Gestione dei processi

SLEEP(3) *sospende il processo per seconds secondi*

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Sospende il thread chiamante fino a quando non sono trascorsi *seconds* secondi, o fino a quando non arriva un segnale che non è ignorato. La sospensione può avere una durata maggiore di *seconds*, in relazione allo scheduling o all'attività del sistema.
@param seconds Il numero di secondi per cui il thread viene sospeso.
@return Restituisce 0 se il tempo richiesto è trascorso, o il numero di secondi rimanenti alla sospensione se la chiamata è stata interrotta da un gestore di segnali.

FORK(2) *duplica il processo*

```
#include <unistd.h>
pid_t fork();
```

Crea un nuovo processo duplicando il processo chiamante. Il nuovo processo, detto figlio, è un duplicato esatto del processo chiamante, detto padre, ad eccezione dei seguenti punti:
• il figlio ha il proprio PID univoco;
• il PPID del figlio è uguale all'ID del processo che ha chiamato la *fork()*;
• il figlio ha una propria copia dei descrittori dei file aperti e dei dati, ed una locazione di memoria dedicata;
• nel figlio tutti gli allarmi precedentemente impostati e i segnali pendenti risultano resettati.
@return In caso di successo, nel padre restituisce il PID del processo figlio, e nel figlio restituisce 0. In caso di fallimento, nel padre restituisce −1 e nessun processo figlio è creato.

GETPID(2) *recupera l'ID del processo*

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid();
```

@return Restituisce l'ID del processo chiamante.

```
pid_t getppid();
```

@return Restituisce l'ID del padre del processo chiamante.

EXIT(3) *provoca la normale terminazione del processo*

```
#include <stdlib.h>
void exit(int status);
```

Termina normalmente il processo chiamante. Al padre restituisce lo stato d'uscita *status* ed invia il segnale *SIGCHLD*. Gli eventuali figli ancora in esecuzione vengono assegnati al processo *init*.
@param status Lo stato di uscita da restituire al processo padre (di solito 0 significa che il processo è terminato con successo).

WAIT(2) *sospende il processo fino alla terminazione di un figlio*

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Sospende l'esecuzione del processo chiamante fino a quando non termina uno dei suoi figli. Quando un figlio termina senza che il padre abbia atteso la sua conclusione, il figlio entra nello stato "zombie", e le risorse associate ad esso non vengono liberate dal sistema per tenere traccia dello stato di uscita. Se il padre chiama *wait()* mentre ci sono uno o più figli in stato di zombie, allora la funzione ritorna immediatamente e tutte le risorse del figlio vengono liberate. Se un processo padre termina, allora gli eventuali figli zombie vengono adottati da *init*, il processo eseguito dal kernel che è padre di tutti i processi (PID = 1), che effettua automaticamente una *wait()* per rimuovere gli zombie.
@retval status Lo stato di uscita del processo figlio, proveniente da un *return* nel *main()* o da una *exit()*. L'header s*ys/wait.h* contiene alcune macro per interpretare lo stato di uscita.
@return Restituisce l'ID del processo figlio che ha terminato, o −1 in caso di fallimento.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Sospende l'esecuzione del processo chiamante fino a quando non termina il figlio il cui ID è pari a *pid*.
@param pid L'ID del processo figlio da attendere.
@retval status Lo stato di uscita del processo figlio.
@param options Se viene specificato *WNOHANG*, ritorna immediatamente se nessun figlio è terminato.
@return Restituisce l'ID del processo figlio che ha terminato, o −1 in caso di fallimento.

Esecuzione di programmi

EXECVE(2) EXEC(3) *esegue un programma*

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

Sostituisce l'immagine del processo chiamante con il programma *path*, i parametri *arg(v)* e l'environment *envp*. Vengono mantenuti solo i descrittori di file esistenti (compresi *stdin*, *stdout*, *stderr*).
@param path Il percorso del programma da eseguire. e*xeclp()* e *execvp()* cercano il programma nel path corrente.
@param arg(v) Gli argomenti da passare al programma. Il primo argomento è il nome con cui verrà identificato il nuovo processo.
@param envp L'environment da passare al programma.
@return Restituisce −1 in caso di fallimento.

SYSTEM(3) *invoca un comando all'interno di una shell*

```
#include <stdlib.h>
int system(const char *command);
```

Esegue all'interno di una shell il comando *command* creando un nuovo environment, e ritorna quando il comando è stato completato. L'esecuzione del comando avviene come se si fosse effettuata una *fork()* ed il figlio avesse eseguito:

```
execl("/bin/sh", "sh", "-c", command, (char*)0);
```

@param command Il comando da invocare.
@return Restituisce lo stato di uscita del comando in caso di successo, o −1 in caso di fallimento.

Segnali

SIGNAL(2) *istanzia un gestore di segnali*

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Stabilisce che, alla ricezione del segnale *signum*, il processo deve passare il controllo al gestore *handler*.
@param signum Il segnale a cui assegnare il gestore di segnali.
@param handler Alla ricezione del segnale:
• se *handler* è *SIG_IGN*, il segnale viene ignorato;
• se *handler* è *SIG_DFL*, viene eseguita l'azione predefinita associata al segnale;
• se *handler* è il puntatore a una funzione utente il cui prototipo è del tipo:

```
void ...(int);
```

il segnale viene catturato e tale funzione viene chiamata in modo sincrono passando ad essa il parametro *signum*.
I segnali *SIGKILL* e *SIGSTOP* non possono essere catturati né ignorati, perciò viene sempre eseguita l'azione predefinita.
@return Restituisce il valore precedente del gestore di segnali in caso di successo, o *SIG_ERR* in caso di errore.

KILL(2) *invia un segnale*

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Invia il segnale *sig* a uno o più processi. Non è possibile inviare segnali al processo *init* per evitare la sua accidentale terminazione.
@param pid Il segnale *sig* viene inviato:
• se *pid* è maggiore di 0, al processo il cui ID è pari a *pid*;
• se *pid* è uguale a −1, a tutti i processi del sistema (tranne *init*);
• se *pid* è minore di −1, a tutti i processi di group ID uguale al valore assoluto di *pid*.
@param sig Il segnale da inviare.
@return Restituisce 0 in caso di successo, −1 in caso di errore.

PAUSE(2) *sospende il processo fino all'arrivo di un segnale*

```
#include <unistd.h>
int pause();
```

Sospende il processo fino all'arrivo di un segnale.
@return Ritorna solo quando viene eseguito un gestore di segnali e questo ritorna. Il valore ritornato è sempre −1.

ALARM(2) *imposta un allarme temporizzato*

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Invia al processo chiamante stesso il segnale *SIGALRM* tra *seconds* secondi. La durata può aumentare in caso di particolari operazioni svolte dal sistema, o per via dello scheduling. Solamente un allarme per volta può essere schedulato: chiamate successive rischedulano l'allarme.
@param seconds Il numero dei secondi tra cui inviare il segnale *SIGALRM*. Se 0, disattiva l'allarme eventualmente impostato in precedenza.
@return Restituisce il numero di secondi rimanenti dell'allarme precedentemente schedulato, o 0 se nessun allarme era schedulato.

Pipe

PIPE(2) *crea una pipe*

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

Crea una pipe, un canale di dati unidirezionale (half-duplex) che può essere usato per la comunicazione tra processi. Il vettore *pipefd* di solito ritorna due descrittori di file che si riferiscono alle estremità della pipe: *pipefd[0]* si riferisce all'estremità di lettura della pipe, *pipefd[1]* si riferisce all'estremità di scrittura della pipe. I dati scritti all'estremità di scrittura della pipe vengono salvati in un buffer dal kernel finché non vengono letti dall'estremità di lettura della pipe. Lettura e scrittura avvengono con politica FIFO.
@retval pipefd Il vettore in cui posizionare i due descrittori di file.
@return Restituisce 0 in caso di successo, −1 in caso di errore.

READ(2) *legge da un descrittore di file*

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Legge fino a *count* caratteri dal descrittore di file *fd* salvandoli nel buffer *buf*, e avanza la posizione del file del numero di caratteri letti. Se la pipe contiene meno caratteri di quanto richiesto, ritorna solo i caratteri letti. Se la pipe è vuota, sospende il processo finché la pipe non viene scritta.
@param fd Il descrittore di file che corrisponde all'estremità di lettura.
@retval buf Il buffer in cui salvare i caratteri letti.
@param count Il numero massimo di caratteri da leggere.
@return Restituisce il numero di caratteri letti (0 indica la fine del file) in caso di successo, −1 in caso di errore.

WRITE(2) *scrive in un descrittore di file*

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Scrive fino a *count* caratteri dal buffer *buf* nel descrittore di file *fd*. Se l'estremità di lettura è chiusa, lancia il segnale *SIGPIPE*. Se la pipe è piena, sospende il processo finché la pipe non viene letta.
@param fd Il descrittore di file che corrisponde all'estremità di scrittura.
@param buf Il buffer da cui leggere i caratteri.
@param count Il numero massimo di caratteri da scrivere.
@return Restituisce il numero di caratteri scritti in caso di successo, −1 in caso di errore.

CLOSE(2) *chiude un descrittore di file*

```
#include <unistd.h>
int close(int fd);
```

Chiude un descrittore di file, e le risorse associate al descrittore di file vengono liberate.
@param fd Il descrittore di file da chiudere.
@return Restituisce 0 in caso di successo, −1 in caso di errore.

Pthread

PTHREAD_CREATE(3) *crea un nuovo thread*

#include <pthread.h>

int **pthread_create**(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

Crea un nuovo thread nel processo chiamante. Il nuovo thread avvia l'esecuzione invocando la funzione *start_routine()*, a cui viene passato il parametro *arg*. Il nuovo thread termina in uno dei seguenti modi:

- chiama *pthread_exit()*;
- ritorna da *start_routine()*;
- è cancellato tramite *pthread_cancel()*;
- uno dei thread nel processo chiama *exit()*, o il thread principale effettua un *return* dal *main()* → tutti i thread nel processo terminano.

Il nuovo thread eredita la signal mask dal creatore, ma il suo set di segnali pendenti è inizialmente vuoto.

@retval thread L'identificatore del thread generato.
@param attr Se viene specificato *NULL*, il nuovo thread viene creato con gli attributi predefiniti.
@param start_routine La funzione di partenza eseguita dal thread all'inizio della sua esecuzione.
@param arg Il parametro passato alla funzione *start_routine*.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

PTHREAD_SELF(3) *ottiene l'ID del thread chiamante*

#include <pthread.h>

pthread_t **pthread_self**();

Ottiene l'identificatore del thread chiamante, che è lo stesso valore ritornato dalla chiamata a *pthread_create()* che ha creato questo thread.
@return Restituisce l'ID del thread chiamante.

PTHREAD_EQUAL(3) *confronta due ID di thread*

#include <pthread.h>

int **pthread_equal**(pthread_t t1, pthread_t t2);

Confronta due identificatori di thread. Essi non possono essere confrontati direttamente perché sono identificatori opachi, cioè non possono essere stampati o confrontati esplicitamente.
@param t1 Il primo ID di thread da confrontare.
@param t2 Il secondo ID di thread da confrontare.
@return Restituisce 0 se i due ID di thread sono diversi.

PTHREAD_EXIT(3) *termina il thread chiamante*

#include <pthread.h>

void **pthread_exit**(void *retval);

Termina il thread chiamante. Vengono rilasciate le risorse specifiche del thread, ma non le risorse condivise a livello di processo (mutex, variabili condizionali, semafori, descrittori di file). Dopo che l'ultimo thread in un processo termina, il processo termina come se fosse chiamata *exit()* con lo stato di uscita 0, e le risorse condivise a livello di processo vengono rilasciate. L'esecuzione di un *return* dalla funzione di partenza del thread (escluso il *main()* del thread principale) comporta una chiamata implicita a *pthread_exit()*, il cui stato di uscita è il valore di ritorno della funzione. Per consentire agli altri thread di continuare l'esecuzione, il thread principale deve chiamare *pthread_exit()* invece di *exit()*, altrimenti tutti i thread del processo terminano.
@param retval Se il thread è joinable, il valore di uscita reso disponibile per un altro thread nello stesso processo che ha chiamato *pthread_join()*.

PTHREAD_JOIN(3) *attende la terminazione di un thread*

#include <pthread.h>

int **pthread_join**(pthread_t thread, void **retval);

Attende la terminazione del thread *thread*. Se quel thread è già terminato, ritorna immediatamente. Il thread *thread* deve essere joinable, altrimenti si verifica l'errore *EINVAL*.
@param thread L'ID del thread da attendere.
@retval retval Il valore di uscita proveniente da *pthread_exit()*. Vale *PTHREAD_CANCELED* se il thread è stato cancellato.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

PTHREAD_CANCEL(3) *termina un thread*

#include <pthread.h>

int **pthread_cancel**(pthread_t thread);

Termina il thread *thread*. È come se il thread indicato eseguisse *pthread_exit()* specificando come valore di uscita *PTHREAD_CANCELED*.
@param thread L'ID del thread da cancellare.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

PTHREAD_DETACH(3) *rende un thread detached*

#include <pthread.h>

int **pthread_detach**(pthread_t thread);

Comunica all'ambiente che le risorse allocate per il thread possono essere rilasciate immediatamente dopo la sua terminazione, in quanto non si è intenzionati ad effettuare *pthread_join()*. Il thread non è più joinable: a qualsiasi successiva *pthread_join()* si verifica l'errore *EINVAL*.
@param thread L'ID del thread da rendere detached.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

Semafori POSIX

SEM_INIT(3) *inizializza un semaforo*

#include <semaphore.h>

int **sem_init**(sem_t *sem, int pshared, unsigned int value);

Inizializza il semaforo *sem* al valore iniziale *value*. Il valore del semaforo non può scendere al di sotto di 0, e *value* è il numero massimo di processi che possono accedere in contemporanea ad una regione critica.

@param sem Il semaforo da inizializzare.

@param pshared Se viene specificato 0, il semaforo è privato a livello di processo (condiviso tra i thread del processo), altrimenti è condiviso con eventuali processi figlio creati successivamente.

@return Restituisce 0 in caso di successo, −1 in caso di errore.

SEM_WAIT(3) *acquisisce il lock di un semaforo*

#include <semaphore.h>

int **sem_wait**(sem_t *sem);

Se il valore del semaforo è maggiore di 0, allora decrementa il semaforo *sem*, acquisisce il lock e ritorna immediatamente. Se il valore del semaforo è pari a 0, allora si blocca finché il valore del semaforo non diventa maggiore di 0, oppure un gestore di segnali non interrompe la chiamata.

@param sem Il semaforo di cui acquisire il lock.

@return Restituisce 0 in caso di successo, −1 in caso di errore (in questo caso il valore del semaforo viene lasciato invariato).

int **sem_trywait**(sem_t *sem);

Se il valore del semaforo è maggiore di 0, allora decrementa il semaforo *sem*, acquisisce il lock e ritorna immediatamente. Se il valore del semaforo è pari a 0, allora si verifica l'errore *EAGAIN* (senza bloccarsi).

@param sem Il semaforo di cui acquisire il lock.

@return Restituisce 0 in caso di successo, −1 in caso di errore (in questo caso il valore del semaforo viene lasciato invariato).

SEM_POST(3) *rilascia il lock di un semaforo*

#include <semaphore.h>

int **sem_post**(sem_t *sem);

Incrementa il semaforo *sem* rilasciandone il lock. Se il valore precedente del semaforo era pari a 0, un processo bloccato in una chiamata *sem_wait()* verrà risvegliato e procederà ad acquisire il lock del semaforo. Quale processo otterrà effettivamente il diritto di procedere non è prevedibile, poiché dipende dalla politica di scheduling.

@param sem Il semaforo di cui rilasciare il lock.

@return Restituisce 0 in caso di successo, −1 in caso di errore (in questo caso il valore del semaforo viene lasciato invariato).

SEM_GETVALUE(3) *recupera il valore di un semaforo*

#include <semaphore.h>

int **sem_getvalue**(sem_t *sem, int *sval);

Recupera il valore corrente del semaforo *sem* (senza bloccarsi).

@param sem Il semaforo di cui esaminare il valore corrente.

@retval sval Il valore corrente del semaforo. Se è positivo, *sval* rappresenta il numero di processi che possono ancora acquisire il lock senza essere bloccati. Se uno o più processi sono bloccati in attesa di acquisire il semaforo, a seconda dell'implementazione *sval* può valere 0 o un numero negativo il cui valore assoluto è pari al numero di processi in attesa (Linux ritorna 0).

@return Restituisce 0 in caso di successo, −1 in caso di errore.

SEM_DESTROY(3) *distrugge un semaforo*

#include <semaphore.h>

int **sem_destroy**(sem_t *sem);

Distrugge il semaforo *sem*.

@param sem Il semaforo da distruggere.

@return Restituisce 0 in caso di successo, −1 in caso di errore. A seconda dell'implementazione, può restituire −1 se si prova a distruggere un semaforo su cui altri processi sono attualmente bloccati in attesa.

Mutex

PTHREAD_MUTEX_INIT(3) *inizializza e distrugge un mutex*

#include <pthread.h>

int **pthread_mutex_init**(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

Inizializza il mutex *mutex*. Non viene specificato il valore iniziale perché il mutex è un semaforo binario: viene inizializzato sempre a 1.
@param mutex Il mutex da inizializzare.
@param attr Se viene specificato *NULL*, il mutex viene inizializzato con gli attributi predefiniti.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

int **pthread_mutex_destroy**(pthread_mutex_t *mutex);

Distrugge il mutex *mutex* rilasciandone la memoria occupata. Il mutex non sarà più utilizzabile.
@param mutex Il mutex da distruggere.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

PTHREAD_MUTEX_LOCK(3) *acquisisce e rilascia un mutex*

#include <pthread.h>

int **pthread_mutex_lock**(pthread_mutex_t *mutex);

Acquisisce il lock del mutex *mutex*. Se il lock del mutex è già acquisito, blocca il chiamante in attesa che il mutex diventi disponibile.
@param mutex Il mutex di cui acquisire il lock.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

int **pthread_mutex_trylock**(pthread_mutex_t *mutex);

Acquisisce il lock del mutex *mutex*. Se il lock del mutex è già acquisito, si verifica immediatamente l'errore EBUSY (senza bloccare il chiamante).
@param mutex Il mutex di cui acquisire il lock.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.

int **pthread_mutex_unlock**(pthread_mutex_t *mutex);

Rilascia il lock del mutex *mutex*.
@param mutex Il mutex di cui rilasciare il lock.
@return Restituisce 0 in caso di successo, un codice di errore in caso di errore.