fork()
vfork()
clone()

Léopold Mols

4 mai 2022

Année : SYSG6 Q2 2021-2022
Professeur : Mme Bastreghi

Table des matières

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 Déclaration de fork(2)

```
1 #include <unistd.h>
2
3 pid_t fork(void);
```

2.3 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).

2.4 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.

2.5 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

2.6 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.

2.7 Suppléments

fork1(), forkall(), forkx(), forkallx() Les fonctions forkx() et forkallx() acceptent un argument flags composé d'un OU inclusif au niveau du bit de zéro ou plusieurs des drapeaux suivants, qui sont définis dans l'en-tête sys/fork.h Si l'argument flags est 0, forkx() est identique à fork() et forkallx() est identique à forkall().

2.8 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.

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

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

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

    /**
    * Le FORK duplique l'espace d'adressage.
    *
    * Donc, le FORK fera les additions de son côté.
    *  Ensuite, le fils sera exit, donc,
    *  il n'affichera pas le résultat de ses additions
    *
    * Du côté du père, l'addition ne sera pas faite puisqu'il exécutera
    * seulement le code suivant le "if"
    *
    * Vu que l'espace d'adressage est dupliqué lors du FORK, les
    * variables ne seront modifiées que dans
    * l'espace d'adressage du FILS.
    *  Donc, les variables du père ne sont pas modifiées
    */
    v = fork();
    if(v == 0) {
        // 10
        a = a + 5;
        // 10
        b = b + 2;
        exit(0);
    }
    // Parent code
    wait(0);
    printf("PID = %d\n", getpid());
    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 currenlty
    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 pus à le
    faire car * stdout ne sera plus dans la TDFO.
61          */
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          /**
71          * Attention : le "exit(0)" le tue, mais ne l'enlève pas
```

```
    de la
72          * table des process et envoie un signal à son parent
73          */
74      }
75      wait(errFils);
76      printf ("After Forking\n");
77      printf ("%d\n", a);
78  }*/
```

2.9 Resultat

```
1  a = 5
2  b = 8
3  Ceci est le process parent et le PID est : 31151
4  I'm the child !
5  Now, a = 10 only in the child
6  Now, b = 10 only in the child
7  PID = 31152
8  PPID = 31151
9  Sum a + b is 20.
10 Let's do a ps to see which process is currenlty running !
11
12 CODES D'ÉTAT DE PROCESSUS
13 Voici les différentes valeurs que les indicateurs de sortie s,
      stat et state (en-tête STAT ou S ) afficheront
      pour décrire l'état d'un processus :
14
15 D    en sommeil non interruptible (normalement entrées et
      sorties) ;
16 R    s'exécutant ou pouvant s'exécuter (dans la file d'exécution
      ) ;
17 S    en sommeil interruptible (en attente d'un événement pour
      finir) ;
18 T    arrêté, par un signal de contrôle des tâches ou parce qu'il
       a été tracé ;
19 W    pagination (non valable depuis le noyau 2.6.xx) ;
20 X    tué (ne devrait jamais être vu) ;
```

```
21 Z    processus zombie (<defunct>), terminé mais pas détruit par
       son parent.
22
23 Pour les formats BSD et quand le mot-clé stat est utilisé, les
       caractères supplémentaires suivants peuvent être affichés :
24
25 <    haute priorité (non poli pour les autres utilisateurs) ;
26 N    basse priorité (poli pour les autres utilisateurs) ;
27 L    avec ses pages verrouillées en mémoire (pour temps réel et
       entrées et sorties personnalisées) ;
28 s    meneur de session ;
29 l    possède plusieurs processus légers (  multi-thread  ,
       utilisant CLONE\_THREAD comme NPTL pthreads le fait) ;
30 +    dans le groupe de processus au premier plan.
31
32 PID = 31151
33 PPID = 28756
34 Value of f is 31152.
35 Sum is 13.
36 Terminated
```

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 Déclaration de vfork(2)

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 pid_t vfork(void);
```

3.3 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.

3.4 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.

Donc semblable à l'appel système fork(), vfork() crée également un processus enfant identique à son processus parent. Cependant, le processus enfant suspend temporairement le processus parent jusqu'à ce qu'il se termine. En effet, les deux processus utilisent le même espace d'adressage, qui contient la pile, le pointeur de pile et le pointeur d'instruction.

3.5 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).

3.6 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 iotcl sont autorisées et des tentatives d'entrées indiqueront une fin de fichier. »

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() ou _exit(). (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.

3.7 Code prouvant l'espace d'adressage partagé

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  #include <unistd.h>
9  #include <pthread.h>
10
11 /**
12  * Le VFORK duplique l'espace d'adressage.
13  * Donc, le VFORK fera les additions de son côté, mais dans l'
      espace d'adressage du PERE
14  *  Ensuite, le fils sera exit, donc, il n'affichera pas le ré
      sultat de ses additions,
15  *  mais les variables sont bien modifiées
16  *
17  * Du côté du père, l'addition ne sera pas faite puisqu'il
      partage l'espace d'adressage du FILS
18  *
19  * Vu que l'espace d'adressage n'est pas dupliqué lors du VFORK,
       les variables seront modifiées dans
20  *  l'espace d'adressage du PERE (qui est aussi celui du fils)
21  *  Donc, les variables du père sont modifiées,
22  *   ce qui permet la prise en compte de la modification des
      valeurs des variables faites par le fils
23  */
24 int main(int argc, char **argv) {
25
26     // Entiers à augmenter dans le fils pour prouver l'espace d'
      adressage commun
```

```
27      int a = 5, b = 8;
28      // Récupérer la valeur de retour de la fonction créant le
        proocess fils
29      int vforkRetNum;
30
31      printf("a = %d\n", a);
32      printf("b = %d\n", b);
33
34      vforkRetNum = vfork();
35
36      if(vforkRetNum == 0) { // La création du fils s'est-elle
        correctement produite ?
37          printf("I'm the child !\n");
38          // a = 10
39          a = a + 5;
40          printf("Now, a = %d in the parent as in the child\n", a)
        ;
41
42          // b = 10
43          b = b + 2;
44          printf("Now, b = %d in the parent as in the child\n", b)
        ;
45
46          printf("PID = %d\n", getpid());
47          printf("PPID = %d\n", getppid());
48  //          printf("Value of vfork is %d.\n", vforkRetNum); //
        Indiquer la valeur de retour de la fonciton$          printf("
        Sum a + b is %d.\n", a + b); // line b
49          printf("Let's do a ps to see which process is currenlty
        running !\n\n");
50          printf("CODES D'ÉTAT DE PROCESSUS \nVoici les diffé
        rentes valeurs que les indicateurs de sortie$
51
52                  *D    en sommeil non interruptible (normalement
        entrées et sorties) ;\n"
53                  *R    s'exécutant ou pouvant s'exécuter (dans la
        file d'exécution) ;\n"
54                  *S    en sommeil interruptible (en attente d'un é
        vénement pour finir) ;\n"
55                  *T    arrêté, par un signal de contrôle des tâ
        ches ou parce qu'il a été tracé ;\n"
```

```
56                *W    pagination (non valable depuis le noyau
      2.6.xx) ;\n"
57                *X    tué (ne devrait jamais être vu) ;\n"
58                *Z    processus zombie (<defunct>), terminé mais
      pas détruit par son parent.\n\n"
59
60        "Pour les formats BSD et quand le mot-clé stat est utilis
      é, les caractères supplémentaires suiva$
61
62                *<    haute priorité (non poli pour les autres
      utilisateurs) ;\n"
63                *N    basse priorité (poli pour les autres
      utilisateurs) ;\n"
64                *L    avec ses pages verrouillées en mémoire (
      pour temps réel et entrées et sorties pers$
65                *s    meneur de session ;\n"
66                *l    possède plusieurs processus légers (
      multi-thread   , utilisant CLONE\_THREAD comme$
67                *+    dans le groupe de processus au premier plan
      .\n\n");
68          while(1){} // Faire en sorte que le fils attende, mais
      en étant en état d'exécution. Un simple $
69          exit(0);
70       }
71       else if (vforkRetNum > 0)
72       {  // Est-ce le process parent ?
73          printf("Ceci est le process parent et le PID est : %d\n"
      , getpid());
74       }
75       else
76       { // Y a-t-il eu une erreur lors de la création du process
      fils ?
77          printf("Problème durant le vfork\n");
78          exit(EXIT_FAILURE);
79       }
80       wait(0); // Pour éviter de faire du fils un zombie
81       printf("PID = %d\n", getpid());
82       printf("PPID = %d\n", getppid());
83       printf("Value of vfork is %d.\n", vforkRetNum); // Indiquer
      la valeur de retour de la fonction créa$
84       // La somme est bien de 20 et non plus ni moins puisque la
```

```
    somme fut faite par le fils avec les mêm$
85      printf("Sum a + b is %d.\n", a + b);
86      printf("Let's do a ps to see which process is currenlty
    running !\n");
87      while(1){} // Simplement pour faire attendre le père que l'
    on fasse un ps pour pouvoir voir son état
88      exit(0);
89  }
```

3.8 Résultat

```
1  a = 5
2  b = 8
3  I'm the child !
4  Now, a = 10 in the parent as in the child
5  Now, b = 10 in the parent as in the child
6  PID = 2216
7  PPID = 2215
8  Sum a + b is 20.
9  Let's do a ps to see which process is currenlty running !
10
11 CODES D'ÉTAT DE PROCESSUS
12 Voici les différentes valeurs que les indicateurs de sortie s,
     stat et state (en-tête STAT ou S ) afficheront
     pour décrire l'état d'un processus :
13
14 D    en sommeil non interruptible (normalement entrées et
     sorties) ;
15 R    s'exécutant ou pouvant s'exécuter (dans la file d'exécution
     ) ;
16 S    en sommeil interruptible (en attente d'un événement pour
     finir) ;
17 T    arrêté, par un signal de contrôle des tâches ou parce qu'il
      a été tracé ;
18 W    pagination (non valable depuis le noyau 2.6.xx) ;
19 X    tué (ne devrait jamais être vu) ;
```

```
20 Z    processus zombie (<defunct>), terminé mais pas détruit par
       son parent.
21
22 Pour les formats BSD et quand le mot-clé stat est utilisé, les
       caractères supplémentaires suivants peuvent être affichés :
23
24 <    haute priorité (non poli pour les autres utilisateurs) ;
25 N    basse priorité (poli pour les autres utilisateurs) ;
26 L    avec ses pages verrouillées en mémoire (pour temps réel et
       entrées et sorties personnalisées) ;
27 s    meneur de session ;
28 l    possède plusieurs processus légers (   multi-thread   ,
       utilisant CLONE\_THREAD comme NPTL pthreads le fait) ;
29 +    dans le groupe de processus au premier plan.
30
31 Ceci est le process parent et le PID est : 2215
32 PID = 2215
33 PPID = 28756
34 Value of vfork is 2216.
35 Sum a + b is 20.
36 Let's do a ps to see which process is currenlty running !
37 Terminated
```

3.9 Code prouvant que le parent est mis en pause

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 /**
7  vfork() affiche le contenu du 'if{} else{}' deux fois, d'abord
      dans l'enfant, puis dans le parent.
8  Vu que les deux processus partagent le même espace d'adressage,
      la première sortie contient la valeur $
```

```
9   Dans le bloc if else, le processus enfant est exécuté en
      premier car il bloque le processus parent lor$
10  */
11 int main()
12 {
13     pid_t pid = vfork();  //creating the child process
14
15     printf("Process parent avant le 'if{} else{}': %d\n", getpid
       ());
16
17     if (pid == 0)
18     {  // Est-ce le process fils ?
19         printf("Ceci est le process fils et le PID est : %d\n\n"
       , getpid());
20         exit(0);
21     }
22     else if (pid > 0)
23     {  // Est-ce le process parent ?
24         printf("Ceci est le process parent et le PID est : %d\n"
       , getpid());
25     }
26     else
27     { // Y a-t-il eu une erreur lors de la création du process
       fils ?
28         printf("Problème durant le fork\n");
29         exit(EXIT_FAILURE);
30     }
31     return 0;
32 }
```

3.10 Résultat

```
1 Process parent avant le 'if{} else{}': 6666
2 Ceci est le process fils et le PID est : 6666
3
4 Process parent avant le 'if{} else{}': 6664
5 Ceci est le process parent et le PID est : 6664
```

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(), cet appel système fournit un contrôle plus précis sur les éléments de contexte d'exécution. Il est partagé entre le processus appelant et le processus enfant. Cet appel système permet également au nouveau processus enfant d'être placé dans des espaces de noms.

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 *stack 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 *stack 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/clon

4.2 Déclaration de clone(2)

```
1 #define _GNU_SOURCE
2 #include <sched.h>
```

```
3
4 int clone(int (*fn)(void *), void *stack, int flags, void *arg,
      ...
5 /* pid_t *parent_tid, void *tls, pid_t *child_tid */ );
```

4.3 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.

4.4 Fonctionnement

Quand le processus enfant est créé par la fonction wrapper 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.

4.5 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 du PID
4                                       (pid_t *) */
5                  u64 child_tid;    /* Où stocker le TID enfant,
6                                       dans la mémoire de l'enfant'
       s memory (pid_t *) */
7                  u64 parent_tid;   /* Où stocker le TID enfant,
8                                       dans la mémoire du parent's
       memory (int *) */
9                  u64 exit_signal;  /* Signal à envoyer au parent
       quand
10                                      l'efant se termine */
11                 u64 stack;        /* Pointeur vers l'octet le
       plus faible de la pile */
12                 u64 stack_size;   /* Taille de la pile */
13                 u64 tls;          /* Emplacement du nouveau TLS
```

```
    */
14              u64 set_tid;        /* Pointeur vers un tableau
    pid_t
15                                     (depuis Linux 5.5) */
16              u64 set_tid_size; /* Nombre d'éléments dans
    set_tid
17                                     (depuis Linux 5.5) */
18              u64 cgroup;         /* Descripteur de fichier du
    cgroup cible
19                                     de l'enfant (depuis Linux
    5.7) */
20          };
```

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

TABLE 1 – clone() vs clone3()

clone()	clone3()	Notes
—	Champ cl_args	
attributs & 0xff	attributs	Pour la plupart des attributs ; détails
parent_tid	pidfd	Voir CLONE_PIDFD
child_tid	child_tid	Voir CLONE_CHILD_SETT
parent_tid	parent_tid	Voir CLONE_PARENT_SET
child_tid	child_tid	Voir CLONE_CHILD_SETT
attributs & 0xff	exit_signal	
pile	pile	
—	stack_size	
tls	tls	Voir CLONE_SETTLS
—	set_tid	
—	set_tid_size	
—	cgroup	Voir CLONE_INTO_CGROU

Quand le processus enfant se termine, un signal peut 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.

4.6 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.

4.7 Code

```
1  // Il est nécessaire de définir \_GNU_SOURCE pour avoir accès à
       clone(2) et CLONE\_*
2
3  #define _GNU_SOURCE
4  #include <sched.h>
5  #include <sys/syscall.h>
6  #include <sys/wait.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11
12 /**
13  child
14  */
15 static int child_func(void* arg) {
16      char* buffer = (char*)arg;
17      printf("Child sees buffer = \"%s\"\n", buffer);
18      strcpy(buffer, "hello from child");
19      return 0;
20 }
21
22 /**
23  Ici, clone() est utilisé de deux manières, une fois avec le
       drapeau CLONE_VM et une fois sans.
24  Un buffer est passé dans le processus enfant, et le processus
       enfant y écrit une chaîne.
25  Une taille de pile est ensuite allouée pour le processus enfant
       et une fonction qui vérifie si nous ex$
```

```
26  De plus, un buffer de 100 octets est créé dans le processus
      parent et une chaîne y est copiée, puis, l$
27
28  Lorsque d'une exécution sans l'argument vm,
29  le drapeau CLONE\_VM n'est pas actif et la mémoire virtuelle du
      processus parent est clonée dans le pro$
30  Le processus enfant peut accéder au message passé par le
      processus parent dans le tampon,
31  mais tout ce qui est écrit dans le tampon par l'enfant n'est
      pas accessible par processus parent.
32  */
33 int main(int argc, char** argv) {
34      // Alloue un stack pour la tâche du fils
35      const int STACK_SIZE = 65536;
36      char* stack = malloc(STACK_SIZE);
37      if (!stack) { // Si 'stack' n'a pas été correctement créé
38          perror("malloc");
39          exit(1);
40      }
41
42      // Lorsqu'il est appelé avec l'argument "vm" en ligne de
    commande, active le flag CLONE_VM.
43      unsigned long flags = 0;
44      if (argc > 1 && !strcmp(argv[1], "vm")) {
45
46          /**
47           int clone(int (*fn)(void *), void *child_stack,
48                     int flags, void *arg, ...
49                     pid_t *ptid, struct user_desc *tls, pid_t *
    ctid );
50           */
51
52          /**
53           Lorsque le processus enfant est créé avec clone(), il
    exécute la fonction fn(arg).
54           (Cela diffère de fork(2), où l'exécution continue dans
    le fils à partir du point d'appel de fo$
55           L'argument fn est un pointeur vers une fonction qui est
     appelée par le processus fils au début$
56           */
57          /**
```

```
58          CLONE_VM (depuis Linux 2.0)
59                      Si CLONE_VM est défini, le parent et l'
    enfant
60                      seront exécuté dans le même espace mémoire
    . En particulier,
61                      les écritures mémoire effectuées par le
    parent ou par l'
62                      enfant sont également visibles dans l'
    autre processus.
63                      De plus, tout mappage ou démappage de mé
    moire effectué avec
64                      mmap(2) ou munmap(2) par le processus
    enfant ou appelant également
65                      affecte l'autre processus.
66                      Si CLONE_VM n'est pas défini, le processus
     enfant s'exécute dans un
67                      copie séparée de l'espace mémoire du
    processus appelant
68                      au moment de l'appel de clone. Les é
    critures effectuées par
69                      les mappages/démappages effectués par la m
    émoire de l'un des processus ne
70                      n'affecte pas l'autre, comme avec fork(2).
71
72                      Si l'indicateur CLONE_VM est spécifié et l
    'indicateur CLONE_VFORK
73                      n'est pas spécifié, alors toute autre pile
     de signaux qui a été
74                      établie par sigaltstack(2) est effacée
    dans l'enfant.
75            */
76          flags |= CLONE\_VM; // 'flags' vaudra 'CLONE\_VM' ou non
77      }
78
79      char buffer[100];
80      strcpy(buffer, "hello from parent"); // Ecrit 'hello from
    parent' dans le buffer
81      // Clone le processus père
82      // Seul appel à 'clone'. Pour avoir les différentes exé
    cutions, il faut ajouter 'vm' comme argument$
```

```
83      // Vu que lorsque CLONE\_VM est défini, l'espace d'adressage
        mémoiire est partaé,
84      // le buffer est le même pour le père et pour le fils, donc,
        le fils override ce que le père a écri$
85      if (clone(child\_func, stack + STACK_SIZE, flags | SIGCHLD,
        buffer) == -1) {
86          perror("clone");
87          exit(1);
88      }
89
90      int status;
91      if (wait(&status) == -1) {
92          perror("wait");
93          exit(1);
94      }
95
96      printf("Child exited with status %d. buffer = \"%s\"\n",
        status, buffer);
97      return 0;
98  }
```

4.8 Résultats

4.8.1 ./clone

```
1 Child sees buffer = "hello from parent"
2 Child exited with status 0. buffer = "hello from parent"
```

4.8.2 ./clone vm

```
1 Child sees buffer = "hello from parent"
2 Child exited with status 0. buffer = "hello from child"
```

5 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'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().

6 Sources

https ://man7.org/linux/man-pages/man2/clone.2.htmll
https : ://cpp.hotexamples.com/fr/examples/-/-/vfork/cpp-vfork-function-examples.htmll
https : ://man7.org/linux/man-pages/man2/vfork.2.htmll
https : ://www.ibm.com/docs/en/SSLTBW$_2$.4.0/*com.ibm.zos.v2r4.bpxbd*
https :: //mindsgrid.com/difference − fork − vfork − exec − clone/
*https :: //stackoverflow.com/questions/*4856255/*the − difference − b*
*https :: //prograide.com/pregunta/*11064/*la − difference − entre − f*
*http : //www.unixguide.net/unix/programming/*1.1.2.*shtml*

https :: //prograide.com/pregunta/12758/differences − entre − exec
https :: //man7.org/linux/man − pages/man2/fork.2.html
https :: //techdifferences.com/difference − between − fork − and −
https :: //www.ibm.com/docs/en/zos/2.4.0?topic = functions − vfork
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/a8a07b0fc75df05f684c23c18d7
https :: //fresh2refresh.com/c − programming/c − buffer − manipu
https :: //stackoverflow.com/questions/66548922/can − a − fork − ch
https :: //news.ycombinator.com/item?id = 30502392
https :: //developer.apple.com/library/archive/documentation/Systen
http : //manpagesfr.free.fr/man/man2/clone.2.html
https :: //gist.github.com/alifarazz/d1ccf716131ed3a369fc7d248d9103
https :: //linux.die.net/man/2/clone
https :: //www.thegeekstuff.com/2012/05/c − mutex − examples/
https :: //docs.oracle.com/cd/E26502$_0$1/html/E35303/gen − 1.html
https :: //mindsgrid.com/difference − fork − vfork − exec − clone/
https :: //stackoverflow.com/questions/21205723/how − many − way
https :: //cpp.hotexamples.com/fr/examples/ − / − /vfork/cpp − vf
https :: //man7.org/linux/man − pages/man2/vfork.2.html
https :: //www.ibm.com/docs/en/zos/2.4.0?topic = functions − vfork
https :: //mindsgrid.com/difference − fork − vfork − exec − clone/
https :: //stackoverflow.com/questions/4856255/the − difference − b
https :: //prograide.com/pregunta/11064/la − difference − entre − f
http : //www.unixguide.net/unix/programming/1.1.2.shtml

$https :: //prograide.com/pregunta/12758/differences - entre - exec$
$https :: //techdifferences.com/difference - between - fork - and -$
$https :: //manpages.ubuntu.com/manpages/hirsute/fr/man2/clone.2$
$http : //manpagesfr.free.fr/man/man2/clone.2.html$
$https :: //github.com/jeremyong/google - coredumper/issues/14$
$https :: //stackoverflow.com/questions/29264322/mmap - error - on$