Mini-man

Ce document contient des *extraits* de la version francophone du manuel Debian GNU/Linux (spécifiquement du paquet manpages-fr-dev version 3.65d1p1-1). Le source d'origine est disponible à http://sources.debian.net/src/manpages-fr/3.65d1p1-1/.

Ce mini-man se découpe de la façon suivante :

— une section « whatis » comme index [1],
— les pages extraites des sections 2 et 3 (appels systèmes et fonctions de la libc),
— quelques pages extraites de la section 7, décrivant des mécanismes.

VALEUR RENVOYÉE (CAS GÉNÉRAL)

Une grande partie des appels systèmes et fonctions documentés ici utilise une même convention pour leur valeur de retour : si l'appel réussit, il renvoie 0 ; s'il échoue, il renvoie -1 et remplit `errno` en conséquence.

Les détails sur la valeur renvoyée n'ont été conservés que pour les cas où ils diffèrent de cette convention. C'est en particulier le cas de toutes les fonctions *pthread_*...

1 whatis

1. La commande `whatis` sert à afficher la description de pages du manuel.

2 access (2)

NOM

access — Vérifier les permissions utilisateur d'un fichier

SYNOPSIS

int access(const char **pathname*, **int** *mode*);

DESCRIPTION

access() vérifie si le processus appelant peut accéder au fichier *pathname*. Si *pathname* est un lien symbolique, il est déréférencé.

Le *mode* indique les vérifications d'accès à effectuer. Il prend la valeur *F_OK* ou un masque contenant un OU binaire d'une des valeurs *R_OK*, *W_OK* et *X_OK*. *F_OK* teste l'existence du fichier. *R_OK*, *W_OK* et *X_OK* testent si le fichier existe et autorise respectivement la lecture, l'écriture et l'exécution.

VALEUR RENVOYÉE

En cas de succès (toutes les permissions demandées sont autorisées, ou *mode* vaut *F_OK* et le fichier existe), 0 est renvoyé. En cas d'erreur (au moins une permission de *mode* est interdite, ou *mode* vaut *F_OK* et le fichier n'existe pas, ou d'autres erreurs se sont produites), -1 est renvoyé et `errno` contient le code d'erreur.

ERREURS

EACCES L'accès serait refusé au fichier lui-même, ou il n'est pas permis de parcourir l'un des répertoires du préfixe du chemin de *pathname* (consultez aussi path_resolution(7)).
ELOOP Trop de liens symboliques ont été rencontrés en parcourant *pathname*.
ENAMETOOLONG *pathname* est trop long.
ENOENT Un composant du chemin d'accès *pathname* n'existe pas ou est un lien symbolique pointant nulle part.
ENOTDIR Un élément du chemin d'accès *pathname* n'est pas un répertoire.
EROFS On demande une écriture sur un système de fichiers en lecture seule.

NOTES

La fonction *access*() déréférence toujours les liens symboliques.

Ces appels renvoient une erreur si l'un des types d'accès de *mode* est refusé, même si d'autres types indiqués dans *mode* sont autorisés.

3 alarm (2)

NOM

alarm — Programmer un réveil pour l'émission d'un signal

SYNOPSIS

unsigned int alarm(unsigned int *nb_sec*);

DESCRIPTION

alarm() programme une temporisation pour qu'elle envoie un signal `SIGALRM` au processus appelant dans *nb_sec* secondes.

Si *seconds* vaut zéro, toute alarme en attente est annulée.

Dans tous les cas, l'appel *alarm*() annule l'éventuelle programmation précédente.

VALEUR RENVOYÉE

alarm() renvoie le nombre de secondes qu'il restait de la programmation précédente (annulée), ou zéro si aucune alarme n'avait été planifiée auparavant.

NOTES

Les alarmes créées par `alarm()` sont conservées lors des appels à execve(2), mais ne le sont pas par les fils créés avec fork(2).

Les délais dûs au multitâche peuvent, comme toujours, retarder le déclenchement d'une alarme d'une durée arbitraire.

4 chdir (2)

NOM

chdir — Changer le répertoire courant

SYNOPSIS

int chdir(const char **path***);**

DESCRIPTION

chdir() remplace le répertoire de travail courant du processus appelant par celui indiqué dans le chemin *path*.

NOTES

Le répertoire de travail est le point de départ pour l'interprétation des chemins relatifs (qui ne commencent pas par « / »).

Un processus fils créé avec fork(2) hérite du répertoire de travail courant de son père. Le répertoire de travail courant n'est pas modifié par un execve(2).

5 chmod, fchmod (2)

NOM

chmod, fchmod — Modifier les permissions d'accès à un fichier

SYNOPSIS

int chmod(const char **pathname*, **mode_t** *mode*);
int fchmod(int *fd*, **mode_t** *mode*);

DESCRIPTION

Les appels système *chmod*() and *fchmod*() modifient les permissions d'un fichier. Ils diffèrent seulement dans la façon dont le fichier est indiqué :

— *chmod*() modifie les permissions du fichier indiqué dont le nom est fourni dans *pathname*, qui est déréférencé s'il s'agit d'un lien symbolique.
— *fchmod*() modifie les permissions du fichier référencé par le descripteur de fichier ouvert *fd*.

Les nouvelles permissions du fichier sont indiquées dans *mode*, qui est un masque de bit créé par un OU bit à bit de zéro ou plusieurs des valeurs suivantes :

***S_ISUID* (04000)** SUID (Définir l'UID effectif d'un processus lors d'un execve(2))
***S_ISGID* (02000)** SGID (Définir le GID effectif d'un processus lors d'un execve(2) ; verrou obligatoire, comme décrit dans fcntl(2) ; prendre un nouveau groupe de fichiers dans le répertoire parent, comme décrit dans chown(2) et mkdir(2))
***S_ISVTX* (01000)** définir le bit « sticky » (attribut de suppression restreinte, comme décrit dans unlink(2))
***S_IRUSR* (00400)** accès en lecture pour le propriétaire
***S_IWUSR* (00200)** accès en écriture pour le propriétaire
***S_IXUSR* (00100)** accès en exécution/parcours par le propriétaire (« parcours » s'applique aux répertoires, et signifie que le contenu du répertoire est accessible)
***S_IRGRP* (00040)** accès en lecture pour le groupe
***S_IWGRP* (00020)** accès en écriture pour le groupe
***S_IXGRP* (00010)** accès en exécution/parcours pour le groupe
***S_IROTH* (00004)** accès en lecture pour les autres
***S_IWOTH* (00002)** accès en écriture pour les autres
***S_IXOTH* (00001)** accès en exécution/parcours pour les autres

6 close (2)

NOM

close — Fermer un descripteur de fichier

SYNOPSIS

int close(int *fd*);

DESCRIPTION

close() ferme le descripteur *fd*, de manière à ce qu'il ne référence plus aucun fichier, et puisse être réutilisé. Tous les verrouillages (consultez fcntl(2)) sur le fichier qui lui était associé, appartenant au processus, sont supprimés (quel que soit le descripteur qui fut utilisé pour obtenir le verrouillage).

Si *fd* est le dernier descripteur de fichier qui se réfère à une description de fichier ouvert sous-jacente (consultez open(2)), les ressources associées à la description de fichier ouvert sont libérées. Si le descripteur était la dernière référence sur un fichier supprimé avec unlink(2), le fichier est effectivement effacé.

7 closedir (3)

NOM

closedir — Fermer un répertoire

SYNOPSIS

int closedir(DIR ***dirp***);**

DESCRIPTION

La fonction *closedir*() ferme le flux du répertoire associé à *dirp*. Un appel réussi à *closedir*() ferme aussi le descripteur de fichier sous-jacent associé à *dirp*. Après cet appel, le descripteur *dirp* du flux du répertoire n'est plus disponible.

ERREURS

EBADF Le descripteur de flux du répertoire, *dirp*, n'est pas valable.

8 dup, dup2 (2)

NOM

dup, dup2 — Dupliquer un descripteur de fichier

SYNOPSIS

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

DESCRIPTION

Ces appels système créent une copie du descripteur de fichier *oldfd*.

dup() utilise le plus petit numéro inutilisé pour le nouveau descripteur.

dup2() transforme *newfd* en une copie de *oldfd*, fermant auparavant *newfd* si besoin est, mais prenez note des points suivants.

— Si *oldfd* n'est pas un descripteur de fichier valable, alors l'appel échoue et *newfd* n'est pas fermé.

— Si *oldfd* est un descripteur de fichier valable et *newfd* a la même valeur que *oldfd*, alors *dup2*() ne fait rien et renvoie *newfd*.
Après un appel réussi à l'un de ces appels système, l'ancien et le nouveau descripteurs peuvent être utilisés de manière interchangeable. Ils référencent la même description de fichier ouvert (consultez open(2)) et ainsi partagent les pointeurs de position et les drapeaux. Par exemple, si le pointeur de position est modifié en utilisant lseek(2) sur l'un des descripteurs, la position est également changée pour l'autre.
Les deux descripteurs ne partagent toutefois pas l'attribut close-on-exec. L'attribut close-on-exec (*FD_CLOEXEC* ; consultez fcntl(2)) de la copie est désactivé, ce qui signifie qu'il ne sera pas fermé lors d'un exec().

VALEUR RENVOYÉE

Ces appels système renvoient le nouveau descripteur en cas de succès, ou -1 en cas d'échec, auquel cas `errno` contient le code d'erreur.

9 execl, execlp, execle, execv, execvp, execvpe (3)

NOM

execl, execlp, execle, execv, execvp, execvpe — Exécuter un fichier

SYNOPSIS

int execl(const char **path*, **const char** **arg*, **...);**
int execlp(const char **file*, **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 **file*, **char *const** *argv*[**]);**
int execvpe(const char **file*, **char *const** *argv*[**], char *const** *envp*[**]);**

DESCRIPTION

La famille des fonctions *exec*() remplace l'image du processus en cours par une nouvelle image du processus. Les fonctions décrites dans cette page sont en réalité des frontaux pour execve(2).

L'argument initial de toutes ces fonctions est le chemin d'accès du fichier à exécuter.

Les arguments *const char *arg* ainsi que les points de suspension des fonctions *execl*(), *execlp*(), et *execle*() peuvent être vues comme *arg0*, *arg1*, ..., *argn*. Ensemble, ils décrivent une liste d'un ou plusieurs pointeurs sur des chaînes de caractères terminées par des caractères nuls, qui constituent les arguments disponibles pour le programme à exécuter. Par convention, le premier argument doit pointer sur le nom du fichier associé au programme à exécuter. La liste des arguments *doit* se terminer par un pointeur NULL, et puisque ce sont des fonctions variadiques, ce pointeur doit être transtypé avec *(char *) NULL*.

Les fonctions *execv*(), *execvp*() et *execvpe*() utilisent un tableau de pointeurs sur des chaînes de caractères terminées par des caractères nuls, qui constituent les arguments disponibles pour le programme à exécuter. Par convention, le premier argument doit pointer sur le nom du fichier associé au programme à exécuter. Le tableau de pointeurs *doit* se terminer par un pointeur NULL.

Les fonctions *execle*() et *execvpe*() permettent à l'appelant d'indiquer l'environnement du processus à exécuter à l'aide du paramètre *envp*. Ce paramètre est un tableau de pointeurs sur des chaînes de caractères terminées par des caractères nuls, qui *doit* se terminer par un pointeur NULL. Les autres fonctions fournissent au nouveau processus l'environnement constitué par la variable externe *environ* du processus appelant.

Sémantique particulière pour execlp() et execvp()

Les fonctions *execlp*(), *execvp*() et *execvpe*() dupliqueront les actions de l'interpréteur de commandes dans la recherche du fichier exécutable si le nom fourni ne contient pas de barre oblique « / ». Le fichier est recherché dans la liste de répertoires, séparés par des deux-points, indiquée dans la variable d'environnement `PATH`.

Si le nom du fichier indiqué inclut une barre oblique, alors `PATH` est ignoré et le fichier dont le chemin est fourni est exécuté.

VALEUR RENVOYÉE

Une sortie des fonctions *exec*() n'intervient que si une erreur s'est produite. La valeur de retour est -1, et `errno` contient le code d'erreur.

10 execve (2)

NOM

execve — Exécuter un programme

SYNOPSIS

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

DESCRIPTION

execve() exécute le programme correspondant au fichier *filename*. Celui-ci doit être un exécutable binaire ou bien un script commençant par une ligne du type :

#! interpréteur [argument-optionnel]

Pour des détails sur ce dernier cas, consultez « Scripts » ci-dessous.

argv est un tableau de chaînes d'arguments passées au nouveau programme. Par convention, la première de ces chaînes devrait contenir le nom de fichier associé au fichier étant exécuté. *envp* est un tableau de chaînes, ayant par convention la forme *clé=valeur*, qui sont passées au nouveau programme comme environnement. *argv* ainsi que *envp* doivent se terminer par un pointeur NULL. Les arguments et l'environnement sont accessibles par le nouveau programme dans sa fonction principale, lorsqu'elle est définie comme :

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

En cas de réussite, *execve*() ne revient pas à l'appelant, et les segments de texte, de données (« data » et « bss »), ainsi que la pile du processus appelant sont remplacés par ceux du programme chargé.

Si le bit Set-UID est positionné sur le fichier *filename*, [...] l'UID effectif du processus appelant est modifié pour prendre celui du propriétaire du fichier. De même, lorsque le bit Set-GID est positionné, le GID effectif est modifié pour correspondre à celui du groupe du fichier.

L'UID effectif du processus est copié dans le Set-UID sauvé ; de la même manière, le GID effectif est copié dans le Set-GID sauvé. Ces copies ont lieu après toute modification d'ID effectif à cause des bits de permission Set-UID et Set-GID.

Si l'exécutable est au format ELF lié dynamiquement, l'interpréteur indiqué dans le segment PT_INTERP sera invoqué pour charger les bibliothèques partagées. Cet interpréteur est généralement `/lib/ld-linux.so.2` pour les fichiers binaires liés avec la glibc 2 (pour les fichiers binaires liés avec l'ancienne libc Linux 5, l'interpréteur était typiquement `/lib/ld-linux.so.1`).

Tous les attributs du processus sont préservés lors d'un *execve*(), à l'exception des suivants :

— Les signaux pour lesquels le processus avait placé un gestionnaire sont réinitialisés à leur valeur par défaut (consultez signal(7)).
— Les projections en mémoire ne sont pas conservées (mmap(2)).
— Les sémaphores nommés POSIX ouverts sont fermés (sem_overview(7)).
— Les temporisations POSIX ne sont pas conservées (timer_create(2)).
— Les flux de répertoires ouverts sont fermés (opendir(3)).

Notez également les points suivants :

— Tous les threads autre que l'appelant sont détruits lors d'un *execve*(). Les mutex, les variables de condition, et les autres objets de pthreads sont détruits.
— Par défaut, les descripteurs de fichier restent ouverts au travers d'un *execve*(). Les descripteurs marqués close-on-exec sont fermés ; consultez la description de *FD_CLOEXEC* dans fcntl(2).

Scripts

Un script est un fichier dont le bit d'exécution est activé et dont la première ligne est de la forme :

#! interpréteur [argument-optionnel]

L'*interpréteur* doit être un nom de fichier valide pour un exécutable qui n'est pas un script lui-même. Si l'argument *filename* de *execve*() indique un script, l'*interpréteur* sera appelé avec les arguments suivants :

interpréteur [argument-optionnel] *filename* arg...

où *arg...* est la liste de mots pointée par l'argument *argv* de *execve*(), commençant par *argv*[1].

VALEUR RENVOYÉE

En cas de réussite, *execve*() ne revient pas, en cas d'échec il renvoie -1 et `errno` contient le code d'erreur.

ERREURS

ENOENT Le fichier *filename* ou un script ou un interpréteur ELF n'existe pas, ou une bibliothèque partagée nécessaire pour le fichier ou l'interpréteur n'est pas disponible.

ENOEXEC Le fichier exécutable n'est pas dans le bon format, ou est destiné à une autre architecture.

NOTES

Linux ignore les bits Set-UID et Set-GID sur les scripts.

11 exit (3)

NOM

exit — Terminer normalement un processus

SYNOPSIS

void exit(int *status***);**

DESCRIPTION

La fonction *exit*() termine normalement un processus et la valeur *status & 0377* est envoyée au processus parent (consultez wait(2)).

Tous les flux ouverts du type stdio(3) sont vidés et fermés.

Le standard C spécifie deux constantes symboliques *EXIT_SUCCESS* et *EXIT_FAILURE* qui peuvent être passées à *exit*() pour indiquer respectivement une terminaison sans ou avec échec.

VALEUR RENVOYÉE

La fonction *exit*() ne revient jamais.

NOTES

Après *exit*(), le code de retour doit être transmis au processus parent. Il y a trois cas. Si le parent a défini *SA_NOCLDWAIT* ou s'il a défini le comportement de `SIGCHLD` à *SIG_IGN*, le code de retour est ignoré. Si le père était en attente de la fin de son fils, il reçoit le code de retour. Dans ces deux cas, le fils meurt immédiatement. Si le parent n'est pas en attente, mais n'a pas indiqué qu'il désire ignorer le code de retour, le processus fils devient un processus « zombie » (ce n'est rien d'autre qu'une coquille enveloppant le code de retour d'un octet), que le processus père pourra consultez ultérieurement grâce à l'une des fonctions wait(2).

Si l'implémentation supporte le signal `SIGCHLD`, celui-ci est envoyé au processus père. Si le père a défini *SA_NOCLDWAIT*, il n'est pas précisé si `SIGCHLD` est envoyé ou non.

12 fcntl (2)

NOM

fcntl — Manipuler un descripteur de fichier

SYNOPSIS

int fcntl(int *fd*, **int** *cmd*, **...** /* *arg* */ **);**

DESCRIPTION

fcntl() permet de se livrer à diverses opérations sur le descripteur de fichier *fd*. L'opération en question est déterminée par la valeur de l'argument *cmd*.

fcntl() prend un troisième paramètre optionnel. La nécessité de fournir ce paramètre dépend de *cmd*. le paramètre doit être du type indiqué entre parenthèses après chaque nom de commande *cmd* (dans la plupart des cas, le type requis est un *int*, et le paramètre est identifié en utilisant le nom *arg*), ou *void* est indiqué si le paramètre n'est pas nécessaire.

Attributs du descripteur de fichier

Les commandes suivantes manipulent les attributs associés à un descripteur de fichier. Actuellement, un seul attribut est défini : il s'agit de *FD_CLOEXEC*, l'attribut « close-on-exec ». Si le bit *FD_CLOEXEC* est 0, le descripteur de fichier reste ouvert au travers d'un execve(2), autrement il sera fermé.

***F_GETFD* (*void*)** Lire les attributs du descripteur de fichier ; *arg* est ignoré.
***F_SETFD* (*int*)** Positionner les attributs du descripteur de fichier avec la valeur précisée par *arg*.

Attribut d'état du fichier

Un descripteur de fichier dispose de certains attributs, initialisés par open(2) et éventuellement modifiés par *fcntl*(). Les attributs sont partagés entre les copies (obtenues avec dup(2), *fcntl*(F_DUPFD), fork(2), etc.) du même descripteur de fichier.

Les attributs et leurs sémantiques sont décrits dans la page open(2).

***F_GETFL* (*void*)** Obtenir le mode d'accès et les attributs d'état du fichier ; *arg* est ignoré.
***F_SETFL* (*int*)** Positionner les nouveaux attributs pour le descripteur de fichier à la valeur indiquée par *arg*. Les bits de mode d'accès (*O_RDONLY*, *O_WRONLY*, *O_RDWR*) et les attributs de création (*O_CREAT*, *O_EXCL*, *O_NOCTTY*, *O_TRUNC*) de *arg* sont ignorés. Sous Linux, cette commande ne peut changer que *O_APPEND*, *O_ASYNC*, *O_DIRECT*, *O_NOATIME* et *O_NONBLOCK*. Modifier les attributs *O_DSYNC* et *O_SYNC* est impossible, consultez *BOGUES* ci-dessous.

Verrouillages coopératifs

F_SETLK, *F_SETLKW* et *F_GETLK* servent à gérer les verrouillages d'enregistrements (de segments ou de régions de fichiers). Le troisième argument, *lock*, est un pointeur sur une structure qui a au moins les champs suivants (dans un ordre non indiqué).

```
struct flock {
    ...
    short l_type;    /* Type de verrouillage : F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* Interprétation de l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Décalage de début du verrouillage */
    off_t l_len;     /* Nombre d'octets du verrouillage */
    pid_t l_pid;     /* PID du processus bloquant notre verrou
                        (F_GETLK seulement) */
    ...
};
```

Les champs *l_whence*, *l_start* et *l_len* de cette structure indiquent l'intervalle d'octets à verrouiller. Des octets après la fin du fichier peuvent être verrouillés, mais pas des octets avant le début du fichier.

l_start est la position de début du verrou, et est interprété de façon relative : au début du fichier (si *l_whence* vaut *SEEK_SET*) ; à la position actuelle dans le fichier (si *l_whence* vaut *SEEK_CUR*) ; à la fin du fichier (si *l_whence* vaut *SEEK_END*). Dans les deux derniers cas, *l_start* peut être un nombre négatif, à partir du moment où la position fournie ne pointe pas avant le début du fichier.

l_len indique le nombre d'octets à verrouiller. Si *l_len* est positif, alors l'intervalle à verrouiller couvre les octets à partir de *l_start* jusqu'à *l_start*+*l_len*-1 (inclus). Indiquer 0 dans *l_len* a une signification particulière : cela verrouille tous les octets à partir de la position indiquée par *l_whence* et *l_start* jusqu'à la fin du fichier, quelle que soit la taille que prendra la fichier.

Le champ *l_type* peut servir à placer un verrou en lecture (*F_RDLCK*) ou en écriture (*F_WRLCK*) sur un fichier. Un nombre quelconque de processus peuvent tenir un verrou en lecture (partagé), sur une région d'un fichier, mais un seul peut avoir un verrou en écriture (exclusif). Un verrou en écriture exclut tous les autres verrous, aussi bien en lecture qu'en écriture. Un processus donné ne peut tenir qu'un seul verrou sur une région d'un fichier, si un nouveau verrou y est appliqué, alors le verrou précédent est converti suivant le nouveau type. Ceci peut entraîner le découpage, la réduction ou l'extension du verrou existant si le nombre d'octets du nouveau verrou ne coïncide pas exactement avec celui de l'ancien.

- ***F_SETLK* (*struct flock* *)** Acquérir (si *l_type* vaut *F_RDLCK* ou *F_WRLCK*) ou libérer (si *l_type* vaut *F_UNLCK*) le verrou sur les octets indiqués par les champs *l_whence*, *l_start*, et *l_len* de *lock*. Si un conflit avec un verrou tenu par un autre processus existe, cet appel renvoie -1 et positionne `errno` aux valeurs EACCES ou EAGAIN.
- ***F_SETLKW* (*struct flock* *)** Comme *F_SETLK*, mais attend la libération du verrou au lieu de retourner une erreur. Si un signal à intercepter est reçu pendant l'attente, l'appel est interrompu et renverra immédiatement (après retour du gestionnaire de signaux) la valeur -1. `errno` sera remplie avec la valeur EINTR ; consultez signal(7).
- ***F_GETLK* (*struct flock* *)** En entrée dans cette routine, *lock* décrit un verrou que nous aimerions placer sur le fichier. Si le verrouillage est possible, *fcntl*() ne le fait pas, mais renvoie

F_UNLCK dans le champ *l_type* de *lock* et laisse les autres champs de la structure inchangés. Si un ou plusieurs verrouillages incompatibles empêchaient l'action, alors *fcntl*() renvoie des informations sur l'un de ces verrous dans les champs *l_type*, *l_whence*, *l_start*, et *l_len* de *lock* et remplit *l_pid* avec le PID du processus tenant le verrou. Notez que l'information renvoyée par *F_GETLK* peut être devenue obsolète au moment où l'appelant l'examine.

Pour pouvoir placer un verrou en lecture, *fd* doit être ouvert au moins en lecture. Pour placer un verrou en écriture, *fd* doit être ouvert en écriture. Pour placer les deux types de verrous, il faut une ouverture en lecture/écriture.

Outre la suppression par un *F_UNLCK* explicite, les verrous sont automatiquement libérés lorsque le processus se termine, ou s'il ferme *l'un* des descripteurs se référant au fichier sur lequel le verrou est placé. C'est dangereux : cela signifie qu'un processus peut perdre un verrou sur un fichier comme `/etc/passwd` ou `/etc/mtab` si, pour une raison quelconque, une fonction de bibliothèque décide de l'ouvrir puis de le refermer.

Les verrouillages d'enregistrements ne sont pas hérités par les enfants lors d'un fork(2), mais sont conservés au travers d'un execve(2).

À cause des tampons gérés par la bibliothèque stdio(3), l'utilisation des verrous d'enregistrements avec les routines de celle-ci est déconseillé. Utilisez plutôt read(2) et write(2).

VALEUR RENVOYÉE

La valeur renvoyée par *fcntl*() varie suivant le type d'opération :

F_GETFD Valeur des attributs du descripteur de fichier.
F_GETFL Valeur des attributs d'état du fichier.

ERREURS

EDEADLK Le verrouillage *F_SETLKW* conduirait à un blocage.

13 flock (2)

NOM

flock — Placer ou enlever un verrou coopératif sur un fichier ouvert

SYNOPSIS

int flock(int *fd*, **int** *operation***);**

DESCRIPTION

Place ou enlève un verrou consultatif sur un fichier ouvert dont le descripteur est *fd*. Le paramètre *operation* est l'un des suivants :

LOCK_SH Verrouillage partagé. Plusieurs processus peuvent disposer d'un verrouillage partagé simultanément sur un même fichier.

LOCK_EX Verrouillage exclusif. Un seul processus dispose d'un verrouillage exclusif sur un fichier, à un moment donné.

LOCK_UN Déverrouillage d'un verrou tenu par le processus.

Un appel *flock*() peut bloquer si un verrou incompatible est tenu par un autre processus. Pour que la requête soit non-bloquante, il faut inclure *LOCK_NB* (par un *OU* binaire « | ») avec la constante précisant l'opération.

Un même fichier ne peut pas avoir simultanément des verrous partagés et exclusifs.

Les verrous créés avec *flock*() sont associés à un fichier, ou plus précisément une entrée de la table des fichiers ouverts. Ainsi, les descripteurs de fichier dupliqués (par exemple avec fork(2) ou dup(2)) réfèrent au même verrou, et celui-ci peut être relâché ou modifié à travers n'importe lequel des descripteurs. De plus, un verrou est relâché par une opération explicite *LOCK_UN* sur l'un quelconque des descripteurs, ou lorsqu'ils ont tous été fermés.

Si un processus utilise open(2) (ou équivalent) plusieurs fois pour obtenir plusieurs descripteurs sur le même fichier, ces descripteurs sont traités indépendamment par *flock*(). Une tentative de verrouiller le fichier avec l'un de ces descripteurs peut être refusée si le processus appelant a déjà placé un verrou en utilisant un autre descripteur.

Un processus ne peut avoir qu'un seul type de verrou (partagé ou exclusif) sur un fichier. En conséquence un appel *flock*() sur un fichier déjà verrouillé modifiera le type de verrouillage.

Les verrous créés par *flock*() sont conservé au travers d'un execve(2).

Un verrou partagé ou exclusif peut être placé sur un fichier quel que soit le mode d'ouverture du fichier.

ERREURS

EINTR Durant l'attente pour acquérir le verrou, l'appel a été interrompu par un signal capturé par un gestionnaire ; consultez signal(7).

EWOULDBLOCK Le fichier est verrouillé et l'attribut *LOCK_NB* a été précisé.

CONFORMITÉ

BSD 4.4 (l'appel système *flock*() est apparu dans BSD 4.2). Une version de *flock*() parfois implémenté à partir de fcntl(2), est apparue sur la plupart des systèmes UNIX.

NOTES

flock() ne verrouille par les fichiers à travers NFS. Utilisez fcntl(2) à la place : il fonctionne avec NFS si la version de Linux est suffisamment récente et si le serveur accepte les verrouillages.

Depuis le noyau 2.0, *flock*() est implémenté par un appel système plutôt que d'être émulé par une routine de la bibliothèque C invoquant fcntl(2). Ceci correspond à la véritable sémantique BSD : il n'y a pas d'interaction entre les verrous placés par *flock*() et fcntl(2), et *flock*() ne détecte pas les cas de blocage (deadlock).

flock() place uniquement des verrous coopératifs : suivant les permissions du fichier un processus peut ignorer l'utilisation de *flock*() et faire des entrées-sorties sur le fichier.

Les sémantiques des verrous placés par *flock*() et fcntl(2) sont différentes en ce qui concerne fork(2) et dup(2). Sur les systèmes qui implémentent *flock*() en utilisant fcntl(2), la sémantique de *flock*() ne sera pas celle décrite ici.

La conversion d'un verrou (de partagé à exclusif et vice versa) n'est pas toujours atomique : tout d'abord le verrou existant est supprimé, puis un nouveau verrou est établi. Entre ces deux étapes, un verrou demandé par un autre processus peut être accordé, ce qui peut causer soit un blocage de la conversion, soit son échec, si *LOCK_NB* était indiqué. (Ceci est le comportement BSD d'origine, et est partagé par de nombreuses implémentations.)

14 fork (2)

NOM

fork — Créer un processus fils

SYNOPSIS

pid_t fork(void);

DESCRIPTION

fork() crée un nouveau processus en copiant le processus appelant. Le nouveau processus, qu'on appelle *fils* (« child »), est une copie exacte du processus appelant, qu'on appelle *père* ou *parent*, avec les exceptions suivantes :

- Le fils a son propre identifiant de processus unique, et ce PID ne correspond à l'identifiant d'aucun groupe de processus existant (setpgid(2)).
- L'identifiant de processus parent (PPID) du fils est l'identifiant de processus (PID) du père.
- Les utilisations de ressources (getrusage(2)) et les compteurs de temps processeur (times(2)) sont remis à zéro dans le fils.
- L'ensemble de signaux en attente dans le fils est initialement vide (sigpending(2)).
- Le fils n'hérite pas des opérations sur les sémaphores de son père (semop(2)).
- Le fils n'hérite pas des verrous d'enregistrements de son père (fcntl(2)).

Notez également les points suivants :

- Le processus fils est créé avec un unique thread — celui qui a appelé *fork*(). L'espace d'adressage virtuel complet du parent est copié dans le fils, y compris l'état des mutex, variables de condition, et autres objets de pthreads ; l'utilisation de pthread_atfork(3) peut être utile pour traiter les problèmes que cela peut occasionner.
- Le fils hérite de copies des descripteurs de fichier ouverts du père. Chaque descripteur de fichier du fils renvoie à la même description de fichier ouvert (consultez open(2)) que le descripteur de fichier correspondant dans le processus parent. Cela signifie que les deux descripteurs partagent les attributs d'état du fichier, le décalage, et les attributs d'E/S liés aux signaux (voir la description de *F_SETOWN* et *F_SETSIG* dans fcntl(2)).
- Le fils hérite d'une copie de l'ensemble des flux de répertoire ouverts par le parent (consultez opendir(3)). POSIX.1-2001 indique que les flux de répertoire correspondant dans le parent ou l'enfant *peuvent* partager le positionnement du flux de répertoire ; sous Linux/-glibc, ce n'est pas le cas.

VALEUR RENVOYÉE

En cas de succès, le PID du fils est renvoyé au parent, et 0 est renvoyé au fils. En cas d'échec -1 est renvoyé au parent, aucun processus fils n'est créé, et `errno` contient le code d'erreur.

ERREURS

EAGAIN Il n'a pas été possible de créer un nouveau processus car la limite ressource *RLIMIT_NPROC* de l'appelant a été rencontrée. Pour franchir cette limite, le processus doit avoir au moins l'une des deux capacités *CAP_SYS_ADMIN* ou *CAP_SYS_RESOURCE*.

NOTES

Sous Linux, *fork*() est implémenté en utilisant une méthode de copie à l'écriture. Ceci consiste à ne faire la véritable duplication d'une page mémoire que lorsqu'un processus en modifie une instance. Tant qu'aucun des deux processus n'écrit dans une page donnée, celle-ci n'est pas vraiment dupliquée. Ainsi les seules pénalisations induites par fork sont le temps et la mémoire nécessaires à la copie de la table des pages du parent ainsi que la création d'une structure de tâche pour le fils.

15 getenv (3)

NOM

getenv — Lire une variable d'environnement

SYNOPSIS

extern char **environ;

char *getenv(const char ****name*);**

DESCRIPTION

La fonction *getenv*() recherche dans la liste des variables d'environnement une variable nommée *name*, et renvoie un pointeur sur la chaîne *value* correspondante.

VALEUR RENVOYÉE

La fonction *getenv*() renvoie un pointeur sur la valeur correspondante, dans l'environnement du processus, ou NULL s'il n'y a pas de correspondance.

NOTES

Les chaînes dans la liste des variables d'environnement sont de la forme *nom=valeur*.

Telle qu'elle est généralement implémentée, *getenv*() renvoie un pointeur vers une chaîne de la liste d'environnement. L'appelant doit faire attention de ne pas modifier cette chaîne car cela modifierait l'environnement du processus.

16 getpid, getppid (2)

NOM

getpid, getppid — Obtenir l'identifiant d'un processus

SYNOPSIS

pid_t getpid(void);
pid_t getppid(void);

DESCRIPTION

getpid() renvoie l'identifiant du processus appelant. Ceci est souvent utilisé par des routines qui créent des fichiers temporaires uniques.

getppid() renvoie le PID du processus père de l'appelant.

ERREURS

Ces fonctions réussissent toujours.

17 kill (2)

NOM

kill — Envoyer un signal à un processus

SYNOPSIS

int kill(pid_t *pid*, **int** *sig*);

DESCRIPTION

L'appel système *kill*() peut être utilisé pour envoyer n'importe quel signal à n'importe quel processus ou groupe de processus.

Si *pid* est positif, le signal *sig* est envoyé au processus dont l'identifiant est indiqué par *pid*.

Si *pid* vaut zéro, alors le signal *sig* est envoyé à tous les processus appartenant au même groupe que le processus appelant.

Si *pid* vaut -1, alors le signal *sig* est envoyé à tous les processus sauf celui de PID 1 (*init*), mais voir plus bas.

Si *pid* est inférieur à -1, alors le signal *sig* est envoyé à tous les processus du groupe dont l'identifiant est `-pid`.

Si *sig* vaut 0, aucun signal n'est envoyé mais les conditions d'erreur sont vérifiées ; ceci peut être utilisé pour vérifier l'existence d'un identifiant de processus ou d'un identifiant de groupes de processus.

Pour qu'un processus puisse envoyer un signal, il doit avoir les privilèges nécessaires, ou l'UID effectif ou réel du processus qui envoie le signal doit être égal au Set-UID sauvé ou réel du processus cible.

NOTES

On ne peut envoyer au processus numéro 1 (*init*) que des signaux pour lesquels il a expressément installé un gestionnaire. Ceci évite que le système soit arrêté accidentellement.

POSIX.1-2001 réclame que *kill(-1,sig)* envoie *sig* à tous les processus accessibles par le processus appelant, sauf à certains processus système dépendant de l'implémentation. Linux autorise un processus à s'envoyer un signal à lui-même, mais l'appel *kill(-1,sig)* n'envoie pas le signal au processus appelant.

18 link (2)

NOM

link — Créer un nouveau nom pour un fichier

SYNOPSIS

int link(const char **oldpath***, const char** **newpath***);**

DESCRIPTION

link() crée un nouveau lien (aussi appelé lien matériel ou hard link) sur un fichier existant.

Si *newpath* existe, il ne sera *pas* écrasé.

Ce nouveau nom pourra être utilisé exactement comme l'ancien quelle que soit l'opération. Les deux noms réfèrent au même fichier (et ont donc les mêmes permissions et propriétaire) et il est impossible de déterminer quel nom était l'original.

ERREURS

EEXIST *newpath* existe déjà.
EXDEV *oldpath* et *newpath* ne résident pas sur le même système de fichiers.

19 lseek (2)

NOM

lseek — Positionner la tête de lecture/écriture dans un fichier

SYNOPSIS

off_t lseek(int *fd***, off_t** *offset***, int** *whence***);**

DESCRIPTION

La fonction *lseek*() place la tête de lecture/écriture à la position *offset* dans le fichier associé au descripteur *fd* en suivant la directive *whence* ainsi :

SEEK_SET La tête est placée à *offset* octets depuis le début du fichier.

SEEK_CUR La tête de lecture/écriture est avancée de *offset* octets.

SEEK_END La tête est placée à la fin du fichier plus *offset* octets.

La fonction *lseek*() permet de placer la tête au-delà de la fin actuelle du fichier (mais cela ne modifie pas la taille du fichier). Si des données sont écrites à cet emplacement, une lecture ultérieure de l'espace intermédiaire (un « trou ») retournera des octets nul (« \0 ») jusqu'à ce que d'autres données y soient écrites.

VALEUR RENVOYÉE

lseek(), si elle réussit, renvoie le nouvel emplacement, mesuré en octets depuis le début du fichier. En cas d'échec, la valeur *(off_t) -1* est renvoyée, et `errno` contient le code d'erreur.

ERREURS

EBADF *fd* n'est pas un descripteur de fichier ouvert.
EINVAL Soit *whence* n'est pas valable, soit la position demandée serait négative, ou après la fin d'un périphérique.
ESPIPE *fd* est associé à un tube (pipe), une socket, ou une file FIFO.

NOTES

Notez que les descripteurs de fichier dupliqués par dup(2) ou fork(2) partagent le même pointeur de position. Ainsi le déplacement sur de tels fichiers peut conduire à des problèmes d'accès concurrents.

20 mkdir (2)

NOM

mkdir — Créer un répertoire

SYNOPSIS

int mkdir(const char **pathname*, **mode_t** *mode***);**

DESCRIPTION

mkdir() crée un nouveau répertoire nommé *pathname*.

Le paramètre *mode* indique les permissions à appliquer au répertoire. Cette valeur peut être modifiée par le *umask* du processus : les permissions du répertoire effectivement créé vaudront (*mode* & ~*umask* & 0777). Les autres bits de mode du répertoire créé dépendent du système d'exploitation. Pour Linux, voir plus loin.

ERREURS

EMLINK Le nombre maximal de liens vers le répertoire parent dépasserait *LINK_MAX*.

21 mkfifo (3)

NOM

mkfifo — Créer un fichier spécial FIFO (un tube nommé)

SYNOPSIS

int mkfifo(const char **pathname***, mode_t** *mode***);**

DESCRIPTION

La fonction *mkfifo*() crée un fichier spécial FIFO (tube nommé) à l'emplacement *pathname*. *mode* indique les permissions d'accès. Ces permissions sont modifiées par la valeur d'*umask* du processus : les permissions d'accès effectivement adoptées sont *(mode & ~umask)*.

Un fichier spécial FIFO est semblable à un tube (pipe), sauf qu'il est créé différemment. Plutôt qu'un canal de communication anonyme, un fichier FIFO est inséré dans le système de fichiers en appelant *mkfifo*().

Une fois qu'un fichier FIFO est créé, n'importe quel processus peut l'ouvrir en lecture ou écriture, comme tout fichier ordinaire. En fait, il faut ouvrir les deux extrémités simultanément avant de pouvoir effectuer une opération d'écriture ou de lecture. L'ouverture d'un FIFO en lecture est généralement bloquante, jusqu'à ce qu'un autre processus ouvre le même FIFO en écriture, et inversement. Consultez fifo(7) pour la gestion non bloquante d'une FIFO.

22 mmap, munmap (2)

NOM

mmap, munmap — Établir/supprimer une projection en mémoire (map/unmap) des fichiers ou des périphériques

SYNOPSIS

void *mmap(void **addr***, size_t** *length***, int** *prot***, int** *flags***, int** *fd***, off_t** *offset***);**
int munmap(void **addr***, size_t** *length***);**

DESCRIPTION

mmap() crée une nouvelle projection dans l'espace d'adressage virtuel du processus appelant. L'adresse de démarrage de la nouvelle projection est indiquée dans *addr*. Le paramètre *length* indique la longueur de la projection.

Si *addr* est NULL, le noyau choisit l'adresse à laquelle démarrer la projection ; c'est la méthode la plus portable pour créer une nouvelle projection. Si *addr* n'est pas NULL, le noyau le considère comme une indication sur l'endroit où placer la projection ; sous Linux, elle sera placée à une frontière de page proche. L'adresse de la nouvelle projection est renvoyée comme résultat de l'appel.

Le contenu d'une projection de fichier est initialisé avec *length* octets à partir de la position *offset* dans le fichier (ou autre objet) correspondant au descripteur de fichier *fd*. *offset* doit être un multiple de la taille de page, renvoyée par *sysconf(_SC_PAGE_SIZE)*.

L'argument *prot* indique la protection que l'on désire pour cette zone de mémoire, et ne doit pas entrer en conflit avec le mode d'ouverture du fichier. Il s'agit soit de *PROT_NONE* (le contenu de la mémoire est inaccessible) soit d'un OU binaire entre les constantes suivantes :

PROT_EXEC On peut exécuter du code dans la zone mémoire.
PROT_READ On peut lire le contenu de la zone mémoire.
PROT_WRITE On peut écrire dans la zone mémoire.
PROT_NONE Les pages ne peuvent pas être accédées.

Le paramètre *flags* détermine si les modifications de la projection sont visibles depuis les autres processus projetant la même région, et si les modifications sont appliquées au fichier sous-jacent. Ce comportement est déterminé en incluant exactement une des valeurs suivantes dans *flags* :

MAP_SHARED Partager la projection. Les modifications de la projection sont visibles dans les autres processus qui projettent ce fichier, et sont appliquées au fichier sous-jacent. En revanche, ce dernier n'est pas nécessairement mis à jour tant qu'on n'a pas appelé msync(2) ou *munmap*().
MAP_PRIVATE Créer une projection privée, utilisant la méthode de copie à l'écriture. Les modifications de la projection ne sont pas visibles depuis les autres processus projetant le même fichier, et ne modifient pas le fichier lui-même. Il n'est pas précisé si les changements effectués dans le fichier après l'appel *mmap*() seront visibles.

La mémoire obtenue par *mmap* est préservée au travers d'un fork(2), avec les mêmes attributs.

munmap()

L'appel système *munmap*() détruit la projection dans la zone de mémoire spécifiée, et s'arrange pour que toute référence ultérieure à cette zone mémoire déclenche une erreur d'adressage. La projection est aussi automatiquement détruite lorsque le processus se termine. À l'inverse, la fermeture du descripteur de fichier ne supprime pas la projection.

VALEUR RENVOYÉE

mmap() renvoie un pointeur sur la zone de mémoire, s'il réussit. En cas d'échec il retourne la valeur *MAP_FAILED* (c.-à-d. *(void *) -1*) et `errno` contient le code d'erreur. *munmap*() renvoie 0 s'il réussit. En cas d'échec, -1 est renvoyé et `errno` contient le code d'erreur (probablement EINVAL).

ERREURS

EACCES Le descripteur ne correspond pas à un fichier normal, ou on demande une projection de fichier mais *fd* n'est pas ouvert en lecture, ou on demande une projection partagée

MAP_SHARED avec protection *PROT_WRITE*, mais *fd* n'est pas ouvert en lecture et écriture (*O_RDWR*). Ou encore *PROT_WRITE* est demandé, mais le fichier est ouvert en ajout seulement.

EINVAL *flags* ne contient ni *MAP_PRIVATE* ni *MAP_SHARED*, ou les contient tous les deux.

L'accès à une zone de projection peut déclencher les signaux suivants :

`SIGSEGV` Tentative d'écriture dans une zone en lecture seule.

`SIGBUS` Tentative d'accès à une portion de la zone qui ne correspond pas au fichier (par exemple après la fin du fichier, y compris lorsqu'un autre processus l'a tronqué).

23 mount (2)

NOM

mount — Monter un système de fichiers

SYNOPSIS

int mount(const char **source*, **const char** **target*, **const char** **filesystemtype*, **unsigned long** *mountflags*, **const void** **data*);

DESCRIPTION

mount() attache le système de fichiers indiqué par *source* (qui est généralement un nom de périphérique, mais peut aussi être un répertoire ou un objet fictif) au répertoire indiqué par *target*.

Des privilèges appropriés (sous Linux : la capacité *CAP_SYS_ADMIN*) sont nécessaires pour monter des systèmes de fichiers.

L'argument *filesystemtype* prend une des valeurs listées dans `/proc/filesystems` (par exemple « ext2 », « minix », « ext3 », « jfs », « xfs », « reiserfs », « msdos », « proc », « nfs », « iso9660 »). Des types supplémentaires peuvent être disponibles lorsque les modules appropriés sont chargés.

L'argument *data* est interprété différemment suivant le type de système de fichiers. Typiquement, c'est une chaîne d'options comprises par le système de fichiers, séparées par des virgules. Consultez mount(8) pour des détails sur les options disponibles pour chaque type de système.

24 open, creat (2)

NOM

open, creat — Ouvrir ou créer éventuellement un fichier

SYNOPSIS

int open(const char **pathname*, **int** *flags*);
int open(const char **pathname*, **int** *flags*, **mode_t** *mode*);

int creat(const char **pathname*, **mode_t** *mode*);

DESCRIPTION

Étant donné le chemin *pathname* d'un fichier, *open*() renvoie un descripteur de fichier (petit entier positif ou nul) qui pourra ensuite être utilisé dans d'autres appels système (read(2), write(2), lseek(2), fcntl(2), etc.). Le descripteur de fichier renvoyé par un appel réussi sera le plus petit descripteur de fichier non actuellement ouvert par le processus.

Par défaut, le nouveau descripteur de fichier est configuré pour rester ouvert après un appel à execve(2) (son attribut *FD_CLOEXEC* décrit dans fcntl(2) est initialement désactivé). L'attribut *O_CLOEXEC* décrit ci-dessous permet de modifier ce comportement par défaut. La position dans le fichier est définie au début du fichier (consultez lseek(2)).

Un appel à *open*() crée une nouvelle *description de fichier ouvert*, une entrée dans la table de fichiers ouverts du système. Cette entrée enregistre la position dans le fichier et les attributs d'état du fichier (modifiables par l'opération *F_SETFL* de fcntl(2)). Un descripteur de fichier est une référence à l'une de ces entrées ; cette référence n'est pas modifiée si *pathname* est ensuite supprimé ou modifié pour correspondre à un autre fichier. La nouvelle description de fichier ouvert n'est initialement partagée avec aucun autre processus, mais ce partage peut apparaître après un fork(2).

Le paramètre *flags* est l'un des éléments *O_RDONLY*, *O_WRONLY* ou *O_RDWR* qui réclament respectivement l'ouverture du fichier en lecture seule, écriture seule, ou lecture/écriture.

De plus, zéro ou plusieurs attributs de création de fichier et attributs d'état de fichier peuvent être indiqués dans *flags* avec un OU binaire.

La liste complète des attributs de création et d'état de fichier est la suivante.

O_APPEND Le fichier est ouvert en mode « ajout ». Initialement, et avant chaque write(2), la tête de lecture/écriture est placée à la fin du fichier comme avec lseek(2).

O_CREAT Créer le fichier s'il n'existe pas.

Le paramètre *mode* indique les droits à utiliser si un nouveau fichier est créé. Ce paramètre doit être fourni quand *O_CREAT* ou *O_TMPFILE* sont indiqués dans *flags* ; si ni *O_CREAT* ni *O_TMPFILE* ne sont précisés, *mode* est ignoré. Les droits effectifs sont modifiées par le *umask* du processus : la véritable valeur utilisée est *(mode & ~umask)*. Remarquez que ce mode ne s'applique qu'aux accès ultérieurs au fichier nouvellement créé. L'appel *open*() qui crée un fichier dont le mode est en lecture seule fournira quand même un descripteur de fichier en lecture et écriture.

Voir *chmod*(2) pour les constantes disponibles pour *mode*.

O_EXCL S'assurer que cet appel crée le fichier : si cet attribut est spécifié en conjonction avec *O_CREAT* et si le fichier *pathname* existe déjà, *open*() échouera.

Lorsque ces deux attributs sont spécifiés, les liens symboliques ne sont pas suivis : si *pathname* est un lien symbolique, *open*() échouera quelque soit l'endroit où pointe le lien symbolique.

En général, le comportement de *O_EXCL* est indéterminé s'il est utilisé sans *O_CREAT*.

O_NONBLOCK* ou *O_NDELAY Le fichier est ouvert en mode « non-bloquant ». Ni la fonction *open*() ni aucune autre opération ultérieure sur ce fichier ne laissera le processus appelant en attente. Pour la manipulation des FIFO (tubes nommés), voir également fifo(7). Pour une explication de l'effet de *O_NONBLOCK* en conjonction avec les verrouillages impératifs et les baux de fichiers, voir fcntl(2).

O_SYNC Les opérations d'écriture dans le fichier se dérouleront selon les conditions d'exécution des opérations E/S synchrones avec garantie d'intégrité du *fichier*

Au moment où write(2) (ou un appel similaire) renvoie une donnée, cette donnée et les métadonnées associées au fichier ont été transmises au matériel sur lequel s'exécute l'appel (autrement dit, comme si chaque appel à write(2) était suivi d'un appel à fsync(2)).

O_TRUNC Si le fichier existe, est un fichier ordinaire et que le mode d'accès permet l'écriture (*O_RDWR* ou *O_WRONLY*), il sera tronqué à une longueur nulle.

creat()

creat() est équivalent à *open*() avec l'attribut *flags* égal à *O_CREAT* | *O_WRONLY* | *O_TRUNC*.

VALEUR RENVOYÉE

open() et *creat*() renvoient le nouveau descripteur de fichier s'ils réussissent, ou -1 s'ils échouent, auquel cas `errno` contient le code d'erreur.

ERREURS

open() et *creat*() peuvent échouer avec les erreurs suivantes :

EACCES L'accès demandé au fichier est interdit, ou la permission de parcours pour l'un des répertoires du chemin *pathname* est refusée, ou le fichier n'existe pas encore et le répertoire parent ne permet pas l'écriture.

EEXIST *pathname* existe déjà et *O_CREAT* et *O_EXCL* ont été indiqués.

EISDIR Une écriture a été demandée alors que *pathname* correspond à un répertoire (en fait, *O_WRONLY* ou *O_RDWR* ont été demandés).

EMFILE Le processus a déjà ouvert le nombre maximal de fichiers.

ENOENT *O_CREAT* est absent et le fichier n'existe pas. Ou un répertoire du chemin d'accès *pathname* n'existe pas, ou est un lien symbolique pointant nulle part.

NOTES

mode accès au fichier

Contrairement aux autres valeurs qui peuvent être indiquées dans *flags*, les valeurs du *mode d'accès O_RDONLY*, *O_WRONLY* et *O_RDWR* ne sont pas des bits individuels. Ils définissent l'ordre des deux bits de poids faible de *flags*, et ont pour valeur respective 0, 1 et 2. En d'autres termes, l'association *O_RDONLY* | *O_WRONLY* est une erreur logique et n'a certainement pas la même signification que *O_RDWR*.

25 opendir (3)

NOM

opendir — Ouvrir un répertoire

SYNOPSIS

DIR *opendir(const char *nom);

DESCRIPTION

La fonction *opendir*() ouvre un flux répertoire correspondant au répertoire *nom*, et renvoie un pointeur sur ce flux. Le flux est positionné sur la première entrée du répertoire.

VALEUR RENVOYÉE

La fonction *opendir*() renvoie un pointeur sur le flux répertoire. Si une erreur se produit, NULL est renvoyé, et `errno` contient le code d'erreur.

ERREURS

EACCES Accès interdit.
ENOENT Le répertoire n'existe pas, ou *nom* est une chaîne vide.
ENOTDIR *nom* n'est pas un répertoire

26 pause (2)

NOM

pause — Attendre un signal

SYNOPSIS

int pause(void);

DESCRIPTION

pause() force le processus (ou le thread) appelant à s'endormir jusqu'à ce qu'un signal soit distribué, qui termine le processus ou provoque l'appel d'une fonction de gestionnaire de signal.

VALEUR RENVOYÉE

pause() ne rend la main que lorsqu'un signal a été intercepté et que le gestionnaire s'est terminé. Dans ce cas *pause*() renvoie -1 et `errno` est positionné à la valeur EINTR.

27 pipe, pipe2 (2)

NOM

pipe, pipe2 — Créer un tube

SYNOPSIS

int pipe(int *pipefd***[2]);**

int pipe2(int *pipefd***[2], int** *flags***);**

DESCRIPTION

pipe() crée un tube, un canal unidirectionnel de données qui peut être utilisé pour la communication entre processus. Le tableau *pipefd* est utilisé pour renvoyer deux descripteurs de fichier faisant référence aux extrémités du tube. *pipefd[0]* fait référence à l'extrémité de lecture du tube. *pipefd[1]* fait référence à l'extrémité d'écriture du tube. Les données écrites sur l'extrémité d'écriture du tube sont mises en mémoire tampon par le noyau jusqu'à ce qu'elles soient lues sur l'extrémité de lecture du tube. Pour plus de détails, consultez pipe(7).

Si *flags* est nul, alors *pipe2*() est identique à *pipe*(). Les valeurs suivantes peuvent être incluses à l'aide d'un OU binaire dans *flags* pour obtenir différents comportements :

- ***O_CLOEXEC*** Placer l'attribut « close-on-exec » (*FD_CLOEXEC*) sur les deux nouveaux descripteurs de fichiers. Consultez la description de cet attribut dans open(2) pour savoir pourquoi ça peut être utile.
- ***O_NONBLOCK*** Placer l'attribut d'état de fichier *O_NONBLOCK* sur les deux nouveaux descripteurs de fichiers ouverts. Utiliser cet attribut économise des appels supplémentaires à fcntl(2) pour obtenir le même résultat.

28 pread, pwrite (2)

NOM

pread, pwrite — Lire ou écrire à une position donnée d'un descripteur de fichier

SYNOPSIS

ssize_t pread(int *fd*, **void** **buf*, **size_t** *count*, **off_t** *offset*);

ssize_t pwrite(int *fd*, **const void** **buf*, **size_t** *count*, **off_t** *offset*);

DESCRIPTION

pread() lit au maximum *count* octets depuis le descripteur *fd*, à la position *offset* (mesurée depuis le début du fichier), et les place dans la zone commençant à l'adresse *buf*. La position de la tête de lecture du fichier n'est pas modifiée par cet appel système.

pwrite() lit au maximum *count* octets dans la zone mémoire pointée par *buf*, et les écrit à la position *offset* (mesurée depuis le début du fichier) dans le descripteur *fd*. La position de la tête d'écriture du fichier n'est pas modifiée.

Dans les deux cas, le fichier décrit par *fd* doit permettre le positionnement.

VALEUR RENVOYÉE

S'ils réussissent, ces appels système renvoient le nombre d'octets lus ou écrits (0 indiquant que rien n'a été écrit dans le cas de *pwrite*(), ou la fin du fichier dans le cas de *pread*()). En cas d'échec, ils renvoient -1, et remplissent `errno` en conséquence.

NOTES

Les appels système *pread*() et *pwrite*() sont particulièrement utiles dans les applications multi-threadées . Ils permettent à plusieurs threads d'effectuer des entrées et sorties sur un même descripteur de fichier sans être affecté des déplacements au sein du fichier dans les autres threads.

29 pthread_attr_init, pthread_attr_destroy (3)

NOM

pthread_attr_init, pthread_attr_destroy — Initialiser et détruire un objet d'attributs de thread

SYNOPSIS

int pthread_attr_init(pthread_attr_t **attr***);**
int pthread_attr_destroy(pthread_attr_t **attr***);**

DESCRIPTION

La fonction *pthread_attr_init*() initialise l'objet d'attributs de thread pointé par *attr* avec des valeurs d'attributs par défaut. Après cet appel, les attributs individuels de cet objet peuvent être modifiés en utilisant diverses fonctions, et l'objet peut alors être utilisé dans un ou plusieurs appels de pthread_create(3) pour créer des threads.

Appeler *pthread_attr_init*() sur un objet d'attributs de thread qui a déjà été initialisé résulte en un comportement indéfini.

Quand un objet d'attributs de thread n'est plus nécessaire, il devrait être détruit en appelant la fonction *pthread_attr_destroy*(). Détruire un objet d'attributs de thread n'a aucun effet sur les threads qui ont été créés en utilisant cet objet.

Dès qu'un objet d'attributs de thread a été détruit, il peut être réinitialisé en appelant *pthread_attr_init*(). Toute autre utilisation d'un objet d'attributs de thread entraîne des résultats indéfinis.

VALEUR RENVOYÉE

En cas de réussite, ces fonctions renvoient 0 ; en cas d'erreur elles renvoient un numéro d'erreur non nul.

NOTES

Le type pthread_attr_t doit être traité comme opaque ; tout accès à l'objet en dehors des fonctions pthreads n'est pas portable et peut produire des résultats indéfinis.

30 pthread_attr_setdetachstate, pthread_attr_getdetachstate (3)

NOM

pthread_attr_setdetachstate, pthread_attr_getdetachstate — Définir ou obtenir l'attribut de l'état de détachement de l'objet d'attributs de thread

SYNOPSIS

int pthread_attr_setdetachstate(pthread_attr_t **attr*, **int** *detachstate***);**
int pthread_attr_getdetachstate(pthread_attr_t **attr*, **int** **detachstate***);**

DESCRIPTION

La fonction *pthread_attr_setdetachstate*() définit l'attribut d'état de détachement de l'objet d'attributs de thread auquel *attr* fait référence à la valeur indiquée par *detachstate*. Cet attribut d'état de détachement détermine si un thread créé en utilisant l'objet d'attributs de thread *attr* sera dans un état joignable ou détaché.

Les valeurs suivantes peuvent être spécifiées dans *detachstate* :

PTHREAD_CREATE_DETACHED Les threads créés avec *attr* seront dans un état détaché.
PTHREAD_CREATE_JOINABLE Les threads créés avec *attr* seront dans un état joignable.

Par défaut, l'attribut d'état de détachement est initialisé à *PTHREAD_CREATE_JOINABLE* dans un objet d'attributs de thread.

La fonction *pthread_attr_getdetachstate*() renvoie, dans le tampon pointé par *detachstate*, l'attribut contenant l'état de détachement de l'objet d'attributs de thread *attr*.

VALEUR RENVOYÉE

En cas de réussite, ces fonctions renvoient 0 ; en cas d'erreur elles renvoient un numéro d'erreur non nul.

NOTES

Consultez pthread_create(3) pour plus de détails sur les threads joignables et détachés.

C'est une erreur de spécifier, lors d'un appel ultérieur à pthread_detach(3) ou pthread_join(3), comme identifiant de thread un thread qui a été créé dans un état détaché.

31 pthread_cancel (3)

NOM

pthread_cancel — Envoyer une requête d'annulation à un thread

SYNOPSIS

int pthread_cancel(pthread_t *thread***);**

DESCRIPTION

La fonction *pthread_cancel*() envoie une requête d'annulation au thread *thread*. Si et quand le thread ciblé réagit à la requête d'annulation dépend de deux attributs qui sont sous le contrôle de ce thread : son état d'annulation (*state*) et son mode d'annulation (*type*).

L'état d'annulation d'un thread, déterminé par pthread_setcancelstate(3), peut être activé (*enabled*), c'est le défaut pour les nouveaux threads, ou désactivé (*disabled*). Si un thread désactive l'annulation, alors une demande d'annulation restera dans la file d'attente jusqu'à ce que le thread active l'annulation. Si un thread active l'annulation, alors son mode d'annulation va déterminer le moment où cette annulation est effectuée.

Le mode d'annulation d'un thread, déterminé par pthread_setcanceltype(3), peut être asynchrone (*asynchronous*) ou retardée (*deferred*), qui est le mode par défaut pour les nouveaux threads. Un mode d'annulation asynchrone signifie que le thread peut être annulé à tout moment (d'ordinaire immédiatement, mais ce n'est pas garanti). Un mode d'annulation retardé signifie que l'annulation peut être retardée jusqu'à ce que le thread appelle une fonction qui est un point d'annulation (*cancellation point*). Une liste des fonctions qui sont ou peuvent être des points d'annulation est donnée dans pthreads(7).

Quand une requête d'annulation est traitée, les étapes suivantes sont effectuées pour *thread* (dans cet ordre) :

1. Les gestionnaires de nettoyage sont dépilés (dans l'ordre inverse dans lequel ils ont été empilés) et appelés (consultez pthread_cleanup_push(3)).
2. Les destructeurs de données spécifiques aux threads sont appelés, dans un ordre non déterminé (consultez pthread_key_create(3)).
3. Le thread est terminé (consultez pthread_exit(3)).

Les étapes ci-dessus sont effectuées de manière asynchrone par rapport à l'appel à *pthread_cancel*(). La valeur de retour de *pthread_cancel*() ne fait qu'informer l'appelant si une requête d'annulation a été correctement mise en file d'attente.

Après qu'un thread annulé s'est terminé, une demande de jointure par pthread_join(3) renvoie *PTHREAD_CANCELED* comme état de sortie du thread. Il faut noter que joindre un thread est la seule manière de savoir si une annulation a terminé.

VALEUR RENVOYÉE

En cas de réussite, *pthread_cancel*() renvoie 0 ; en cas d'erreur, elle renvoie un numéro d'erreur non nul.

32 pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait (3)

NOM

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait — Opérations sur les conditions

SYNOPSIS

pthread_cond_t *cond* **= PTHREAD_COND_INITIALIZER;**

int pthread_cond_init(pthread_cond_t **cond*, **pthread_condattr_t** **cond_attr*);

int pthread_cond_signal(pthread_cond_t **cond*);

int pthread_cond_broadcast(pthread_cond_t **cond*);

int pthread_cond_wait(pthread_cond_t **cond*, **pthread_mutex_t** **mutex*);

int pthread_cond_timedwait(pthread_cond_t **cond*, **pthread_mutex_t** **mutex*, **const struct timespec** **abstime*);

int pthread_cond_destroy(pthread_cond_t **cond*);

DESCRIPTION

Une condition (abréviation pour « variable condition ») est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition (un prédicat) sur des données partagées soit vérifiée. Les opérations fondamentales sur les conditions sont : signaler la condition (quand le prédicat devient vrai) et attendre la condition en suspendant l'exécution du thread jusqu'à ce qu'un autre thread signale la condition.

Une variable condition doit toujours être associée à un mutex, pour éviter une condition concurrente où un thread se prépare à attendre une condition et un autre thread signale la condition juste avant que le premier n'attende réellement.

pthread_cond_init() initialise la variable condition *cond*, en utilisant les attributs de condition spécifiés par *cond_attr*, ou les attributs par défaut si *cond_attr* vaut *NULL*.

Les variables de type pthread_cond_t peuvent également être statiquement initialisées, en utilisant la constante *PTHREAD_COND_INITIALIZER*.

pthread_cond_signal() relance l'un des threads attendant la variable condition *cond*. S'il n'existe aucun thread répondant à ce critère, rien ne se produit. Si plusieurs threads attendent sur *cond*, seul l'un d'entre eux sera relancé, mais il est impossible de savoir lequel.

pthread_cond_broadcast() relance tous les threads attendant sur la variable condition *cond*. Rien ne se passe s'il n'y a aucun thread attendant sur *cond*.

pthread_cond_wait() déverrouille atomiquement le *mutex* (comme *pthread_unlock_mutex*()) et attend que la variable condition *cond* soit signalée. L'exécution du thread est suspendue et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée. Le *mutex* doit être verrouillé par le thread appelant à l'entrée de *pthread_cond_wait*(). Avant de rendre la main au thread appelant, *pthread_cond_wait*() reverrouille *mutex* (comme *pthread_lock_mutex*()).

Le déverrouillage du mutex et la suspension de l'exécution sur la variable condition sont liés atomiquement. Donc, si tous les threads verrouillent le mutex avant de signaler la condition, il est garanti que la condition ne peut être signalée (et donc ignorée) entre le moment où un thread verrouille le mutex et le moment où il attend sur la variable condition.

pthread_cond_timedwait() déverrouille le *mutex* et attend sur *cond*, en liant atomiquement ces deux étapes, comme le fait *pthread_cond_wait*(), cependant l'attente est bornée temporellement. Si *cond* n'a pas été signalée après la période spécifiée par *abstime*, le mutex *mutex* est reverrouillé et *pthread_cond_timedwait*() rend la main avec l'erreur ETIMEDOUT. Le paramètre *abstime* spécifie un temps absolu, avec la même origine que time(2) et gettimeofday(2) : un *abstime* de 0 correspond à 00:00:00 GMT, le 1er janvier 1970.

pthread_cond_destroy() détruit une variable condition, libérant les ressources qu'elle possède. Aucun thread ne doit attendre sur la condition à l'entrée de *pthread_cond_destroy*().

ANNULATION

pthread_cond_wait() et *pthread_cond_timedwait*() sont des points d'annulation. Si un thread est annulé alors qu'il est suspendu dans l'une de ces fonctions, son exécution reprend immédiatement, reverrouillant le paramètre *mutex* à *pthread_cond_wait*() et *pthread_cond_timedwait*(), et exécute finalement l'annulation. Aussi, les gestionnaires d'annulation sont assurés que *mutex* est verrouillé lorsqu'ils sont exécutés.

ASYNC-SIGNAL SAFETY

Ces fonctions ne sont pas fiables par rapport aux signaux asynchrones et ne doivent donc pas être utilisées dans des gestionnaires de signaux [NdT : sous peine de perdre leur propriété d'atomicité]. En particulier, appeler *pthread_cond_signal*() ou *pthread_cond_broadcast*() dans un gestionnaire de signal peut placer le thread appelant dans une situation de blocage définitif.

VALEUR RENVOYÉE

Toutes ces fonctions renvoient 0 en cas de succès et un code d'erreur non nul en cas de problème.

ERREURS

pthread_cond_init(), *pthread_cond_signal*(), *pthread_cond_broadcast*() et *pthread_cond_wait*() ne renvoient jamais de code d'erreur.

33 pthread_create (3)

NOM

pthread_create — Créer un nouveau thread

SYNOPSIS

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

DESCRIPTION

La fonction *pthread_create*() démarre un nouveau thread dans le processus appelant. Le nouveau thread commence par appeler *start_routine*() ; *arg* est passé comme unique argument de *start_routine*().

Le nouveau thread se termine d'une des manières suivantes :

— Il appelle pthread_exit(3), en indiquant une valeur de sortie qui sera disponible pour pour un autre thread du même processus qui appelle pthread_join(3).

— Il sort de la routine *start_routine*(). C'est équivalent à appeler pthread_exit(3) avec la valeur fournie à l'instruction *return*.
— Il est annulé (voir pthread_cancel(3)).
— Un des threads du processus appelle exit(3), ou le thread principal sort de la routine *main*(). Cela entraine l'arrêt de tous les threads du processus.

L'argument *attr* pointe sur une structure pthread_attr_t dont le contenu est utilisé pendant la création des threads pour déterminer les attributs du nouveau thread. Cette structure est initialisée avec pthread_attr_init(3) et les fonctions similaires. Si *attr* est NULL, alors le thread est créé avec les attributs par défaut.

Avant de revenir, un appel réussi à *pthread_create*() stocke l'identifiant du nouveau thread dans le tampon pointé par *thread*. Cet identifiant est utilisé pour se référer à ce thread dans les appels ultérieurs aux autres fonctions de pthreads.

VALEUR RENVOYÉE

En cas de réussite, *pthread_create*() renvoie 0 ; en cas d'erreur, elle renvoie un numéro d'erreur, et le contenu de **thread* est indéfini.

ERREURS

EAGAIN Ressources insuffisantes pour créer un nouveau thread, ou une limite sur le nombre de threads imposée par le système a été atteinte. Ce dernier cas peut arriver de deux façons : la limite souple *RLIMIT_NPROC* (changée par setrlimit(2)), qui limite le nombre de processus pour un identifiant d'utilisateur réel, a été atteinte ; ou alors la limite imposée par le noyau sur le nombre total de threads, `/proc/sys/kernel/threads-max`, a été atteinte.

NOTES

Consultez pthread_self(3) pour des informations plus détaillées sur l'identifiant de thread renvoyé dans **thread* par *pthread_create*(). Sauf si une politique d'ordonnancement temps-réel est employée, après un appel à *pthread_create*(), on ne sait pas quel thread — l'appelant ou le nouveau thread — sera exécuté ensuite.

Un thread peut être dans un état soit joignable (*joinable*), soit détaché (*detached*). Si un thread est joignable, un autre thread peut appeler pthread_join(3) pour attendre que ce thread se termine, et récupérer sa valeur de sortie. Ce n'est que quand un thread terminé et joignable a été joint que ses ressources sont rendues au système. Quand un thread détaché se termine, ses ressources sont automatiquement rendues au système ; il n'est pas possible de joindre un tel thread afin d'en obtenir la valeur de sortie. Mettre un thread dans l'état détaché est pratique pour certains types de démons qui ne se préoccupent pas de la valeur de sortie de ses threads. Par défaut, un nouveau thread est créé dans l'état joignable, à moins qu'*attr* n'ait été modifié (avec pthread_attr_setdetachstate(3)) pour créer le thread dans un état détaché.

Sous Linux/x86-32, la taille de la pile par défaut pour un nouveau thread est de 2 mégaoctets. Avec l'implémentation NPTL, si la limite souple *RLIMIT_STACK* a une valeur autre qu'« unlimited » *au moment où le programme a démarré*, alors elle détermine la taille de la pile par défaut pour les nouveaux threads. Afin d'obtenir une taille de pile différente de la valeur par défaut, il faut appeler pthread_attr_setstacksize(3) avec la valeur souhaitée sur l'argument *attr* utilisé pour créer un thread.

34 pthread_detach (3)

NOM

pthread_detach — Détacher un thread

SYNOPSIS

int pthread_detach(pthread_t *thread***);**

DESCRIPTION

La fonction *pthread_detach*() marque l'identifiant de thread identifié par *thread* comme détaché. Quand un thread détaché se termine, ses ressources sont automatiquement rendues au système sans avoir besoin d'un autre thread pour joindre le thread terminé.

Essayer de détacher un thread qui est déjà détaché résulte en un comportement indéfini.

VALEUR RENVOYÉE

En cas de réussite, *pthread_detach*() renvoie 0 ; en cas d'erreur, elle renvoie un numéro d'erreur.

NOTES

Une fois qu'un thread est détaché, il ne peut plus être joint avec pthread_join(3) ou être fait de nouveau joignable.

Un nouveau thread peut être créé dans un état détaché en utilisant pthread_attr_setdetachstate(3) pour positionner l'attribut détaché de l'argument *attr* de pthread_create(3).

L'attribut d'état de détachement détermine principalement le comportement du système quand le thread se termine. Il n'empêche pas le thread de terminer si le processus se termine avec exit(3) (ou, de manière équivalente, si le thread principal sort de sa routine d'appel).

Soit pthread_join(3), soit *pthread_detach*() devrait être appelé pour chaque thread créé par une application, afin que les ressources système du thread puissent être libérées. Notez cependant que les ressources de tous les threads sont libérées quand le processus se termine.

35 pthread_equal (3)

NOM

pthread_equal — Comparer des identifiants de threads

SYNOPSIS

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

DESCRIPTION

La fonction *pthread_equal*() compare deux identifiants de threads.

VALEUR RENVOYÉE

Si les deux identifiants de threads sont égaux, *pthread_equal*() renvoie une valeur non nulle. Autrement, elle renvoie 0.

NOTES

La fonction *pthread_equal*() est nécessaire car les identifiants de threads devraient être considérés comme étant opaques. Il n'existe pas de façon portable de comparer directement deux valeurs de pthread_t.

36 pthread_exit (3)

NOM

pthread_exit — Terminer le thread appelant

SYNOPSIS

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

DESCRIPTION

La fonction *pthread_exit*() termine le thread appelant et renvoie une valeur à travers *retval* qui, si le thread est joignable, est rendue disponible à un autre thread dans le même processus s'il appelle pthread_join(3).

Tous les gestionnaires de nettoyage ajoutés par pthread_cleanup_push(3) qui n'ont pas encore été dépilés sont dépilés (dans l'ordre inverse dans lequel ils ont été empilés) et exécutés. Si le thread contient des données spécifiques au thread, alors les destructeurs de ces données sont appelées, dans un ordre indéfini, une fois que tous les gestionnaires de nettoyage ont été exécutés.

Quand un thread se termine, les ressources partagées au niveau du processus (comme les verrous mutuellement exclusifs, des variables de condition, des sémaphores et des descripteurs de fichiers) ne sont pas libérées, et les fonctions enregistrées avec atexit(3) ne sont pas appelées.

Quand le dernier thread d'un processus se termine, le processus s'arrête en appelant exit(3) avec une valeur de sortie de zéro. Ainsi, les ressources partagées au niveau du processus sont libérées, et les fonctions enregistrées avec atexit(3) sont appelées.

VALEUR RENVOYÉE

Cette fonction ne retourne jamais vers l'appelant.

NOTES

Tout thread autre que le thread principal qui sort de la fonction initiale entraîne un appel implicite à *pthread_exit*(), en utilisant la valeur de retour de la fonction comme état de sortie du thread.

Afin de permettre aux autres threads de continuer l'exécution, le thread principal devrait se terminer en appelant *pthread_exit*() plutôt que exit(3).

La valeur pointée par *retval* ne devrait pas être placée sur la pile du thread appelant, puisque le contenu de cette pile devient indéfini quand le thread se termine.

37 pthread_join (3)

NOM

pthread_join — Joindre un thread terminé

SYNOPSIS

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

DESCRIPTION

La fonction *pthread_join*() attend que le thread spécifié par *thread* se termine. Si ce thread s'est déjà terminé, *pthread_join*() revient tout de suite. Le thread spécifié par *thread* doit être joignable.

Si *retval* n'est pas NULL, *pthread_join*() copie la valeur de sortie du thread cible (c'est-à-dire la valeur que le thread cible a fournie à pthread_exit(3)) dans l'emplacement pointé par **retval*. Si le thread cible est annulé, *PTHREAD_CANCELED* est placé dans **retval*.

Si plusieurs threads essaient simultanément de joindre le même thread, le résultat est indéfini. Si le thread appelant *pthread_join*() est annulé, le thread cible reste joignable (c'est-à-dire qu'il ne sera pas détaché).

VALEUR RENVOYÉE

En cas de réussite, *pthread_join*() renvoie 0 ; en cas d'erreur, elle renvoie un numéro d'erreur.

ERREURS

EDEADLK Un verrou perpétuel (*deadlock*) a été détecté, par exemple deux threads essaient de se joindre mutuellement ; ou bien *thread* est aussi le thread appelant.
EINVAL *thread* n'est pas un thread joignable.
EINVAL Un autre thread attend déjà de joindre ce thread.
ESRCH Aucun thread avec pour identifiant *thread* n'a pu être trouvé.

NOTES

Après un appel réussi à *pthread_join*(), l'appelant est sûr que le thread cible s'est terminé.

Joindre un thread qui avait préalablement été joint résulte en un comportement indéfini.

Un échec à joindre un thread qui est joignable (c'est-à-dire non détaché) produit un « thread zombie ». Il faut l'éviter, car chaque thread zombie consomme des ressources du système, et si trop de threads zombies s'accumulent, il ne sera plus possible de créer de nouveaux threads (ou de nouveaux processus).

Il n'existe pas d'analogue pthreads à *waitpid(-1, &status, 0)* pour joindre tout thread non terminé. Si vous pensez avoir besoin de cette fonctionnalité, vous devez probablement repenser la conception de votre application.

Tous les threads dans un processus sont au même niveau : tout thread peut joindre tout autre thread du processus.

38 pthread_kill (3)

NOM

pthread_kill — Envoyer un signal à un thread

SYNOPSIS

int pthread_kill(pthread_t *thread*, **int** *sig*);

DESCRIPTION

La fonction *pthread_kill*() envoie le signal *sig* à *thread*, un thread du même processus que l'appelant. Le signal est dirigé de manière asynchrone vers *thread*.

Si *sig* est 0, aucun signal n'est envoyé, mais la détection d'erreur est quand même effectuée.

VALEUR RENVOYÉE

En cas de réussite, *pthread_kill*() renvoie 0 ; en cas d'erreur, elle renvoie un numéro d'erreur, et aucun signal n'est envoyé.

NOTES

Les dispositions d'un signal sont définies au niveau du processus. Si un gestionnaire de signal est installé, le gestionnaire sera invoqué dans le thread *thread*, mais si la disposition du signal est « stop », « continue » ou « terminate », cette action affectera le processus entier.

La norme POSIX.1-2008 recommande que lorsqu'une implémentation détecte l'utilisation de l'identifiant d'un thread qui n'est plus en vie, l'appel *pthread_kill*() renvoie le message d'erreur ESRCH. L'implémentation de glibc renvoie cette erreur dans les cas où un identifiant de thread non valide est détecté. Il est à noter que POSIX précise également que l'utilisation d'un identifiant de thread dont l'exécution est terminée produit des effets indéfinis, et que l'utilisation d'un identifiant de thread invalide dans l'appel à *pthread_kill*() peut, par exemple, provoquer une erreur de segmentation (segmentation fault).

39 pthread_mutex_init, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock, pthread_mutex_destroy (3)

NOM

pthread_mutex_init, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock, pthread_mutex_destroy — Opérations sur les mutex

SYNOPSIS

pthread_mutex_t *fastmutex* **= PTHREAD_MUTEX_INITIALIZER;**

pthread_mutex_t *recmutex* **= PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;**

pthread_mutex_t *errchkmutex* **= PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;**

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

int pthread_mutex_lock(pthread_mutex_t **mutex*);

int pthread_mutex_trylock(pthread_mutex_t **mutex*);

int pthread_mutex_unlock(pthread_mutex_t **mutex*);

int pthread_mutex_destroy(pthread_mutex_t **mutex*);

DESCRIPTION

Un mutex est un objet d'exclusion mutuelle (MUTual EXclusion), et il est très pratique pour protéger des données partagées de modifications simultanées et pour implémenter des sections critiques et moniteurs.

Un mutex peut être dans deux états : déverrouillé (pris par aucun thread) ou verrouillé (pris par un thread). Un mutex ne peut être pris que par un seul thread à la fois. Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.

pthread_mutex_init() initialise le mutex pointé par *mutex* selon les attributs de mutex spécifié par *mutexattr*. Si *mutexattr* vaut *NULL*, les paramètres par défaut sont utilisés.

L'implémentation LinuxThreads ne gère qu'un seul attribut, le *type de mutex*, qui peut être soit « rapide », « récursif » ou à « vérification d'erreur ». Le type de mutex détermine s'il peut être verrouillé plusieurs fois par le même thread. Le type par défaut est « rapide ». Voyez pthread_mutexattr_init(3) pour plus d'informations sur les attributs de mutex.

Les variables de type pthread_mutex_t peuvent aussi être initialisées de manière statique, en utilisant les constantes *PTHREAD_MUTEX_INITIALIZER* (pour les mutex « rapides »), *PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP* (pour les mutex « récursifs ») et *PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP* (pour les mutex à « vérification d'erreur »).

pthread_mutex_lock() verrouille le mutex. Si le mutex est déverrouillé, il devient verrouillé et il est possédé par le thread appelant et *pthread_mutex_lock*() rend la main immédiatement. Si le mutex est déjà verrouillé par un autre thread, *pthread_mutex_lock* suspend le thread appelant jusqu'à ce que le mutex soit déverrouillé.

Si le mutex est déjà verrouillé par le thread appelant, le comportement de *pthread_mutex_lock*() dépend du type du mutex. Si ce dernier est de type « rapide », le thread appelant est suspendu jusqu'à ce que le mutex soit déverrouillé, plaçant ainsi le thread appelant en situation de blocage définitif. Si le mutex est de type « vérification d'erreur », *pthread_mutex_lock*() rend

la main immédiatement avec le code d'erreur EDEADLK. Si le mutex est de type « récursif », *pthread_mutex_lock*() rend la main immédiatement avec un code de retour indiquant le succès, enregistrant le nombre de fois où le thread appelant a verrouillé le mutex. Un nombre égal d'appels à *pthread_mutex_unlock*() doit être réalisé avant que le mutex retourne à l'état déverrouillé.

pthread_mutex_trylock() se comporte de la même manière que *pthread_mutex_lock*(), excepté qu'elle ne bloque pas le thread appelant si le mutex est déjà verrouillé par un autre thread (ou par le thread appelant dans le cas d'un mutex « rapide »). Au contraire, *pthread_mutex_trylock*() rend la main immédiatement avec le code d'erreur EBUSY.

pthread_mutex_unlock() déverrouille le mutex. Celui-ci est supposé verrouillé, et ce par le thread courant en entrant dans *pthread_mutex_unlock*(). Si le mutex est de type « rapide », *pthread_mutex_unlock*() le réinitialise toujours à l'état déverrouillé. S'il est de type « récursif », son compteur de verrouillage est décrémenté (du nombre d'opérations *pthread_mutex_lock*() réalisées sur le mutex par le thread appelant), et déverrouillé seulement quand ce compteur atteint 0.

Sur les mutex « vérification d'erreur » et « récursif », *pthread_mutex_unlock*() vérifie lors de l'exécution que le mutex est verrouillé en entrant, et qu'il est verrouillé par le même thread que celui appelant *pthread_mutex_unlock*(). Si ces conditions ne sont pas réunies, un code d'erreur est renvoyé et le mutex n'est pas modifié. Les mutex « rapides » ne réalisent pas de tels tests, permettant à un mutex verrouillé d'être déverrouillé par un thread autre que celui l'ayant verrouillé. Ce comportement n'est pas portable et l'on ne doit pas compter dessus.

pthread_mutex_destroy() détruit un mutex, libérant les ressources qu'il détient. Le mutex doit être déverrouillé. Dans l'implémentation LinuxThreads, aucune ressource ne peut être associée à un mutex, aussi *pthread_mutex_destroy*() ne fait rien si ce n'est vérifier que le mutex n'est pas verrouillé.

ANNULATION

Aucune des primitives relatives aux mutex n'est un point d'annulation, ni même *pthread_mutex_lock*(), malgré le fait qu'il peut suspendre l'exécution du thread pour une longue durée. De cette manière, l'état des mutex aux points d'annulation est prévisible, permettant aux gestionnaires d'annulation de déverrouiller précisément ces mutex qui nécessitent d'être déverrouillés avant que l'exécution du thread ne s'arrête définitivement. Aussi, les threads travaillant en mode d'annulation retardée ne doivent jamais verrouiller un mutex pour de longues périodes de temps.

ASYNC-SIGNAL SAFETY

Les fonctions relatives aux mutex ne sont pas fiables par rapport aux signaux asynchrones et ne doivent donc pas être utilisées dans des gestionnaires de signaux. En particulier, appeler *pthread_mutex_lock*() ou *pthread_mutex_unlock*() dans un gestionnaire de signal peut placer le thread appelant dans une situation de blocage définitif.

VALEUR RENVOYÉE

pthread_mutex_init() retourne toujours 0. Les autres fonctions renvoient 0 en cas de succès et un code d'erreur non nul en cas de problème.

ERREURS

La fonction *pthread_mutex_lock*() renvoie l'un des codes d'erreur suivants en cas de problème :

EINVAL Le mutex n'a pas été initialisé.
EDEADLK Le mutex est déjà verrouillé par le thread appelant (mutex à « vérification d'erreur » seulement).

La fonction *pthread_mutex_trylock*() renvoie l'un des codes d'erreur suivants en cas de problème :

EBUSY Le mutex ne peut être verrouillé car il l'est déjà.
EINVAL Le mutex n'a pas été initialisé.

40 pthread_self (3)

NOM

pthread_self — Obtenir l'identifiant du thread appelant

SYNOPSIS

pthread_t pthread_self(void);

DESCRIPTION

La fonction *pthread_self*() renvoie l'identifiant du thread appelant. C'est la même valeur qui est renvoyée dans **thread* dans l'appel à pthread_create(3) qui a créé ce thread.

VALEUR RENVOYÉE

Cette fonction réussit toujours, et renvoie l'identifiant du thread appelant.

NOTES

POSIX.1 laisse la liberté aux implémentations de choisir le type utilisé pour représenter l'identifiant des threads ; par exemple, une représentation par un type arithmétique ou par une structure est permise. Cependant, des variables de type pthread_t ne peuvent pas être comparées de manière portable en utilisant l'opérateur d'égalité C (==). Il faut utiliser pthread_equal(3) à la place.

Les identifiants de threads doivent être considérés comme opaques. Toute tentative pour utiliser un identifiant de thread autre part que dans des appels à pthreads n'est pas portable et peut entraîner des résultats indéfinis.

Les identifiants de threads ne sont garantis d'être uniques qu'à l'intérieur d'un processus. Un identifiant de thread peut être réutilisé après qu'un thread terminé a été rejoint, ou après qu'un thread détaché s'est terminé.

41 pthread_setcancelstate, pthread_setcanceltype (3)

NOM

pthread_setcancelstate, pthread_setcanceltype — Définir l'état et le type d'annulation

SYNOPSIS

int pthread_setcancelstate(int *state***, int** **oldstate***);**
int pthread_setcanceltype(int *type***, int** **oldtype***);**

DESCRIPTION

La fonction *pthread_setcancelstate*() définit l'état d'annulation du thread appelant à la valeur indiquée par *state*. L'ancien état d'annulation du thread est renvoyé dans le tampon pointé par *oldstate*. L'argument *state* doit avoir une des valeurs suivantes :

PTHREAD_CANCEL_ENABLE Le thread peut être annulé. C'est l'état d'annulation par défaut pour tous les nouveaux threads, y compris le thread initial. Le type d'annulation du thread détermine quand un thread annulable répondra à une requête d'annulation.

PTHREAD_CANCEL_DISABLE Le thread n'est pas annulable. Si une requête d'annulation arrive, elle est bloquée jusqu'à ce que l'annulation soit activée.

La fonction *pthread_setcanceltype*() définit le type d'annulation du thread appelant à la valeur indiquée par *type*. L'ancien type d'annulation du thread est renvoyé dans le tampon pointé par *oldtype*. L'argument *type* doit avoir une des valeurs suivantes :

PTHREAD_CANCEL_DEFERRED Une requête d'annulation est retardé jusqu'à ce que le thread appelle une fonction qui est un point d'annulation (consultez pthreads(7)). C'est le type d'annulation par défaut pour tous les nouveaux threads, y compris le thread initial.

PTHREAD_CANCEL_ASYNCHRONOUS Le thread peut être annulé à tout moment. Typiquement, il sera annulé dès réception de la requête d'annulation, mais ce n'est pas garanti par le système.

Les opérations *set/get* effectuées par ces fonctions sont atomiques, eu égard aux autres threads du processus qui appellent la même fonction.

VALEUR RENVOYÉE

En cas de réussite, ces fonctions renvoient 0 ; en cas d'erreur elles renvoient un numéro d'erreur non nul.

NOTES

Pour des détails sur ce qui se passe quand un thread est annulé, voyez pthread_cancel(3).

Désactiver brièvement l'annulation peut être pratique si un thread effectue une action critique qui ne doit pas être interrompue par une requête d'annulation. Mais attention de ne pas désactiver l'annulation sur de longues périodes, ou autour d'opérations qui peuvent bloquer pendant un long moment, car cela empêcherait le thread de répondre aux requêtes d'annulation.

Le type d'annulation est rarement mis à *PTHREAD_CANCEL_ASYNCHRONOUS*. Comme le thread pourrait être annulé n'importe quand, il ne pourrait pas réserver de ressources (par exemple en allouant de la mémoire avec malloc(3)) de manière sûre, acquérir des verrous exclusifs (*mutex*), des sémaphores, des verrous, etc. Réserver des ressources n'est pas sûr, car l'application n'a aucun moyen de connaître l'état de ces ressources quand le thread est annulé ; en d'autres termes, l'annulation arrive-t-elle avant que les ressources n'aient été réservées, pendant qu'elles sont réservées, ou après qu'elles ont été libérées ? De plus, certaines structures de données internes (par exemple la liste chaînée des blocs libres gérée par la famille de fonctions malloc(3)) pourraient se retrouver dans un état incohérent si l'annulation se passe au milieu d'un appel de fonction. En conséquence de quoi les gestionnaires de nettoyage perdent toute utilité. Les fonctions qui peuvent sans risque être annulées de manière asynchrone sont appelées des *fonctions async-cancel-safe*. POSIX.1-2001 nécessite seulement que pthread_cancel(3), *pthread_setcancelstate*() et *pthread_setcanceltype*() soient async-cancel-safe. En général, les autres fonctions de la bibliothèque ne peuvent pas être appelées de manière sûre depuis un thread annulable immédiatement. Une des rares circonstances dans lesquelles une annulation immédiate est utile est pour l'annulation d'un thread qui est dans une boucle qui ne fait que des calculs.

Les implémentations de Linux autorisent l'argument *oldstate* de *pthread_setcancelstate*() à être NULL, auquel cas l'information au sujet de l'état antérieur d'annulation n'est pas renvoyé à l'appelant. Beaucoup d'autres implémentations autorisent aussi un argument *oldstat* NULL, mais POSIX.1-2001 ne spécifie pas ce point, si bien que les applications portables devraient toujours donner une valeur non NULL à *oldstate*. Le même type de raisonnement s'applique à l'argument *oldtype* de *pthread_setcanceltype*().

42 read (2)

NOM

read — Lire depuis un descripteur de fichier

SYNOPSIS

ssize_t read(int *fd***, void** **buf***, size_t** *count***);**

DESCRIPTION

read() lit jusqu'à *count* octets depuis le descripteur de fichier *fd* dans le tampon pointé par *buf*.

Sur les fichiers permettant le positionnement, l'opération de lecture a lieu à la position actuelle dans le fichier et elle est déplacée du nombre d'octets lus. Si la position actuelle dans le fichier est à la fin du fichier ou après, aucun octet n'est lu et *read*() renvoie zéro.

VALEUR RENVOYÉE

read() renvoie -1 s'il échoue, auquel cas `errno` contient le code d'erreur, et la position de la tête de lecture est indéfinie. Sinon, *read* renvoie le nombre d'octets lus (0 en fin de fichier), et avance la tête de lecture de ce nombre. Le fait que le nombre renvoyé soit plus petit que le nombre demandé n'est pas une erreur. Ceci se produit à la fin du fichier, ou si on lit depuis un tube ou un terminal, ou encore si *read*() a été interrompu par un signal.

ERREURS

EAGAIN ou EWOULDBLOCK Le descripteur de fichier *fd* fait référence à un fichier autre qu'une socket et a été marqué comme non bloquant (*O_NONBLOCK*), et la lecture devrait bloquer.
EBADF *fd* n'est pas un descripteur de fichier valable ou n'est pas ouvert en lecture.
EINTR *read*() a été interrompu par un signal avant d'avoir eu le temps de lire quoi que ce soit ; consultez signal(7).

43 readdir (3)

NOM

readdir — Consulter un répertoire

SYNOPSIS

struct dirent *readdir(DIR *_dirp_);

DESCRIPTION

La fonction *readdir*() renvoie un pointeur sur une structure *dirent* représentant l'entrée suivante du flux répertoire pointé par *dirp*. Elle renvoie NULL a la fin du répertoire, ou en cas d'erreur.

Avec Linux, la structure *dirent* est définie comme suit :

```
struct dirent {
    ino_t          d_ino;       /* numéro d'inoeud */
    ...
    char           d_name[256]; /* nom du fichier */
};
```

Les données renvoyées par *readdir*() sont écrasées lors de l'appel suivant à *readdir*() sur le même flux répertoire.

VALEUR RENVOYÉE

En cas de succès, *readdir*() renvoie un pointeur sur une structure *dirent* (cette structure peut avoir été allouée statiquement ; n'essayez pas de la désallouer avec free(3)). Lorsque la fin du flux répertoire est atteinte, NULL est renvoyé et `errno` n'est pas modifiée. En cas d'erreur, NULL est renvoyée et `errno` contient le code d'erreur.

ERREURS

EBADF Le descripteur de flux du répertoire, *dirp*, n'est pas valable.

44 readlink (2)

NOM

readlink — Lire le contenu d'un lien symbolique

SYNOPSIS

ssize_t readlink(const char **pathname*, **char** **buf*, **size_t** *bufsiz***);**

DESCRIPTION

readlink() place le contenu du lien symbolique *pathname* dans le tampon *buf*, dont la taille est *bufsiz*. *readlink*() n'ajoute pas de caractère *NUL* dans le tampon *buf*. Il tronquera le contenu (à la longueur *bufsiz*) si le tampon est trop petit pour recevoir les données.

VALEUR RENVOYÉE

S'il réussit, cet appel renvoie le nombre d'octets placés dans *buf*. S'il échoue, il renvoie -1 et écrit `errno` en conséquence.

45 rmdir (2)

NOM

rmdir — Supprimer un répertoire

SYNOPSIS

int rmdir(const char **pathname*);

DESCRIPTION

rmdir() supprime un répertoire, lequel doit être vide.

46 sched_yield (2)

NOM

sched_yield — Céder le processeur

SYNOPSIS

int sched_yield(void);

DESCRIPTION

sched_yield() force le thread appelant à libérer le CPU. Le thread est déplacé à la fin de la liste des processus prêts de sa priorité, et un autre thread sera exécuté.

VALEUR RENVOYÉE

sched_yield() renvoie 0 s'il réussit ou -1 s'il échoue auquel cas `errno` contient le code d'erreur.

NOTES

Si le thread appelant est le seul avec la priorité la plus élevée au moment de l'appel, il continuera son exécution après un appel à *sched_yield*().

Les systèmes POSIX sur lesquels *sched_yield*() est disponible définissent *_POSIX_PRIORITY_SCHEDULING* dans *<unistd.h>*.

Des appels stratégiques à *sched_yield*() peuvent améliorer les performances en donnant à d'autres thread ou processus une chance de s'exécuter quand des ressources (très) demandées (par exemple, des mutex) sont libérées par l'appelant. Évitez d'appeler *sched_yield*() si ce n'est pas nécessaire ou inapproprié (par exemple, si les ressources nécessaires pour d'autres threads pouvant être ordonnancés sont encore tenues par l'appelant), puisqu'en faisant ainsi provoquera des changements de contexte non nécessaire, qui dégraderont les performances du système.

47 sem_destroy (3)

NOM

sem_destroy — Détruire un sémaphore non nommé

SYNOPSIS

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

DESCRIPTION

sem_destroy() détruit le sémaphore non nommé situé à l'adresse pointée par *sem*.

Seul un sémaphore initialisé avec sem_init(3) peut être détruit avec *sem_destroy*().

La destruction d'un sémaphore sur lequel des processus ou threads sont bloqués (dans sem_wait(3)) produira un comportement indéfini.

L'utilisation d'un sémaphore détruit produira des résultats indéfinis jusqu'à ce que le sémaphore soit réinitialisé avec sem_init(3).

NOTES

Un sémaphore non nommé devrait être détruit avec *sem_destroy*() avant que la mémoire dans laquelle il est situé ne soit libérée. Ne pas le faire peut provoquer des fuites de ressource sur certaines implémentations.

48 sem_init (3)

NOM

sem_init — Initialiser un sémaphore non nommé

SYNOPSIS

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

DESCRIPTION

sem_init() initialise le sémaphore non nommé situé à l'adresse pointée par *sem*. L'argument *value* spécifie la valeur initiale du sémaphore.

L'argument *pshared* indique si ce sémaphore sera partagé entre les threads d'un processus ou entre processus.

Si *pshared* vaut 0, le sémaphore est partagé entre les threads d'un processus et devrait être situé à une adresse visible par tous les threads (par exemple, une variable globale ou une variable allouée dynamiquement dans le tas).

Si *pshared* n'est pas nul, le sémaphore est partagé entre processus et devrait être situé dans une région de mémoire partagée (puisqu'un fils créé avec fork(2) hérite de la projection mémoire du père, il peut accéder au sémaphore). Tout processus qui peut accéder à la région de mémoire partagée peut opérer sur le sémaphore avec sem_post(3), sem_wait(3), etc.

L'initialisation d'un sémaphore qui a déjà été initialisé résulte en un comportement indéfini.

49 sem_post (3)

NOM

sem_post — Déverrouiller un sémaphore

SYNOPSIS

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

DESCRIPTION

sem_post() incrémente (déverrouille) le sémaphore pointé par *sem*. Si, à la suite de cet incrément, la valeur du sémaphore devient supérieure à zéro, un autre processus ou thread bloqué dans un appel sem_wait(3) sera réveillé et procédera au verrouillage du sémaphore.

VALEUR RENVOYÉE

sem_post() renvoie 0 s'il réussit. S'il échoue, la valeur du sémaphore n'est pas modifiée, il renvoie -1 et écrit `errno` en conséquence.

50 sem_wait, sem_timedwait, sem_trywait (3)

NOM

sem_wait, sem_timedwait, sem_trywait — Verrouiller un sémaphore

SYNOPSIS

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

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

int sem_timedwait(sem_t **sem*, **const struct timespec** **abs_timeout***);**

DESCRIPTION

sem_wait() décrémente (verrouille) le sémaphore pointé par *sem*. Si la valeur du sémaphore est plus grande que 0, la décrémentation s'effectue et la fonction revient immédiatement. Si le sémaphore vaut zéro, l'appel bloquera jusqu'à ce que soit il devienne disponible d'effectuer la décrémentation (c'est-à-dire la valeur du sémaphore n'est plus nulle), soit un gestionnaire de signaux interrompe l'appel.

sem_trywait() est pareil que *sem_wait*(), excepté que si la décrémentation ne peut pas être effectuée immédiatement, l'appel renvoie une erreur (`errno` vaut EAGAIN) plutôt que bloquer.

sem_timedwait() est pareil que *sem_wait*(), excepté que *abs_timeout* spécifie une limite sur le temps pendant lequel l'appel bloquera si la décrémentation ne peut pas être effectuée immédiatement. L'argument *abs_timeout* pointe sur une structure qui spécifie un temps absolu en secondes et nanosecondes depuis l'époque, 1er janvier 1970 à 00:00:00 (UTC). Cette structure est définie de la manière suivante :

```
struct timespec {
    time_t tv_sec;      /* Secondes */
    long   tv_nsec;     /* Nanosecondes [0 .. 999999999] */
};
```

Si le délai est déjà expiré à l'heure de l'appel et si le sémaphore ne peut pas être verrouillé immédiatement, *sem_timedwait*() échoue avec l'erreur d'expiration de délai (`errno` vaut ETIMEDOUT).

Si l'opération peut être effectuée immédiatement, *sem_timedwait*() n'échoue jamais avec une valeur d'expiration de délai, quelque soit la valeur de *abs_timeout*. De plus, la validité de *abs_timeout* n'est pas vérifiée dans ce cas.

VALEUR RENVOYÉE

Toutes ces fonctions renvoient 0 si elles réussissent. Si elles échouent, la valeur du sémaphore n'est pas modifiée, elles renvoient -1 et écrivent `errno` en conséquence.

ERREURS

EINTR L'appel a été interrompu par un gestionnaire de signal.

L'erreur supplémentaire suivante peut survenir pour *sem_trywait*() :

EAGAIN L'opération ne peut pas être effectuée sans bloquer (c'est-à-dire, le sémaphore a une valeur nulle).

Les erreurs supplémentaires suivantes peuvent survenir pour *sem_timedwait*() :

EINVAL La valeur de *abs_timeout.tv_nsecs* est plus petite que 0 ou supérieure ou égale à 1 milliard.
ETIMEDOUT Le délai a expiré avant que le sémaphore ait pu être verrouillé.

NOTES

Un gestionnaire de signaux interrompra toujours un appel bloqué à l'une de ces fonctions, quelque soit l'utilisation de l'attribut *SA_RESTART* de sigaction(2).

51 setpgid, getpgid, setpgrp, getpgrp (2)

NOM

setpgid, getpgid, setpgrp, getpgrp — Définir ou lire le groupe du processus

SYNOPSIS

int setpgid(pid_t *pid*, **pid_t** *pgid***);**
pid_t getpgid(pid_t *pid***);**

pid_t getpgrp(void);

DESCRIPTION

Toutes ces interfaces sont disponibles sous Linux et sont utilisées pour récupérer et définir l'identifiant du groupe de processus (PGID : « Process Group ID ») d'un processus. La façon préférée, celle spécifiée par POSIX.1 est : *getpgrp*(void) pour récupérer le PGID du processus appelant et *setpgid*() pour définir le PGID d'un processus.

setpgid() définit à *pgid* le PGID du processus mentionné par *pid*. Si *pid* vaut zéro, alors le PID du processus appelant est utilisé. Si *pgid* vaut zéro, alors le PGID du processus indiqué par *pid* est positionné à la même valeur que l'identifiant du processus.

getpgid() renvoie le PGID du processus indiqué par *pid*. Si *pid* vaut zéro, le PID du processus appelant est utilisé. La récupération du PGID d'un processus autre que l'appelant est rarement utilisée et *getpgrp*() est préférée pour cette tâche.

VALEUR RENVOYÉE

L'appel *getpgrp*() renvoie le PGID du processus appelant.

getpgid() renvoie le groupe du processus s'il réussit. En cas d'erreur -1 est renvoyé et `errno` contient le code d'erreur.

NOTES

Un processus fils créé par fork(2) hérite du PGID de son père. Le PGID est conservé au travers d'un execve(2).

Les appels *setpgid*() et *getpgrp*() sont utilisés par des programmes comme bash(1) pour créer des groupes de processus pour l'implémentation du contrôle des travaux depuis l'interpréteur de commande.

52 sigaction (2)

NOM

sigaction — Examiner et modifier l'action associée à un signal

SYNOPSIS

int sigaction(int *signum*, **const struct sigaction** **act*, **struct sigaction** **oldact*);

DESCRIPTION

L'appel système *sigaction*() sert à modifier l'action effectuée par un processus à la réception d'un signal spécifique.

signum indique le signal concerné, à l'exception de `SIGKILL` et `SIGSTOP`.

Si *act* n'est pas NULL, la nouvelle action pour le signal *signum* est définie par *act*. Si *oldact* n'est pas NULL, l'ancienne action est sauvegardée dans *oldact*.

La structure *sigaction* est définie par quelque chose comme :

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    int        sa_flags;
};
```

sa_handler indique l'action affectée au signal *signum*, et peut être *SIG_DFL* pour l'action par défaut, *SIG_IGN* pour ignorer le signal, ou un pointeur sur une fonction de gestion de signaux.

sa_mask spécifie un masque de signaux à bloquer (c'est-à-dire ajoutés au masque de signaux du thread dans lequel le gestionnaire est appelé) pendant l'exécution du gestionnaire. De plus le signal ayant appelé le gestionnaire est bloqué à moins que l'attribut *SA_NODEFER* soit précisé.

sa_flags spécifie un ensemble d'attributs qui modifient le comportement du signal. Il est formé par un OU binaire entre les options suivantes :

SA_NODEFER Ne pas empêcher un signal d'être reçu depuis l'intérieur de son propre gestionnaire.

SA_RESETHAND Rétablir l'action à son comportement par défaut à l'entrée dans le gestionnaire de signal. Cet attribut n'a de sens que lors de la mise en place d'un gestionnaire de signal.

SA_RESTART Fournir un comportement compatible avec la sémantique BSD en redémarrant automatiquement les appels système lents interrompus par l'arrivée du signal. Cet attribut n'a de sens que lors de la mise en place d'un gestionnaire de signal. Consultez signal(7) pour une discussion sur le redémarrage d'un appel système.

ERREURS

EINVAL Un signal invalide est indiqué. Ceci se produit également si l'on tente de modifier l'action associée aux signaux `SIGKILL` ou `SIGSTOP`, qui ne peuvent pas être interceptés ou ignorés.

NOTES

La seule méthode complètement portable pour s'assurer que les fils ne deviennent pas des zombies à leur terminaison est d'intercepter le signal `SIGCHLD` et d'invoquer wait(2) ou équivalent.

53 sigprocmask (2)

NOM

sigprocmask — Examiner et modifier les signaux bloqués

SYNOPSIS

int sigprocmask(int *how*, **const sigset_t** **set*, **sigset_t** **oldset*);

DESCRIPTION

L'appel *sigprocmask*() est utilisé pour récupérer et/ou changer le masque de signaux du thread appelant. Le masque de signaux est l'ensemble des signaux dont la distribution est actuellement bloquée pour l'appelant.

Son comportement est dépendant de la valeur de *how*, avec les conventions suivantes :

SIG_BLOCK L'ensemble des signaux bloqués est l'union de l'ensemble actuel et de l'argument *set*.

SIG_UNBLOCK Les signaux dans l'ensemble *set* sont supprimés de la liste des signaux bloqués. Il est permis de débloquer un signal non bloqué.

SIG_SETMASK L'ensemble des signaux bloqués est égal à l'argument *set*.

Si *oldset* n'est pas NULL, la valeur précédente du masque de signaux est stockée dans *oldset*.

Si *set* est NULL, le masque de signaux n'est pas modifié (*how* est donc ignoré), mais la valeur actuelle du masque de signaux est tout de même renvoyée dans *oldset* (s'il n'est pas NULL).

L'utilisation de *sigprocmask*() dans un processus multithreadé n'est pas définie ; consultez pthread_sigmask(3).

NOTES

Il est impossible de bloquer `SIGKILL` or `SIGSTOP` avec l'appel sigprocmask. Les tentatives seront ignorées silencieusement.

Un processus fils créé avec fork(2) hérite d'une copie du masque de signaux de son père ; le masque de signaux est conservé au travers d'un execve(2).

Si l'un des signaux `SIGBUS`, `SIGFPE`, `SIGILL` ou `SIGSEGV` est généré alors qu'il est bloqué, le résultat est indéfini, sauf si le signal a été généré par kill(2), sigqueue(3) ou raise(3).

Consultez sigsetops(3) pour les détails concernant les ensembles de signaux.

54 sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember (3)

NOM

sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember — Opérations sur les ensembles de signaux POSIX

SYNOPSIS

int sigemptyset(sigset_t **set***);**

int sigfillset(sigset_t **set***);**

int sigaddset(sigset_t **set***, int** *signum***);**

int sigdelset(sigset_t **set***, int** *signum***);**

int sigismember(const sigset_t **set***, int** *signum***);**

DESCRIPTION

Ces fonctions permettent la manipulation des ensembles de signaux POSIX.

sigemptyset() vide l'ensemble de signaux fourni par *set*, tous les signaux étant exclus de cet ensemble.

sigfillset() remplit totalement l'ensemble de signaux *set* en incluant tous les signaux.

sigaddset() et *sigdelset*() ajoutent ou suppriment respectivement le signal *signum* de l'ensemble *set*.

sigismember() teste si le signal *signum* est membre de l'ensemble *set*.

Les objets de type sigset_t doivent être initialisés par un appel à *sigemptyset*() ou *sigfillset*() avant d'être passé aux fonctions *sigaddset*(), *sigdelset*() et *sigismember*(). Les résultats ne sont pas définis si ce n'est pas fait.

VALEUR RENVOYÉE

sigismember() renvoie 1 si le signal *signum* est dans l'ensemble *set*, 0 si *signum* n'y est pas, et -1 en cas d'erreur. En cas d'erreur, `errno` contient le code d'erreur.

55 sigsuspend (2)

NOM

sigsuspend — Attendre un signal

SYNOPSIS

int sigsuspend(const sigset_t **mask***);**

DESCRIPTION

L'appel *sigsuspend*() remplace temporairement le masque de signaux bloqués par celui fourni dans *mask* puis endort le processus jusqu'à arrivée d'un signal qui déclenche un gestionnaire de signal ou termine le processus.

Si le signal termine le processus, *sigsuspend*() ne retourne pas à l'appelant. Si le signal est intercepté, *sigsuspend*() retourne après l'exécution du gestionnaire, et le masque de signaux bloqués est restauré à sa valeur précédant l'appel à *sigsuspend*().

NOTES

En général, *sigsuspend*() est utilisé conjointement avec sigprocmask(2) pour empêcher l'arrivée d'un signal pendant l'exécution d'une section de code critique. L'appelant commence par bloquer les signaux avec sigprocmask(2). Après la fin de la section critique, l'appelant attend les signaux avec *sigsuspend*() utilisé avec le masque renvoyé par sigprocmask(2) (dans l'argument *oldset*).

56 sleep (3)

NOM

sleep — Endormir le processus pour une durée déterminée

SYNOPSIS

unsigned int sleep(unsigned int *nb_sec***);**

DESCRIPTION

sleep() endort le thread appelant jusqu'à ce que *nb_sec* secondes se soient écoulées, ou jusqu'à ce qu'un signal non ignoré soit reçu.

VALEUR RENVOYÉE

sleep() renvoie zéro si le temps prévu s'est écoulé, ou le nombre de secondes restantes si l'appel a été interrompu par un gestionnaire de signal.

BOGUES

sleep() peut être implémenté en utilisant `SIGALRM`; ainsi l'utilisation conjointe de alarm(2) et *sleep*() est une très mauvaise idée.

57 printf, fprintf, sprintf, snprintf (3)

NOM

printf, fprintf, sprintf, snprintf — Formatage des sorties

SYNOPSIS

int printf(const char **format*, **...);**
int fprintf(FILE **stream*, **const char** **format*, **...);**
int sprintf(char **str*, **const char** **format*, **...);**
int snprintf(char **str*, **size_t** *size*, **const char** **format*, **...);**

DESCRIPTION

Les fonctions de la famille *printf*() produisent des sorties en accord avec le *format* décrit plus bas. Les fonctions *printf*() et *vprintf*() écrivent leur sortie sur *stdout*, le flux de sortie standard. *fprintf*() et *vfprintf*() écrivent sur le flux *stream* indiqué. *sprintf*(), *snprintf*(), *vsprintf*() et *vsnprintf*() écrivent leurs sorties dans la chaîne de caractères *str*.

Les fonctions *snprintf*() et *vsnprintf*() écrivent au plus *size* octets (octet nul (« \0 ») final compris) dans *str*.

Ces fonctions créent leurs sorties sous le contrôle d'une chaîne de *format* qui indique les conversions à apporter aux arguments suivants.

VALEUR RENVOYÉE

En cas de succès, ces fonctions renvoient le nombre de caractères affichés (sans compter l'octet nul final utilisé pour terminer les sorties dans les chaînes).

Les fonctions *snprintf*() et *vsnprintf*() n'écrivent pas plus de *size* octets (y compris l'octet nul final). Si la sortie a été tronquée à cause de la limite, la valeur de retour est le nombre de caractères (octet nul final non compris) qui auraient été écrits dans la chaîne s'il y avait eu suffisamment de place. Ainsi, une valeur de retour *size* ou plus signifie que la sortie a été tronquée.

Si une erreur de sortie s'est produite, une valeur négative est renvoyée.

58 socketpair (2)

NOM

socketpair — Créer une paire de sockets connectées

SYNOPSIS

int socketpair(int *domain*, **int** *type*, **int** *protocol*, **int** *sv*[**2**]);

DESCRIPTION

La fonction *socketpair*() crée une paire de sockets connectées, sans noms, dans le domaine de communication *domain*, du *type* indiqué, en utilisant le protocole optionnel *protocol*. Pour plus de détails sur ces paramètres, consultez socket(2).

Les descripteurs correspondant aux deux sockets sont placés dans *sv*[0] et *sv*[1]. Les deux sockets ne sont pas différenciables.

59 stat, fstat, lstat (2)

NOM

stat, fstat, lstat — Obtenir l'état d'un fichier (file status)

SYNOPSIS

int stat(const char **pathname*, **struct stat** **buf*);
int fstat(int *fd*, **struct stat** **buf*);
int lstat(const char **pathname*, **struct stat** **buf*);

DESCRIPTION

Ces fonctions renvoient des renseignements sur le fichier indiqué, dans le tampon pointé par *stat*. Vous n'avez besoin d'aucun droit d'accès au fichier pour obtenir les informations, mais vous devez — dans le cas de *stat*() et *lstat*() — avoir le droit de parcours de tous les répertoires mentionnés dans le chemin menant au fichier.

lstat() est identique à *stat*(), sauf que dans le cas où *pathname* est un lien symbolique, auquel cas il renvoie des renseignements sur le lien lui-même plutôt que celui du fichier visé.

fstat() est identique à *stat*(), sauf que le fichier dont les renseignements sont à récupérer est référencé par le descripteur de fichier *fd*.

Les trois fonctions renvoient une structure *stat* contenant les champs suivants :

```
struct stat {
    dev_t     st_dev;         /* Périphérique                   */
    ino_t     st_ino;         /* Numéro d'inoeud                */
    mode_t    st_mode;        /* Protection                     */
    nlink_t   st_nlink;       /* Nombre de liens physiques      */
    uid_t     st_uid;         /* UID du propriétaire            */
    gid_t     st_gid;         /* GID du propriétaire            */
    dev_t     st_rdev;        /* Type de périphérique           */
    off_t     st_size;        /* Taille totale en octets        */
    blksize_t st_blksize;     /* Taille de bloc pour E/S        */
    blkcnt_t  st_blocks;      /* Nombre de blocs de 512 o alloués */
```

```
    struct timespec st_atim;  /* Heure dernier accès              */
    struct timespec st_mtim;  /* Heure dernière modification      */
    struct timespec st_ctim;  /* Heure dernier changement état    */
};
```

Le champ *st_dev* décrit le périphérique sur lequel ce fichier réside. Les macros major(3) et minor(3) peuvent être utiles pour décomposer l'identifiant de périphérique de ce champ.

Le champ *st_rdev* indique le périphérique que ce fichier (inoeud) représente.

Le champ *st_size* indique la taille du fichier (s'il s'agit d'un fichier ordinaire ou d'un lien symbolique) en octets. La taille d'un lien symbolique est la longueur de la chaîne représentant le chemin d'accès qu'il vise, sans le caractère NUL final.

Le champ *st_blocks* indique le nombre de blocs de 512 octets alloués au fichier. Cette valeur peut être inférieure à *st_size*/512 si le fichier a des trous.

Le champ *st_blksize* donne la taille de bloc « préférée » pour des entrées-sorties efficaces. Des écritures par blocs plus petits peuvent entraîner un cycle lecture/modification/réécriture inefficace.

Les macros POSIX suivantes sont fournies pour vérifier le type de fichier (dans le champ *st_mode*) :

***S_ISREG*(m)** un fichier ordinaire ?
***S_ISDIR*(m)** un répertoire ?
***S_ISCHR*(m)** un périphérique caractère ?
***S_ISBLK*(m)** un périphérique bloc ?
***S_ISFIFO*(m)** FIFO (tube nommé) ?
***S_ISLNK*(m)** un lien symbolique ? (Pas dans POSIX.1-1996).
***S_ISSOCK*(m)** une socket ? (Pas dans POSIX.1-1996).

60 strcat, strncat (3)

NOM

strcat, strncat — Concaténer deux chaînes

SYNOPSIS

char *strcat(char ****dest*, const char ****src***);**
char *strncat(char ****dest*, const char ****src***, size_t** ***n*****);**

DESCRIPTION

La fonction *strcat*() ajoute la chaîne *src* à la fin de la chaîne *dest* en écrasant l'octet nul (« \0 ») final à la fin de *dest*, puis en ajoutant un nouvel octet nul final. Les chaînes ne doivent pas se chevaucher, et la chaîne *dest* doit être assez grande pour accueillir le résultat. Si *dest* n'est pas assez grande, le comportement du programme est imprévisible. *Les dépassements de tampon font partie des moyens préférés pour attaquer les programmes sécurisés*.

La fonction *strncat*() est similaire, à la différence que :

— elle ne prend en compte que les *n* premiers octets de *src*,
— *src* n'a pas besoin de se terminer par un caractère nul si elle contient au moins *n* octets.

Comme pour *strcat*(), la chaîne résultante dans *dest* est toujours terminée par un caractère nul.

Si *src* contient au moins *n* octets, *strncat*() écrit *n+1* octets dans *dest* (*n* octets de *src* plus le caractère nul final). De ce fait, *dest* doit être au moins de taille *strlen(dest)+n+1*.

VALEUR RENVOYÉE

Les fonctions *strcat*() et *strncat*() renvoient un pointeur sur la chaîne résultat *dest*.

61 strcmp, strncmp (3)

NOM

strcmp, strncmp — Comparaison de deux chaînes

SYNOPSIS

int strcmp(const char **s1*, **const char** **s2*);
int strncmp(const char **s1*, **const char** **s2*, **size_t** *n*);

DESCRIPTION

La fonction *strcmp*() compare les deux chaînes *s1* et *s2*. Elle renvoie un entier négatif, nul, ou positif, si *s1* est respectivement inférieure, égale ou supérieure à *s2*.

La fonction *strncmp*() est identique sauf qu'elle ne compare que les *n* (au plus) premiers octets de *s1* et *s2*.

VALEUR RENVOYÉE

Les fonctions *strcmp*() et *strncmp*() renvoient un entier inférieur, égal ou supérieur à zéro si *s1* (ou ses *n* premiers octets) est respectivement inférieure, égale ou supérieure à *s2*.

62 strcpy, strncpy (3)

NOM

strcpy, strncpy — Copier une chaîne

SYNOPSIS

char *strcpy(char **dest*, **const char** **src*);
char *strncpy(char **dest*, **const char** **src*, **size_t** *n*);

DESCRIPTION

La fonction *strcpy*() copie la chaîne pointée par *src*, y compris le caractère nul (« \0 ») final dans la chaîne pointée par *dest*. Les deux chaînes ne doivent pas se chevaucher. La chaîne *dest* doit être assez grande pour accueillir la copie. *Attention aux dépassements de tampon !* (consultez *BUGS*).

La fonction *strncpy*() est identique, sauf qu'au plus *n* octets de *src* sont copiés. *Attention* : s'il n'y a pas de caractère nul dans les *n* premiers octets de *src*, la chaîne résultante dans *dest* ne disposera pas de caractère nul final.

Si la longueur de *src* est inférieure à *n*, *strncpy*() écrit des caractères nuls supplémentaires vers *dest* pour s'assurer qu'un total de *n* octets ont été écrits.

VALEUR RENVOYÉE

Les fonctions *strcpy*() et *strncpy*() renvoient un pointeur sur la chaîne destination *dest*.

BUGS

Si la chaîne de destination d'un *strcpy*() n'est pas suffisamment grande, n'importe quoi peut survenir. Un dépassement de tampon pour une chaîne de taille fixe est la technique favorite de pirates pour prendre le contrôle d'une machine. À chaque fois qu'un programme lit ou copie des données dans un tampon, le programme doit d'abord vérifier qu'il y a suffisamment de place. Ça peut ne pas être nécessaire si vous pouvez montrer qu'un dépassement est impossible, mais faites attention : les programmes changent au cours du temps, et ce qui était impossible peut devenir possible.

63 strdup (3)

NOM

strdup — Dupliquer une chaîne

SYNOPSIS

char *strdup(const char *s);

DESCRIPTION

La fonction *strdup*() renvoie un pointeur sur une nouvelle chaîne de caractères qui est dupliquée depuis *s*. La mémoire occupée par cette nouvelle chaîne est obtenue en appelant malloc(3), et peut (doit) donc être libérée avec free(3).

VALEUR RENVOYÉE

En cas de succès, la fonction *strdup*() renvoie un pointeur sur la chaîne dupliquée. NULL est renvoyé s'il n'y avait pas assez de mémoire et `errno` contient le code d'erreur.

64 symlink (2)

NOM

symlink — Créer un nouveau nom pour un fichier

SYNOPSIS

int symlink(const char **target***, const char** **linkpath***);**

DESCRIPTION

symlink() crée un lien symbolique avec le nom *linkpath* indiqué, et qui pointe sur *target*.

Les liens sont interprétés à l'exécution, comme si le contenu du lien était remplacé par le chemin d'accès pour trouver un fichier ou un répertoire.

Les liens symboliques peuvent contenir `..` pour le chemin, qui (s'il est utilisé au début du lien) se réfère aux répertoires parents du lien.

Un lien symbolique (aussi nommé « soft link ») peut pointer vers un fichier existant ou sur un fichier non existant.

Si *linkpath* existe, il ne sera *pas* écrasé.

NOTES

Il n'y a pas de vérification de l'existence de *target*.

Effacer le nom référencé par un lien symbolique effacera effectivement le fichier (à moins qu'il ait d'autres liens matériels). Si ce comportement est indésirable, utiliser *link*().

65 system (3)

NOM

system — Exécuter une commande shell

SYNOPSIS

int system(const char **command***);**

DESCRIPTION

La fonction *system*() exécute la commande indiquée dans *command* en appelant `/bin/sh -c` *command*, et revient après l'exécution complète de la commande. Durant cette exécution, le signal `SIGCHLD` est bloqué, et les signaux `SIGINT` et `SIGQUIT` sont ignorés.

VALEUR RENVOYÉE

La valeur renvoyée est -1 en cas d'erreur (par exemple échec de fork(2)) ou le code de retour de la commande en cas de succès. Ce dernier code est dans le format indiqué dans wait(2). Ainsi, le retour de la commande sera *WEXITSTATUS(status)*. Dans le cas où `/bin/sh` ne peut pas être exécuté, le code de retour sera identique à celui d'une commande effectuant un *exit(127)*.

66 umask (2)

NOM

umask — Définir le masque de création de fichiers

SYNOPSIS

mode_t umask(mode_t *mask***);**

DESCRIPTION

umask() définit le masque de création de fichiers à la valeur *mask* & 0777 (c'est-à-dire seuls les bits relatifs aux permissions des fichiers de *mask* sont utilisés), et renvoie la valeur précédente du masque.

Ce masque est utilisé par open(2), mkdir(2) et autres pour positionner les permissions d'accès initiales sur les fichiers nouvellement créés. Les bits contenus dans l'umask sont éliminés de l'argument *mode* de l'appel open(2) ou mkdir(2).

VALEUR RENVOYÉE

Cet appel système n'échoue jamais, et la valeur précédente du masque est renvoyée.

67 unlink (2)

NOM

unlink — Détruire un nom et éventuellement le fichier associé

SYNOPSIS

int unlink(const char **pathname***);*

DESCRIPTION

unlink() détruit un nom dans le système de fichiers. Si ce nom était le dernier lien sur un fichier, et si aucun processus n'a ouvert ce fichier, ce dernier est effacé, et l'espace qu'il utilisait est rendu disponible.

Si le nom était le dernier lien sur un fichier, mais qu'un processus conserve encore le fichier ouvert, celui-ci continue d'exister jusqu'à ce que le dernier descripteur le référençant soit fermé.

Si le nom correspond à un lien symbolique, le lien est supprimé.

Si le nom correspond à une socket, une FIFO, ou un périphérique, le nom est supprimé mais les processus qui ont ouvert l'objet peuvent continuer à l'utiliser.

ERREURS

EACCES L'accès en écriture au répertoire contenant *pathname* n'est pas autorisé pour l'UID effectif du processus, ou bien l'un des répertoires de *pathname* n'autorise pas le parcours.

68 wait, waitpid (2)

NOM

wait, waitpid — Attendre la fin d'un processus

SYNOPSIS

pid_t wait(int **status***);**

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

DESCRIPTION

Tous ces appels système attendent qu'un des fils du processus appelant change d'état, et permettent d'obtenir des informations sur le fils en question. Un processus est considéré comme changeant d'état s'il termine, s'il est stoppé par un signal, ou s'il est relancé par un signal. Dans le cas d'un fils qui se termine, l'attendre permet au système de libérer les ressources qui lui étaient allouées ; si le processus n'est pas attendu, il reste en état de « zombie » (voir les NOTES plus bas).

Si un fils a déjà changé d'état, ces appels système retournent immédiatement. Sinon, ils bloquent jusqu'à ce qu'un fils change d'état ou qu'un gestionnaire de signal interrompe l'appel (sauf si les appels système sont relancés automatiquement par l'file `SA_RESTART` de sigaction(2)). Dans la suite de cette page, un fils qui a changé d'état et qui n'a pas été attendu est appelé *prêt* (*waitable*).

L'appel système *wait*() suspend l'exécution du processus appelant jusqu'à ce que l'un de ses enfants se termine. L'appel *wait(&status)* est équivalent à :

```
waitpid(-1, &status, 0);
```

L'appel système *waitpid*() suspend l'exécution du processus appelant jusqu'à ce qu'un fils spécifié par l'argument *pid* change d'état. Par défaut, *waitpid*() n'attend que les fils terminés, mais ce comportement peut être modifié par l'argument *options*, de la façon décrite ci-dessous.

La valeur de *pid* peut être l'une des suivantes :

< -1 Attendre la fin de n'importe quel processus fils appartenant au groupe de processus d'ID `-pid`.

-1 Attendre n'importe lequel des processus fils.

0 Attendre la fin de n'importe quel processus fils du même groupe que l'appelant.
> 0 Attendre la fin du processus numéro *pid*.

La valeur de l'argument file `options` est un *OU* binaire entre les constantes suivantes :

WNOHANG Ne pas bloquer si aucun fils ne s'est terminé.
WUNTRACED Recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.
WCONTINUED **(Depuis Linux 2.6.10)** Renvoyer également si un processus fils stoppé a été relancé par le signal `SIGCONT`.

Si *status* n'est pas NULL, *wait*() et *waitpid*() stockent l'état du fils dans la variable de type *int* pointée. Cet entier peut être évalué avec les macros suivantes (qui prennent l'entier lui-même comme argument, et pas un pointeur vers celui-ci, comme le font *wait*() et *waitpid*() !) :

WIFEXITED(status) Vrai si le fils s'est terminé normalement, c'est-à-dire par un appel à exit(3) ou _exit(2), ou par un *return* depuis main().
WEXITSTATUS(status) Donne le code de retour, consistant en les 8 bits de poids faibles du paramètre *status* fourni à exit(3) ou _exit(2) ou dans le *return* de la routine main(). Cette macro ne peut être évaluée que si *WIFEXITED* est non nul.
WIFSIGNALED(status) Vrai si le fils s'est terminé à cause d'un signal non intercepté.
WTERMSIG(status) Donne le numéro du signal qui a causé la fin du fils. Cette macro ne peut être évaluée que si *WIFSIGNALED* est non nul.
WIFSTOPPED(status) Vrai si le fils est actuellement arrêté. Cela n'est possible que si l'on a effectué l'appel avec l'file `WUNTRACED`.
WSTOPSIG(status) Donne le numéro du signal qui a causé l'arrêt du fils. Cette macro ne peut être évaluée que si *WIFSTOPPED* est non nul.
WIFCONTINUED(status) (Depuis Linux 2.6.10) Vrai si le processus fils a été relancé par `SIGCONT`.

VALEUR RENVOYÉE

wait() : en cas de réussite, l'identifiant du processus fils terminé est renvoyé ; en cas d'erreur, la valeur de retour est -1.

waitpid() : s'il réussit, l'appel renvoie l'identifiant du processus fils dont l'état a changé ; si *WNOHANG* est utilisé et un fils (ou plus) spécifié par *pid* existe, mais n'a toujours pas changé d'état, la valeur de retour est 0. En cas d'erreur, -1 est renvoyé.

ERREURS

ECHILD (pour *wait*()) Le processus appelant n'a pas de fils qui n'ont pas été attendus.
ECHILD (pour *waitpid*()) Le processus indiqué par *pid* n'existe pas, ou n'est pas un fils du processus appelant.
EINTR *WNOHANG* n'est pas indiqué, et un signal à intercepter ou `SIGCHLD` a été reçu ; consultez signal(7).

NOTES

Un fils qui se termine mais n'a pas été attendu devient un « zombie ». Le noyau conserve des informations minimales sur le processus zombie (identifiant, code de retour, informations d'utilisation des ressources) pour permettre au parent de l'attendre plus tard et d'obtenir des informations sur le fils. Tant que le zombie n'est pas effacé du système par une attente, il prendra un

emplacement dans la table des processus du noyau, et si cette table est remplie, il sera impossible de créer de nouveaux processus. Si un processus parent se termine, ses fils zombies sont adoptés par init(8), qui les attend automatiquement pour les supprimer.

69 write (2)

NOM

write — Écrire dans un descripteur de fichier

SYNOPSIS

ssize_t write(int *fd*, **const void** **buf*, **size_t** *count*);

DESCRIPTION

write() lit au maximum *count* octets dans la zone mémoire pointée par *buf*, et les écrit dans le fichier référencé par le descripteur *fd*.

Le nombre d'octets écrits peut être inférieur à *count* par exemple si la place disponible sur le périphérique est insuffisante, ou l'appel est interrompu par un gestionnaire de signal après avoir écrit moins de *count* octets. (Consultez aussi pipe(7).)

Pour un fichier sur lequel lseek(2) est possible (par exemple un fichier ordinaire), l'écriture a lieu à la position actuelle dans le fichier, et elle est déplacée du nombre d'octets effectivement écrits. Si le fichier était ouvert avec *O_APPEND*, la position avant l'écriture est à la fin du fichier. La modification de la position et l'écriture sont effectuées de façon atomique.

VALEUR RENVOYÉE

write() renvoie le nombre d'octets écrits (0 signifiant aucune écriture), ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

ERREURS

EBADF *fd* n'est pas un descripteur de fichier valable, ou n'est pas ouvert en écriture.
ENOSPC Le périphérique correspondant à *fd* n'a plus de place disponible.
EPIPE *fd* est connecté à un tube (pipe) ou une socket dont l'autre extrémité est fermée. Quand ceci se produit, le processus écrivain reçoit un signal `SIGPIPE`. (Ainsi la valeur de retour de write n'est vue que si le programme intercepte, bloque ou ignore ce signal.

NOTES

Si un *write*() est interrompu par un gestionnaire de signaux avant d'avoir écrit quoi que ce soit, l'appel échoue avec EINTR ; s'il est interrompu après avoir écrit au moins un octet, l'appel réussit et renvoie le nombre d'octets écrits.

70 fifo (7)

NOM

fifo — Fichier spécial file FIFO, tube nommé

DESCRIPTION

Un fichier spécial de file FIFO (First In, First Out) est l'équivalent d'un tube (pipeline), sauf qu'il est accessible en tant que partie du système de fichiers. Il peut être ouvert par plusieurs processus, tant en lecture qu'en écriture. Lorsque des processus échangent des données par le biais d'une file FIFO, le noyau transfère les informations de manière interne, sans passer par une écriture réelle dans le système de fichiers. Ainsi, le fichier spécial FIFO n'a pas de véritable contenu ; c'est essentiellement un point de référence pour que les processus puissent accéder au tube en employant un nom dans le système de fichiers.

Le noyau assimile exactement un tube à chaque fichier spécial FIFO ouvert par au moins un processus. La file FIFO doit être ouverte aux deux extrémités (lecture et écriture) avant que des données puissent y transiter. Normalement, l'ouverture d'une file FIFO est bloquante jusqu'à ce que l'autre côté soit aussi ouvert.

Un processus peut ouvrir une FIFO en mode non bloquant. Dans ce cas, l'ouverture en lecture seule réussira même si personne n'a encore ouvert le côté écriture. L'ouverture en écriture seule échouera avec l'erreur ENXIO (aucun périphérique ou adresse) si l'autre extrémité n'a pas encore été ouverte.

Sous Linux, l'ouverture d'une file FIFO en lecture et écriture réussira aussi bien en mode bloquant que non bloquant. POSIX ne précise pas ce comportement. Ceci peut servir à ouvrir une FIFO en écriture, même si aucun lecteur n'est prêt. Un processus qui utilise les deux côtés d'une FIFO pour communiquer avec lui-même doit être très prudent pour éviter les situations de blocage.

NOTES

Quand un processus essaye d'écrire dans une FIFO qui n'a pas été ouverte en lecture de l'autre côté, le processus reçoit un signal `SIGPIPE`.

71 hier (7)

NOM

hier — Description de la hiérarchie du système de fichiers

DESCRIPTION

Un système Linux typique contient, entre autres, les répertoires suivants :

`/` Le répertoire racine (root). Le point de départ de toute l'arborescence.

`/bin` Ce répertoire contient les programmes exécutables nécessaires en mode mono-utilisateur pour démarrer ou réparer le système.

/boot Contient les fichiers statiques utilisés par le chargeur du système.

/dev Fichiers spéciaux ou fichiers se rapportant à des périphériques physiques. Consultez mknod(1).

/etc Contient les fichiers de configuration spécifiques à la machine.

/home Sur les machines offrant des répertoires personnels pour les utilisateurs, ils sont généralement placés sous ce répertoire, directement ou avec une arborescence définie par l'administration locale.

/lib Ce répertoire doit contenir les bibliothèques partagées nécessaires pour démarrer le système et utiliser les commandes de la partition racine.

/media Contient des points de montage pour les périphériques amovibles comme les CD, les DVD, ou les clés USB.

/proc est le point de montage pour le système de fichiers *proc*, qui fournit des informations sur les processus en cours et sur le noyau. Ce pseudosystème de fichiers est décrit dans proc(5).

/root Ce répertoire est habituellement celui personnel du superutilisateur.

/sbin Comme /bin, ce répertoire contient les commandes nécessaires au démarrage du système, mais qui ne sont pas exécutées par des utilisateurs normaux.

/tmp Ce répertoire sert à contenir des fichiers temporaires que l'on peut détruire régulièrement, par un script périodique, ou au démarrage du système.

/usr Ce répertoire est généralement monté depuis une partition séparée. Il ne devrait contenir que des données partageables, en lecture seule, afin d'être monté par plusieurs machines utilisant Linux.

/usr/bin Il s'agit du répertoire principal pour les programmes exécutables. La plupart des programmes nécessaires aux utilisateurs, et pas indispensables pour démarrer ou réparer le système sont placés ici, à l'exception des programmes installés uniquement sur cette machine.

/usr/include Fichiers d'en-tête pour le compilateur C.

/usr/share/doc Documentation à propos des programmes installés.

/var Ce répertoire contient des fichiers qui peuvent changer régulièrement comme les fichiers des files d'attente, ou les fichiers de journalisation.

BOGUES

Cette liste n'est pas exhaustive, certains systèmes peuvent être configurés différemment.

72 pipe (7)

NOM

pipe — Panorama des tubes et des FIFO

DESCRIPTION

Les tubes et les FIFO (ou tubes nommés) fournissent un canal de communication interprocessus unidirectionnel. Un tube a une *entrée* et une *sortie*. Les données écrites à l'entrée du tube peuvent être lues à sa sortie.

Un tube est créé avec l'appel système pipe(2), qui crée un nouveau tube et renvoie deux descripteurs de fichier, l'un correspondant à l'entrée du tube, et l'autre à la sortie. Les tubes peuvent être utilisés pour créer un canal de communication entre des processus liés ; consultez pipe(2) pour un exemple.

Un FIFO (abréviation de First In First Out) a un nom sur le système de fichiers (créé avec mkfifo(3)), et est ouvert avec open(2). Tout processus peut ouvrir un FIFO, si les permissions du fichier l'autorisent. La sortie est ouverte avec l'option `O_RDONLY` ; l'entrée est ouverte avec l'option `O_WRONLY`. Consultez fifo(7) pour plus de détails. *Note* : même si les FIFO ont un nom sur le système de fichiers, les entrées/sorties sur un FIFO n'impliquent pas d'opérations sur le périphérique sous-jacent (s'il y en a un).

E/S sur les tubes et les FIFO

La seule différence entre les tubes et les FIFO est la manière dont ils sont créés et ouverts. Une fois ces tâches accomplies, les E/S sur les tubes et les FIFO ont strictement les mêmes sémantiques.

Si un processus essaie de lire dans un tube vide, read(2) bloquera jusqu'à ce que des données soient disponibles. Si un processus essaie d'écrire dans un tube plein (voir ci-dessous), write(2) bloque jusqu'à ce que suffisamment de données aient été lues dans le tube avant de permettre la réussite de l'écriture. Des E/S non bloquantes sont possibles en utilisant l'opération *F_SETFL* de fcntl(2) pour activer l'attribut *O_NONBLOCK*.

Le canal de communication fourni par un tube est un *flot d'octets* : il n'y a pas de notion de limite entre messages.

Si tous les descripteurs de fichier correspondant à l'entrée d'un tube sont fermés, une tentative de lecture sur le tube renverra une condition de fin de fichier (read(2) renverra 0). Si tous les descripteurs de fichier correspondant à la sortie d'un tube sont fermés, une tentative d'écriture provoquera l'envoi du signal `SIGPIPE` au processus appelant. Si le processus appelant ignore ce signal, write(2) échoue avec l'erreur EPIPE. Une application utilisant pipe(2) et fork(2) doit utiliser des appels à close(2) afin de fermer les descripteurs de fichier superflus ; ceci permet d'assurer que la condition de fin de ficher et `SIGPIPE`/EPIPE sont renvoyés correctement.

Il n'est pas possible d'invoquer lseek(2) sur un tube.

Capacité d'un tube

Un tube a une capacité limitée. Si le tube est plein, un write(2) bloquera ou échouera, selon que l'attribut *O_NONBLOCK* est activé ou non (voir ci-dessous). Différentes implémentations ont différentes limites de capacité des tubes. Les applications ne doivent pas dépendre d'une capacité particulière, mais être conçues pour qu'un processus lecteur lise les données dès qu'elles sont disponibles, pour qu'un processus écrivain ne soit pas bloqué.

Dans les versions de Linux antérieures à 2.6.11, la capacité d'un tube était la taille d'une page système (p.ex. 4096 octets sur i386). Depuis Linux 2.6.11, la capacité d'un tube est de 65536 octets.

PIPE_BUF

POSIX.1-2001 indique que les écritures de moins que *PIPE_BUF* octets doivent être atomiques : les données sont écrites dans le tube de façon contiguë. Les écritures de plus que *PIPE_BUF* peuvent ne pas être atomiques : le noyau peut entrelacer les données avec des données écrites par d'autres processus. POSIX.1-2001 demande que *PIPE_BUF* soit au moins 512 octets. (Sous Linux, *PIPE_BUF* vaut 4096 octets.) La sémantique précise dépend de l'attribut non-bloquant du descripteur de fichier (*O_NONBLOCK*), du nombre d'écrivains dans le tube, et de *n*, le nombre d'octets à écrire :

O_NONBLOCK* désactivé, *n* <= *PIPE_BUF Les *n* octets sont écrits de manière atomique ; write(2) peut bloquer s'il n'y a pas de place pour écrire *n* octets immédiatement.

O_NONBLOCK* activé, *n* <= *PIPE_BUF S'il y a la place d'écrire *n* octets dans le tube, write(2) réussit immédiatement, en écrivant les *n* octets ; sinon, write(2) échoue, et définit `errno` à EAGAIN.

O_NONBLOCK* désactivé, *n* > *PIPE_BUF L'écriture est non atomique : les données fournies à write(2) peuvent être entrelacées avec des écritures d'autres processus ; l'écriture bloque jusqu'à ce que *n* octets aient été écrits.

O_NONBLOCK* activé, *n* > *PIPE_BUF Si le tube est plein, write(2) échoue, en plaçant `errno` à EAGAIN. Sinon, entre 1 et *n* octets peuvent être écrits (une « écriture partielle » peut se produire ; l'appelant doit vérifier la valeur de retour de write(2) pour voir combien d'octets ont réellement été écrits), et ces octets peuvent être entrelacés avec des écritures d'autres processus.

Attributs d'état de fichier ouvert

Les seuls attributs d'état de fichier ouvert qui peuvent s'appliquer aux tubes et aux FIFO sont *O_NONBLOCK* et *O_ASYNC*.

Activer l'attribut *O_ASYNC* à la sortie d'un tube provoque l'envoi d'un signal (`SIGIO` par défaut) lorsque de nouvelles données sont disponibles sur le tube (consultez fcntl(2) pour les détails). Sous Linux, *O_ASYNC* n'est possible sur les tubes et les FIFO que depuis le noyau 2.6.

73 pthreads (7)

NOM

pthreads — Threads POSIX

DESCRIPTION

POSIX.1 décrit une série d'interfaces (fonctions et fichiers d'en-têtes) pour la programmation multithread, couramment appelée threads POSIX, ou pthreads. Un unique processus peut contenir plusieurs threads, qui exécutent tous le même programme. Ces threads partagent la même mémoire globale (segments de données et tas), mais chaque thread a sa propre pile (variables automatiques).

POSIX.1 requiert aussi que les threads partagent une série d'autres attributs (ces attributs sont par processus, plutôt que par thread) :

— identifiant de processus (PID)
— identifiant de processus père (PPID)
— identifiant de groupe de processus (PGID) et identifiant de session (SID)
— identifiants d'utilisateur et de groupe
— descripteurs de fichier ouverts
— verrouillages d'enregistrements (consultez fcntl(2))
— gestion de signaux
— masque de création de fichier (umask(2))
— répertoire de travail (chdir(2)) et répertoire racine (chroot(2))
— temporisations d'intervalle (setitimer(2)) et temporisations POSIX (timer_create(2))
— valeur de politesse (setpriority(2))

— limites de ressources (setrlimit(2))
— mesures de consommation de temps CPU (times(2)) et de ressources (getrusage(2))

En plus de la pile, POSIX.1 indique que plusieurs autres attributs sont distincts pour chaque thread, dont les suivants :

— identifiant de thread (le type de donnée pthread_t)
— masque de signaux (pthread_sigmask(3))
— la variable `errno`

Valeurs de retour des fonctions pthreads

La plupart des fonctions pthreads renvoient 0 en cas de succès et un numéro d'erreur en cas d'échec. Notez que les fonctions pthreads ne positionnent pas `errno`. Pour chacune des fonctions pthreads qui peuvent produire une erreur, POSIX.1-2001 spécifie que la fonction ne peut pas échouer avec l'erreur EINTR.

Identifiants de thread

Chacun des threads d'un processus a un unique identifiant de thread (stocké dans le type pthread_t). Cet identifiant est renvoyé à l'appelant de pthread_create(3) et un thread peut obtenir son propre identifiant de thread en utilisant pthread_self(3). Les identifiants de thread n'ont la garantie d'être uniques qu'à l'intérieur d'un processus. Un identifiant de thread peut être réutilisé après qu'un thread qui s'est terminé a été rejoint ou qu'un thread détaché se soit terminé. Pour toutes les fonctions qui acceptent un identifiant de thread en paramètre, cet identifiant de thread se réfère par définition à un thread du même processus que l'appelant.

Fonctions sûres du point de vue des threads

Une fonction sûre du point de vue des threads est une fonction qui peut être appelée en toute sûreté (c'est-à-dire qu'elle renverra le même résultat d'où qu'elle soit appelée) par plusieurs threads en même temps.

POSIX.1-2001 et POSIX.1-2008 exigent que toutes les fonctions indiquées dans la norme soient sûres du point de vue des threads, excepté les fonctions suivantes (voir la page de manuel complète).

Fonctions pour annulations sûres asynchrones

Une fonction pour annulations sûres asynchrones peut être appelée sans risque dans une application où l'état d'annulation est activé (consultez pthread_setcancelstate(3)).

POSIX.1-2001 et POSIX.1-2008 exigent que seules les fonctions suivantes soient pour annulations sûres asynchrones : pthread_cancel(), pthread_setcancelstate(), pthread_setcanceltype().

Points d'annulation

POSIX.1 spécifie que certaines fonctions doivent, et certaines autres fonctions peuvent, être des points d'annulation. Si un thread est annulable, que son type d'annulation est retardé (« deferred ») et qu'une demande d'annulation est en cours pour ce thread, alors le thread est annulé quand il appelle une fonction qui est un point d'annulation.

74 signal (7)

NOM

signal — Panorama des signaux

DESCRIPTION

Linux prend en charge à la fois les signaux POSIX classiques (« signaux standards ») et les signaux POSIX temps-réel.

Dispositions de signaux

Chaque signal a une *disposition* courante, qui détermine le comportement du processus lorsqu'il reçoit ce signal.

Les symboles de la colonne « Action » indiquent l'action par défaut pour chaque signal, avec la signification suivante :

Term Par défaut, terminer le processus.
Ign Par défaut, ignorer le signal.
Core Par défaut, créer un fichier core et terminer le processus (consultez core(5)).
Stop Par défaut, arrêter le processus.
Cont Par défaut, continuer le processus s'il est actuellement arrêté.

Un processus peut changer la disposition d'un signal avec sigaction(2). Avec cet appel système, un processus peut choisir de se comporter de l'une des façons suivantes lorsqu'il reçoit ce signal : effectuer l'action par défaut, ignorer le signal, ou rattraper le signal avec un *gestionnaire de signal*, c'est-à-dire une fonction définie par le programme, qui est invoquée automatiquement lorsque le signal est distribué.

Un fils créé par fork(2) hérite d'une copie des dispositions de signaux de son père. Lors d'un execve(2), les dispositions des signaux pris en charge sont remises aux valeurs par défaut ; les dispositions des signaux ignorés ne sont pas modifiées.

Envoyer un signal

Les appels système suivants permettent à l'appelant d'envoyer un signal :

raise(3) Envoie un signal au thread appelant.
kill(2) Envoie un signal au processus indiqué, à tous les membres du groupe de processus indiqué, ou à tous les processus du système.

Attente de la capture d'un signal

Les appels système suivants suspendent l'exécution du processus ou du thread appelant jusqu'à ce qu'un signal soit attrapé (ou qu'un signal non pris en charge termine le processus) :

pause(2) Suspend l'exécution jusqu'à ce que n'importe quel signal soit reçu.
sigsuspend(2) Change temporairement le masque de signaux (voir ci-dessous) et suspend l'exécution jusqu'à ce qu'un des signaux masqué soit reçu.

Masque de signaux et signaux en attente

Un signal peut être *bloqué*, ce qui signifie qu'il ne sera pas reçu par le processus avant d'être débloqué. Entre sa création et sa réception, le signal est dit *en attente*. Si plusieurs instances d'un signal arrivent alors qu'il est bloqué, une seule instance sera mémorisée.

Chaque thread d'un processus a un *masque de signaux* indépendant, qui indique l'ensemble des signaux bloqués par le thread. Dans une application traditionnelle, à un seul thread, sigprocmask(2) peut être utilisée pour modifier le masque de signaux.

Un processus fils créé avec fork(2) hérite d'une copie du masque de signaux de son père ; le masque de signaux est conservé au travers d'un execve(2).

Un signal envoyé à un processus peut être traité par n'importe lequel des threads qui ne le bloquent pas. Si plus d'un thread ne bloque pas le signal, le noyau choisit l'un de ces threads arbitrairement, et lui envoie le signal.

Un fils créé avec fork(2) démarre avec un ensemble de signaux en attente vide ; l'ensemble de signaux en attente est conservé au travers d'un execve(2).

Signaux standards

Signal	Action	Commentaire
SIGHUP	Term	Déconnexion détectée sur le terminal de contrôle ou mort du processus de contrôle.
SIGINT	Term	Interruption depuis le clavier.
SIGQUIT	Core	Demande « Quitter » depuis le clavier.
SIGILL	Core	Instruction illégale.
SIGABRT	Core	Signal d'arrêt depuis abort(3).
SIGFPE	Core	Erreur mathématique virgule flottante.
SIGKILL	Term	Signal « KILL ».
SIGSEGV	Core	Référence mémoire invalide.
SIGPIPE	Term	Écriture dans un tube sans lecteur.
SIGALRM	Term	Temporisation alarm(2) écoulée.
SIGTERM	Term	Signal de fin.
SIGUSR1	Term	Signal utilisateur 1.
SIGUSR2	Term	Signal utilisateur 2.
SIGCHLD	Ign	Fils arrêté ou terminé.
SIGCONT	Cont	Continuer si arrêté.
SIGSTOP	Stop	Arrêt du processus.
SIGTSTP	Stop	Stop invoqué depuis le terminal.
SIGTTIN	Stop	Lecture sur le terminal en arrière-plan.
SIGTTOU	Stop	Écriture dans le terminal en arrière-plan.

Les signaux `SIGKILL` et `SIGSTOP` ne peuvent être ni capturés ni bloqués ni ignorés.

Interruption des appels système et des fonctions de bibliothèque par des gestionnaires de signal

Si un gestionnaire de signal est invoqué pendant qu'un appel système ou une fonction de bibliothèque est bloqué, alors :

— soit l'appel est automatiquement redémarré après le retour du gestionnaire de signal ;
— soit l'appel échoue avec l'erreur EINTR.

Lequel de ces deux comportements se produira dépend de l'interface et de si le gestionnaire de signal a été mis en place avec l'attribut *SA_RESTART* (consultez sigaction(2)).

La fonction sleep(3) n'est jamais relancée si elle est interrompue par un gestionnaire, mais elle renvoie un code de retour de succès, le nombre de secondes restantes pour le sommeil.