fork()
vfork()
clone()

Léopold Mols

13 août 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 (si l'on atteint le nombre maximum de processus s'exécutant pour un même utilisateur (le kernel n'a pas de limite), si la mémoire est pleine et ne peut pas être vidée,...).

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' ou 'père'. Le 'processus enfant' ou 'fils' 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, mais uniquement lors de la première modification de ressource grâce à la méthode COW (Copy On Write).

La méthode COW trouve son utilisation principale dans le partage de la mémoire virtuelle des processus du système d'exploitation : dans la mise en œuvre de l'appel système de bifurcation. Typiquement, le processus ne modifie aucune mémoire et exécute immédiatement un nouveau processus, remplaçant complètement l'espace d'adressage. Ainsi, il serait inutile de copier toute la mémoire du processus pendant une bifurcation, et au lieu de cela, la technique COW est utilisée.
La méthode COW peut être mise en œuvre efficacement en utilisant le tableau des pages en marquant certaines pages de mémoire comme étant en lecture seule et en comptant le nombre de références à la page. Lorsque des données sont écrites sur ces pages, le noyau du système d'exploitation intercepte la tentative d'écriture et alloue une nouvelle page physique initialisée avec les données de COW, bien que l'allocation puisse être ignorée s'il n'y a qu'une seule référence. Le noyau met ensuite à jour la table des pages avec la nouvelle page, décrémente le nombre de ré-

férences et effectue l'écriture. La nouvelle allocation garantit qu'un changement dans la mémoire d'un processus n'est pas visible dans un autre.
La technique de COW peut être étendue pour prendre en charge une allocation efficace de mémoire en ayant une page de mémoire physique remplie de zéros. Lorsque la mémoire est allouée, toutes les pages renvoyées se réfèrent à la page de zéros et sont toutes marquées COW. De cette façon, la mémoire physique n'est pas allouée pour le processus tant que les données ne sont pas écrites, ce qui permet aux processus de réserver plus de mémoire virtuelle que de mémoire physique et d'utiliser la mémoire avec modération, au risque de manquer d'espace d'adressage virtuel. L'algorithme combiné est similaire à la pagination à la demande.

— **Le processus enfant** est une copie exacte du processus parent sauf pour
  * L'enfant a son propre ID de processus, ID unique
  * L'enfant n'hérite pas des threads de son père et n'en crée pas de nouveaux.
    ('cat /proc/$PID/status' -> section 'Threads' ou 'Thr')
  * L'enfant n'hérite pas des verrous de mémoire de son parent :
    ipcs -s (A MONTRER)
    ('ps -aux' pour voir que l'état du père est 'SLl+'. Le 'L' indique que ses pages sont verrouillées en mé-

moire. L'état du fils indique qu'il n'a pas de pages verrouillées en mémoire.).

* La table des signaux est remise à 0 pour l'enfant : ceci peut être faux en fonction de l'environnement.
* Quelques autres exceptions qui ne sont pas prouvables sur n'importe quel environnement...

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 si, par exemple, la mémoire est pleine et qu'elle contient des pages qui ne peuvent être supprimées.

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 seules les variables du fils sont modifiées puisque le père et le fils ne partagent pas le même espace d'adressage.
Il prouve également que les threads créés par le père ne sont pas transmis au fils ou recréés pour être attribués au fils.
Enfin, il prouve que les verrous de mémoire ne sont pas transmis au fils.

```
1  #include <sys/types.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  #include <pthread.h>
6  #include <mm_malloc.h>
7  #include <spawn.h>
8  #include <sys/mman.h>
9  #include <signal.h>
10 #include <ctype.h>
11 #include <string.h>
12
13
14
15 // Pour compiler avec les threads :
16 // gcc -pthread -o fork fork.c
```

```
17
18
19
20 void continueProgram()
21 {
22     printf("Pour continuer le programme, entrez 'continue' ou 'c
       ' : ");
23     int c, numberCounter = 0, letterCounter = 0;
24     while ((c = getchar()) != 'c')
25             if (isalpha(c))
26                 letterCounter++;
27             else if (isdigit(c))
28                 numberCounter++;
29 }
30
31 int lock_memory(char * address, size_t size)
32 {
33     //https://linuxhint.com/mlock-2-c-function/
34     unsigned long page_offset, page_size;
35     page_size = sysconf(_SC_PAGE_SIZE);
36     page_offset = (unsigned long) address % page_size;
37     address -= page_offset; // ajuste l'adresse à la limite de
       la page
38     size += page_offset; // ajuste la taille avec l'offset de la
       page
39     return (mlock(address, size));
40 }
41
42 int unlock_memory(char * address, size_t size)
43 {
44     //https://linuxhint.com/mlock-2-c-function/
45     unsigned long page_offset, page_size;
46     page_size = sysconf(_SC_PAGE_SIZE);
47     page_offset = (unsigned long) address % page_size;
48     address -= page_offset; // ajuste l'adresse à la limite de
       la page
49     size += page_offset; // ajuste la taille avec l'offset de la
       page
50     return (munlock(address, size));
51 }
52
```

```
53 void signal_handler(int signal_nb)
54 {
55     printf("\nChange le numéro d'un signal\n");
56     signal(SIGINT, SIG_DFL);
57 }
58
59 // Fonction exécutée comme un thread
60 // lorsque son nom est présent comme argument dans l'appel '
   pthread_create'
61 void *threadCreation(void *arg)
62 {
63     printf("Fonction liée à la création de thread appelée \n");
64     sleep(50); // Ceci pour permettre d'avoir le temps de
   prouver que le fils n'hérite pas des threads du père ni en cr
   ée de nouveaux
65     return NULL;
66 }
67
68 /**
69  * Le FORK duplique l'espace d'adressage.
70  * Donc, le process créé par FORK fera les additions de son côté
   .
71  *  Ensuite, le fils sera exit, donc, il n'affichera pas le ré
   sultat de ses additions
72  *
73  * Du côté du père, l'addition ne sera pas faite puisqu'il exé
   cutera seulement le code suivant le "if"
74  *
75  * Vu que l'espace d'adressage est dupliqué lors du FORK, les
   variables ne seront modifiées que dans
76  *  l'espace d'adressage du FILS.
77  *  Donc, les variablses du père ne sont pas modifiées
78  */
79 int main(int argc, char **argv) {
80     printf("\n\n\nCODES D'ÉTAT DE PROCESSUS \nVoici les diffé
   rentes valeurs que les indicateurs de sortie s, stat et state
    (en-tête "STAT" ou "S") afficheront pour décrire l'état d'un
    processus :\n\n"
81
82                 "D    en sommeil non interruptible (normalement
   entrées et sorties) ;\n"
```

```
83                "R    s'exécutant ou pouvant s'exécuter (dans la
      file d'exécution) ;\n"
84                "S    en sommeil interruptible (en attente d'un é
      vénement pour finir) ;\n"
85                "T    arrêté, par un signal de contrôle des tâ
      ches ou parce qu'il a été tracé ;\n"
86                "W    pagination (non valable depuis le noyau
      2.6.xx) ;\n"
87                "X    tué (ne devrait jamais être vu) ;\n"
88                "Z    processus zombie (<defunct>), terminé mais
      pas détruit par son parent.\n\n"
89
90        "Pour les formats BSD et quand le mot-clé stat est utilis
      é, les caractères supplémentaires suivants peuvent être
      affichés :\n\n"
91
92                "<    haute priorité (prioritaire par rapport aux
       autre utilisateurs) ;\n"
93                "N    basse priorité (non prioritaire par rapport
       aux autres utilisateurs) ;\n"
94                "L    les pages du processus sont verrouillées en
       mémoire;\n"
95                "s    meneur de session ;\n"
96                "l    possède plusieurs processus légers ("multi-
      thread", utilisant CLONE_THREAD comme NPTL pthreads le fait)
      ;\n"
97                "+    dans le groupe de processus au premier plan
      .\n\n\n\n\n\n");
98
99
100
101     // Entiers à incrémenter dans le fils pour prouver l'espace
      d'adressage commun
102     int a = 5, b = 8;
103     // Récupérer la valeur de retour de la fonction créant le
      proocess fils
104     int forkRetNum;
105
106     printf("PID du père = %d\n", getpid());
107     printf ("Ces 2 variables sont créées par le père :\n");
108     printf("a = %d\n", a);
```

```
109     printf("b = %d\n", b);
110
111     // A tester sur les machines de l'école : il faut que la
        quantité de mémoire allouée au process fils soit différente
        de celle allouée au process père
112     int dataSize = 2048;
113     char dataLock[dataSize];
114     if (lock_memory(dataLock, dataSize) == -1)
115         perror("Error with locking memory\n");
116     else
117         printf ("\nDe la mémoire a été réservée en RAM par le pè
        re\n");
118
119
120     /*signal(SIGINT, signal_handler);
121     sleep(30);
122     for(int i=1; ; i++)
123     {
124     printf("%d, Le programme est dans la fonction main\n", i);
125         sleep(1);
126     }*/
127
128     continueProgram();
129
130     printf("\nCeci est avant que le père ne crée un thread\n\n")
        ;
131     continueProgram();
132
133     int i;
134     pthread_t tid;
135     // Let us create three threads
136     for (i = 0; i < 3; i++)
137         pthread_create(&tid, NULL, threadCreation, (void *)&tid)
        ;
138
139     printf("%s", "");
140
141     forkRetNum = fork();
142
143     if(forkRetNum == 0) { // La création du fils s'est-elle
        correctement produite ?
```

```
144         printf("Le processus fils vient d'être créé. La suite
        est affichée par le fils.\n");
145         // a = 10 but only the one of the chile. not the one of
        the parent
146         a = a + 5;
147         printf("Maintenant, a = %d et ce, uniquement dans l'
        espace d'adressage du fils\n", a);
148
149         // b = 10 but only the one of the chile. not the one of
        the parent
150         b = b + 2;
151         printf("Maintenant, b = %d et ce, uniquement dans l'
        espace d'adressage du fils\n", b);
152
153      //lock_memory();
154
155         printf("PID (du fils, donc) = %d\n", getpid());
156         printf("PID du père = %d\n", getppid());
157         printf("a + b = %d.\n", a + b);
158         printf("\nDans une autre fenêtre de terminal, entrez la
        commande 'ps' pour voir quel process est en cours et plus d'
        informations à leurs propos !\n\n");
159         printf("Sous la section 'Status', vous pouvez voir que
        le statut du père est 'SLl+'. Le 'L' signifie que de la mé
        moire est verrouillée en RAM par le process !\n\n");
160         printf("RSS signifie Resident Set Size et montre la
        quantité de RAM utilisée au moment de la sortie de la
        commande. "
161                "Il convient également de noter qu'il affiche toute
        la pile de mémoire physiquement allouée.\n\n");
162         printf("VSZ est l'abréviation de Virtual Memory Size. C'
        est la quantité totale de mémoire à laquelle un processus
        peut hypothétiquement accéder. "
163                "Il tient compte de la taille du binaire lui-mê
        me, de toutes les bibliothèques liées et de toutes les
        allocations de pile ou de tas.\n");
164         printf("\n\nDans une autre fenêtre de terminal, entrez
        la commande 'cat /proc/$PID/status' pour voir les
        informations du process !\n");
165
166         printf("\n\nLe fils est en train de tourner à l'infini
```

```
        via un 'while(1)' pour prouver qu'il n'est pas en sommeil (
        cfr 'ps'). Pour l'arrêter, dans une autre fenêtre de terminal
        , entrez la commande 'kill $PID' !\n");
167
168         //continueProgram();
169         while(1){} // Faire en sorte que le fils attende, mais
        en étant en état d'exécution. Un simple ps le montrera
170         exit(0);
171       }
172       else if (forkRetNum > 0)
173       {  // Est-ce le process parent ?
174         printf("Ceci est le process parent et le PID est : %d\n"
        , getpid());
175       }
176       else
177       { // Y a-t-il eu une erreur lors de la création du process
        fils ?
178         printf("Problème durant la duplication\n");
179         exit(EXIT_FAILURE);
180       }
181       // Parent code
182       wait(0); // Pour éviter de faire du fils un zombie
183       printf ("Le fils est terminé\n");
184       printf("PID = %d\n", getpid());
185       printf("PPID = %d\n", getppid());
186       // La somme est bien de 13 et non plus ni moins puisque la
        somme fut faite par le fils, mais uniquement avec ses propres
         variables et non celles du père
187       printf("a + b = %d.\n", a + b);
188       printf("Vu que a + b = 20 dans le fils et que a + b = 13
        dans le père, cela prouve que l'espace d'adressage d'un
        process créé au moyen de fork n'est pas celui du père car il
        a été dupliqué par rapport à celui du père. Chaque process a
        donc ses propres variables,...\n");
189       printf("Let's do a ps to see which process is currenlty
        running !");
190
191       if (unlock_memory(dataLock, dataSize) == -1)
192         perror("Error with locking memory\n");
193       else
194         printf ("Memory unlocked in RAM\n");
```

```
195
196     printf ("\n\nLe programme ne se termine pas pour laisser le
        temps de faire un ps et voir quels process sont en cours d'ex
        écution. Pour le terminer, faites un 'kill $PID' dans une
        autre fenêtre de terminal ou faites un CTRL + C\n");
197
198     pthread_join(tid, NULL);
199     printf("After Thread\n");
200
201     while(1){} // Simplement pour faire attendre le père que l'
        on fasse un ps pour pouvoir voir son état
202
203     exit(0);
204 }
```

2.9 Resultat

```
1  CODES D'ÉTAT DE PROCESSUS
2  Voici les différentes valeurs que les indicateurs de sortie s,
       stat et state (en-tête "STAT" ou "S") afficheront pour dé
       crire l'état d'un processus :
3
4  D    en sommeil non interruptible (normalement entrées et
       sorties) ;
5  R    s'exécutant ou pouvant s'exécuter (dans la file d'exécution
       ) ;
6  S    en sommeil interruptible (en attente d'un événement pour
       finir) ;
7  T    arrêté, par un signal de contrôle des tâches ou parce qu'il
        a été tracé ;
8  W    pagination (non valable depuis le noyau 2.6.xx) ;
9  X    tué (ne devrait jamais être vu) ;
10 Z    processus zombie (<defunct>), terminé mais pas détruit par
       son parent.
11
12 Pour les formats BSD et quand le mot-clé stat est utilisé, les
       caractères supplémentaires suivants peuvent être affichés :
```

```
13
14 <    haute priorité (non poli pour les autres utilisateurs) ;
15 N    basse priorité (poli pour les autres utilisateurs) ;
16 L    les pages du processus sont verrouillées en mémoire;
17 s    meneur de session ;
18 l    possède plusieurs processus légers ("multi-thread",
       utilisant CLONE_THREAD comme NPTL pthreads le fait) ;
19 +    dans le groupe de processus au premier plan.
20
21
22
23
24
25
26 PID du père = 9059
27 Ces 2 variables sont créées par le père :
28 a = 5
29 b = 8
30
31 De la mémoire a été réservée en RAM par le père
32 Pour continuer le programme, entrez 'continue' ou 'c' : c
33
34 Ceci est avant que le père ne crée un thread
35
36 Pour continuer le programme, entrez 'continue' ou 'c' : c
37 Fonction liée à la création de thread appelée
38 Fonction liée à la création de thread appelée
39 Fonction liée à la création de thread appelée
40 Fonction liée à la création de thread appelée
41 Le processus fils vient d'être créé. La suite est affichée par
       le fils.
42 Ceci est le process parent et le PID est : 9059
43 Maintenant, a = 10 et ce, uniquement dans l'espace d'adressage
       du fils
44 Maintenant, b = 10 et ce, uniquement dans l'espace d'adressage
       du fils
45 PID (du fils, donc) = 9063
46 PID du père = 9059
47 a + b = 20.
48
```

```
49 Dans une autre fenêtre de terminal, entrez la commande 'ps' pour
       voir quel process est en cours et plus d'informations à
     leurs propos !
50
51 Sous la section 'Status', vous pouvez voir que le statut du père
       est 'SLl+'. Le 'L' signifie que de la mémoire est verrouillé
     e en RAM par le process !
52
53 RSS signifie Resident Set Size et montre la quantité de RAM
     utilisée au moment de la sortie de la commande. Il convient é
     galement de noter qu'il affiche toute la pile de mémoire
     physiquement allouée.
54
55 VSZ est l'abréviation de Virtual Memory Size. C'est la quantité
     totale de mémoire à laquelle un processus peut hypothé
     tiquement accéder. Il tient compte de la taille du binaire
     lui-même, de toutes les bibliothèques liées et de toutes les
     allocations de pile ou de tas.
56
57
58 Dans une autre fenêtre de terminal, entrez la commande 'cat /
     proc/$PID/status' pour voir les informations du process !
59
60
61 Le fils est en train de tourner à l'infini via un 'while(1)'
     pour prouver qu'il n'est pas en sommeil (cfr 'ps'). Pour l'
     arrêter, dans une autre fenêtre de terminal, entrez la
     commande 'kill $PID' !
62 Le fils est terminé
63 PID = 9059
64 PPID = 3532
65 a + b = 13.
66 Vu que a + b = 20 dans le fils et que a + b = 13 dans le père,
     cela prouve que l'espace d'adressage d'un process créé au
     moyen de fork n'est pas celui du père car il a été dupliqué
     par rapport à celui du père. Chaque process a donc ses
     propres variables,...
67 Let's do a ps to see which process is currenlty running !Memory
     unlocked in RAM
68
69
```

```
70 Le programme ne se termine pas pour laisser le temps de faire un
       ps et voir quels process sont en cours d'exécution. Pour le
      terminer, faites un 'kill $PID' dans une autre fenêtre de
      terminal ou faites un CTRL + C
71 After Thread
72 ^C
```

---

3 vfork()

3.1 Qu'est-ce

vfork() est un appel système ou fonction qui 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 avant d'appeler _exit() ou une des fonctions de la famille exec().
vfork() est un cas particulier de **clone()** que nous verrons plus tard. Il est utilisé pour créer de nouveaux processus sans copier les tables de pages du processus parent. **vfork()** diffère de **fork()** car le père appelant est suspendu jusqu'à ce que l'enfant se termine, ou il fait un appel à une fonction de la famille exec(2). Jusqu'à ce moment-là, l'enfant partage toute la mémoire avec son parent. Il peut être utile dans les applications qui doivent utiliser le minimum des ressources du système. Le processus enfant créé appelle alors immédiatement une fonction de la famille exec() pour se dissocier du père, ce qui changera le statut du père de "en pause" à "en exécution". L'appel **vfork()** ne diffère de **fork()** que dans le traitement de l'espace d'adressage virtuel, comme décrit ci-dessus. Les signaux envoyés au parent arrivent après que l'enfant ait libéré la mémoire du parent (c'est-à-dire après sa fin ou après l'appel de à une fonction de la famille exec()).

Sous Linux, **fork()** est implémenté en utilisant des pages copy-

on-write, donc la seule pénalité encourue par **fork()** est le temps et la mémoire nécessaire pour dupliquer les tables de pages du parent et pour créer un structure de tâches unique pour l'enfant. Cependant, auparavant, **fork()** nécessitait de faire une copie complète du l'espace d'adressage du père, souvent inutilement, car généralement immédiatement par la suite, un appel à une fonction de la famille exec() est effectué. Ainsi, pour une plus grande efficacité, BSD a introduit l'appel système **vfork()**, qui ne copie pas entièrement l"espace d'adressage du père, mais emprunte la mémoire et le fil d'exécution jusqu'à un appel à execve(2) ou une sortie. Le processus parent est suspendu pendant que l'enfant utilise ses ressources. L'utilisation de **vfork()** a été délicate : par exemple, ne pas modifier les données dans le processus père dépendait de savoir quelles variables étaient conservées dans un registre et lesquelles ne l'étaient dans le but de savoir lesquelles il était autorisé de modifier.

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(). NetBSD l'a réintroduit pour qu'il foncitonne à nouveau en tant que vfork() : voir urlhttp ://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

Suite à la duplication de processus via **vfork()**, l'espace d'adressage du fils n'est pas une duplication de celui du père comme pour un fork(), mais il sera le même : celui du père. Cela peut permettre de faire en sorte que, si la duplication du processus fils s'est correctement déroulée, le père n'aura plus d'utilité parce que le fils aura le même espace d'adressage (cfr. la famille d'exec qui remplace l'espace d'adressage du père lors de leur création).

vfork(), tout comme **fork()**, crée un processus fils à partir du processus appelant. **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 rapi-

dité d'exécution, si le fils doit invoquer immédiatement un appel execve().

vfork() diffère aussi de **fork()** car le processus père reste en pause 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 retourner du père.

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'instructions.

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 n'hérite pas des threads de son père et n'en crée pas de nouveaux.

('cat /proc/$PID/status' -> section 'Threads' ou 'Thr')
Toutes les pages de manuel vfork(2) 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 du père.

3.6 Problèmes éventuels

A PARLER SUR POWERPOINT

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, le système fera qu'il se comportera comme fork(2).

Les détails de la gestion des signaux sont compliqués, et varient suivant les systèmes.

A PARLER SUR POWERPOINT

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 er-*

rors 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). Il indique que vfork() est déprécié (malgré le fait que Linux l'ait ressucité) et qu'il vaut mieux utiliser posix_spawn puisqu'il est considéré comme son successeur.

A PARLER SUR POWERPOINT

vfork() a un inconvénient : 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 appelle _exit() ou une fonction de la famille exec(). (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).

3.7 Code prouvant l'espace d'adressage partagé

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.
Il prouve également que les threads créés par le père ne sont pas transmis au fils ou recréés pour être attribués au fils.

```
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 <ctype.h>
10 #include <sys/wait.h>
11 #include <sys/mman.h>
12 #include <signal.h>
13
14
15 // gcc -o vfork vfork.c
16 // Pour compiler avec les threads : gcc -pthread -o vfork vfork.
       c
17
18
19
20 // Cette fonction permet à l'utilisateur de choisir à quel
       moment reprendre
21 // l'exécution du programme pour lui laisser le temps de faire
       les manipulations qu'il désire
22 void continueProgram()
23 {
24     printf("Pour continuer le programme, entrez 'continue' ou 'c
       ' : ");
25     int c, numberCounter = 0, letterCounter = 0;
26     while ((c = getchar()) != 'c')
27             if (isalpha(c))
28                 letterCounter++;
29             else if (isdigit(c))
30                 numberCounter++;
31 }
32
33 // Cette fonction a pour but d'être exécutée
34 // lorsque son nom est spécifié comme argument dans
       pthread_create()
35 void *threadCreation(void *arg)
36 {
37     printf("Fonction liée à la création de thread appelée \n");
```

```
38      sleep(50); // Ceci pour permettre d'avoir le temps de
        prouver que le fils n'hérite pas des threads du père ni en cr
        ée de nouveaux
39      return NULL;
40  }
41
42  /**
43   * Le VFORK ne duplique pas l'espace d'adressage du père.
44   * Donc, le VFORK fera les additions de son côté, mais dans l'
        espace d'adressage du PERE.
45   *
46   * Du côté du père, l'addition sera faite puisqu'il partage l'
        espace d'adressage avec le FILS.
47   *
48   * Vu que l'espace d'adressage est dupliqué lors du VFORK, les
        variables seront modifiées dans l'espace d'adressage du PERE
        (qui est aussi celui du fils)
49   *  Donc, les variables du père sont modifiées, ce qui permet la
         prise en compte de la modification des valeurs des variables
        , modifications faites par le fils
50   */
51  int main(int argc, char **argv)
52  {
53      printf("\n\n\nCODES D'ÉTAT DE PROCESSUS \nVoici les diffé
        rentes valeurs que les indicateurs de sortie s, stat et state
         (en-tête "STAT" ou "S") afficheront pour décrire l'état d'un
         processus :\n\n"
54
55                  "D    en sommeil non interruptible (normalement
        entrées et sorties) ;\n"
56                  "R    s'exécutant ou pouvant s'exécuter (dans la
        file d'exécution) ;\n"
57                  "S    en sommeil interruptible (en attente d'un é
        vénement pour finir) ;\n"
58                  "T    arrêté, par un signal de contrôle des tâ
        ches ou parce qu'il a été tracé ;\n"
59                  "W    pagination (non valable depuis le noyau
        2.6.xx) ;\n"
60                  "X    tué (ne devrait jamais être vu) ;\n"
61                  "Z    processus zombie (<defunct>), terminé mais
        pas détruit par son parent.\n\n"
```

```
62
63          "Pour les formats BSD et quand le mot-clé stat est utilis
        é, les caractères supplémentaires suivants peuvent être
        affichés :\n\n"
64
65                  "<    haute priorité (non poli pour les autres
        utilisateurs) ;\n"
66                  "N    basse priorité (poli pour les autres
        utilisateurs) ;\n"
67                  "L    les pages du processus sont verrouillées en
         mémoire;\n"
68                  "s    meneur de session ;\n"
69                  "l    possède plusieurs processus légers ("multi-
        thread", utilisant CLONE_THREAD comme NPTL pthreads le fait)
        ;\n"
70                  "+    dans le groupe de processus au premier plan
        .\n\n\n\n\n\n");
71
72
73
74      // Entiers à incrémenter dans le fils pour prouver l'espace
        d'adressage commun
75      int a = 5, b = 8;
76      // Récupérer la valeur de retour de la fonction créant le
        proocess fils
77      int vforkRetNum;
78
79      printf("PID du père = %d\n", getpid());
80      printf ("Ces 2 variables sont créées et initialisées par le
        père :\n");
81      printf("a = %d\n", a);
82      printf("b = %d\n", b);
83
84      continueProgram();
85
86      printf("\nCeci est avant que le père ne crée un thread\n\n")
        ;
87
88      continueProgram();
89
90      pthread_t tid;
```

```
91      // Let us create three threads
92      for (unsigned int i = 0; i < 3; i++)
93          pthread_create(&tid, NULL, threadCreation, (void *)&tid)
        ;
94
95      printf("%s", "");
96
97      vforkRetNum = vfork();
98
99      if(vforkRetNum == 0)
100     { // La création du fils s'est-elle correctement produite ?
101         printf("Le processus fils vient d'être créé. La suite
        est affichée par le fils.\n");
102         // a = 10
103         a = a + 5;
104         printf("Maintenant, a = %d et ce, dans l'espace d'
        adressage du fils qui est aussi celui du père\n", a);
105
106         // b = 10
107         b = b + 2;
108         printf("Maintenant, b = %d et ce, dans l'espace d'
        adressage du fils qui est aussi celui du père\n", b);
109
110         printf("PID (du fils, donc) = %d\n", getpid());
111         printf("PID du père = %d\n", getppid());
112 //          printf("Value of vfork is %d.\n", vforkRetNum); //
        Indiquer la valeur de retour de la fonciton créant le process
113         printf("a + b = %d.\n", a + b); // line b
114         printf("\nDans une autre fenêtre de terminal, entrez la
        commande 'ps -aux' pour voir quel process est en cours et
        plus d'informations à leurs propos !\n\n");
115     printf("Sous la section 'Status', vous pouvez voir que le
        statut du père est 'SLl+'.\nLe 'L' signifie que de la mémoire
         est verrouillée en RAM par le process !\nLe 'l' signifie que
         le proccess possède plusieurs processus légers : les threads
         qu'il a créés\n\n");
116         printf("RSS signifie Resident Set Size et montre la
        quantité de RAM utilisée au moment de la sortie de la
        commande. "
117             "Il convient également de noter qu'il affiche toute
        la pile de mémoire physiquement allouée.\n\n");
```

```
118         printf("VSZ est l'abréviation de Virtual Memory Size. C'
        est la quantité totale de mémoire à laquelle un processus
        peut hypothétiquement accéder. "
119                 "Il tient compte de la taille du binaire lui-mê
        me, de toutes les bibliothèques liées et de toutes les
        allocations de pile ou de tas.\n");
120         printf("\n\nDans une autre fenêtre de terminal, entrez
        la commande 'cat /proc/$PID/status' pour voir les
        informations du process !\n");
121
122         // wait est un processus bloquant. Donc, la suite ne
        sera pas exécutée tant qu'une condition ne sera pas remplie.
        Si l'on met un pointeur d'un nombre, alors, on pourra récupé
        rer le code de terminaison du processus enfant. Pareil pour
        exit
123
124         printf("\n\nLe fils est en train de tourner à l'infini
        via un 'while(1)' pour prouver qu'il n'est pas en sommeil (
        cfr 'ps -aux'). Pour l'arrêter, dans une autre fenêtre de
        terminal, entrez la commande 'kill $PID' !\n");
125         while(1){} // Faire en sorte que le fils attende, mais
        en étant en état d'exécution. Un simple 'ps -aux' le montrera
126         exit(0);
127       }
128       else if (vforkRetNum > 0)
129       {  // Est-ce le process parent ?
130         printf("Ceci est le process parent et le PID est : %d\n"
        , getpid());
131       }
132       else
133       { // Y a-t-il eu une erreur lors de la création du process
        fils ?
134         printf("Problème durant le vfork\n");
135         exit(EXIT_FAILURE);
136       }
137
138       wait(0); // Pour éviter de faire du fils un zombie
139       printf ("Le fils est terminé\n");
140       printf("PID (du père, donc) = %d\n", getpid());
141       printf("PPID = %d\n", getppid());
142       // La somme est bien de 20 puisque la somme fut faite par le
```

```
        fils avec les mêmes variables que celles du père
143     printf("a + b = %d.\n", a + b);
144     printf("Vu que a + b = 20 dans le fils et que a + b = 20
        dans le père, cela prouve que l'espace d'adressage d'un
        process créé au moyen de vfork est celui du père car il est
        partagé avec le père.\n");
145     printf("\nDans une autre fenêtre de terminal, entrez la
        commande 'ps -aux' pour voir quel process est en cours et
        plus d'informations à leurs propos !\n\n");
146
147     printf ("\n\nLe programme ne se termine pas pour laisser le
        temps de faire un 'ps -aux' et voir quels process sont en
        cours d'exécution. Pour le terminer, faites un 'kill $PID'
        dans une autre fenêtre de terminal ou faites un CTRL + C\n");
148
149     pthread_join(tid, NULL);
150     printf("After Thread\n");
151
152     while(1){} // Simplement pour faire attendre le père que l'
        on fasse un 'ps -aux' pour pouvoir voir son état
153     exit(0);
154 }
```

3.8 Résultat

```
1 CODES D'ÉTAT DE PROCESSUS
2 Voici les différentes valeurs que les indicateurs de sortie s,
     stat et state (en-tête "STAT" ou "S") afficheront pour dé
     crire l'état d'un processus :
3
4 D    en sommeil non interruptible (normalement entrées et
     sorties) ;
5 R    s'exécutant ou pouvant s'exécuter (dans la file d'exécution
     ) ;
6 S    en sommeil interruptible (en attente d'un événement pour
     finir) ;
```

```
 7 T    arrêté, par un signal de contrôle des tâches ou parce qu'il
       a été tracé ;
 8 W    pagination (non valable depuis le noyau 2.6.xx) ;
 9 X    tué (ne devrait jamais être vu) ;
10 Z    processus zombie (<defunct>), terminé mais pas détruit par
      son parent.
11
12 Pour les formats BSD et quand le mot-clé stat est utilisé, les
      caractères supplémentaires suivants peuvent être affichés :
13
14 <    haute priorité (non poli pour les autres utilisateurs) ;
15 N    basse priorité (poli pour les autres utilisateurs) ;
16 L    les pages du processus sont verrouillées en mémoire;
17 s    meneur de session ;
18 l    possède plusieurs processus légers ("multi-thread",
      utilisant CLONE_THREAD comme NPTL pthreads le fait) ;
19 +    dans le groupe de processus au premier plan.
20
21
22
23
24
25
26 PID du père = 16307
27 Ces 2 variables sont créées et initialisées par le père :
28 a = 5
29 b = 8
30 Pour continuer le programme, entrez 'continue' ou 'c' : c
31
32 Ceci est avant que le père ne crée un thread
33
34 Pour continuer le programme, entrez 'continue' ou 'c' : c
35 Fonction liée à la création de thread appelée
36 Fonction liée à la création de thread appelée
37 Fonction liée à la création de thread appelée
38 Le processus fils vient d'être créé. La suite est affichée par
      le fils.
39 Maintenant, a = 10 et ce, dans l'espace d'adressage du fils qui
      est aussi celui du père
40 Maintenant, b = 10 et ce, dans l'espace d'adressage du fils qui
      est aussi celui du père
```

```
41 PID (du fils, donc) = 16312
42 PID du père = 16307
43 a + b = 20.
44
45 Dans une autre fenêtre de terminal, entrez la commande 'ps -aux'
       pour voir quel process est en cours et plus d'informations à
       leurs propos !
46
47 Sous la section 'Status', vous pouvez voir que le statut du père
       est 'SLl+'.
48 Le 'L' signifie que de la mémoire est verrouillée en RAM par le
      process !
49 Le 'l' signifie que le proccess possède plusieurs processus lé
      gers : les threads qu'il a créés
50
51 RSS signifie Resident Set Size et montre la quantité de RAM
      utilisée au moment de la sortie de la commande. Il convient é
      galement de noter qu'il affiche toute la pile de mémoire
      physiquement allouée.
52
53 VSZ est l'abréviation de Virtual Memory Size. C'est la quantité
      totale de mémoire à laquelle un processus peut hypothé
      tiquement accéder. Il tient compte de la taille du binaire
      lui-même, de toutes les bibliothèques liées et de toutes les
      allocations de pile ou de tas.
54
55
56 Dans une autre fenêtre de terminal, entrez la commande 'cat /
      proc/$PID/status' pour voir les informations du process !
57
58
59 Le fils est en train de tourner à l'infini via un 'while(1)'
      pour prouver qu'il n'est pas en sommeil (cfr 'ps -aux'). Pour
       l'arrêter, dans une autre fenêtre de terminal, entrez la
      commande 'kill $PID' !
60 ^[[ACeci est le process parent et le PID est : 16307
61 Le fils est terminé
62 PID (du père, donc) = 16307
63 PPID = 3126
64 a + b = 20.
```

```
65 Vu que a + b = 20 dans le fils et que a + b = 20 dans le père,
       cela prouve que l'espace d'adressage d'un process créé au
       moyen de vfork est celui du père car il est partagé avec le p
       ère.
66
67 Dans une autre fenêtre de terminal, entrez la commande 'ps -aux'
       pour voir quel process est en cours et plus d'informations à
       leurs propos !
68
69
70
71 Le programme ne se termine pas pour laisser le temps de faire un
       'ps -aux' et voir quels process sont en cours d'exécution.
       Pour le terminer, faites un 'kill $PID' dans une autre fenê
       tre de terminal ou faites un CTRL + C
72 Terminated
```

3.9 Code prouvant que le parent est mis en pause

Ce code effectue un affichage par le père et par le fils. Le fait que le fils effectue le sien en premier et le père en second prouve que le père 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 du PID correspondant
       au process fils.
```

```
 9  Dans le bloc if else, le processus fils est exécuté EN PREMIER
      car il bloque le processus parent lors de son exécution, donc
      , le père est MIS EN PAUSE.
10  */
11 int main()
12 {
13
14     printf("\n\n\nvfork() affiche le contenu du 'if{} else{}'
       deux fois, d'abord dans l'enfant, puis dans le parent.\n"
15          "Vu que les deux processus partagent le même espace d'
       adressage, la première sortie contient la valeur du PID
       correspondant au process fils.\n"
16          "Dans le bloc if else, le processus fils est exécuté EN
       PREMIER car il bloque le processus parent lors de son exé
       cution, donc, le père est MIS EN PAUSE.\n\n\n");
17     //pid_t pid = vfork();  // Crée le process fils
18
19     printf("Process fils avant le 'if{} else{}': %d\n", getpid()
       );
20
21     pid_t pid = vfork();  // Crée le process fils
22
23     if (pid > 0)
24     {  // Est-ce le process fils ?
25          printf("Ceci est le process parent et le PID est : %d\n"
       , getpid());
26          exit(0);
27     }
28     else if (pid == 0)
29     {  // Est-ce le process parent ?
30          printf("Ceci est le process fils et le PID est : %d\n\n"
       , getpid());
31
32     }
33     else
34     { // Y a-t-il eu une erreur lors de la création du process
       fils ?
35          printf("Problème durant le fork\n");
36          exit(EXIT_FAILURE);
37     }
38     return 0;
```

```
39 }
```

3.10 Résultat

```
1
2 vfork() affiche le contenu du 'if{} else{}' deux fois, d'abord
     dans l'enfant, puis dans le parent.
3 Vu que les deux processus partagent le même espace d'adressage,
     la première sortie contient la valeur du PID correspondant au
      process fils.
4 Dans le bloc if else, le processus fils est exécuté EN PREMIER
     car il bloque le processus parent lors de son exécution, donc
     , le père est MIS EN PAUSE.
5
6
7 Process fils avant le 'if{} else{}': 14839
8 Ceci est le process fils et le PID est : 14840
9 Ceci est le process parent et le PID est : 14839
```

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/clone.2.html

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()

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

Signal de fin de l'enfant
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 ob-

tenir 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 ancien de cache, 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 acces à
       clone(2) et aux flags 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 static int child_func(void* arg) {
13     char* buffer = (char*)arg;
14     printf("Child sees buffer = \"%s\"\n", buffer);
15     strcpy(buffer, "Hello from child");
16     return 0;
17 }
18
19 /**
```

```
20  Ici, clone() est utilisé de deux manières : une fois avec le
      flag CLONE_VM (CLONE_VM = clone virtual memory) et une fois
      sans.
21  Un buffer est passé dans le processus enfant, et le processus
      enfant y écrit un string.
22  Une pile de taille 65536 ensuite allouée pour le processus
      enfant et une fonction qui vérifie si nous exécutons le
      fichier en utilisant l'option 'vm' (correspondant donc au
      flag 'CLONE_VM').
23  De plus, un buffer de 100 octets est créé dans le processus
      parent et une chaîne y est copiée, puis, l'appel système
      clone() est exécuté et les erreurs sont vérifiées.
24
25  Lorsque d'une exécution sans l'argument 'vm' se produit, le
      flag CLONE_VM n'est pas actif et la mémoire virtuelle du
      processus parent est clonée dans le processus enfant.
26  Le processus enfant peut accéder au message passé par le
      processus parent dans le buffer, mais tout ce qui est écrit
      dans le buffer par l'enfant n'est pas accessible par
      processus parent puisque la mémoire virtuelle est dupliquée
      pour être allouée au processus enfant.
27  */
28  int main(int argc, char** argv) {
29      // Alloue un stack pour la tâche du fils
30      const int STACK_SIZE = 65536;
31      char* stack = malloc(STACK_SIZE);
32      if (!stack) { // Si 'stack' n'a pas été correctement créé
33          perror("malloc");
34          exit(1);
35      }
36
37      // Lorsqu'il est appelé avec l'argument 'vm' en ligne de
        commande, active le flag CLONE_VM.
38      unsigned long flags = 0;
39      if (argc > 1 && !strcmp(argv[1], "vm")) {
40
41          /**
42            int clone(int (*fn)(void *), void *child_stack,
43                      int flags, void *arg, ...
44                      pid_t *ptid, struct user_desc *tls, pid_t *
        ctid );
```

```
45          */
46
47          /**
48          Lorsque le processus enfant est créé avec clone(), il exécute la fonction fn(arg).
49          (Cela diffère de fork(2) dans lequel l'exécution continue dans le fils à partir du point d'appel de fork(2).)
50          L'argument fn est un pointeur vers une fonction qui est appelée par le processus fils au début de son exécution. L'argument 'arg' est passé à la fonction fn.
51          */
52          /**
53          CLONE_VM (depuis Linux 2.0)
54                      Si CLONE_VM est défini, le parent et l'enfant seront exécuté dans le même espace mémoire. En particulier les écritures mémoire effectuées par le parent ou par l'enfant sont également visibles dans l'autre processus.
55                      De plus, tout mappage ou démappage de mémoire effectué avec mmap(2) ou munmap(2) par le processus enfant ou appelant également affecte l'autre processus.
56
57                      Si CLONE_VM n'est pas défini, le processus enfant s'exécute dans un copie séparée de l'espace mémoire du processus appelant au moment de l'appel de clone. Les écritures effectuées par les mappages/démappages par un des processus n'affecte pas l'autre, comme avec fork(2).
58          */
59          flags |= CLONE_VM; // 'flags' vaudra 'CLONE_VM' ou non en fonction du fait que l'option 'vm' soit spécifiée ou non.
60      }
61
62      char buffer[100];
63      strcpy(buffer, "Hello from parent"); // Ecrit 'hello from parent' dans le buffer
64      // Clone le processus père
65      // Seul appel à 'clone'. Pour avoir les différentes exécutions, il faut ajouter 'vm' comme argument lors de l'appel en ligne de commande
66      // Vu que lorsque CLONE_VM est défini, l'espace d'adressage mémoiire est partaé,
```

```
67      // le buffer est le même pour le père et pour le fils, donc,
        le fils override ce que le père a écrit par 'hello from
        child'
68      if (clone(child_func, stack + STACK_SIZE, flags | SIGCHLD,
        buffer) == -1) {
69          perror("clone");
70          exit(1);
71      }
72
73      int status;
74      if (wait(&status) == -1) {
75          perror("wait");
76          exit(1);
77      }
78
79      printf("Child exited with status %d. buffer = \"%s\"\n",
        status, buffer);
80      return 0;
81  }
```

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 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 Conclusion

Pour finir, nous pouvons constater que fork() et vfork() sont des fonctions et appels système qui existent depuis longtemps. clone() a pris le relais de par sa puissance et sa modularité.
Donc, pour tout programme un minimum lourd, il est meilleur d'implémenter clone() avec les flags nécessaires afin d'utiliser la mémoire virtuelle de manière pertinente et efficace.

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.html
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-clonee
http://www.unixguide.net/unix/programming/1.1.2.shtml
https://prograide.com/pregunta/12758/differences-entre-exec-et-fourche
https://man7.org/linux/man-pages/man2/fork.2.html
https://techdifferences.com/difference-between-fork-and-vfork.html
https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-vfork-create-new-process
https://man7.org/linux/man-pages/man2/fork.2.html
https://www.linuxjournal.com/article/5211
https://man7.org/linux/man-pages/man2/clone.2.html
http://www-igm.univ-mlv.fr/~dr/CS/node88.html

https://gist.github.com/nicowilliams/
a8a07b0fc75df05f684c23c18d7db234
https://fresh2refresh.com/c-programming/
c-buffer-manipulation-function/
https://stackoverflow.com/questions/66548922/
can-a-fork-child-determine-whether-it-is-a-fork-or-a-vfork
https://news.ycombinator.com/item?id=30502392
https://developer.apple.com/library/archive/
documentation/System/Conceptual/ManPages_iPhoneOS/
man2/vfork.2.html
http://manpagesfr.free.fr/man/man2/clone.2.html
https://gist.github.com/alifarazz/
d1ccf716131ed3a369fc7d248d910330
https://linux.die.net/man/2/clone
https://www.thegeekstuff.com/2012/05/
c-mutex-examples/
https://docs.oracle.com/cd/E26502_01/html/E35303/
gen-1.html
https://mindsgrid.com/difference-fork-vfork-exec-clone/
https://stackoverflow.com/questions/21205723/
how-many-ways-we-can-create-a-process-in-linux-using-c
https://cpp.hotexamples.com/fr/examples/-/-/vfork/
cpp-vfork-function-examples.html
https://man7.org/linux/man-pages/man2/vfork.2.html
https://www.ibm.com/docs/en/zos/2.4.0?topic=
functions-vfork-create-new-process
https://mindsgrid.com/difference-fork-vfork-exec-clone/

https://stackoverflow.com/questions/4856255/
the-difference-between-fork-vfork-exec-and-clone
https://prograide.com/pregunta/11064/
la-difference-entre-fork-vfork-exec-et-clone
http://www.unixguide.net/unix/programming/1.1.2.
shtml
https://prograide.com/pregunta/12758/
differences-entre-exec-et-fourche
https://techdifferences.com/
difference-between-fork-and-vfork.html
https://manpages.ubuntu.com/manpages/hirsute/fr/
man2/clone.2.html
http://manpagesfr.free.fr/man/man2/clone.2.html
https://github.com/jeremyong/google-coredumper/
issues/14
https://stackoverflow.com/questions/29264322/
mmap-error-on-linux-using-somethingelse
https://man7.org/linux/man-pages/man7/signal.7.html
https://stackoverflow.com/questions/9361816/
maximum-number-of-processes-in-linux
https://www.2daygeek.com/kill-terminate-a-process-in-linux-us