

# fork() vfork() clone()

Léopold MOLS

1<sup>er</sup> avril 2022

Année : SYSG6 Q2 2021-2022  
Professeur : Mme BASTREGHI

## Table des matières

<b>1</b>	<b>Que sont-ils ?</b>	<b>2</b>
<b>2</b>	<b>fork()</b>	<b>2</b>
2.1	Qu'est-ce . . . . .	2
2.2	Historique . . . . .	2
2.3	Fonctionnement . . . . .	3
2.4	particularités . . . . .	3
2.5	Problèmes éventuels . . . . .	3
2.6	Suppléments . . . . .	4
2.7	Code . . . . .	4
<b>3</b>	<b>vfork()</b>	<b>6</b>
3.1	Qu'est-ce . . . . .	6
3.2	Historique . . . . .	6
3.3	Fonctionnement . . . . .	6
3.4	particularités . . . . .	7
3.5	Problèmes éventuels . . . . .	7
3.6	Code . . . . .	8
<b>4</b>	<b>clone()</b>	<b>11</b>
4.1	Qu'est-ce . . . . .	11
4.2	Historique . . . . .	12
4.3	Fonctionnement . . . . .	12
4.4	particularités . . . . .	13
4.5	Problèmes éventuels . . . . .	15
4.6	Code . . . . .	15
4.6.1	Code 1 . . . . .	15

4.6.2	Code 2 . . . . .	17
4.6.3	Code 3 . . . . .	19
4.6.4	Code 4 . . . . .	20
4.6.5	Code 1 . . . . .	21
<b>5</b>	<b>Results and Conclusions</b>	<b>23</b>
<b>6</b>	<b>Aller plus loin</b>	<b>23</b>
<b>7</b>	<b>Sources</b>	<b>24</b>

## 1 Que sont-ils ?

`fork()` (cfr 2.1 : *fork()*), `vfork()` (cfr 3.1 : *vfork()*), `clone()` (cfr 4.1 : *clone()*) permettent de créer des processus s'exécutant.

Au démarrage d'un système Unix, un seul processus existe (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`, `forkx()`, `forkall()`, `forkallx()`, `vforkx()` ou d'autres moyens sont des appels système standard d'UNIX (norme POSIX) permettant de créer des processus.

## 2 `fork()`

### 2.1 Qu'est-ce

`fork()` crée un nouveau processus en dupliquant le processus appelant.

Le nouveau processus est appelé *processus enfant*. L'appel processus est appelé *processus parent*. Le *processus enfant* et le *processus parent* s'exécutent dans une mémoire séparée. Au moment de **`fork()`**, les deux *espaces mémoire* ont le même contenu.

Écritures en mémoire, mappages de fichiers (`mmap()`) et démappages (`munmap()`) exécutés par l'un des processus n'affecte pas l'autre.

### 2.2 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).

## 2.3 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, aucun processus enfant n'est créé et `errno` est modifié pour indiquer l'erreur.

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.

## 2.4 particularités

- **L'espace d'adressage** est dupliqué  
Le **processus enfant** est une copie exacte du processus parent sauf pour
- \* L'enfant a son propre ID de processus unique, et ce PID est unique
- \* L'enfant n'hérite pas des verrous de mémoire de son parent
- \* La table des signaux est remise à 0 pour l'enfant
- \* L'enfant n'hérite pas des sémaphores de son parent
- \* L'enfant n'hérite pas des verrous d'enregistrement associés au processus parent
- \* L'enfant n'hérite pas des minuteries de son parent
- \* L'enfant n'hérite pas des E/S asynchrones en suspens ni des contextes d'E/S asynchrones de son parent

## 2.5 Problèmes éventuels

Au vu du fait que `fork` duplique l'espace d'adressage et d'autres fonctionnement du parent (comme les threads,...), cela peut vite remplir la mémoire, ralentir l'ordinateur et empêcher la création de nouveaux processus.

fork() 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.

## 2.6 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().

## 2.7 Code

Ce code effectue une addition de 2 variables par le fils pour prouver que les variables du père sont modifiées par le fils puisque le père et le fils partagent le même espace d'adressage.

---

```
1  #include <sys/types.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <pthread.h>
5  #include <mm_malloc.h>
6  #include <spawn.h>
7
8  int main(int argc, char **argv) {
9
10     int a = 5, b = 8;
11     int v;
12
13     /**
14      * Le FORK duplique l'espace d'adressage.
15      *
16      * Donc, le FORK fera les additions de son côté.
17      * Ensuite, le fils sera exit, donc,
18      * il n'affichera pas le résultat de ses additions
19      *
20      * Du côté du père, l'addition ne sera pas faite puisqu'il exécutera
21      * seulement le code suivant le "if"
22      *
23      * Vu que l'espace d'adressage est dupliqué lors du FORK, les
24      * variables ne seront modifiées que dans
25      * l'espace d'adressage du FILS.
26      * Donc, les variables du père ne sont pas modifiées
27      */
28     v = fork();
```

```

29     if(v == 0) {
30         // 10
31         a = a + 5;
32         // 10
33         b = b + 2;
34         exit(0);
35     }
36     // Parent code
37     wait(0);
38     printf("PID = %d\n", getpid());
39     printf("PPID = %d\n", getppid());
40     printf("Value of v is %d.\n", v); // line a
41     printf("Sum is %d.\n", a + b); // line b
42     sleep(4000);
43     printf("Let's do a ps to see which process is currently running !");
44     exit(0);
45 }
46
47 /*int main()
48 {
49     int a = 10, errFils;
50     printf ("Before Forking\n");
51     printf ("%d\n", a);
52     if ((errFils = fork()) == 0)
53     {
54         a = 20;
55         /**
56         * exit(0); // Remplacer par "wait(0)" pour montrer la différence
57         * pour démontrer que seul le fils exécutera la suite puisque c'est
58         * le même espace d'adressage, donc, la même TDFO, donc, une fois
59         * que la variable sera changée et que le fils se sera occupé de
60         * print sur la sortie standard, le père n'aura plus à le faire car
61         * stdout ne sera plus dans la TDFO.
62         */
63         /**
64         * wait est un processus bloquant. Donc, la suite ne sera pas
65         * exécutée tant qu'une condition ne sera pas remplie. Si l'on met
66         * un pointeur d'un nombre, alors, on pourra récupérer le code de
67         * terminaison du processus enfant. Pareil pour exit
68         */
69         /**
70         * Attention : le "exit(0)" le tue, mais ne l'enlève pas de la
71         * table des process et envoie un signal à son parent
72         */
73     }
74     wait(errFils);
75     printf ("After Forking\n");
76     printf ("%d\n", a);
77 }*/

```

---

## 3 `vfork()`

### 3.1 Qu'est-ce

`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()`.

### 3.2 Historique

L'appel système `vfork()` est apparu dans BSD 3.0. Dans BSD 4.4, il est devenu synonyme de `fork()`, mais NetBSD l'a réintroduit à nouveau : voir <http://www.netbsd.org/Documentation/kernel/vfork.html>.

Sous Linux, il fut l'équivalent de `fork()` 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.

### 3.3 Fonctionnement

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.

`vfork()`, tout comme `fork()`, crée un processus fils à partir du processus appelant. Pour plus de détails sur les valeurs renvoyées et les erreurs possibles, voir `fork()`. `vfork()` est conçu comme un cas particulier de `clone()`. Il sert à créer un nouveau processus sans effectuer de copie de la table des pages mémoire du processus père. Ceci peut être utile dans des applications nécessitant une grande rapidité d'exécution, si le fils doit invoquer immédiatement un appel `execve()`.

`vfork()` diffère aussi de `fork()` car le processus père reste suspendu jusqu'à ce que le fils invoque `execve()`, ou `_exit()`. Le fils partage toute la mémoire avec son père, y compris la pile, jusqu'à ce que `execve()` soit appelé par le fils. Le processus fils ne doit donc pas revenir de la fonction en cours, ni invoquer une nouvelle routine. Il ne doit pas appeler `exit(3)`, mais à la place `_exit()`.

Les gestionnaires de signaux sont hérités mais pas partagés. Les signaux pour le processus père sont délivrés après que le fils ait mis à jour la mémoire du père.

Sous Linux, `fork()` est implémenté en utilisant un mécanisme de copie en écriture, ainsi ses seuls coûts sont le temps et la mémoire nécessaire pour dupliquer la table des pages mémoire du processus père, et créer une structure de tâche pour le fils. Toutefois, jadis `fork()` nécessitait malheureusement une copie complète de l'espace d'adresse du père, souvent inutile car un appel `exec(3)` est souvent réalisé immédiatement par le fils. Pour améliorer les performances, BSD a introduit un appel

système `vfork()` qui ne copie pas l'espace d'adressage du père, mais emprunte au père son espace d'adressage et son fil de contrôle jusqu'à un appel à `execve()` ou `exit`. Le processus père était suspendu tant que le fils utilisait les ressources. L'utilisation de `vfork()` était loin d'être facile, car pour éviter de modifier les données du processus père, il fallait être capable de déterminer quelles variables se trouvaient dans des registres du processeur.

### 3.4 particularités

— **L'espace d'adressage** est dupliqué

**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`

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).

### 3.5 Problèmes éventuels

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 `ioctl` sont autorisées et des tentatives d'entrées indiqueront une fin de fichier. »

L'appel système `vfork()` échouera comme `fork()`. 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 des fonctions de `libc` qui ne font aucun appel système autre que ceux-là) est appelé après avoir appelé `vfork()`.

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)`

Lors de l'utilisation de `vfork()`, il arrive souvent que ce message apparaisse lors de la compilation, ce qui montre, par exemple, que l'exécution diffère d'un système à un autre : *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)`.*

Mais `vfork()` a les avantages de `fork`, et aucun de ses inconvénients ! `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.

### 3.6 Code

Ce code effectue une addition de 2 variables par le fils pour prouver que seules les variables du fils sont modifiées puisque le père et le fils ne partagent pas le même espace d'adressage.

---

```
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <pthread.h>
5 #include <mm_malloc.h>
6 #include <spawn.h>
```



```

7  #include <string.h>
8
9  /*int main()
10 {
11     int a = 10;
12     pid_t errFils = 0;
13     int * address = malloc(sizeof(int));
14     char buff[10];
15     printf ("\nBefore Forking\n");
16     printf ("Je suis le père\n");
17     printf ("a = %d\n", a);
18     printf ("Adress of malloc : %d\n", address);
19     printf ("Valeur de retour du vfork : %d\n", errFils);
20     printf ("PID = %d\n", getpid());
21     printf ("PPID = %d\n\n", getppid());
22     if ((errFils = vfork()) == 0)
23     {
24         a = 20;
25         printf ("Je suis le fils\n");
26         char * name = "Léo";
27         strcpy(buff, *name);
28         //printf ("PID = %d\n", getpid());
29         //printf ("PPID = %d\n", getppid());
30         exit(0); // Remplacer par "wait(0)" pour montrer la différence
31         pour démontrer que seul le fils exécutera la suite puisque c'est
32         le même espace d'adressage, donc, la même TDFO, donc, une fois que
33         la variable sera changée et que le fils se sera occupé de print
34         sur la sortie standard, le père n'aura pas à le faire car stdout
35         ne sera plus dans la TDFO.
36
37         // wait est un processus bloquant. Donc, la suite ne sera pas
38         exécutée tant qu'une condition ne sera pas remplie. Si l'on met un
39         pointeur d'un nombre, alors, on pourra récupérer le code de
40         terminaison du processus enfant. Pareil pour exit
41
42         // Attention : le "exit(0)" le tue, mais ne l'enlève pas de la
43         table des process et envoie un signal à son parent
44
45         // Si l'on met cette ligne en commentaire (celle avec le "exit(0)"
46         ou le "wait(0)"), alors, une erreur de segmentation sera renvoyée
47     }
48     //wait(errFils);
49     printf ("After Forking\n");
50     printf ("a = %d\n", a);
51     printf ("Adress of malloc : %d\n", address);
52     printf ("Valeur de retour du vfork : %d\n", errFils);
53     printf ("PID = %d\n", getpid());
54     printf ("PPID = %d\n\n", getppid());
55 }*/
56
57
58 int main(int argc, char **argv) {
59
60     int a = 5, b = 8;
61     int v;
62
63     /**
64      * Le VFORK duplique l'espace d'adressage.

```

```

65      * Donc, le VFORK fera les additions de son côté, mais dans l'espace
66      * d'adressage du PERE
67      * Ensuite, le fils sera exit, donc, il n'affichera pas le résultat
68      * de ses additions, mais les variables sont bien modifiées
69      *
70      * Du côté du père, l'addition sera pas faite puisqu'il partage
71      * l'espace d'adressage du FILS
72      *
73      * Vu que l'espace d'adressage n'est pas dupliqué lors du VFORK, les
74      * variables seront modifiées dans
75      * l'espace d'adressage du PERE (qui est aussi celui du fils)
76      * Donc, les variables du père sont modifiées,
77      * ce qui permet la prise en compte de la modification des valeurs
78      * des variables
79      */
80  v = vfork();
81  if(v == 0) {
82      // a = 10
83      a = a + 5;
84      // b = 10
85      b = b + 2;
86      exit(0);
87  }
88  // Parent code
89  wait(0);
90  printf("PID = %d\n", getpid());
91  printf("PPID = %d\n", getppid());
92  printf("Value of v is %d.\n", v); // line a
93  printf("Sum is %d.\n", a + b); // line b
94  sleep(4000);
95  printf("Let's do a ps to see which process is currently running !");
96  exit(0);
97  }

```

---

## 4 clone()

### 4.1 Qu'est-ce

**clone()** crée un nouveau processus. La fonction **clone()** a le même effet que **fork()**, sauf qu'il permet, si l'on le souhaite, de choisir si le processus enfant partage l'espace d'adressage du parent ou non.

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 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.

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.

Les *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 :

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

## 4.2 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.

## 4.3 Fonctionnement

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.

## 4.4 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 :

---

```
1 struct clone_args {
2     u64 flags;           /* Masque de bit d'attribut */
3     u64 pidfd;           /* Où stocker le descripteur de fichier
4         du PID
5         (pid_t *) */
6     u64 child_tid;       /* Où stocker le TID enfant,
7         dans la mémoire de l'enfant's memory (
8         pid_t *) */
9     u64 parent_tid;      /* Où stocker le TID enfant,
10        dans la mémoire du parent's memory (
11        int *) */
12    u64 exit_signal;      /* Signal à envoyer au parent quand
13        l'enfant se termine */
14    u64 stack;            /* Pointeur vers l'octet le plus faible
15        de la pile */
16    u64 stack_size;       /* Taille de la pile */
17    u64 tls;              /* Emplacement du nouveau TLS */
18    u64 set_tid;          /* Pointeur vers un tableau pid_t
19        (depuis Linux 5.5) */
20    u64 set_tid_size;     /* Nombre d'éléments dans set_tid
21        (depuis Linux 5.5) */
22    u64 cgroup;           /* Descripteur de fichier du cgroup cible
23        de l'enfant (depuis Linux 5.7) */
24};
```

---

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()`

Signal de fin de l'enfant Quand le processus enfant se termine, un signal peut

TABLE 1 – clone() vs clone3()

clone()	clone3()	Notes
—	Champ <code>cl_args</code>	
attributs & 0xff	attributs	Pour la plupart des attributs ; détails ci-dessous
parent_tid	pidfd	Voir <code>CLONE_PIDFD</code>
child_tid	child_tid	Voir <code>CLONE_CHILD_SETTID</code>
parent_tid	parent_tid	Voir <code>CLONE_PARENT_SETTID</code>
child_tid	child_tid	Voir <code>CLONE_CHILD_SETTID</code>
attributs & 0xff	exit_signal	
pile	pile	
—	stack_size	
tls	tls	Voir <code>CLONE_SETTLS</code>
—	set_tid	
—	set_tid_size	
—	cgroup	Voir <code>CLONE_INTO_CGROUP</code>

être envoyé au parent. Le signal de fin est indiqué dans l'octet de poids faible de flags (clone()) ou dans `cl_args.exit_signal` (clone3()). Si ce signal est différent de `SIGCHLD`, le processus parent doit également spécifier les options `__WALL` ou `__WCLONE` lorsqu'il attend la fin de l'enfant avec `wait(2)`. Si aucun signal n'est indiqué (donc zéro), le processus parent ne sera pas notifié de la terminaison de l'enfant.

— **L'espace d'adressage** est dupliqué

**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`

**Valeur de retour** 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.

## 4.5 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 :

---

```
1      #include <syscall.h>
2
3      pid_t mypid;
4
5      mypid = syscall(SYS_getpid);
```

---

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.

## 4.6 Code

Voici 5 codes qui peuvent ne pas s'exécuter de la même manière suivant l'OS, la machine et le compilateur

### 4.6.1 Code 1

---

```
1  #define _GNU_SOURCE
2      #include <sys/wait.h>
3      #include <sys/utsname.h>
4      #include <sched.h>
5      #include <string.h>
6      #include <stdint.h>
7      #include <stdio.h>
8      #include <stdlib.h>
9      #include <unistd.h>
10     #include <sys/mman.h>
```

```

11
12     #define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
13                             } while (0)
14
15     static int              /* Start function for cloned child */
16     childFunc(void *arg)
17     {
18         struct utsname uts;
19
20         /* Change hostname in UTS namespace of child. */
21
22         if (sethostname(arg, strlen(arg)) == -1)
23             errExit("sethostname");
24
25         /* Retrieve and display hostname. */
26
27         if (uname(&uts) == -1)
28             errExit("uname");
29         printf("uts.nodename in child: %s\n", uts.nodename);
30
31         /* Keep the namespace open for a while, by sleeping.
32            This allows some experimentation—for example, another
33            process might join the namespace. */
34
35         sleep(200);
36
37         return 0;           /* Child terminates now */
38     }
39
40     #define STACK_SIZE (1024 * 1024)    /* Stack size for cloned child
41     */
42
43     int
44     main(int argc, char *argv[])
45     {
46         char *stack;          /* Start of stack buffer */
47         char *stackTop;       /* End of stack buffer */
48         pid_t pid;
49         struct utsname uts;
50
51         if (argc < 2) {
52             fprintf(stderr, "Usage: %s <child-hostname>\n", argv[0]);
53             exit(EXIT_SUCCESS);
54         }
55
56         /* Allocate memory to be used for the stack of the child. */
57
58         stack = mmap(NULL, STACK_SIZE, PROT_READ | PROT_WRITE,
59                     MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
60         if (stack == MAP_FAILED)
61             errExit("mmap");
62
63         stackTop = stack + STACK_SIZE; /* Assume stack grows downward
64     */
65
66         /* Create child that has its own UTS namespace;
67            child commences execution in childFunc(). */

```



```

66         pid = clone(childFunc, stackTop, CLONE_NEWUTS | SIGCHLD, argv
67         [1]);
68         if (pid == -1)
69             errExit("clone");
70         printf("clone() returned %jd\n", (intmax_t) pid);
71         /* Parent falls through to here */
72
73         sleep(1);          /* Give child time to change its hostname
74         */
75         /* Display hostname in parent's UTS namespace. This will be
76         different from hostname in child's UTS namespace. */
77
78         if (uname(&uts) == -1)
79             errExit("uname");
80         printf("uts.nodename in parent: %s\n", uts.nodename);
81
82         if (waitpid(pid, NULL, 0) == -1)    /* Wait for child */
83             errExit("waitpid");
84         printf("child has terminated\n");
85
86         exit(EXIT_SUCCESS);
87     }

```

---

#### 4.6.2 Code 2

---

```

1  #define _GNU_SOURCE
2  #include <sched.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8
9  #define STACK_SIZE (1024 * 1024) /* Stack size for cloned child */
10 #define ERREXIT(msg)
11     \
12     {
13         \
14         perror(msg);
15         \
16         exit(EXIT_FAILURE);
17     }
18 #define CHECKALLOC(ptr, msg)
19     \
20     ({
21         \
22         void *p = ptr;
23         \
24         if (p == NULL)
25             \
26             ERREXIT(msg);
27     }

```

```

20     p;
21     \
22 })
23 static int
24 childFunc(void *arg __attribute__((unused)))
25 {
26     puts("child: start");
27     sleep(2);
28     puts("child: terminate");
29     return 0; /* Child terminates now */
30 }
31
32 int
33 main(int argc, char *argv[])
34 {
35     char **stacks; /* Start of stack buffer */
36     pid_t *pids; /* Child process's pids */
37     size_t nproc, i;
38
39     if (argc != 2) {
40         puts("Wrong way to execute the program:\n"
41             "\t\t./waitpid nProcesses\n"
42             "example:\t\t./waitpid 2");
43
44         return EXIT_FAILURE;
45     }
46
47     nproc = atoi(argv[1]); /* Process count */
48
49     stacks = CHECKALLOC(malloc(nproc * sizeof(void *)), "malloc");
50     pids = CHECKALLOC(malloc(nproc * sizeof(pid_t)), "malloc");
51
52     for (i = 0; i < nproc; i++) {
53         char *stackTop; /* End of stack buffer */
54         stacks[i] = CHECKALLOC(malloc(STACK_SIZE), "stack malloc");
55         stackTop = stacks[i] + STACK_SIZE;
56
57         pids[i] = clone(childFunc, stackTop,
58             CLONE_CHILD_CLEARTID | CLONE_CHILD_SETTID | SIGCHLD,
59             NULL);
60         if (pids[i] == -1)
61             ERREXIT("clone");
62         printf("clone() returned %ld\n", (long)pids[i]);
63     }
64
65     sleep(1);
66
67     for (i = 0; i < nproc; i++) {
68         if (waitpid(pids[i], NULL, 0) == -1)
69             ERREXIT("waitpid");
70         printf("child %ld has terminated\n", (long)pids[i]);
71     }
72
73     for (i = 0; i < nproc; i++)
74         free(stacks[i]);
75     free(stacks);
76     free(pids);

```

```
76     return EXIT_SUCCESS;
77 }
```

---

### 4.6.3 Code 3

---

```
1  #define _GNU_SOURCE
2
3  #include <sched.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/wait.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10 #include <sched.h>
11
12 #define SIZESTACK ( 1024 * 1024 )
13
14 int child_proc(void *arg)
15 {
16     printf ("Child process arg = %s\n", (char *) arg);
17     return 0;
18 }
19
20 int main(int argc, char *argv[])
21 {
22     char *stack;
23     char *stackhead;
24     char *str = "Hello world! 'n";
25     pid_t pid;
26     int status;
27     stack = (char *) malloc (SIZESTACK);
28     if (!stack)
29     {
30         fprintf (stderr, "Unable to allocate stack. \n") ;
31         exit(1);
32     }
33     stackhead = stack + SIZESTACK - 1;
34
35     /* pid = clone(child_proc, stackhead, SIGCHLD, str): */
36     pid = clone(child_proc, stackhead, CLONE_VM | CLONE_FS | CLONE_FILES |
37         CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |
38         CLONE_CHILD_CLEARTID | CLONE_SYSVSEM | CLONE_DETACHED , str);
39
40     if (pid == -1)
41     {
42         fprintf (stderr, "Unable to clone in");
43         free(stack);
44         exit(1);
45     }
46     printf ("Child pid = %d n", pid);
47     waitpid(pid, &status, 0);
48     printf("Done!In");
49     sleep(2);
```

```
48     return 0;
49 }
```

---

#### 4.6.4 Code 4

---

```
1  #define _GNU_SOURCE
2  #include <sched.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8
9  #define STACK_SIZE (1024 * 1024) /* Stack size for cloned child */
10 #define ERREXIT(msg)
11     {
12         perror(msg);
13         exit(EXIT_FAILURE);
14     }
15 #define CHECKALLOC(ptr, msg)
16     ({
17         void *p = ptr;
18         if (p == NULL)
19             ERREXIT(msg);
20         p;
21     })
22
23 static int
24 childFunc(void *arg __attribute__((unused)))
25 {
26     puts("child: start");
27     sleep(2);
28     puts("child: terminate");
29     return 0; /* Child terminates now */
30 }
31
32 int
33 main(int argc, char *argv[])
34 {
35     char **stacks; /* Start of stack buffer */
36     pid_t *pids; /* Child process's pids */
37     size_t nproc, i;
38
39     if (argc != 2) {
```

```

40     puts("Wrong way to execute the program:\n"
41          "\t\t./waitpid nProcesses\n"
42          "example:\t./waitpid 2");
43
44     return EXIT_FAILURE;
45 }
46
47 nproc = atoi(argv[1]); /* Process count */
48
49 stacks = CHECKALLOC(malloc(nproc * sizeof(void *)), "malloc");
50 pids = CHECKALLOC(malloc(nproc * sizeof(pid_t)), "malloc");
51
52 for (i = 0; i < nproc; i++) {
53     char *stackTop; /* End of stack buffer */
54     stacks[i] = CHECKALLOC(malloc(STACK_SIZE), "stack malloc");
55     stackTop = stacks[i] + STACK_SIZE;
56
57     pids[i] = clone(childFunc, stackTop, CLONE_CHILD_CLEARTID |
58                  CLONE_CHILD_SETTID | SIGCHLD, NULL);
59     if (pids[i] == -1)
60         ERREXIT("clone");
61     printf("clone() returned %ld\n", (long)pids[i]);
62 }
63
64 sleep(1);
65
66 for (i = 0; i < nproc; i++) {
67     if (waitpid(pids[i], NULL, 0) == -1)
68         ERREXIT("waitpid");
69     printf("child %ld has terminated\n", (long)pids[i]);
70 }
71
72 for (i = 0; i < nproc; i++)
73     free(stacks[i]);
74 free(stacks);
75 free(pids);
76 return EXIT_SUCCESS;
77 }

```

---

#### 4.6.5 Code 1

---

```

1  #define _GNU_SOURCE
2  #include <sched.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8
9  #define STACK_SIZE (1024 * 1024) /* Stack size for cloned child */
10 #define ERREXIT(msg)
11     {
12         \

```

```

12     perror(msg);
13     exit(EXIT_FAILURE);
14 }
15 #define CHECKALLOC(ptr, msg)
16 ({
17     void *p = ptr;
18     if (p == NULL)
19         ERREXIT(msg);
20     p;
21 })
22
23 static int
24 childFunc(void *arg __attribute__((unused)))
25 {
26     puts("child: start");
27     sleep(2);
28     puts("child: terminate");
29     return 0; /* Child terminates now */
30 }
31
32 int main(int argc, char *argv[])
33 {
34     char **stacks; /* Start of stack buffer */
35     pid_t *pids; /* Child process's pids */
36     size_t nproc, i;
37
38     if (argc != 2) {
39         puts("Wrong way to execute the program:\n"
40             "\t\t./waitpid nProcesses\n"
41             "example:\t./waitpid 2");
42
43         return EXIT_FAILURE;
44     }
45
46     nproc = atoi(argv[1]); /* Process count */
47
48     stacks = CHECKALLOC(malloc(nproc * sizeof(void *)), "malloc");
49     pids = CHECKALLOC(malloc(nproc * sizeof(pid_t)), "malloc");
50
51     for (i = 0; i < nproc; i++) {
52         char *stackTop; /* End of stack buffer */
53         stacks[i] = CHECKALLOC(malloc(STACK_SIZE), "stack malloc");
54         stackTop = stacks[i] + STACK_SIZE;
55
56         pids[i] = clone(childFunc, stackTop, CLONE_CHILD_CLEARTID |
57             CLONE_CHILD_SETTID | SIGCHLD, NULL);
58         if (pids[i] == -1)
59             ERREXIT("clone");
60         printf("clone() returned %ld\n", (long)pids[i]);
61     }

```

```

61
62     sleep(1);
63
64     for (i = 0; i < nproc; i++) {
65         if (waitpid(pids[i], NULL, 0) == -1)
66             ERREXIT("waitpid");
67         printf("child %ld has terminated\n", (long)pids[i]);
68     }
69
70     for (i = 0; i < nproc; i++)
71         free(stacks[i]);
72     free(stacks);
73     free(pids);
74     return EXIT_SUCCESS;
75 }

```

---

## 5 Results and Conclusions

The atomic weight of magnesium is concluded to be  $24 \text{ g mol}^{-1}$ , as determined by the stoichiometry of its chemical combination with oxygen. This result is in agreement with the accepted value.

FIGURE 1 – Figure caption.

## 6 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 `filedescriptor` 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'entête `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().

## 7 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://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>  
<https://news.ycombinator.com/item?id=30502392>  
[https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages\\_iPhoneOS/man2/vfork.2.html](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://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>  
<https://cpp.hotexamples.com/fr/examples/-/-/vfork/cpp-vfork-function-examples.html>



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>