Cours "Programmation système"

R3.05

2ème année BUT

Département d'informatique

les processus

Table des matières

Partie 1 :

I/ Introduction : Rappels et vocabulaire

Qu'est-ce que le multitâche ?

Un système d'exploitation est dit multitâche (*multitasking*) **s'il est capable d'exécuter de façon apparemment simultanée plusieurs programmes informatiques.**

La simultanéité apparente n'est que le résultat de l'alternance d'exécution de programmes présents en mémoire (temps partagé ou multiplexage). Le passage de l'exécution d'un programme à celle d'un autre est appelée ***Commutation de contexte.***

Deux types de commutations existent selon leurs initiations :

- ✓ Par les programmes eux-mêmes : ***multitâche coopératif***
- ✓ Par le système d'exploitation : ***multitâche préemptif***

Le multitâche coopératif n'est plus utilisé. Unix (Linux), Windows et MacOS sont des systèmes basés sur le multitâche préemptif.

A quoi sert le multitâche ?

À paralléliser les traitements par l'exécution simultanée de programmes informatiques. Par exemple pour permettre à plusieurs utilisateurs de travailler sur la même machine, ou utiliser un traitement de texte tout en surfant sur le web, ou encore transférer plusieurs fichiers en même temps. On peut également améliorer la conception en écrivant plusieurs programmes simples et les faire coopérer pour exécuter les tâches nécessaires.

Comment marche le multitâche ?

Le multitâche préemptif est assuré par **l'ordonnanceur** *(sheduler)* sur un système multitâche. La **préemption** étant la capacité d'un système d'exploitation à suspendre l'exécution d'un programme au profit de celle d'un autre. Elle se définit comme la réquisition du processeur pour exécuter une seule tâche pendant un temps déterminé.

L'ordonnanceur désigne le composant du noyau du système d'exploitation dont le rôle est de distribuer le temps du processeur entre les différents processus.

Le quantum est la quantité de temps définie attribuée par l'ordonnanceur : un processus ne peut donc pas s'attribuer un temps non défini pour s'exécuter dans le processeur.

L'ordonnancement statique à base de priorités permet à un processus de remplacer un autre processus de priorité plus élevée s'il est à l'état prêt.

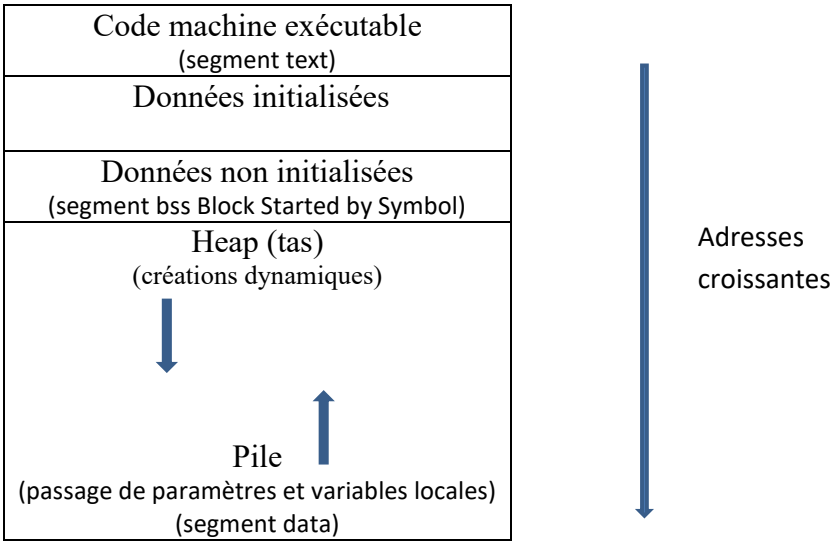
II/Les processus

II-1/ Introduction

Linux est un système d'exploitation multitâches ce qui signifie que plusieurs programmes peuvent s'exécuter en même temps sur la même machine. On appelle processus une instance de programme. Cad **c'est un programme en cours d'exécution** par un système d'exploitation.

Un processus s'exécute dans son contexte, quand il y a changement de processus courant il y a **commutation ou changement de contexte**. Le noyau s'exécute alors dans le nouveau contexte.

Un processus ne se résume pas au programme qu'il exécute. Un processus comporte du code machine exécutable, une zone mémoire pour les données allouées par le processus et une pile pour les variables locales et la gestion des fonctions (passage de paramètres) par thread. Si le programme est multithread, il y a plusieurs piles. Tout cet ensemble est accessible au travers d'un ensemble d'adresses que l'on appelle espace d'adressage virtuel.



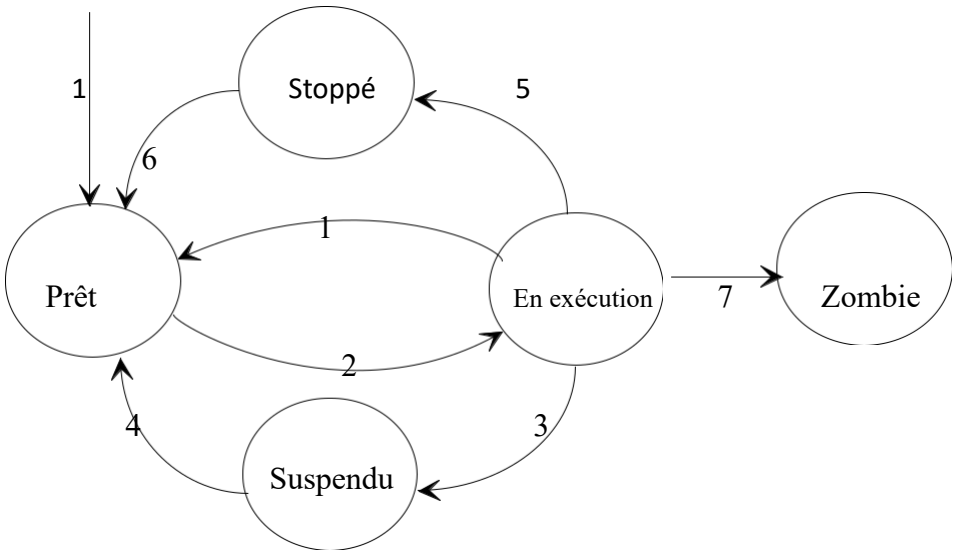
Exemple :

```
  int Globale1; /*donnée non initialisée */
  int Globale2 = 1234; /*donnée initialisée */
  static int Globale3; /*donnée non initialisée */
/* … */
int ma_fonction(paramètres)
{
   int Locale1;     /*pile d'exécution */
   static int Locale2 = 4321;/*donnée initialisée (car static) */
   static int lOcale3; /*donnée non initialisée (car static)*/
   /*… */
}
```

II-2/ État d'un processus

Au cours de son exécution, un processus change d'état. L'état d'un processus est défini par son activité courante. Les différents états d'un processus sont les suivants :

- ✓ En exécution : le processus est exécuté par le processeur,
- ✓ Prêt : le processus est éligible, il attend que le système lui donne processeur,
- ✓ Suspendu : le processus est en attente d'une ressource,
- ✓ Stoppé : le processus a été suspendu par une intervention extérieure,
- ✓ Zombie : le processus a terminé son exécution mais est toujours référencé dans le système.



II-3/ Caractéristiques d'un processus

Pour assurer le contrôle des différents processus existants, le système associe à chaque processus un ensemble d'informations permettant son identification et donnant ses principales caractéristiques. Cet ensemble d'informations s'appelle : le bloc de contrôle PCB.

Le contexte d'un processus *(Process Control Block)* est l'ensemble des informations :

- Son état
- Son identificateur et celui de son père
- L'identité des utilisateurs (réel et effectif) des groupes (réel et effectif)
- Des informations utilisées par le noyau pour procéder à l'ordonnancement (priorité, l'événement attendu …),
- Des informations concernant l'espace d'adressage
- Des informations concernant les entrées/sorties effectuées par le processus (descripteur de fichiers ouverts, répertoire courant …)
- Son entrée dans la table des processus
- Ses données privées

- Les piles *user et système*
- Les zones de code et de données

Lorsqu'un processus est terminé, il passe dans l'état zombie. Dans cet état, le bloc de contrôle ne contient plus qu'un minimum d'informations :

- ✓ Son code de retour,
- ✓ Ses temps d'exécutions (mode utilisateur et mode noyau),
- ✓ Son identité : PID et celle de son père : PPID.

Puisque les processus père et fils se déroulent en concurrence, le code de retour du fils doit rester afin que le père puisse le lire. C'est la raison d'être de cet état zombie.

III/Création d'un processus

III-1/Création sous un Shell

Lorsqu'on entre une commande dans un shell, le shell lance un processus pour l'exécuter. Le shell attend ensuite la fin de ce processus, puis demande la commande suivante. Chaque processus, d'identificateur PID, a donc un processus père, d'identificateur PPID. Un processus n'a qu'un seul père mais peut lancer l'exécution de plusieurs processus fils. Lors de l'initialisation du système, un premier processus, nommée ***init***, est créé (avec 1 comme PID) et il est l'ancêtre de tous les autres processus. Le mécanisme de création d'un processus est toujours le même :

- ✓ Le shell se duplique (on a donc deux processus shell identiques),
- ✓ Le shell père se met en attente de la fin du fils,
- ✓ Le shell fils remplace son code exécutable par celui de la commande passée,
- ✓ La commande s'exécute et logiquement se termine, le processus fils disparaît,
- ✓ Le père est alors réveillé et affiche à nouveau le prompt.

III-2/Création en langage C

Le mécanisme de création sera évidemment le même que sous le shell. Il faut donc des fonctions pour dupliquer le processus et attendre la fin du fils. La fonction ***pid_t fork(void)*** permet (quand cela est possible) de dupliquer le processus. Les deux processus exécutent le même code, mais la valeur de retour de la fonction ***fork*** n'est pas la même. Pour le processus fils, cette valeur vaut 0 et pour le processus père c'est la valeur du PID du fils qui est retournée. En testant cette valeur de retour, le père et le fils n'exécuteront pas la même partie de code.

Exemple de code C créant un processus :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main(int argc, char* argv[])
{
   pid_t pid;
   if ((pid = fork()) > 0)
   {
      /* process père */
      printf("Je suis le pere : %d (de pere %d)\n", getpid(), getppid());
   }
   else
   {
      if (pid == 0)
      {
         /* processus fils */
         printf("\t\t\tJe suis le fils : %d (de pere %d)\n", getpid(), getppid());
         return EXIT_SUCCESS;
      }
      perror("fork");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

Autre exemple de code C créant un processus :

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char* argv[])
{
   pid_t pid;
   switch (pid = fork())
   {
      case -1: printf("Erreur de creation du processus fils\n"); return EXIT_FAILURE;
      case 0:
         /* processus fils */
         printf("\t\t\tJe suis le fils : %d (de pere %d)\n", getpid(), getppid());
         return EXIT_SUCCESS;
      default:
         /* processus pere */
         system("ps l");
         printf("Je suis le pere : %d (de pere %d)\n", getpid(), getppid());
   }
   return EXIT_SUCCESS;
}
```

L'exécution de cet exemple donne l'affichage suivant :

```
Je suis le fils : 20171 (de pere 20170)
F   UID   PID   PPID   PRI  NI  VSZ   RSS  WCHAN  STAT  TTY  TIME  COMMAND
0  501  20038 20037 20  0  5616  3216   wait     Ss     pts/0   0:00   -bash
0  501  20170  20038  20  0 1508  276    wait    S+   pts/0  0:00    ./processus2
1  501  20171  20170  20  0   0       0       exit    Z+   pts/0  0:00    [processus2]
0 501  20172  20170  20   0  2384 800       -       R+   pts/0  0:00     ps l
Je suis le pere : 20170 (de pere 20038)
```

Le processus père exécute la commande système ***ps l*** qui demande des informations sur les processus courants. On peut voir que le processus fils est terminé et qu'il est dans l'état zombie : Z+ dans la colonne STAT (le shell le montre aussi en ajoutant à la fin de la commande.

Il n'occupe plus de mémoire : 0 dans la colonne RSS. On peut aussi vérifier que le processus fils (celui qui est zombie) a pour PID 20171 et comme père le processus de PID 20170 ce qui est confirmé par les affichages du programme. Le PPID du père est le PID du shell. L'exécution de la commande système ps l demande du temps au processus père ce qui permet au fils de se terminer.

Si on fait exécuter cette commande par le fils, le père se terminera avant le fils.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char* argv[])
{
   pid_t pid;
   switch (pid = fork())
   {
      case -1: perror("fork"); return EXIT_FAILURE;
      case 0:
         /* processus fils */
         system("ps l");
         printf("\t\t\t Je suis le fils : %d (de pere %d)\n", getpid(), getppid());
         return EXIT_SUCCESS;
      default:
         /* processus pere */
         printf("Je suis le pere : %d (de pere %d)\n", getpid(), getppid());
   }
   return EXIT_SUCCESS;
}
```

L'exécution de cet exemple donne l'affichage suivant :

```
Je suis le pere : 16465(de pere 16402)
F UID   PID    PPID    PRI NI  VSZ   RSS   WCHAN  STAT  TTY   TIME COMMAND
0 501 16402 16400 20   0  6776 3644   n_tty_  Ss+   pts/2  0:00   - bash
1 501 16466    1      20   0 1620   48      wait      S     pts/2  0:00   ./ processus3
0 501 16468  16466 20  0 4300   772     -          R     pts /2   0:00   ps l
 Je suis le fils : 16466(de pere 1)
```

Le processus père est effectivement achevé : il n'y a pas de trace lors de l'exécution de la commande système ps l. Cependant le processus fils ayant perdu son père est automatiquement adopté par le processus INIT de PID 1.

Afin de protéger la structure arborescente des processus, il est nécessaire que ce processus soit rattaché à un autre processus qui doit évidemment exister quel que soit le contexte. Il est donc "adopté" par le processus d'initialisation (de numéro 1). Cette adoption permettra l'élimination des processus zombies.

III-3/Fonctions permettant de récupérer des informations de processus

Il est possible, grâce à des fonctions, de connaître une partie des informations contenues dans le BCP :

- ✓ Le PID : pid_t getpid(void),
- ✓ Le PPID : pid_t getppid(void),
- ✓ Le propriétaire réel : uid_t getuid(void),
- ✓ Le propriétaire effectif : uid_t geteuid(void),
- ✓ Le groupe propriétaire réel : gid_t getgid(void),
- ✓ Le groupe propriétaire effectif : gid_t getegid(void),
- ✓ Le répertoire de travail : long *getcwd (char *tampon, unsigned long longueur),
- ✓ Les statistiques de durée du processus : clock_t times(struct tms *buf),
- ✓ La priorité du processus : int getpriority(int which, int who),
- ✓ L'utilisation courante des ressources : int getrusage (int who, struct rusage *usage)

IV/ Le recouvrement :

Jusqu'à maintenant, le fils exécutait le code qui avait été écrit. Il pourrait être intéressant qu'il exécute le code d'un autre programme. Il existe pour cela des primitives qui permettent de réaliser la troisième étape de la création d'un processus sous le shell : le shell fils remplace son code exécutable par celui de la commande passée. Ce mécanisme, appelé recouvrement, a pour effet d'écraser l'espace d'adressage du processus fils. Un nouveau programme étant chargé, il s'exécute. Ces primitives font partie de la famille exec :

- ✓ int execve (const char *fichier, char * constargv [],
- ✓ char * constenvp[]),
- ✓ 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[])

Elles se différencient essentiellement par les paramètres transmis mais on peut connaître les paramètres attendus grâce aux lettres ajoutées à exec :

- ✓ **p** indique un nom d'exécutable ce qui suppose qu'il est situé dans un des répertoires de la variable d'environnement PATH, en l'absence de p, le nom de l'exécutable doit être précisé avec son chemin,
- ✓ **v** indique que les paramètres passés seront dans un tableau de pointeur sur des chaînes de caractères (type char *argv[] du main),
- ✓ **l** indique que la fonction admet un nombre variable d'arguments. Ces arguments sont des chaînes de caractères et seront passés en paramètre à l'exécutable. Le dernier argument sera NULL,
- ✓ **e** indique qu'un nouvel environnement est transmis dans un tableau sur des chaînes de caractères : char *env[]. Chaque chaîne de caractère est du type nom=valeur.

IV-1/ Exemple d'utilisation :

Exemple 1 : utilisation de la primitive execl

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char* argv[])
{
	switch (fork())
	{
	case -1: perror("fork"); break;
	case 0: /*	fils	*/
		execl("/bin/tar", "tar", "zcvf", "archive.tar.gz", "/home/user1/Processus/", NULL);
		break;
	default:/*pere*/
		wait(NULL);
	}
	return EXIT_SUCCESS;
}
```

*Le premier paramètre de la primitive **execl** est le chemin complet de l'exécutable, le second paramètre est le nom de la commande. Ensuite on trouve les paramètres passés à cette commande. La fin des paramètres est signalée par un pointeur NULL.*

Exemple 2 : utilisation de la primitive execlp

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char* argv[])
{
	switch (fork())
	{
	case -1: perror("fork"); break;
	case 0: /*	fils	*/
		execlp("tar", "tar", "zcvf", "archive.tar.gz", "/home/user1/Processus", NULL);
		break;
	default: /*pere*/
		wait(NULL);
	}
	return EXIT_SUCCESS;
}
```

*Seul le premier paramètre change par rapport à la primitive **execl**. Le chemin complet n'est pas nécessaire car le programme va utiliser un des champs de la variable d'environnement PATH pour trouver la commande. Si aucun des chemins ne contient la commande, il y aura échec d'exécution.*

Exemple 3 : utilisation de la primitive execv

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char* argv[])
{
	char* new_argv[5];
```

```
        switch (fork())
        {
        case -1: perror("fork"); break;
        case 0: /*          fils        */
                new_argv[0] = "tar";
                new_argv[1] = "zcvf";
                new_argv[2] = "archive.tar.gz";
                new_argv[3] = "/home/user1/Processus";
                new_argv[4] = NULL;
                execv("/bin/tar", new_argv);
                break;
        default:/*pere*/
                wait(NULL);
        }
        return EXIT_SUCCESS;
}
```

Pour cette primitive, il faut passer la commande et ses paramètres dans un tableau. Le premier paramètre est le nom complet de la commande.

Exemple 4 : utilisation de la primitive execvp

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <string.h>

int main(int argc, char* argv[])
{
        char* new_argv[5];
        switch (fork())
        {
        case -1: perror("fork"); break;
        case 0: /*          fils        */
                new_argv[0] = "tar";
                new_argv[1] = "zcvf";
                new_argv[2] = "archive.tar.gz";
                new_argv[3] = "/home/user1/Processus";
                new_argv[4] = NULL;
                execvp("tar", new_argv);
                break;
        default:/*pere*/
                wait(NULL);
        }
        return EXIT_SUCCESS;
}
```

V/Documentation des fonctions :

V-1/fork()

NOM fork - Créer un processus fils (child).

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

DESCRIPTION

fork crée un processus fils qui diffère du processus parent uniquement par ses valeurs PID et PPID et par le fait que toutes les statistiques d'utilisation des ressources sont remises à zéro. Les verrouillages de fichiers, et les signaux en attente ne sont pas hérités.

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.

VALEUR RENVOYÉE

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

ERREURS

ENOMEM Impossible d'allouer assez de mémoire pour copier la table des pages du père et d'allouer une structure de tâche pour le fils.

EAGAIN Impossible de trouver un emplacement vide dans la table des processus.

V-2/getpid() – getppid()

NOM getpid, getppid - Obtenir l'identifiant d'un processus.

SYNOPSIS (#include <sys/types.h> #include <unistd.h>)

```
pid_t getpid(void);

pid_t getppid(void);
```

DESCRIPTION

getpid retourne l'ID du processus actif. Ceci est souvent utilisé par des routines qui créent des fichiers temporaires uniques. getppid retourne le PID du processus parent de celui en cours. Ceci est régulièrement utilise après un fork (2) pour établir la communication entre les deux processus issus du même programme.

ERREURS

Ces fonctions réussissent toujours.

V-3/ wait() – waitpid()

```
NOM wait, waitpid - Attendre la fin d'un processus.

SYNOPSIS (#include <sys/types.h> #include <sys/wait.h>)

 pid_t wait(int *status)

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

DESCRIPTION

 La fonction wait suspend l'exécution du processus courant jusqu'à ce qu'un
enfant se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si un
processus fils s'est déjà terminé au moment de l'appel (il est devenu
"zombie"), la fonction revient immédiatement. Toutes les ressources utilisées
par le fils sont libérées.

 La fonction waitpid suspend l'exécution du processus courant jusqu'à ce que
le processus fils numéro pid se termine, ou jusqu'à ce qu'un signal à
intercepter arrive. Si le fils mentionné par pid s'est déjà terminé au moment
de l'appel (il est devenu "zombie"), la fonction revient immédiatement.
Toutes les ressources utilisées par le fils sont libérées.

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

 <-1 attendre la fin de n'importe quel processus fils appartenant à un
 groupe de processus d'ID pid.


 -1 attendre la fin de n'importe quel fils. C'est le même comportement que
wait.

 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 option 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.

 Si status est non NULL, wait et waitpid y stockent l'information sur la
terminaison du fils.

 Cette information peut être analysée avec les macros suivantes, qui
réclament en argument le buffer status (un int, et non pas un pointeur sur
ce buffer).

 WIFEXITED(status) non nul si le fils s'est terminé normalement

 WEXITSTATUS(status) donne le code de retour tel qu'il a été mentionné dans
l'appel exit() ou dans le return de la routine main. Cette macro ne peut être
évaluée que si WIFEXITED est non nul.

 WIFSIGNALED(status) indique que le fils s'est terminé à cause d'un signal
non intercepté.
```

WTERMSIG(status) donne le nombre de signaux qui ont causé la fin du fils. Cette macro ne peut être évaluée que si WIFSIGNALED est non nul.

WIFSTOPPED(status) indique que le fils est actuellement arrêté. Cette macro n'a de sens que si l'on a effectué l'appel avec l'option WUNTRACED.

WSTOPSIG(status) donne le nombre de signaux qui ont causé l'arrêt du fils.

Cette macro ne peut être évaluée que si WIFSTOPPED est non nul.

VALEUR RENVOYÉE

En cas de réussite, le PID du fils qui s'est terminé est renvoyé, en cas d'échec -1 est renvoyé et errno contient le code d'erreur.

ERREURS

ECHILD Le processus indiqué par pid n'existe pas, ou n'est pas un fils du processus appelant. (Ceci peut arriver pour son propre fils si l'action de SIGCHLD est placé sur SIG IGN, voir également le passage de la section NOTES concernant les threads).

EINVAL L'argument options est invalide.

ERESTARTSYS WNOHANG n'est pas indiqué, et un signal à intercepter ou SIGCHLD a été reçu. Cette erreur est renvoyée par l'appel système. La routine de bibliothèque d'interface n'est pas autorisée à renvoyer ERESTARTSYS, mais renverra EINTR.

NOTES

Les spécifications Single Unix décrivent un attribut SA NOCLDWAIT (absent sous Linux) permettant (lorsqu'il est positionné) aux processus fils se terminant de ne pas devenir zombies, comme quand l'action pour SIGCHLD est fixée à SIG IGN (ce qui toutefois n'est pas autorisé par POSIX). Un appel à wait() ou waitpid() bloquera jusqu'à ce qu'un fils se termine, puis échouera avec errno contenant ECHILD.

Dans le noyau Linux, un thread ordonnancé par le noyau n'est pas différent d'un simple processus. En fait, un thread est juste un processus qui est créé à l'aide de la routine - spécifique Linux - clone(2). Les routines portables, comme pthread_create(3) sont implémentées en appelant clone(2). Ainsi, si deux threads A et B sont frères, alors le thread A ne peut pas se mettre en attente sur un processus créé par le thread B ou ses descendants, car un oncle ne peut pas attendre un neveu. Sur d'autres systèmes Unix, où de multiples threads sont implémentés au sein d'un unique processus, le thread A peut naturellement attendre la fin de n'importe quel processus lancé par un thread frère B. Pour qu'un programme qui se base sur ce comportement fonctionne sous Linux, il faudra en modifier le code.

V-4/getuid() -geteuid()

NOM getuid, geteuid - Obtenir l'identifiant de l'utilisateur.

SYNOPSIS (#include <unistd.h> #include <sys/types.h>)

uid_t getuid(void);

```
 uid_t geteuid(void);

DESCRIPTION

 getuid retourne l'UID réel du processus en cours.

 geteuid retourne l'UID effectif du processus en cours.

 L'UID réel correspond à celui du processus appelant, l'UID effectif dépend
de l'état du bit Set-UID du fichier exécuté.

ERREURS

 Ces fonctions réussissent toujours.
```

V-5/getgid() – getegid()

```
NOM getgid, getegid - Obtenir l'identifiant du groupe.

SYNOPSIS (#include <unistd.h> #include <sys/types.h>)


 gid_t getgid(void);

 gid_t getegid(void);

DESCRIPTION

 getgid retourne le GID réel du processus en cours.

 getegid retourne le GID effectif du processus en cours.

 Le GID réel correspond au GID du processus appelant, le GID effectif est
dépendant de l'état du bit Set-GID du fichier exécuté.

ERREURS

 Ces fonctions réussissent toujours.
```

V-6/getcwd()

```
NOM getcwd - Obtenir le répertoire courant.

SYNOPSIS (#include <linux/unistd.h>)

 long *getcwd (char *tampon, unsigned long longueur);

DESCRIPTION

 La fonction getcwd() copie le chemin d'accès absolu du répertoire de travail
courant dans la chaîne pointée par tampon, qui est de longueur longueur.

 Si le chemin du répertoire en cours nécessite un tampon plus long que
longueur octets, la fonction renvoie NULL, et errno contient le code d'erreur
ERANGE. Une application doit détecter cette erreur et allouer un tampon plus
grand si besoin est.

VALEUR RENVOYÉE
```

```
 La longueur du chemin en cas de réussite, une valeur négative en cas d'échec
(par exemple si le répertoire en cours n'est pas lisible).

NOTES

 La version de la bibliothèque libc (getcwd(3)) renvoie un pointeur sur un
tableau de caractères. L'appel système du noyau renvoie la longueur du tampon
rempli (incluant le '\0' final), ou un nombre négatif pour signaler une
erreur. La version de la libc devrait donc être :

 char *getcwd(char * buf, longueur_t longueur)
 {
 int retval = sys getcwd(buf, longueur);
 if (retval >= 0)
 return buf;

 errno = -retval;
 return NULL;
 }

 Cette version serait plus rapide que la version actuelle getcwd(3) qui
scanne récursivement '..' jusqu'à atteindre / alors que getcwd(2) utilise
les possibilités du dentries cache.

ERREURS

 EFAULT Erreur lors de la copie de la page tampon allouée dans le noyau vers
le tampon alloué par la libc dans l'espace utilisateur.

 ENOENT répertoire inexistant.

 ENOMEM pas assez de mémoire pour allouer une page intermédiaire.

 ERANGE longueur n'est pas assez grande pour stocker le chemin absolu.
```

V-7/ chdir()

```
NOM chdir, fchdir - Changer le répertoire courant.



SYNOPSIS (#include <unistd.h>)

 int chdir(const char *path);

 int fchdir(int fd);

DESCRIPTION

 chdir remplace le répertoire courant par celui indiqué dans le chemin
path.

 fchdir est identique à chdir, sauf que le répertoire cible est fourni sous
forme de descripteur de fichier.

VALEUR RENVOYÉE

 chdir et fchdir renvoient 0 s'ils réussissent, ou -1 s'ils échouent,
auquel cas errno contient le code d'erreur.
```

```
ERREURS

 Suivant le type de système de fichiers, plusieurs erreurs peuvent être
renvoyées, les plus courantes pour chdir sont les suivantes :

 EFAULT path pointe en dehors de l'espace d'adressage accessible.

 ENAMETOOLONG path est trop long.

 ENOENT Le fichier n'existe pas.

 ENOMEM Pas assez de mémoire pour le noyau.

 ENOTDIR Un élément du chemin d'accès n'est pas un répertoire.

 EACCES L'accès n'est pas autorisé sur un élément du chemin.

 ELOOP path contient des références circulaires (à travers un lien
symbolique) EIO Erreur générique d'entrée/sortie.

 Les erreurs courantes pour fchdir sont :

 EBADF fd n'est pas un descripteur de fichier valide.

 EACCES Le répertoire ouvert sur fd n'autorise pas le parcours.
```

V-8/ umask()

```
NOM umask - Fixer le masque de création de fichiers.

SYNOPSIS (#include <sys/stat.h>)

 int umask(int mask);

DESCRIPTION

 umask fixe le masque de création de fichiers à la valeur mask & 0777.

 Ce masque est utilisé par open(2) pour positionner les permissions d'accès
initiales sur les fichiers nouvellement créés. Les bits contenus dans le
umask sont éliminés de la valeur 0666 pour créer les nouvelles permissions.
Par exemple la valeur par défaut 022 pour le umask fournit une autorisation
d'accès 0666 & ~022 = 0755 = rw-r--r--.

VALEUR RENVOYÉE

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

V-9/ getpriority()- setpriority()

```
NOM getpriority, setpriority - Lire / écrire la priorité d'ordonnancement du
processus.

SYNOPSIS (#include <sys/time.h> #include <sys/resource.h>)

 int getpriority(int which, int who);
```

```
int setpriority(int which, int who, int prio);

DESCRIPTION

 La priorité d'ordonnancement du processus, du groupe de processus ou de
l'utilisateur, comme précisé dans which et who est lue avec getpriority et
fixée avec setpriority. Which doit être l'un des éléments PRIO_PROCESS,
PRIO_PGRP, ou PRIO_USER, et who est interprété en fonction de which (un ID
de processus pour PRIO_PROCESS, un ID de groupe de processus pour PRIO_PGRP,
et un ID d'utilisateur pour PRIO_USER).

 Une valeur nulle pour who indique le processus, groupe ou utilisateur
courant. Prio est une valeur dans l'intervalle -20 à 20. La priorite par
défaut est 0, les priorités les plus faibles indiquant un ordonnancement le
plus favorable.

 La fonction getpriority retourne la plus haute priorité (la plus basse
valeur numérique) dont a bénéficié le processus. La fonction setpriority fixe
la priorité des processus indiqués à la valeur fournie. Seul le Super-User
peut diminuer la valeur numérique de la priorité (favoriser le processus).

VALEUR RENVOYÉE

 Comme getpriority peut tout à fait renvoyer la valeur -1, il faut effacer
la variable externe errno avant l'appel afin de vérifier si une valeur -1
indique une erreur ou une priorité légitime.

 setpriority renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno
contient le code d'erreur.

ERREURS

 ESRCH Aucun processus ne correspond aux valeurs de which et who.

 EINVAL Which n'était ni PRIO_PROCESS, ni PRIO_PGRP, ni PRIO_USER.

 De plus setpriority échouera pour les erreurs suivantes :

 EPERM Un processus correspond bien aux valeurs indiquées,mais ni son UID
réel, ni son UID effectif ne correspondent à ceux de l'appelant.

 EACCES Tentative de favoriser un processus sans être Super-User.
```

V-10/ times()

```
NOM times - Obtenir les statistiques de durée du processus.

SYNOPSIS (#include <sys/times.h>) clock_t times(struct tms *buf);

DESCRIPTION times stocke les durées statistiques du processus en cours dans
buf.

 struct tms est définie ainsi dans /usr/include/sys/times.h:

 struct tms {

 clock_t tms_utime; /* durée utilisateur
 */ clock_t tms_stime; /* durée système */
```

```
clock_t tms_cutime; /* durée utilisateur des fils
*/ clock_t tms_cstime; /* durée système des fils */
};
```

times renvoie le nombre de tops d'horloge écoulés depuis que le système a démarré.

V-11/ getrlimit() – getrusage() – setrlimit()

NOM

getrlimit, getrusage, setrlimit - Lire / écrire les limites d'utilisation des ressources systèmes.

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrlimit (int resource, struct rlimit *rlim);
int getrusage (int who, struct rusage *usage);

int setrlimit (int resource, const struct rlimit *rlim);
```

DESCRIPTION getrlimit et setrlimit lisent ou écrivent les limites des ressources systèmes. resource doit être l'un des éléments suivants :

```
RLIMIT_CPU /* Temps CPU en secondes */

RLIMIT_FSIZE /* Taille maximale d'un fichier */

RLIMIT_DATA /* Taille maximale zone de données */

RLIMIT_STACK /* Taille maximale de la pile */

RLIMIT_CORE /* Taille maximale fichier core */

RLIMIT_RSS /* Taille résidente maximale */

RLIMIT_NPROC /* Nombre maximal de processus */

RLIMIT_NOFILE /* Nombre maximal fichiers ouverts */

RLIMIT_MEMLOCK /* Verrouillage maximal en mémoire */
```

Une ressource peut être illimitée si l'on précise RLIM_INFINITY.

RLIMIT_OFILE est le nom "BSD" pour RLIMIT_NOFILE.

La structure rlimit est définie comme suit :

```
struct rlimit
{
rlim_t rlim_cur;

rlim_t rlim_max;
};
```

```
 Sous Linux 2.2, rlim_t est équivalent à un long int.

 getrusage renvoie l'utilisation courante des ressources, pour who
correspondant à RUSAGE_SELF, RUSAGE_CHILDREN, ou RUSAGE_BOTH correspondant
aux ressources consommées respectivement par le processus appelant,
l'ensemble des processus fils terminés, ou la somme de ces deux statistiques.


 struct rusage

 {

 struct timeval ru_utime; /* Temps utilisateur écoulé
 */ struct timeval ru_stime; /* Temps système écoulé */
 long ru_maxrss; /* Taille résidente maximale */ long
 ru_ixrss; /* Taille de mémoire partagée */

 long ru_idrss; /* Taille des données non partagées
 */ long ru_isrss; /* Taille de pile */ long
 ru_minflt; /* Demandes de pages */

 long ru_majflt; /* Nombre de fautes de pages
 */ long ru_nswap; /* Nombre de swaps */

 long ru_inblock; /* Nombre de lectures de blocs */
 long ru_oublock; /* Nombre d'écritures de blocs */
 long ru_msgsnd; /* Nombre de messages émis */

 long ru_msgrcv; /* Nombre de messages reçus */
 long ru_nsignals; /* Nombre de signaux reçus */
 long ru_nvcsw; /* Chgmnts de contexte volontaires */
 long ru_nivcsw; /* Chgmnts de contexte involontaires*/
 };

 En fait, sous Linux 2.2, seuls les membres ru_utime, ru_stime, ru_minflt,
ru_majflt, et ru_nswap sont remplis. Les autres sont mis à zéro.


VALEUR RENVOYÉE Ces fonctions renvoient 0 si elles réussissent, ou -1 si
elles échouent, auquel cas errno contient le code d'erreur.

ERREURS

 EFAULT rlim ou usage pointent en dehors de l'espace d'adressage disponible.

 EINVAL getrlimit ou setrlimit est appelé avec un mauvais argument resource,
ou getrusage est appelé avec un mauvais argument who.

 EPERM Tentative d'utiliser setrlimit() sans être Super-User pour augmenter
ses limites, ou alors le Super-User essaye d'augmenter les limites au-dessus
des maxima du noyau.
```

V-12/ sched yield

```
NOM sched yield - Céder le processeur.

SYNOPSIS (#include <sched.h>) int sched yield(void);
```

```
DESCRIPTION

 Un processus peut volontairement libérer le processeur sans se bloquer en
appelant sched_yield. Le processus sera alors déplacé à la fin de la liste
des processus prêts de sa priorité, et un autre processus sera exécuté.


 Note: Si le processus est le seul avec une priorité élevée, 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>.

VALEUR RENVOYÉE

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

V-13/ setregid() – setegid()

```
NOM setregid, setegid - Fixer le GID réel ou effectif.

SYNOPSIS (#include <sys/types.h> #include <unistd.h>)

 int setregid(gid_t rgid, gid_t egid);

 int setegid(gid_t egid);


DESCRIPTION

 setregid fixe les GID réel et effectif du processus en cours. Les
utilisateurs non-privilégiés peuvent changer leur GID réel pour le GID
effectif et inversement.

 Depuis Linux 1.1.37, il est également possible de fixer le GID effectif à
la valeur du GID sauvé.

 Seul le Super-User peut effectuer d'autres changements.

 Fournir une valeur -1 pour l'un ou l'autre des GID réel ou effectif conduit
le système à laisser ce GID inchangé.

 Actuellement (libc-4.x.x), setegid(egid) est fonctionnellement équivalent à
setregid(-1, egid).

 Si le GID réel est changé, ou si le GID effectif est positionné à une valeur
différente du GID réel précédent, le GID sauvé va prendre la valeur du nouveau
GID effectif.

VALEUR RENVOYÉE

 setregid et setegid renvoient zéro s'ils réussissent, et -1 s'ils échouent,
auquel cas errno contient le code d'erreur.
```

```
ERREURS

 EPERM le processus en cours n'est pas Super-User et d'autres changements
que (i) l'échange des GID effectif et réel, (ii) positionner l'un des GID à
la valeur de l'autre, ou (iii) placer le GID effectif à la valeur du GID
sauvé, ont été demandés.
```

V-14/ setreuid() – seteuid()

```
NOM setreuid, seteuid - Fixer l'UID réel ou effectif.

SYNOPSIS (#include <sys/types.h> #include <unistd.h>)


 int setreuid(uid_t ruid, uid_t euid);

 int seteuid(uid_t euid);

DESCRIPTION

 setreuid fixe les UID réel et effectif du processus en cours. Les
utilisateurs non-privilégiés peuvent changer leur UID réel pour l'UID
effectif et inversement.

 Avant Linux 1.1.37, le principe des ID sauvegardés, utilisés avec setreuid
ou seteuid ne fonctionnait pas.

 Depuis Linux 1.1.37, il est également possible de fixer l'UID effectif à la
valeur de l'UID sauvé.

 Seul le Super-User peut effectuer d'autres changements.

 Fournir une valeur -1 pour l'un ou l'autre des UID réel ou effectif conduit
le système à laisser cet UID inchangé.

 Actuellement seteuid(euid) est fonctionnellement équivalent à setreuid(-1,
euid).

 Si l'UID réel est changé, ou si l'UID effectif est positionné à une valeur
différente de l'ID réel précédent, l'UID sauve va prendre la valeur du nouvel
UID effectif.

VALEUR RENVOYÉE

 setreuid et seteuid renvoient zéro s'ils réussissent, et -1 s'ils échouent,
auquel cas errno contient le code d'erreur.

ERREURS

 EPERM le processus en cours n'est pas Super-User et d'autres changements
que (i) l'échange des UID effectif et réel, (ii) positionner l'un des UID à
la valeur de l'autre, ou (iii) placer l'UID effectif à la valeur de l'UID
sauvé, ont été demandés.
```

V-15/ setgid()

```
NOM setgid - Fixer l'ID de groupe.
```

```
SYNOPSIS (#include <unistd.h>) int setgid(gid_t gid)

DESCRIPTION

 setgid fixe le GID effectif du processus en cours. Si l'appelant est le
Super-User, les GID réels et sauvés sont également fixés.

 Sous Linux setgid est implémenté comme la version POSIX avec l'option
POSIX_SAVED_IDS. Ceci permet à un programme Set-GID (autre que root)
d'abandonner tous ses privilèges de groupe, d'effectuer des tâches non-
privilégiées, et de retrouver son GID effectif de manière sécurisée.

 Si l'utilisateur est le Super-User, ou si le programme est Set-GID root,
des précautions particulières doivent être prises. La fonction setgid vérifie
le GID effectif de l'appelant et si c'est le Super-User, tous les GID du
processus sont mis à gid. Une fois ceci effectué, il est impossible au
programme de retrouver ses privilèges de Super-User.

 Ainsi un programme Set-GID root désireux d'abandonner temporairement ses
privilèges, en prenant l'identité d'un programme de groupe non-root, puis de
récupérer ses privilèges par la suite ne doit pas utiliser setgid. On peut
accomplir ceci en utilisant l'appel (non-POSIX, BSD) setegid.

VALEUR RENVOYÉE


 setgid renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient
le code d'erreur.

ERREURS


 EPERM L'utilisateur n'est pas Super-User et gid ne correspond ni au GID
effectif, ni au GID sauvé du processus.
```

V-16/ setuid()

```
NOM setuid - Fixer l'ID de l'utilisateur.

SYNOPSIS (#include <sys/types.h> #include <unistd.h>)

 int setuid(uid_t uid)

DESCRIPTION

 setuid fixe l'UID effectif du processus en cours. Si cet UID effectif est
celui du Super-User, les UID réels et sauvés sont également fixés.

 Sous Linux setuid est implémenté comme le spécifie Posix, avec l'option
POSIX SAVED IDS. Ceci permet à un programme Set-UID (autre que root)
d'abandonner tous ses privilèges, d'effectuer des tâches non-privilégiées,
et de retrouver son UID effectif de manière sécurisée.

 Si l'utilisateur est le Super-User, ou si le programme est Set-UID root,
des précautions particulières doivent être prises. La fonction setuid vérifie
l'UID effectif de l'appelant et si c'est le Super-User, tous les UID du
processus sont mis à uid. Une fois ceci effectué, il est impossible au
programme de retrouver ses privilèges de Super-User.
```

Ainsi un programme Set-UID root désireux d'abandonner temporairement ses privilèges, en prenant l'identité d'un utilisateur non-root, puis de récupérer ses privilèges par la suite ne doit pas utiliser setuid. On peut accomplir ceci en utilisant l'appel (non-POSIX, BSD) seteuid.

VALEUR RENVOYÉE

setuid renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

ERREURS

EPERM L'utilisateur n'est pas Super-User et uid ne correspond ni à l'UID effectif, ni à l'UID sauvé du processus.

SPÉCIFICITÉS LINUX

Linux dispose d'un concept d'UID de système de fichiers, qui est normalement égal à l'UID effectif. L'appel setuid peut également fixer l'UID de système de fichiers du processus en cours. Voir setfsuid(2).

Si l'uid est différent de l'ancien UID effectif, le processus ne pourra pas laisser d'image mémoire (core dump) sur le disque.

V-17/ execl() – execlp() - execle () – execv() – execvpo()

NOM execl, execlp, execle, execv, execvp - Exécuter un programme.

SYNOPSIS (#include <unistd.h>)

```
extern char **environ;

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[]);
```

DESCRIPTION

La famille de fonctions exec remplace l'image mémoire du processus en cours par un nouveau processus. Les fonctions décrites dans cette page sont en réalité des frontaux pour l'appel système execve(2). (Voir la page de execve pour des informations détaillées sur le remplacement du processus en cours.)

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 des 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.

Les fonctions execv et execvp 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 pointeur doit se terminer par un pointeur NULL.

La fonction execle peut également indiquer l'environnement du processus à exécuter en faisant suivre le pointeur NULL qui termine la liste d'arguments, ou le pointeur NULL de la table par un paramètre supplémentaire. 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.

Certaines de ces fonctions ont une sémantique spécifique.

Les fonctions execlp et execvp agiront comme le shell dans la recherche du fichier exécutable si le nom fourni ne contient pas de slash (/). Le chemin de recherche est spécifié dans la variable d'environnement PATH. Si cette variable n'est pas définie, le chemin par défaut sera ``/bin:/usr/bin:''. De plus certaines erreurs sont traitées de manière spécifique.

Si la permission d'accès au fichier est refusée (execve renvoie EACCES), ces fonctions continueront à parcourir le reste du chemin de recherche. Si aucun fichier n'est trouvé, elles reviendront, et errno contiendra le code d'erreur EACCES.

Si l'en-tête d'un fichier n'est pas reconnu (execve renvoie ENOEXEC), ces fonctions exécuteront un shell avec le chemin d'accès au fichier en tant que premier argument. Si ceci échoue, aucune recherche supplémentaire n'est effectuée.

VALEUR RENVOYÉE

Si l'une quelconque des fonctions exec revient à l'appelant, c'est qu'une erreur a eu lieu. La valeur de retour est -1, et errno contient le code d'erreur.

ERREURS

Toutes ces fonctions peuvent échouer et positionner errno sur n'importe quelle erreur décrite dans la fonction execve(2).

V-18/ execve()

NOM execve - Exécuter un programme.

SYNOPSIS (#include <unistd.h>)

```
int execve (const char *fichier, char * constargv [], char * constenvp[]);
```

DESCRIPTION

execve() exécute le programme correspondant au fichier. Celui-ci doit être un exécutable binaire ou bien un script commençant par une ligne du type "#! interpreteur [arg]". Dans ce dernier cas, l'interpréteur doit être indiqué

par un nom complet, avec son chemin d'accès, et qui sera invoqué sous la forme interprèteur [arg] fichier.

argv est un tableau de chaînes d'arguments passées au nouveau programme. envp est un tableau de chaînes, ayant par convention la forme cle=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, ainsi que la pile du processus appelant sont remplacés par ceux du programme chargé. Le programme invoqué hérite du PID du processus appelant, et de tous les descripteurs de fichiers qui ne possèdent pas le drapeau Close-on-Exec. Les signaux en attente destinés au processus parent sont effacés. Les signaux prêts à être intercepté par le processus appelant reprennent leur comportement par défaut.

Si l'on effectuait un ptrace(2) sur le programme appelant, un signal SIGTRAP est envoyé après la réussite de execve().

Si le bit Set-UID est positionné sur le fichier du programme, 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. Si l'exécutable est un fichier binaire a.out lié dynamiquement, et contenant des appels aux bibliothèques partagées, le linker dynamique de Linux ld.so(1) est appelé avant l'exécution, afin de charger les bibliothèques partagées nécessaires en mémoire, et d'effectuer l'édition des liens de l'exécutable.

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.1 pour les fichiers binaires liés avec la libc Linux version 5, ou /lib/ld-linux.so.2 pour ceux liés avec la GNU libc version 2.

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

EACCES Le fichier n'est pas un fichier régulier.

EACCES L'autorisation d'exécution est refusée pour le fichier, ou un script, ou un interpréteur ELF.

EPERM Le système de fichiers est monté avec l'attribut noexec.

EPERM Le système de fichiers est monté avec l'attribut nosuid et le fichier a un bit Set-UID ou Set-GID positionné.

E2BIG La liste d'arguments est trop grande.

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

```
 EFAULT L'argument fichier pointe en dehors de l'espace d'adressage
accessible.

 ENAMETOOLONG La chaîne de caractères fichier est trop longue.

 ENOENT Le fichier n'existe pas.

 ENOMEM Pas assez de mémoire pour le noyau.

 ENOTDIR Un élément du chemin d'accès n'est pas un répertoire.

 ELOOP Le chemin d'accès au fichier contient une référence circulaire (à
travers un lien symbolique)

 ETXTBSY Le fichier exécutable a été ouvert en écriture par un ou plusieurs
processus.

 EIO Une erreur d'entrée/sortie de bas-niveau s'est produite.

 ENFILE Le nombre maximal de fichiers ouverts sur le système est atteint

 EMFILE Le nombre maximal de fichiers ouverts par processus est atteint.

 EINVAL Un exécutable ELF a plusieurs segments PT INTERP (indique plusieurs
interpréteurs).

 EISDIR L'interpréteur ELF cité est un répertoire.

 ELIBBAD L'interpréteur ELF mentionné n'est pas dans un format connu.

NOTES

 Les processus Set-UID et Set-GID ne peuvent pas être suivis par ptrace()
en positionnant effectivement leur UID ou leur GID.

 La première ligne d'un shell script exécutable (#!) a une longueur
maximale de 127 caractères.

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