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 chmod (2)

NOM

chmod — Modifier les permissions d'accès à un fichier

SYNOPSIS

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

DESCRIPTION

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.

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

4 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.

5 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.

6 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.

7 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.

8 memcmp (3)

NOM

memcmp — Comparaison de zones mémoire

SYNOPSIS

int memcmp(const void **s1*, **const void** **s2*, **size_t** *n*);

DESCRIPTION

La fonction *memcmp*() compare les *n* premiers octets (chacun interprété comme *unsigned char*) des zones mémoire *s1* et *s2*.

VALEUR RENVOYÉE

La fonction *memcmp*() renvoie un entier négatif, nul ou positif si les *n* premiers octets de *s1* sont respectivement inférieurs, égaux ou supérieurs aux *n* premiers octets de *s2*.

Lorsque le code retour est différent de zéro, son signe dépend de celui de la différence entre les deux premiers octets (interprétés comme *unsigned char*) qui diffèrent dans *s1* et *s2*.

9 memcpy (3)

NOM

memcpy — Copier une zone mémoire

SYNOPSIS

void *memcpy(void **dest*, **const void** **src*, **size_t** *n*);

DESCRIPTION

La fonction *memcpy*() copie *n* octets depuis la zone mémoire *src* vers la zone mémoire *dest*. Les deux zones ne doivent pas se chevaucher. Si c'est le cas, utilisez plutôt *memmove*(3).

VALEUR RENVOYÉE

La fonction *memcpy*() renvoie un pointeur sur *dest*.

10 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*.

11 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.

12 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.

13 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.

14 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.

15 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.

16 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.

17 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.

18 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.

19 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.

20 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).

21 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.

22 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.

23 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*().

24 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).

25 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.

26 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.

27 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.

28 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).

29 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.

30 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.

31 stat, lstat (2)

NOM

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

SYNOPSIS

int stat(const char **pathname*, **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 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.

Les deux 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).

32 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*.

33 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*.

34 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.

35 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.

36 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.

37 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.

38 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.