.

Al contrario, esso viene caricato in memoria ed eseguito al posto del parent, sostituendolo a tutti gli effetti. E' del tutto equivalente utilizzare

spawnv(P\_OVERLAY,"myutil",childArgv) o

execv("myutil",childArgv);

dup e dup2

Per ridirezionare un file un C si utilizza la funzione dup e dup2

**#include <unistd.h>**

**int dup(int oldfd);**

**int dup2(int oldfd, int newfd);**

La funzioni dup() e la funzione dup2() creano una copia del file

descriptor oldfd, dup() attribuisce al nuovo file descriptor, il più

piccolo intero non usato, dup2() crea newfd come copia di oldfd,

chiudendo prima newfd se è necessario.

Il vecchio e nuovo file descriptor possono essere utilizzati

interscambiabilmente, essi condividono locks, puntatori di file

position e flag, ad eccezione del flag close-on-exec.

Per esempio è possibile effettuare una lseek() su un file descriptor

e ritrovarsi la posizione modificata su entrambi i file descriptor.

Le funzioni dup() e dup2() ritornano il nuovo file descriptor in

caso di sucesso, oppure -1 in caso di errore.

Non è necessario fornire il percorso assoluto dei programmi da

eseguire poiché utilizziamo execvp.

Segnali

I segnali vengono lanciati attraverso la chiamata della funzione

kill che ha la seguente sintassi

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

pid indica il numero del processo al quale si vuole inviare un

determinato segnale, e sig il segnale che si vuole inviare.

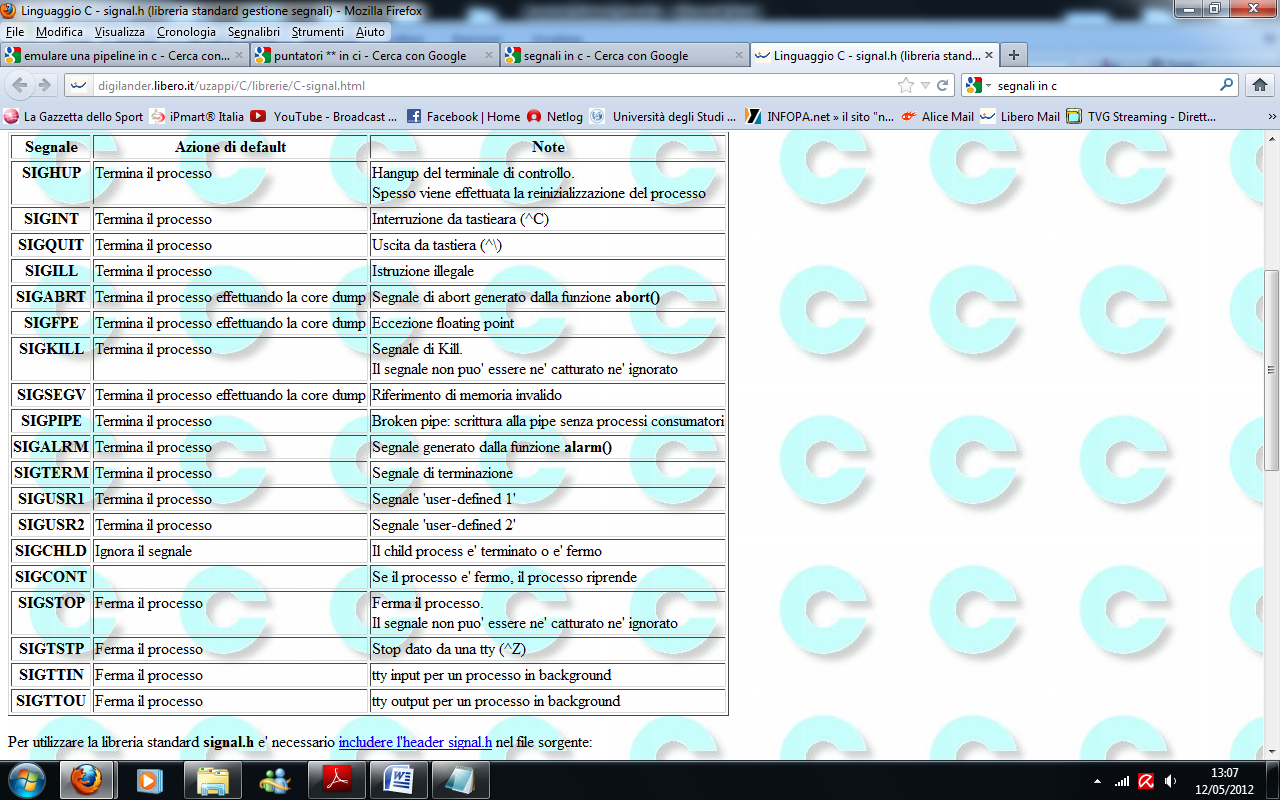
pid può anche essere:

0 per tutti i processi del gruppo di quello corrente

-1 per tutti i processi tranne il primo nella tavola dei processi

<-1 il segnale è spedito a tutti i processi del gruppo -pid

sig il segnale che si vuole inviare è può essere uno di questi:



signal()

L’intercettazione di un segnale utente avviene attraverso una

funzione definita dal programmatore attraverso La

funzione signal() che stabilisce a quale funzione il controllo deve

passare quando il segnale è catturato.

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

La funzione signal() stabilisce che quando il processo corrente

riceve il segnale signum, il controllo deve passare come è

specificato dal signal handler sighandler.

Per specificare il segnale signum può essere impiegato uno dei

valori costanti definiti nell'header standard (o in un altro header

da questo richiamato) oppure un valore numerico.

sighandler puoò essere o una funzione utente oppure uno dei

valori SIG\_IGN o SIG\_DFL.

Quando il processo riceve il segnale signum, esso può

comportarsi in 3 modi differenti a seconda che sighandler sia

stato impostato come:

SIG\_IGN Il segnale viene ignorato.

SIG\_DFLViene 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 né

ignorati né catturati.

In caso di successo la funzione ritorna il precedente valore della

funzione signal handler , SIG\_ERR in caso di errore.

Il prototype della funzione signal può essere semplificato nella

seguente forma:

**#include <signal.h>**

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

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

Ora appare più 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

uno di tipo puntatore a funzione (la quale a sua volta accetta

un argomento intero e torna void).

Il tipo ritornato da signal è un puntatore a funzione che accetta un

argomento intero e torna void.

raise()

L'altra funzione da considerare è raise(), con la quale si attiva

volontariamente un segnale, dal quale poi dovrebbero o

potrebbero sortire delle conseguenze, come stabilito in una fase

precedente attraverso signal(). La funzione raise() è molto

semplice:

**int raise (int sig);**

La funzione ha come argomento il segnale da attivare e

restituisce:

0 in caso di successo,

!=0 altrimenti.

Vediamo un esempio di raise, con un programma che all'invio dei

segnali relativi mostra un messaggio sullo schermo.

Quindi dopo l'esecuzione del programma, viene inviato il segnale

SIGUSR1 con il comando kill -s SIGUSR1 <pid>.

Se si inviano altri segnali SIGUSR1 finché il signal handler

relativo non termina tramite invio seguito da "bg <numero job>"

(il segnale di stop SIGSTOP è invocato dallo stesso processo con

raise), questi non vengono eseguiti ma restano sospesi.

Dopo aver pressato l'invio comunque il gestore verrà eseguito

soltanto una volta, anche se sono stati ricevuti più di un segnale

SIGUSR1.

La ricezione di SIGUSR2 (kill -s SIGUSR2 <pid>), invece

interrompe il signal handler di SIGUSR1.

Attenzione, la funzione printf non è sicura con i segnali.

Pause e Alarm

Un’altra funzione utile è pause() che sospende il processo

chiamante sino alla ricezione di un segnale e risulta utile per

sincronizzazioni interprocesso basate su segnali.

**int pause(void);**

La funzione alarm() permette di lanciare un segnale di tipo

SIGALRM ad un determinato processo dopo un certo numero di

secondi, ritorna il numero di secondi mancanti all'invio del

segnale.

**unsigned int alarm (unsigned int secs);**

Bloccare i Segnali

sigset\_t

particolare tipo di dato al fine di definire quali

segnali debbano essere bloccati: sigset\_t, implementato sia come

un intero che come una struttura e definito in signal.h.

L'inizializzazione di una variabile di questo tipo deve essere fatto

in uno dei seguenti modi:

inizializzato vuoto attraverso la funzione

int sigemptyset(sigset\_t \*SET)

in seguito si aggiungono singolarmente i segnali che devono

far parte dell'insieme.

inizializzato pieno attraverso la funzione

int sigfillset(sigset\_t \*SET)

in seguito si tolgono singolarmente quei segnali che non

devono far parte dell'insieme.

L'insieme dei segnali correntemente bloccati, durante l'esecuzione

di un processo, prende il nome di signal mask ed ogni processo

ne ha uno.

Ogni processo figlio eredita dal padre la propria signal mask,

ogni processo può comunque intervenire su di essa,

modificandola, attraverso l'uso delle funzioni:

int sigaddset (sigset\_t \*SET, int SIGNUM)

per aggiungere il segnale *SIGNUM* al *signal set* puntato

da *SET*. Ritorna 0 in caso di successo e -1 in caso di fallimento.

Il valore della variabile *errno* può assumere il

valore *EINVAL* nel caso si tenti di passare alla funzione un

segnale non valido.

int sigdelset (sigset\_t \*SET, int SIGNUM)

per elimina il segnale SIGNUM dal *signal set* puntato da SET.

Per il resto valgono le stesse caratteristiche della funzione

precedentemente trattata.

int sigismember (const sigset\_t \*SET, int SIGNUM)

Come facilmente intuibile questa è una funzione di test. Viene

infatti verificato che il segnale SIGNUM appartenga al signal

set puntato da SET. In caso affermativo viene restituito il valore

1 altrimenti, in caso negativo, 0. Qualora si verificasse un

errore il valore restituito è -1. La variabile errno può assumere

il valore EINVAL nel caso si tenti di passare alla funzione un

segnale non valido.

int sigprocmask (int HOW, const sigset\_t \*restrict SET, sigset\_t

\*restrict OLDSET)

Questa funzione è utilizzata, per la manipolazione della *signal*

*mask* , e l'effettivo comportamento di questa chiamata è

determinato dalla variabile *HOW*.

Se *HOW=SIG\_BLOCK* l'insieme dei segnali definito

in *SET* viene aggiunto all'insieme dei segnali della *signal*

*mask* corrente.

Se *HOW=SIG\_UNBLOCK* l'insieme dei segnali definito

in *SET* viene rimosso dalla *signal mask*.

Se *HOW=SIG\_SETMASK* l'insieme dei segnali definito

in *SET* viene utilizzato come nuova *signal mask*, la

variabile *OLDSET* è utilizzata per tenere traccia della *signal*

*mask* precedente alla modifica, qualora si volesse, ad esempio,

tornare a riutilizzarla. In caso di successo la funzione ritorna il

valore 0, -1 in caso di fallimento.

int sigpending (sigset\_t \*SET)

Questa funzione è utilizzata per conoscere quali segnali sono

pendenti in un determinato momento. Le informazioni relative vengono memorizzate in SET. Il valore 0 viene ritornato in

caso di successo, -1 in caso di errore.

Infine la funzione sigsuspend la cui sintassi è la seguente

int sigsuspend(sigset\_t \*SET)

rimpiazza la *signal mask* con la variabile *SET*, bloccando quindi

tutti i segnali in esso definiti.

Il processo viene sospeso fino all'arrivo di un segnale che non fa

parte di *SET*. In seguito all'avvenuta ricezione di questo segnale

viene eseguito l'eventuale signal handler e la funzione ritorna

ripristinando la *signal mask* precedente alla sua chiamata.

Consideriamo come esempio il seguente programma che durante

l'esecuzione del signal handler, determinati segnali vengono

bloccati.

Si procede in modo che dopo l'esecuzione del programma viene

inviato il segnale SIGUSR1 al processo con il comando

kill -s SIGUSR1 <pid>.

Se SIGUSR2 viene inviato prima di digitare invio all'interno del

signal handler per SIGUSR1, esso viene sospeso e bloccato, dopo

il primo invio verrà sbloccato, e la ricezione di SIGUSR2 dopo il

primo invio viene invece gestita immediatamente.

Il blocco di SIGUSR2 si ottiene tramite le funzioni sigemptyset,

sigaddset, e sigprocmask.

SIGCHLD

utilizziamo il segnale SIGCHLD per

gestire la terminazione dei processi figli con relativo rilascio delle

risorse.

Il padre genera dei processi figli che terminano in un tempo

casuale differente e attende la loro terminazione.

All'arrivo di SIGCHLD il padre controlla se ci sono figli che

hanno terminato e rilascia le sue risorse.

Poiché è possibile che mentre è in esecuzione l'handler di

SIGCHLD terminano più di un figlio, e solo un SIGCHLD risulta

pendente è necessario un ciclo while.

Per comunicare tra processi tramite variabili esse devono essere

di tipo volatile.

Il qualificatore volatile serve segnalare al compilatore che la

variabile può essere modificata in modo non deterministico,

quindi non prevedibile in base al programma, da qualcosa di esterno, come per esempio un altro processo, il sistema operativo,

o un dispositivo hardware.

Se ad esempio in un programma si utilizza un ciclo in cui si testa

continuamente il valore di una variabile, il compilatore può

effettuare delle ottimizzazioni sul codice, inserendo la variabile in

un registro in memoria, in modo da rendere più veloce l'accesso.

Così facendo però non si potranno vedere le modifiche che

occorrono, perché un dispositivo o un altro processo, accede

all'indirizzo della variabile in memoria cambiandola.

Per evitare ciò basta dichiarare la variabile di tipo volatile