# SYSG6 - FORK(), VFORK()et CLONE()

17 février 2022

—

## Participant

Léopold Mols - 53212

## Table des matières

# Sujets

## fork()

1. Qu’est-ce ?

La fonction **fork** fait partie des **appels système** standard d'**UNIX**

Cette fonction permet à un **processus** de **donner naissance** à un **nouveau processus** qui est **sa copie conforme**, par exemple, en vue de réaliser un **second traitement parallèlement au premier**.

Le **créateur** d'un nouveau processus est appelé le **père** et le **nouveau** processus, le **fils**. La plupart des attributs système du père (par exemple les droits sur le système de fichier) sont transmis au fils, de la même manière que l'héritage.

Au **démarrage** d'un **système Unix**, un seul processus existe (de numéro 1). Tous les autres processus qui peuvent exister au cours de la vie du système descendent de ce **premier processus**, appelé `init`, via des appels système comme `fork`, `vfork` ou d'autres moyens.

2. Historique

Sur les premiers UNIX (1969 -> années 1990), seul l'appel système `fork` permet de créer de nouveaux processus.

La fonction **fork** fait partie des **appels système** standard d'**UNIX** (norme **POSIX** (Portable Operating System Interface et le X exprime l'héritage UNIX) qui est une famille de normes techniques définie depuis 1988 par l'Institute of Electrical and Electronics Engineers (IEEE), et formellement désignée par IEEE 1003. Ces normes ont émergé d'un projet de standardisation des interfaces de programmation des logiciels destinés à fonctionner sur les variantes du système d'exploitation UNIX).

3. Fonctionnement

L'appel système fork fournit une valeur résultat qui est entière. Pour différencier le père du fils, il suffit de regarder la valeur de retour du fork() qui peut être :

- le PID du fils, auquel cas nous sommes dans le processus père
- 0 auquel cas nous sommes dans le processus fils.
- -1 qui témoigne une erreur lors de l’exécution de la commande.

Il est possible d'interagir entre processus de plusieurs manières différentes. Premièrement, on peut envoyer des **signaux.** En langage de commande *kill <pid>* permet de tuer le processus ayant pour pid ce que l'on entre dans la commande.

Il est possible de **faire attendre un processus** grâce à *sleep(n)* pour bloquer le processus pendant n secondes, ou en utilisant *pause()* qui bloque jusqu'à la réception d'un signal.

Pour mettre fin à un processus, on peut utiliser *exit(state)* sachant que *state* est un code de fin, par convention 0 si ok, code d'erreur sinon.

Il peut être très pratique que **le père attende la fin de l'un de ses fils**, pour ce faire on utilise *pid_t wait(int *ptr_state)* qui donne comme valeur de retour le pid du fils qui a terminé, et le code de fin est stocké dans ptr_state.

On peut également **attendre la fin du fils grâce à son pid** : *pid_t waitpid(pid_t pid, int *ptr_state, int options).*

Un terme commun dans la partie « Système » de l'informatique est ce que l'on appelle les **processus zombies**. Cela arrive quand le processus est terminé mais que le père n'a pas attendu son fils, c'est-à-dire qu'il n'a pas fait d'appels à *wait()*. C'est une situation qu'il convient d'éviter absolument car le processus ne peut plus s'exécuter mais consomme encore des ressources.

4. Particularités
5. Problèmes éventuels
6. Suppléments

fork1(), forkall(), forkx(), forkallx()
The `forkx()` and `forkallx()` functions accept a `flags` argument consisting of a bitwise inclusive-OR of zero or more of the following flags, which are defined in the header `sys/fork.h`

If the flags argument is 0 `forkx ()` is identical to `fork()` and `forkallx ()` is identical to `forkall()`.

## vfork()

1. Qu'est-ce ?
2. Historique
3. Fonctionnement
4. Particularités
5. Problèmes éventuels

La différence fondamentale entre vfork et fork est que lorsqu'un nouveau processus est créé avec vfork(), le processus parent est temporairement suspendu et le processus enfant peut emprunter l'espace d'adressage du parent. Cet état étrange se poursuit jusqu'à

ce que le processus enfant se termine ou appelle execve(), après quoi le processus parent se poursuit.

Cela signifie que le processus enfant d'une vfork() doit faire attention à éviter de modifier de manière inattendue les variables du processus parent. En particulier, le processus enfant ne doit pas retourner à partir de la fonction contenant l'appel vfork(), et il ne doit pas appeler exit() (s'il doit quitter, il doit utiliser _exit(); en fait, c'est également vrai pour l'enfant d'une fork() normale).

La différence avec fork est que vfork a le même UID, à l'instar de fork()

### clone()

1. Qu'est-ce ?
2. Historique
3. Fonctionnement
4. Particularités
5. Problèmes éventuels

## Notes

- **Insérez votre texte ici** Insérez votre texte ici.
- **Insérez votre texte ici** Insérez votre texte ici Insérez votre texte ici Insérez votre texte ici Insérez votre texte ici.
    - Insérez votre texte ici.

## Tâches

But : ne PAS reproduire une page de manuel en expliquant quels sont les arguments de fork en les expliquant.

Il faut trouver des choses supplémentaires, ce qui m'impressionne ou ce que je découvre et l'expliquer. Pas réexpliquer ce que sont ces appels systèmes / commandes

### fork()

7. Qu'est-ce ?
8. Historique

9. Fonctionnement
10. Particularités
11. Problèmes éventuels

## vfork()

6. Qu’est-ce ?

La fonction vfork() crée un nouveau processus. La fonction vfork() a le même effet que fork(), sauf que le comportement n'est pas défini, si le processus créé par vfork() tente d'appeler toute autre fonction C/370 avant d'appeler exec() ou _exit(). Le nouveau processus (le *processus enfant*) est un duplicata exact du processus qui appelle vfork() (le *processus parent*), à l'exception de ce qui suit :

- Le processus enfant a un ID de processus (PID) unique, qui ne correspond à aucun ID de groupe de processus actif.
- L'enfant a sa propre copie de la TDFO du parent. Chaque descripteur de fichier dans l'enfant fait référence au même descripteur de fichiers ouverts que le descripteur de fichier correspondant dans le parent.
- L'enfant a sa propre copie des flux de répertoires ouverts du parent. Le flux de répertoires ouverts de chaque enfant peut partager le positionnement du flux de répertoires avec le flux de répertoires du parent correspondant.
- L'enfant n'hérite d'aucun verrou de fichier précédemment défini par le parent.
- Le processus enfant n'a pas d'alarmes définies (semblable aux résultats d'un appel à alarm() avec une valeur d'argument de 0).
- L'enfant n'a pas de signaux en attente.
- Les minuteries d'intervalle sont réinitialisées dans le processus enfant.
- Ces éléments sont mis à 0 dans le fils :
    - `tms_utime`
    - `tms_stime`
    - `tms_cutime`
    - `tms_cstime`

7. Historique

L'appel système **vfork**() est apparu dans BSD 3.0. Dans BSD 4.4, il est devenu synonyme de **fork**(2), mais NetBSD l'a réintroduit à nouveau : voir *http://www.netbsd.org/Documentation/kernel/vfork.html*. Sous Linux, il fut l'équivalent de **fork**(2) jusqu'au noyau 2.2.0-pre-6. Depuis le 2.2.0-pre-9 il s'agit d'un appel système indépendant. Le support dans la bibliothèque a été introduit dans la glibc 2.0.112.

## 8. Fonctionnement

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <mm_malloc.h>
#include <spawn.h>
#include <string.h>

/*int main()
{
   int a = 10;
   pid_t errFils = 0;
   int * adress = malloc(sizeof(int));
   char buff[10];
   printf ("\nBefore Forking\n");
   printf ("Je suis le père\n");
   printf ("a = %d\n", a);
   printf ("Adress of malloc : %d\n", adress);
   printf ("Valeur de retour du vfork : %d\n", errFils);
   printf ("PID = %d\n", getpid());
   printf ("PPID = %d\n\n", getppid());
   if ((errFils = vfork()) == 0)
   {
      a = 20;
      printf ("Je suis le fils\n");
      char * name = "Léo";
      strcpy(buff, *name);
      //printf ("PID = %d\n", getpid());
      //printf ("PPID = %d\n", getppid());
      exit(0); // Remplacer par "wait(0)" pour montrer la différence pour démontrer que seul le fils exécutera la suite puisque c'est le même espace d'adressage, donc, la même TDFO, donc, une fois que la variable sera changée et que le fils se sera occupé de print sur la sortie standard, le père n'aura pus à le faire car stdout ne sera plus dans la TDFO.

      // wait est un processus bloquant. Donc, la suite ne sera pas exécutée tant qu'une condition ne sera pas remplie. Si l'on met un pointeur d'un nombre, alors, on pourra récupérer le code de terminaison du processus enfant. Pareil pour exit

      // Attention : le "exit(0)" le tue, mais ne l'enlève pas de la table des process et envoie un signal à son parent

      // Si l'on met cette ligne en commentaire (celle avec le "exit(0)" ou le "wait(0)"), alors, une erreur de segmentation sera renvoyée
   }
```

```
    //wait(errFils);
    printf ("After Forking\n");
    printf ("a = %d\n", a);
    printf ("Adress of malloc : %d\n", adress);
    printf ("Valeur de retour du vfork : %d\n", errFils);
    printf ("PID = %d\n", getpid());
    printf ("PPID = %d\n\n", getppid());
}*/


int main(int argc, char **argv) {

    int a = 5, b = 8;
    int v;

    /**
     * Le VFORK duplique l'aspace d'adressage.
     * Donc, le VFORK fera les additions de son côté, mais dans l'espace d'adressage du PERE
     *  Ensuite, le fils sera exit, donc, il n'affichera pas le résultat de ses additions,
     *  mais les variables sont bien modifiées
     *
     * Du côté du père, l'addition sera pas faite puisqu'il partage l'espace d'adressage du FILS
     *
     * Vu que l'espace d'adressage n'est pas dupliqué lors du VFORK, les variables seront modifiées dans
     *  l'espace d'adressage du PERE (qui est aussi celui du fils)
     *  Donc, les variables du père sont modifiées,
     *   ce qui permet la prise en compte de la modification des valeurs des variables
     */
    v = vfork();
    if(v == 0) {
        // a = 10
        a = a + 5;
        // b = 10
        b = b + 2;
        exit(0);
    }
    // Parent code
    wait(0);
    printf("PID = %d\n", getpid());
    printf("PPID = %d\n", getppid());
    printf("Value of v is %d.\n", v); // line a
    printf("Sum is %d.\n", a + b); // line b
    sleep(4000);
    printf("Let's do a ps to see which process is currenlty running !");
    exit(0);

}
```

9. Particularités

L'espace d'adressage n'est pas un nouveau par rapport au père lors de la duplication comme pour un fork(), mais l'espace d'adressage sera le même. Cela peut permettre de faire en sorte que, si le processus fils se passe correctement, le père n'aura plus rien à faire, plutôt que de d'office reprendre la main après le fils.

10. Problèmes éventuels

```
The vfork() system call will fail for any of the reasons described in the
     fork man page.  In addition, it will fail if:

[EINVAL]           A system call other than _exit() or execve() (or libc
                        functions that make no system calls other than
those)
                        is called following calling a vfork() call.


L'appel système vfork() échouera pour l'une des raisons décrites dans le
     page de manuel de fourche. De plus, il échouera si :

[EINVAL] (valeur qu'un programme doit renvoyer lorsqu'il reçoit un
argument invalide.) Un appel système autre que _exit() ou execve() (ou
libc
                        fonctions qui ne font aucun appel système autre
que ceux-là)
                        est appelé après avoir appelé un appel vfork().
```

Il est regrettable que Linux ait ressuscité ce spectre du passé. La page de manuel de BSD indique que cet appel système sera supprimé quand des mécanismes de partage appropriés seront implémentés, et qu'il ne faut pas essayer de tirer profit du partage mémoire induit par **vfork**(), car dans ce cas, il sera rendu synonyme de **fork**(2).

Les détails de la gestion des signaux sont compliqués, et varient suivant les systèmes. La page de manuel BSD indique : « Pour éviter de possibles situations de blocage, les processus qui sont des fils au milieu d'un **vfork**() ne reçoivent jamais les signaux **SIGTTOU** ou **SIGTTIN** ; à la place, des sorties ou des requêtes *iotcl* sont autorisées et des tentatives d'entrées indiqueront une fin de fichier. »

Partager l'espace d'adressage avec le parent laisse la possibilité de le modifier, donc, de corrompre l'exécution du père lorsqu'il reprendra la main.

man vorf()

"The behavior is undefined if the process created by vfork() either modifies any data other than a variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling _exit(2) or one of the exec(3) family of functions"

"le comportement est indéfini si le processus créé par vfork() modifie des données autres qu'une variable de type pid_t utilisée pour stocker la valeur de retour de vfork(), ou revient de la fonction dans laquelle vfork() a été appelé, ou appelle toute autre fonction avant d'appeler avec succès _exit(2) ou l'une des fonctions de la famille exec(3)"

This system call is deprecated. In a future release, it may begin to return errors in all cases, or may be removed entirely. It is extremely strongly recommended to replace all uses with fork(2) or, ideally, posix_spawn(3).

**→ pointer les différences systèmes entre fork et vfork (avec le noyau) avec des comportements annexes.**

## Peut-être que c’est le noyau qui alloue l’espace d’adressage, du coup, il y a une différence au niveau du noyau pour cette allocation.

## Parler aussi des entrées dans la table TDFO

## montrer avec la commande ps que le père est en pause avec vfork et non fork

fork(2) a donné aux créateurs d'Unix la possibilité de déplacer toute cette complexité du kernel-land vers le user-land, où il est beaucoup plus facile de développer des logiciels. Cela les a rendus plus productifs, peut-être beaucoup plus. Le prix que les créateurs d'Unix ont payé pour cette élégance était la nécessité de copier les espaces d'adressage. Étant donné qu'à l'époque, les programmes et les processus étaient petits, l'inélégance était facile à négliger ou à ignorer. Mais maintenant, les processus ont tendance à être énormes et multithreads, ce qui rend extrêmement coûteux la copie même de l'ensemble résident d'un parent et la manipulation de la table des pages pour le reste.

Mais vfork() a ces avantages, et aucun des inconvénients de fork() !

vfork() a un inconvénient : que le parent (en particulier : le thread dans le parent qui appelle vfork()) et l'enfant partagent une pile, ce qui nécessite que le parent (thread) soit arrêté jusqu'à ce que l'enfant exec()s ou _exit( )s. (Cela peut être pardonné en raison des longs threads précédents de vfork(2) -- lorsque les threads sont apparus, le besoin d'une pile séparée pour chaque nouveau thread est devenu tout à fait clair et inévitable. La solution pour les threads était d'utiliser une nouvelle pile pour le nouveau thread et utilisez une fonction de rappel et un argument comme principal () pour cette nouvelle pile.) Mais le blocage est mauvais car le comportement synchrone est mauvais, en particulier lorsque vfork (2) (ou clone (2), utilisé comme vfork (2 )) est la seule alternative performante à fork(2), mais cela aurait pu être mieux.

Toutes les pages de manuel vfork(2) que j'ai vues indiquent que le processus parent est arrêté jusqu'à ce que l'enfant quitte/exécute, mais cela est antérieur aux threads. Linux, par exemple, n'arrête que le seul thread du parent qui a appelé vfork(), pas tous les threads. Je pense que c'est la bonne chose à faire, mais les autres systèmes d'exploitation de l'IIRC arrêtent tous les threads du processus parent (ce qui est une erreur, IMO).

## clone()

6. Qu’est-ce ?

Contrairement à fork(), ces appels système fournissent un contrôle plus précis sur les éléments de contexte d'exécution. Ils sont partagés entre le processus appelant et le processus enfant. Par exemple, en utilisant cet appel système, l'appelant peut contrôler si oui ou non les deux les processus partagent l'espace d'adressage virtuel, la TDFO et le tableau des gestionnaires de signaux. Ces système les appels permettent également au nouveau processus enfant d'être placé dans des espaces de noms.

Contrairement à **fork**(2), cette routine permet le partage d'une partie du contexte d'exécution entre le processus fils et le processus appelant. Le partage peut s'appliquer sur l'espace mémoire, sur la table des descripteurs de fichier ou la table des gestionnaires de signaux. (Notez que sur cette page de manuel, le « processus appelant » correspond normalement au « processus père », mais voyez quand même la description de **CLONE_PARENT** plus bas).

L'appel système **clone**() est principalement utilisé pour permettre l'implémentation des threads : un programme est scindé en plusieurs lignes de contrôle, s'exécutant simultanément dans un espace mémoire partagée.

Quand le processus fils est créé, avec **clone**(), il exécute la fonction *fn*(*arg*) de l'application. (Ceci est différent de **fork**(2) avec lequel l'exécution continue dans le fils au point de l'appel **fork**(2)) L'argument *fn* est un pointeur sur la fonction appelée par le processus fils lors de son démarrage. L'argument *arg* est transmis à la fonction *fn* lors de son invocation.

Quand la fonction *fn*(*arg*) revient, le processus fils se termine. La valeur entière renvoyée par *fn* est utilisée comme code de retour du processus fils. Ce dernier peut également se terminer de manière explicite en invoquant la fonction **exit**(2) ou après la réception d'un signal fatal.

L'argument *pile_fils* indique l'emplacement de la pile utilisée par le processus fils. Comme les processus fils et appelant peuvent partager de la mémoire, il n'est généralement pas

possible pour le fils d'utiliser la même pile que son père. Le processus appelant doit donc préparer un espace mémoire pour stocker la pile de son fils, et transmettre à **clone**() un pointeur sur cet emplacement. Les piles croissent vers le bas sur tous les processeurs implémentant Linux (sauf le HP PA), donc *pile_fils* doit pointer sur la plus haute adresse de l'espace mémoire prévu pour la pile du processus fils.

L'octet de poids faible de *flags* contient le numéro du *signal de terminaison* qui sera envoyé au père lorsque le processus fils se terminera. Si ce signal est différent de **SIGCHLD**, le processus parent doit également spécifier les options__**WALL** ou **__WCLONE** lorsqu'il attend la fin du fils avec **wait**(2). Si aucun signal n'est indiqué, le processus parent ne sera pas notifié de la terminaison du fils.

*flags* permet également de préciser ce qui sera partagé entre le père et le fils, en effectuant un OU binaire entre zéro ou plusieurs des constantes suivantes :

7. Historique

Peut-être que Linux aurait dû avoir un appel système de création de threads - Linux aurait alors pu s'épargner la douleur de la première implémentation de pthread pour Linux. (Beaucoup d'erreurs ont été commises sur le chemin du NPTL.) Linux aurait dû apprendre de Solaris/SVR4, où l'émulation des sockets BSD via libsocket au-dessus de STREAMS s'est avérée être une erreur qui a pris beaucoup de temps et beaucoup d'argent à corriger . L'émulation d'une API à partir d'une autre API avec des décalages d'impédance est généralement au mieux difficile.

Depuis lors, clone(2) est devenu un couteau suisse - il a évolué pour avoir des fonctionnalités d'entrée dans les zones/prison, mais seulement en quelque sorte : Linux n'a pas de zones/prison appropriées, à la place, Linux a ajouté de nouveaux drapeaux clone(2) à pour indiquer les espaces de noms qui ne doivent pas être partagés avec le parent. Et au fur et à mesure que de nouveaux drapeaux clone(2) liés au conteneur sont ajoutés, l'ancien code pourrait souhaiter les avoir utilisés... il faudra modifier et reconstruire le monde appelant clone(2), et ce n'est décidément pas élégant.

8. Fonctionnement

/**

Ces appels système créent un nouveau processus ("fils"), d'une manière

similaire à fork(2).

Contrairement à fork(2), ces appels système fournissent de

contrôler quels éléments de contexte d'exécution sont partagés entre le processus appelant et le processus enfant. Par exemple, en utilisant ces appels système, l'appelant peut contrôler si oui ou non les deux les processus partagent l'espace d'adressage virtuel, la table des fichiers descripteurs ouverts et le tableau des gestionnaires de signaux. Ces appels systèmes permettent également au processus enfant d'être placé dans des espaces de noms.

Lorsque le processus enfant est créé avec la fonction clone(), il commence l'exécution en appelant la fonction pointée par l'argument fn. (Cela diffère de fork(2), où l'exécution continue dans l'enfant à partir du point de la bifurcation.). L'argument arg est passé comme argument de la fonction fn.

Lorsque la fonction fn(arg) revient, le processus enfant se termine. L'entier retourné par fn est le statut de sortie pour l'enfant traité. Le processus enfant peut également se terminer explicitement en appelant exit(2) ou après avoir reçu un signal de mauvaise exécution se traduisant par une erreur.

L'argument pile spécifie l'emplacement de la pile utilisée par le processus fils. Étant donné que l'enfant et le processus d'appel peuvent partager la mémoire, il n'est pas possible pour le processus enfant de s'exécuter dans la même pile que le processus appelant. Le processus d'appel doit configurez donc de l'espace mémoire pour la pile de l'enfant et passez un pointeur vers cet espace pour clone(). Les piles poussent vers le bas sur tout processus qui exécutent Linux (à l'exception des processeurs HP PA), donc empiler pointe généralement vers l'adresse la plus haute de l'espace mémoire configuré pour la pile de l'enfant.
Les arguments restants à clone() sont discutés ci-dessous.

Lorsque le processus enfant se termine, un signal peut être envoyé au parent. Le signal de terminaison est spécifié dans l'octet de poids faible de

flags (clone3()) ou dans cl_args.exit_signal (clone3()). Si ce

signal est spécifié autrement que SIGCHLD, alors le

le processus parent doit spécifier les options __WALL ou __WCLONE lorsqu'il attend l'enfant avec wait(2). Si aucun signal n'est

spécifié, le processus parent n'est pas signalé lorsque l'enfant

se termine.

*/

Quand le processus enfant est créé par la fonction enveloppe **clone**(), il débute son exécution par un appel à la fonction vers laquelle pointe l'argument fn (cela est différent de **fork**(2), pour lequel l'exécution continue dans le processus enfant à partir du moment de l'appel de **fork**(2)). L'argument arg est passé comme argument de la fonction

fn.

Quand la fonction fn(arg) renvoie, le processus enfant se termine. La valeur entière renvoyée par fn est utilisée comme code de retour du processus enfant. Ce dernier peut

également se terminer de manière explicite en invoquant la fonction **exit**(2) ou après la

réception d'un signal fatal.

L'argument stack indique l'emplacement de la pile utilisée par le processus enfant. Comme

les processus enfant et appelant peuvent partager de la mémoire, il n'est généralement pas

possible pour l'enfant d'utiliser la même pile que son parent. Le processus appelant doit

donc préparer un espace mémoire pour stocker la pile de son enfant, et transmettre à **clone**

un pointeur sur cet emplacement. Les piles croissent vers le bas sur tous les processeurs

implémentant Linux (sauf le HP PA), donc stack doit pointer sur la plus haute adresse de

l'espace mémoire prévu pour la pile du processus enfant. Remarquez que **clone**() ne fournit

aucun moyen pour que l'appelant puisse informer le noyau de la taille de la zone de la

pile.

9. Particularités

L'appel système **clone3**() fournit un sur-ensemble de la fonctionnalité de l'ancienne

interface de **clone**(). Il offre également un certain nombre d'améliorations de l'API dont :

un espace pour des bits d'attributs supplémentaires, une séparation plus propre dans

l'utilisation de plusieurs paramètres et la possibilité d'indiquer la taille de la zone de

la pile de l'enfant.

Comme avec **fork**(2), **clone3**() renvoie à la fois au parent et à l'enfant. Il renvoie **0** dans

le processus enfant et il renvoie le PID de l'enfant dans le parent.

Le paramètre cl_args de **clone3**() est une structure ayant la forme suivante :

```
struct clone_args {
        u64 flags;        /* Masque de bit d'attribut */
        u64 pidfd;        /* Où stocker le descripteur de fichier du PID
                            (pid_t *) */
        u64 child_tid;    /* Où stocker le TID enfant,
                            dans la mémoire de l'enfant's memory (pid_t *) */
        u64 parent_tid;   /* Où stocker le TID enfant,
                            dans la mémoire du parent's memory (int *) */
        u64 exit_signal;  /* Signal à envoyer au parent quand
                            l'efant se termine */
```

```
        u64 stack;      /* Pointeur vers l'octet le plus faible de la pile */
        u64 stack_size;  /* Taille de la pile */
        u64 tls;        /* Emplacement du nouveau TLS */
        u64 set_tid;    /* Pointeur vers un tableau pid_t
                           (depuis Linux 5.5) */
        u64 set_tid_size; /* Nombre d'éléments dans set_tid
                           (depuis Linux 5.5) */
        u64 cgroup;     /* Descripteur de fichier du cgroup cible
                           de l'enfant (depuis Linux 5.7) */
    };
```

Le paramètre size fourni à **clone3**() doit être initialisé à la taille de cette structure (l'existence du paramètre size autorise des extensions futures de la structure clone_args).

La pile du processus enfant est indiquée avec cl_args.stack, qui pointe vers l'octet le plus faible de la zone de la pile, et avec cl_args.stack_size, qui indique la taille de la pile en octets. Si l'attribut **CLONE_VM** est indiqué (voir ci-dessous), une pile doit être explicitement allouée et indiquée. Sinon, ces deux champs peuvent valoir NULL et **0**, ce qui amène l'enfant à utiliser la même zone de pile que son parent (dans l'espace d'adressage virtuel de son propre enfant).

**Équivalence entre les paramètres de clone() et de clone3()**

| **clone()** | **clone3()** | **Notes** |
|---|---|---|
| | Champ cl_args | |
| attributs & ~0xff | attributs | Pour la plupart des attributs ; détails ci-dessous |
| parent_tid | pidfd | Voir CLONE_PIDFD |
| child_tid | child_tid | Voir CLONE_CHILD_SETTID |

parent_tid parent_tid Voir CLONE_PARENT_SETTID

attributs & 0xff exit_signal

pile pile

--- stack_size

tls tls Voir CLONE_SETTLS

--- set_tid Voir ci-dessous pour des détails

--- set_tid_size

--- cgroup Voir CLONE_INTO_CGROUP

\begin{itemize}

\begin{description}

\item[L'espace d'adressage] est dupliqué

\item[Le processus enfant] est un duplicata exact du processus qui appelle vfork() (le processus parent), à l'exception de ce qui suit :

\end{description}

\begin{itemize}

\item[$\ast$] Le processus enfant a un ID de processus (PID) unique, qui ne correspond à aucun ID de groupe de processus actif.

\item[$\ast$] L'enfant a sa propre copie de la TDFO du parent. Chaque descripteur de fichier dans l'enfant fait référence au même descripteur de fichiers ouverts que le descripteur de fichier correspondant dans le parent.

\item[$\ast$] L'enfant a sa propre copie des flux de répertoires ouverts du parent. Le flux de répertoires ouverts de chaque enfant peut partager le positionnement du flux de répertoires avec le flux de répertoires du parent correspondant.

\item[$\ast$] L'enfant n'hérite d'aucun verrou de fichier précédemment défini par le parent.

\item[$\ast$] Le processus enfant n'a pas d'alarmes définies (semblable aux résultats d'un appel à alarm() avec une valeur d'argument de 0).

\item[$\ast$] L'enfant n'a pas de signaux en attente.

\item[$\ast$] Les minuteries d'intervalle sont réinitialisées dans le processus enfant.

\item[$\ast$] Ces éléments sont mis à 0 dans le fils : tms_utime, tms_stime, tms_cutime, tms_cstime

\end{itemize}

\end{itemize}

\\\\\begin{large}

Valeur de retour

\end{large}

En cas de réussite, le TID du processus enfant est renvoyé dans le thread d'exécution de l'appelant. En cas d'échec, -1 est renvoyé dans le contexte de l'appelant, aucun enfant n'est créé, et errno contiendra le code d'erreur.

10. Problèmes éventuels

Les versions de la bibliothèque C GNU jusqu'à la 2.24 comprise contenaient une fonction enveloppe pour getpid(2) qui effectuait un cache des PID. Ce cache nécessitait une prise en charge par l'enveloppe de clone() de la glibc, mais des limites dans l'implémentation faisaient que le cache pouvait ne pas être à jour sous certaines circonstances. En particulier, si un signal était distribué à un enfant juste après l'appel à clone(), alors un appel à getpid(2) dans le gestionnaire de signaux du signal pouvait renvoyer le PID du processus appelant (le parent), si l'enveloppe de clone n'avait toujours pas eu le temps de mettre le cache de PID à jour pour l'enfant. (Ce point ignore le cas où l'enfant a été créé en utilisant CLONE_THREAD, quand getpid(2) doit renvoyer la même valeur pour l'enfant et pour le processus qui a appelé clone(), puisque l'appelant et l'enfant se trouvent dans le même groupe de threads. Ce problème de cache n'apparaît pas non plus si le paramètre flags contient CLONE_VM.) Pour obtenir la véritable valeur, il peut être nécessaire d'utiliser quelque chose comme ceci :

\begin{lstlisting}

#include <syscall.h>

pid_t mypid;

mypid = syscall(SYS_getpid);

\end{lstlisting}

\\\\Suite à un problème de cache ancien, ainsi qu'à d'autres problèmes traités dans getpid(2), la fonctionnalité de mise en cache du PID a été supprimée de la glibc 2.25.

## Aller plus loin

L'ajout de la fonction *forkall*() au standard a été considéré et rejeté. La fonction *forkall*() permet à tous les threads du parent d'être dupliqués dans l'enfant. Ceci reproduit essentiellement l'état du parent chez l'enfant. Cela permet aux threads de l'enfant de poursuivre le traitement et permet de préserver les verrous et l'état sans code *pthread_atfork*() explicite.

Le processus appelant doit s'assurer que l'état de traitement des threads qui est partagé entre le parent et l'enfant (c'est-à-dire les descripteurs de fichiers ou la mémoire MAP_SHARED) se comporte correctement après *forkall*(). Par exemple, si un thread lit un descripteur de fichier dans le parent lorsque *forkall*() est appelée, alors deux threads (un dans le parent et un dans le child) lisent le fichier fileescriptor après la *forkall*(). Si ce n'est pas un comportement souhaité, le processus parent doit se synchroniser avec de tels threads avant d'appeler *forkall*().

Les fonctions forkx() et forkallx() acceptent un argument flags constitué d'un OU inclusif bit à bit de zéro ou plus des drapeaux suivants, qui sont définis dans l'en-tête sys/fork.h:

**FORK_NOSIGCHLD**
Ne postez pas de signal SIGCHLD au processus parent lorsque le processus enfant se termine, quelle que soit la disposition du signal SIGCHLD dans le parent. Les signaux SIGCHLD sont toujours possibles pour les actions d'arrêt et de poursuite du contrôle des tâches si le parent les a demandés.

**FORK_WAITPID**
Ne permettez pas que les wait-for-multiple-pids par le parent, comme dans wait(), waitid(P_ALL ), ou waitid(P_PGID), récolter l'enfant et ne permettez pas que l'enfant soit récolté automatiquement en raison de la disposition du signal SIGCHLD configuré pour être ignoré dans le parent. Seule une attente spécifique pour l'enfant, comme dans waitid (P_PID, pid), est autorisée et elle est requise, sinon lorsque l'enfant sortira, il restera un zombie jusqu'à ce que le parent quitte.

Si l'argument des flags est à 0, alors forkx() aura le même comportement que fork() et forkallx() aura le même comportement que forkall().

## Sources


https://man7.org/linux/man-pages/man2/clone.2.html

https://cpp.hotexamples.com/fr/examples/-/-/vfork/cpp-vfork-function-examples.html

https://man7.org/linux/man-pages/man2/vfork.2.html

https://www.ibm.com/docs/en/SSLTBW_2.4.0/com.ibm.zos.v2r4.bpxbd00/rvfork.htm

https://mindsgrid.com/difference-fork-vfork-exec-clone//

https://stackoverflow.com/questions/4856255/the-difference-between-fork-vfork-exec-and-clone

https://prograide.com/pregunta/11064/la-difference-entre-fork-vfork-exec-et-clone

http://www.unixguide.net/unix/programming/1.1.2.shtml

https://prograide.com/pregunta/12758/differences-entre-exec-et-fourche

https://man7.org/linux/man-pages/man2/fork.2.html

https://techdifferences.com/difference-between-fork-and-vfork.html

https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-vfork-create-new-process

https://man7.org/linux/man-pages/man2/fork.2.html

https://www.linuxjournal.com/article/5211

https://man7.org/linux/man-pages/man2/clone.2.html

http://www-igm.univ-mlv.fr/~dr/CS/node88.html

https://gist.github.com/nicowilliams/a8a07b0fc75df05f684c23c18d7db234

https://fresh2refresh.com/c-programming/c-buffer-manipulation-function/

https://stackoverflow.com/questions/66548922/can-a-fork-child-determine-whether-it-is-a-fork-or-a-vfork

https://news.ycombinator.com/item?id=30502392

https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man2/vfork.2.html

http://manpagesfr.free.fr/man/man2/clone.2.html

https://gist.github.com/alifarazz/d1ccf716131ed3a369fc7d248d910330

https://linux.die.net/man/2/clone

https://www.thegeekstuff.com/2012/05/c-mutex-examples/

https://docs.oracle.com/cd/E26502_01/html/E35303/gen-1.html

https://mindsgrid.com/difference-fork-vfork-exec-clone/

https://stackoverflow.com/questions/21205723/how-many-ways-we-can-create-a-process-in-linux-using-c

https://cpp.hotexamples.com/fr/examples/-/-/vfork/cpp-vfork-function-examples.html

https://man7.org/linux/man-pages/man2/vfork.2.html

https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-vfork-create-new-process

https://mindsgrid.com/difference-fork-vfork-exec-clone/

https://stackoverflow.com/questions/4856255/the-difference-between-fork-vfork-exec-and-clone

https://prograide.com/pregunta/11064/la-difference-entre-fork-vfork-exec-et-clone

http://www.unixguide.net/unix/programming/1.1.2.shtml

https://prograide.com/pregunta/12758/differences-entre-exec-et-fourche

https://techdifferences.com/difference-between-fork-and-vfork.html

https://manpages.ubuntu.com/manpages/hirsute/fr/man2/clone.2.html

http://manpagesfr.free.fr/man/man2/clone.2.html

https://github.com/jeremyong/google-coredumper/issues/14

https://stackoverflow.com/questions/29264322/mmap-error-on-linux-using-somethingelse

(comment programmer vfork)