fork()
vfork()
clone()

Léopold Mols

20 août 2022

Année : SYSG6 Q2 2021-2022
Professeur : Mme Bastreghi

Table des matières

1 Que sont-ils ? 5

2 fork() 6

2.1 Qu'est-ce 6
2.2 Déclaration de fork(2) 6
2.3 Historique 6
2.4 Fonctionnement 7
2.5 particularités 9
2.6 Problèmes éventuels 11
2.7 Suppléments 12
2.8 Code prouvant que l'espace d'adressage n'est pas partagé, que les threads ne sont pas transmis au fils et que les verrous de mémoire ne sont pas transmis au fils 12
2.9 Resultat 13
2.10 Code prouvant que l'espace d'adressage n'est partagé en appelant plusieurs fois fork 16
2.11 Resultat 16

3 vfork() 19

3.1 Qu'est-ce 19
3.2 Déclaration de vfork(2) 20
3.3 Historique 21
3.4 Fonctionnement 21
3.5 particularités 22

3.6 Problèmes éventuels . . . 23
3.7 Code prouvant l'espace d'adressage partagé . . . 25
3.8 Résultat . . . 25
3.9 Code prouvant que le parent est mis en pause . . . 28
3.10 Résultat . . . 29
3.11 Code prouvant que l'espace d'adressage est partagé entre un père et son fils même si vfork est exécuté plusieurs fois . . . 29
3.12 Résultat . . . 29

4 clone() 34
4.1 Qu'est-ce . . . 34
4.2 Déclaration de clone(2) . . . 36
4.3 Historique . . . 36
4.4 Fonctionnement . . . 37
4.5 particularités . . . 38
4.5.1 Signal de fin de l'enfant . . . 40
4.6 Problèmes éventuels . . . 41
4.7 Code . . . 42
4.8 Résultats . . . 43
4.8.1 ./clone . . . 43
4.8.2 ./clone vm . . . 43

5 Aller plus loin 44

6 Conclusion 45

7 Codes **46**
7.1 FORK 46
7.1.1 fork.c 46
7.1.2 forkMoreProcesses.c 55
7.2 VFORK 62
7.2.1 vfork.c 62
7.2.2 vforkMoreProcesses.c 69
7.2.3 vforkParentPAUSED.c 80
7.3 CLONE 83
7.3.1 clone.c 83

8 Sources **89**

1 Que sont-ils ?

fork(), vfork() et clone() permettent de créer des process s'exécutant sur un OS.

Au démarrage d'un système UNIX (norme POSIX (Portable Operating System Interface et le X exprime l'héritage UNIX)), un seul process existe. UNIX 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).
Il porte le PID 1 et est souvent nommé 'init'. Tous les autres process du système descendent de ce premier process, via des appels système comme, notamment, fork(), vfork(), forkx(), forkall(), forkallx(), vforkx().

2 fork()

2.1 Qu'est-ce

fork() crée un nouveau process en dupliquant le process appelant.

Le nouveau process est appelé 'process enfant' ou 'fils'. Le process appelant est appelé 'process parent' ou 'père'. Lors de la création du process, la mémoire du père est copiée dans une nouvelle zone mémoire séparée et allouée au fils. Juste après l'exécution de **fork()**, les deux espaces mémoire ont exactement le même contenu.
Écritures en mémoire propre, mappages de fichiers (mmap()) et démappages (munmap()) exécutés par l'un des process 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 process. En 1969, fork

a été implémenté. Il a été fortement utilisé jusqu'en 1990, année de dépréciation au profit de, notamment, POSIX_SPAWN.

2.4 Fonctionnement

L'appel système fork() fournit une valeur de retour entière qui peut être utilisée pour différencier le père du fils ou indiquer une erreur. La valeur de retour peut être :

- un entier supérieur à 0 qui correspond PID du fils, auquel cas nous sommes dans le process père.
- 0 auquel cas nous sommes dans le process fils.
- -1 qui témoigne une erreur lors de l'exécution de la commande. Dans ce cas, aucun process enfant n'est créé et errno est modifié pour indiquer l'erreur.

Il est possible d'interagir entre process de plusieurs manières différentes.
Premièrement, il est possible envoyer des signaux. Par exemple, la commande bash 'kill <pid>' permet d'envoyer le signal SIGINT à un process. La majorité des process recevant ce signal se termineront. Si l'on veut envoyer un autre signal, il faut le préciser dans la commande de cette manière : 'kill -<numéro du signal> <pid>'. Pour éviter qu'un fils devienne un zombie, le père doit l'attendre. Pour ce faire, il peut utiliser la fonction pid_t wait(int *ptr_state). La valeur de retour de cette fonction est le pid du fils qui a terminé. Un process devient un zombie lorsque le process se termine, mais que le père n'a pas attendu son fils, c'est-à-dire qu'il n'a, par exemple, pas fait d'appels à

wait(). C'est une situation qu'il convient d'éviter absolument car le process ne peut plus s'exécuter mais consomme encore des ressources.
On peut également attendre la fin du fils grâce à son pid : pid_t waitpid(pid_t pid, int *ptr_state, int options).

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) et ce, sur les systèmes récents. Auparavant, il était directement dupliqué.

La méthode COW trouve son utilisation principale dans le partage de la mémoire virtuelle des process du système d'exploitation : dans la mise en œuvre de l'appel système de bifurcation. Typiquement, le process ne modifie aucune mémoire et exécute immédiatement un nouveau process, remplaçant complètement l'espace d'adressage. Ainsi, il serait inutile de copier toute la mémoire du process 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 références et effectue l'écriture. La nouvelle allocation garantit qu'un changement dans la mémoire d'un process 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 process tant que les données ne sont pas écrites, ce qui permet aux process 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 process enfant** est une copie presque'exacte du process parent. Notamment, ces éléments diffèrent :
  * L'enfant a son propre ID de process, 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

('ps -aux' pour voir que l'état du père est 'SLl+'. Le 'L' indique que ses pages sont verrouillées en mémoire. 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 (sur PopOS, MacOS ARM, MacOS Intel, machine virtuelle sur Mac,..., la table des signaux n'est pas remise à 0).

2.6 Problèmes éventuels

Au vu du fait que fork() duplique l'espace d'adressage , cela représent un risque de remplir la mémoire, ralentir l'ordinateur et empêcher la création de nouveaux process si, par exemple, la mémoire est pleine de pages non évincables.

Lors de son implémentation, fork() n'était pas utilisable par les développeurs. Depuis que les développeurs d'Unix l'ont ouvert au user-land (pour que les développeurs puissent l'utiliser dans leurs logiciels), cela a permis de développer des logiciels qui fonctionnent sur plusieurs process et non sur un seul et même process. Cela les a rendus plus productifs, peut-être beaucoup plus. Le prix que les créateurs d'Unix ont payé pour cette pratique était la nécessité de copier les espaces d'adressage. Étant donné qu'à l'époque, les programmes et les process étaient petits, le manque de possibilité de répartir des tâches

sur plusieurs process était facile à négliger ou à ignorer. Mais maintenant, les process ont tendance à être très importants et multithreads, ce qui rend extrêmement coûteux la copie même de l'espace d'adressage d'un parent.

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 prouvant que l'espace d'adressage n'est pas partagé, que les threads ne sont pas transmis au fils et que les verrous de mémoire ne sont pas transmis au fils

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.

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 = 98479
27 Ces 2 variables sont créées et initialisées 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 Ceci est le process parent et le PID est : 98479
41 Le processus fils vient d'être créé. La suite est affichée par
      le fils.
42 Maintenant, a = 10 et ce, uniquement dans l'espace d'adressage
      du fils
43 Maintenant, b = 10 et ce, uniquement dans l'espace d'adressage
      du fils
44 PID (du fils, donc) = 98483
45 PID du père = 98479
46 a + b = 20.
47
48 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 !
49
50 Sous la section 'Status', vous pouvez voir que le statut du père
       est 'SLl+'.
51 Le 'L' signifie que de la mémoire est verrouillée en RAM par le
      process !
52 Le 'l' signifie que le proccess possède plusieurs processus lé
      gers : les threads qu'il a créés
53
54 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.
55
56 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.
```

```
57
58
59 Dans une autre fenêtre de terminal, entrez la commande 'cat /
       proc/$PID/status' pour voir les informations du process !
60
61
62 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' !
63
64
65 Ce signal indique qu'un processus fils s'est arrêté ou a fini
       son exécution. Par défaut ce signal est ignoré. SIGHUP : numé
       ro 1
66
67 Le fils est terminé
68 PID (du père, donc) = 98479
69 PPID (id du process à l'origine de la création du programme) =
       3126
70 a + b = 13.
71 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,...
72
73 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 !
74
75 Memory unlocked in RAM
76
77
78 Le programme ne se termine pas pour laisser le temps de faire un
        'ps -aux' et voir quels process sont en cours d'exécution et
        leurs états. Pour le terminer, faites un 'kill $PID' dans
       une autre fenêtre de terminal ou faites un CTRL + C ici
79 After Thread
80 Terminated
```

2.10 Code prouvant que l'espace d'adressage n'est partagé en appelant plusieurs fois fork

Ce code effectue plusieurs fois l'appel à fork() pour prouver qu'à chaque duplication de process par ce moyen, un nouvel espace d'adressage sera alloué au process fils.

2.11 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
27 Dans une autre fenêtre de terminal, entrez la commande 'top'
      pour voir quels process sont en cours et plsu d'informations,
       dont leur utilisation de la mémoire et ce, en temps réel !
28  Ceci permettra d'observer que 5 lignes seront créées dans le
      tableau du résultat de la commande car fork() crée des
      process à part entière.
29
30 Pour continuer le programme, entrez 'continue' ou 'c' : c
31 PID du père = 98535
32 Ceci est le process parent et le PID est : 98535
33 Ceci est le process fils et le PID est : 98536
34 Ceci est le process parent et le PID est : 98535
35 Ceci est le process fils et le PID est : 98537
36 Ceci est le process parent et le PID est : 98535
37 Ceci est le process fils et le PID est : 98538
38 Ceci est le process parent et le PID est : 98535
39 Ceci est le process fils et le PID est : 98539
40 Ceci est le process parent et le PID est : 98535
41
42
43 Les fils sont en train de tourner à l'infini via un 'while(1)'
      pour montrer la mémoire qu'ils occupent via la commande 'top'
       (cfr 'ps -aux'). Pour les arrêter, dans une autre fenêtre de
       terminal, entrez la commande 'kill {$PID_du_premier_fils..
      $PID_su_dernier_fils' !
44
45 Ceci est le process fils et le PID est : 98540
```

```
46
47 Les fils sont terminés
48 PID (du père, donc) = 98535
49 PPID (id du process à l'origine de la création du programme) =
      3126
50
51 Vu que lorsque vous tuez 1 process via un autre terminal, l'entr
      ée  liée au process de l'affichage généré par la commande '
      top' disparait, cela prouve que chaque process a bien son
      propre espace d'adressage.
52
53 Terminated
```

---

3 vfork()

3.1 Qu'est-ce

vfork() est un appel système ou fonction qui crée un nouveau process. La fonction **vfork()** a le même effet que **fork()**, sauf que le comportement n'est pas défini si le process 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 process sans copier les tables de pages du process 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 process 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 process 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 process 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 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

Suite à la duplication de process 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 process 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 process fils à partir du process appelant. **vfork()** est conçu comme un cas particulier de **clone()**. Il sert à créer un nouveau process sans effectuer de copie de la table des pages mémoire du process 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 process 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 process fils ne doit donc pas retourner du père.

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

3.5 particularités

- **L'espace d'adressage** n'est pas dupliqué, mais partagé entre le père et le fils.
- **Le process enfant** est un duplicata exact du process qui appelle vfork() (le process parent), à l'exception de ce qui suit :
  * Le process enfant a un ID de process (PID) unique, qui ne correspond à aucun ID de groupe de process 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 process 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

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.

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). 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.

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.

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 = 98625
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) = 98629
42 PID du père = 98625
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 98629' !
60 Ceci est le process parent et le PID est : 98625
61 Le fils est terminé
62 PID (du père, donc) = 98625
63 PPID (id du process à l'origine de la création du programme) =
       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 et
        leurs états. Pour le terminer, faites un 'kill 98625' dans
```

```
    une autre fenêtre de terminal ou faites un CTRL+C ici
72  Terminated
```

3.9 Code prouvant que le parent est mis en pause

Ce code effectue 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 car, au vu du fait que le père est mis en pause lors de la duplication de process, le fils prend la main et effectue son affichage avant que le père ne puisse effectuer le sien.

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
```

3.11 Code prouvant que l'espace d'adressage est partagé entre un père et son fils même si vfork est exécuté plusieurs fois

Ce code effectue plusieurs fois l'appel à vfork() pour prouver qu'à chaque duplication de process par ce moyen, le fils aura accès au même espace d'adressage que le père, donc, ce dernier ne sera pas dupliqué.

3.12 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
27 Dans une autre fenêtre de terminal, entrez la commande 'top'
    pour voir quels process sont en cours et plsu d'informations,
     dont leur utilisation de la mémoire et ce, en temps réel !
28
29 Pour continuer le programme, entrez 'continue' ou 'c' : c
30 PID du père = 98674
31 Ceci est le process fils et le PID est : 98675
32 Ceci est le process fils et le PID est : 98676
33 Ceci est le process fils et le PID est : 98678
```

```
34 Ceci est le process fils et le PID est : 98679
35 Ceci est le process fils et le PID est : 98680
36
37 Vous pouvez observer que, dans le tableau de la commande 'top',
      une seule ligne concernant ce programme a été créée. Cela
      signifie que vfork() a créé des process en créant des threads
      . A ce stade, les fils créés par vfork ne sont que des
      threads. Ils deviendront des process lorsque les fils fils
      appelleront une fonction de la famille exec().
38
39
40
41 Le fils est en train de tourner à l'infini via un 'while(1)'
      pour montrer la mémoire utilisée par l'espace des process lié
       à ce programme (puisque l'espace d'adressage est partagé
      entre le père et le fils).
42
43 Pour l'arrêter, dans une autre fenêtre de terminal, entrez la
      commande 'kill 98680' !
44 Ceci est le process parent et le PID est : 98679
45
46
47 Le fils est en train de tourner à l'infini via un 'while(1)'
      pour montrer la mémoire utilisée par l'espace des process lié
       à ce programme (puisque l'espace d'adressage est partagé
      entre le père et le fils).
48
49 Pour l'arrêter, dans une autre fenêtre de terminal, entrez la
      commande 'kill 98679' !
50 Ceci est le process parent et le PID est : 98678
51
52
53 Le fils est en train de tourner à l'infini via un 'while(1)'
      pour montrer la mémoire utilisée par l'espace des process lié
       à ce programme (puisque l'espace d'adressage est partagé
      entre le père et le fils).
54
55 Pour l'arrêter, dans une autre fenêtre de terminal, entrez la
      commande 'kill 98678' !
56 Ceci est le process parent et le PID est : 98676
57
```

```
58
59 Le fils est en train de tourner à l'infini via un 'while(1)'
      pour montrer la mémoire utilisée par l'espace des process lié
       à ce programme (puisque l'espace d'adressage est partagé
      entre le père et le fils).
60
61 Pour l'arrêter, dans une autre fenêtre de terminal, entrez la
      commande 'kill 98676' !
62 Ceci est le process parent et le PID est : 98675
63
64
65 Le fils est en train de tourner à l'infini via un 'while(1)'
      pour montrer la mémoire utilisée par l'espace des process lié
       à ce programme (puisque l'espace d'adressage est partagé
      entre le père et le fils).
66
67 Pour l'arrêter, dans une autre fenêtre de terminal, entrez la
      commande 'kill 98675' !
68 Ceci est le process parent et le PID est : 98674
69 Les fils sont terminés
70
71
72 Au fur et à mesure que vous killiez les process un à un, vous
      avez pu observer que la mémoire occupée par le process
      courant n'a pas diminué, ce qui prouve que l'espace d'
      adressage est partagé entre un process père et un process
      fils si le moyen de duplication de process est 'vfork()' !
73
74
75 PID (du père, donc) = 98674
76
77 PPID (id du process à l'origine de la création du programme) =
      3126
78
79 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 !
80
81
82
```

```
83 Le programme ne se termine pas pour laisser le temps de faire un
       'ps -aux' et voir quels process sont en cours d'exécution et
       leurs états. Pour le terminer, faites un 'kill 98674' dans
      une autre fenêtre de terminal ou faites un CTRL+C ici. Vous
      verrez alors dans l'affichage généré par la commande 'top'
      que la mémoire occupée par le process courant est libérée
84 Terminated
```

---

4 clone()

4.1 Qu'est-ce

clone() crée un nouveau process. La fonction **clone()** a le même effet que **fork()**, sauf qu'il permet, si l'on le souhaite, de choisir si le process 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 process appelant et le process enfant.

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

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

L'argument *stack indique l'emplacement de la pile utilisée par le process fils. Comme les process 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 process 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 process fils.

L'octet de poids faible de flags contient le numéro du signal de terminaison qui sera envoyé au père lorsque le process fils se terminera. Si ce signal est différent de SIGCHLD, le process 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 process parent ne sera pas notifié de la terminaison du fils.

Les *flags* permettent également de préciser ce qui sera partagé entre le père et le fils, en effectuant un OU binaire entre zéro ou plusieurs de ces constantes.

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 appropriées, à la place, Linux a ajouté de nouveaux drapeaux à clone(2) pour indiquer ce qui ne doit pas être partagé avec le parent. Et au fur et à mesure que de nouveaux drapeaux de

clone(2) liés au conteneur furent ajoutés, les anciens codes ayant utilisé clone(2) pouvait souhaiter les avoir utilisés... il faudra modifier et reconstruire le monde appelant clone(2).

4.4 Fonctionnement

Quand le process enfant est créé par la fonction 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 process 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 process enfant se termine. La valeur entière renvoyée par fn est utilisée comme code de retour du process 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 process enfant. Comme les process 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 process 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 process enfant. Pourtant, 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 process 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
2 struct clone_args {
3                 u64 flags;        /* Masque de bit d'attribut */
4                 u64 pidfd;        /* Où stocker le descripteur de
      fichier du PID
5                                      (pid_t *) */
6                 u64 child_tid;    /* Où stocker le TID enfant,
7                                      dans la mémoire de l'enfant'
     s memory (pid_t *) */
8                 u64 parent_tid;   /* Où stocker le TID enfant,
```

```
9                                       dans la mémoire du parent's
    memory (int *) */
10              u64 exit_signal;  /* Signal à envoyer au parent
    quand
11                                      l'efant se termine */
12              u64 stack;        /* Pointeur vers l'octet le
    plus faible de la pile */
13              u64 stack_size;   /* Taille de la pile */
14              u64 tls;          /* Emplacement du nouveau TLS
    */
15              u64 set_tid;      /* Pointeur vers un tableau
    pid_t
16                                      (depuis Linux 5.5) */
17              u64 set_tid_size; /* Nombre d'éléments dans
    set_tid
18                                      (depuis Linux 5.5) */
19              u64 cgroup;       /* Descripteur de fichier du
    cgroup cible
20                                      de l'enfant (depuis Linux
    5.7) */
21          };
```

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 process 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, 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

4.5.1 Signal de fin de l'enfant

Quand le process 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 process 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 process parent ne sera pas notifié de la terminaison de l'enfant.

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 process 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 process 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);
```

Donc, 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

Ce code permet de dupliquer un process au moyen de l'appel système clone(). Grâce à option 'vm' entrée lors de l'exécution, l'espace d'adressage du père sera partagé avec le fils. Si cette option n'est pas précisée, le clone() ne se conduira pas comme l'appel système fork() mais comme l'appel système vfork().

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 rejeté. La fonction forkall() permet à tous les threads du parent d'être dupliqués dans l'enfant. Cela permet aux threads de l'enfant de poursuivre le traitement et permet de préserver les verrous.

Le parent doit s'assurer que l'état de traitement des threads partagé entre le parent et l'enfant 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 l'enfant) lisent le fichier filedescriptor après le 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

A PARLER DANS LE POWERPOINT 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 vfork() ou clone() avec les flags nécessaires afin d'utiliser la mémoire virtuelle de manière pertinente et efficace.

7 Codes

7.1 FORK

7.1.1 fork.c

```
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 #include <sys/wait.h>
13
14
15
16 // Pour compiler avec les threads :
17 // gcc -pthread -o fork fork.c
18
19
20 /**
21  Cette fonction permet à l'utilisateur de choisir à quel
   ↪  moment reprendre
22  l'exécution du programme pour lui laisser le temps de faire
   ↪  les manipulations qu'il désire
23  */
24 void continueProgram()
25 {
```

```
26          printf("Pour continuer le programme, entrez
            ↪  'continue' ou 'c' : ");
27          int c, numberCounter = 0, letterCounter = 0;
28          while ((c = getchar()) != 'c')
29                      if (isalpha(c))
30                          letterCounter++;
31                      else if (isdigit(c))
32                          numberCounter++;
33  }
34
35  /**
36   Cette fonction permet de réserver de la mémoire en RAM
37
38   @param addressPpour réserver les adresses en mémoire
39   @param size Pour la taille de la mémoire à allouer
40   */int lock_memory(char * address, size_t size)
41  {
42          //https://linuxhint.com/mlock-2-c-function/
43          unsigned long page_offset, page_size;
44          page_size = sysconf(_SC_PAGE_SIZE);
45          page_offset = (unsigned long) address % page_size;
46          address -= page_offset; // adjust address to
            ↪  pageboundary
47          size += page_offset; // adjust size with page_offset
48          return (mlock(address, size));
49  }
50
51  /**
52   Cette fonction permet de libérer la mémoire réservée au
    ↪  process en question
53
54   @param address Pour libérer les adresses en mémoire
55   @param size Pour la taille de la mémoire à libérer
56   */
```

```
57 int unlock_memory(char * address, size_t size)
58 {
59         //https://linuxhint.com/mlock-2-c-function/
60         unsigned long page_offset, page_size;
61         page_size = sysconf(_SC_PAGE_SIZE);
62         page_offset = (unsigned long) address % page_size;
63         address -= page_offset; // adjust address to
           ↪  pageboundary
64         size += page_offset; // adjust size with page_offset
65         return (munlock(address, size));
66 }
67
68 /**
69  Cette fonction a pour but d'être exécutée lorsque son nom
   ↪  est spécifié comme argument dans pthread_create()
70  */
71 void *threadCreation(void *arg)
72 {
73     printf("Fonction liée à la création de thread appelée
       ↪  \n");
74     sleep(50); // Ceci pour permettre d'avoir le temps de
       ↪  prouver que le fils n'hérite pas des threads du père
       ↪  ni en crée de nouveaux
75     return NULL;
76 }
77
78 /**
79  * Le FORK duplique l'aspace d'adressage.
80  * Donc, le fils fera les additions de son côté, dans son
   ↪  propre espace d'adressage
81  *
82  * Du côté du père, l'addition ne sera pas faite puisqu'il
   ↪  exécutera seulement le code suivant le "if" qui vérifie
   ↪  le bon code de retour de l'appel à fork()
```

```
83   *
84   * Vu que l'espace d'adressage est dupliqué lors du FORK, les
     ↪  variables ne seront modifiées que dans
85   *  l'espace d'adressage du FILS.
86   *  Donc, les variablses du père ne sont pas modifiées
87   */
88  int main(int argc, char **argv) {
89      printf("\n\n\n\nCODES D'ÉTAT DE PROCESSUS \nVoici 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 :\n\n"
90
91                 "D    en sommeil non interruptible
                    ↪  (normalement entrées et sorties) ;\n"
92                 "R    s'exécutant ou pouvant s'exécuter (dans
                    ↪  la file d'exécution) ;\n"
93                 "S    en sommeil interruptible (en attente
                    ↪  d'un événement pour finir) ;\n"
94                 "T    arrêté, par un signal de contrôle des
                    ↪  tâches ou parce qu'il a été tracé ;\n"
95                 "W    pagination (non valable depuis le noyau
                    ↪  2.6.xx) ;\n"
96                 "X    tué (ne devrait jamais être vu) ;\n"
97                 "Z    processus zombie (<defunct>), terminé
                    ↪  mais pas détruit par son parent.\n\n"
98
99          "Pour les formats BSD et quand le mot-clé stat est
             ↪  utilisé, les caractères supplémentaires suivants
             ↪  peuvent être affichés :\n\n"
100
101                "<    haute priorité (non poli pour les autres
                    ↪  utilisateurs) ;\n"
102                "N    basse priorité (poli pour les autres
                    ↪  utilisateurs) ;\n"
```

```
103                 "L    les pages du processus sont verrouillées
                    ↪  en mémoire;\n"
104                 "s    meneur de session ;\n"
105                 "l    possède plusieurs processus légers («
                    ↪  multi-thread », utilisant CLONE_THREAD
                    ↪  comme NPTL pthreads le fait) ;\n"
106                 "+    dans le groupe de processus au premier
                    ↪  plan.\n\n\n\n\n\n\n");
107
108
109
110     // Entiers à augmenter dans le fils pour prouver l'espace
        ↪  d'adressage commun
111     int a = 5, b = 8;
112     // Récupérer la valeur de retour de la fonction créant le
        ↪  proocess fils
113     int forkRetNum;
114
115     printf("PID du père = %d\n", getpid());
116     printf ("Ces 2 variables sont créées et initialisées par
        ↪  le père :\n");
117     printf("a = %d\n", a);
118     printf("b = %d\n", b);
119
120     int dataSize = 2048;
121     char dataLock[dataSize];
122     if (lock_memory(dataLock, dataSize) == -1)
123             perror("Error with locking memory\n");
124     else
125             printf ("\nDe la mémoire a été réservée en RAM
                ↪  par le père\n");
126
127     continueProgram();
128
```

```
129     printf("\nCeci est avant que le père ne crée un
        ↪  thread\n\n");
130
131     continueProgram();
132
133     pthread_t tid;
134     // Crée 3 threads
135     for (unsigned int i = 0; i < 3; i++)
136         pthread_create(&tid, NULL, threadCreation, (void
            ↪  *)&tid);
137
138     printf("%s", "");
139     //printf("Les threads ont été créés par le père\n");
140
141     //continueProgram();
142     forkRetNum = fork();
143
144     if(forkRetNum == 0)
145     { // La création du fils s'est-elle correctement produite
        ↪  ?
146         printf("Le processus fils vient d'être créé. La suite
            ↪  est affichée par le fils.\n");
147         // a = 10 mais seulement la variable 'a' du fils et
            ↪  non celle du père
148         a = a + 5;
149         printf("Maintenant, a = %d et ce, uniquement dans
            ↪  l'espace d'adressage du fils\n", a);
150
151         // b = 10 mais seulement la variable 'b' du fils et
            ↪  non celle du père
152         b = b + 2;
153         printf("Maintenant, b = %d et ce, uniquement dans
            ↪  l'espace d'adressage du fils\n", b);
154
```

```
155         //lock_memory();
156
157         printf("PID (du fils, donc) = %d\n", getpid());
158         printf("PID du père = %d\n", getppid());
159         printf("a + b = %d.\n", a + b);
160         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");
161         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");
162         printf("RSS signifie Resident Set Size et montre la
            ↪  quantité de RAM utilisée au moment de la sortie
            ↪  de la commande. "
163                 "Il convient également de noter qu'il affiche
                    ↪  toute la pile de mémoire physiquement
                    ↪  allouée.\n\n");
164         printf("VSZ est l'abréviation de Virtual Memory Size.
            ↪  C'est la quantité totale de mémoire à laquelle un
            ↪  processus peut hypothétiquement accéder. "
165                 "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.\n");
166         printf("\n\nDans une autre fenêtre de terminal,
            ↪  entrez la commande 'cat /proc/$PID/status' pour
            ↪  voir les informations du process !\n");
167
```

```
168         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");
169         printf("\n\nCe signal indique qu'un processus fils
            ↪ s'est arrêté ou a fini son exécution. Par défaut
            ↪ ce signal est ignoré. SIGHUP : n°1");
170         printf("\n\nUn processus qui effectue une division
            ↪ par zéro reçoit un signal SIGFPE : n°8");
171
172         //continueProgram();
173         while(1){} // Faire en sorte que le fils attende,
            ↪ mais en étant en état d'exécution. Un simple 'ps
            ↪ -aux' le montrera
174         exit(0);
175     }
176     else if (forkRetNum > 0)
177     {  // Est-ce le process parent ?
178         printf("Ceci est le process parent et le PID est :
            ↪ %d\n", getpid());
179     }
180     else
181     { // Y a-t-il eu une erreur lors de la création du
        ↪ process fils ?
182         printf("Problème durant la duplication\n");
183         exit(EXIT_FAILURE);
184     }
185     // Code du père
186     wait(0); // Pour éviter de faire du fils un zombie
187     printf ("Le fils est terminé\n");
188     printf("PID (du père, donc) = %d\n", getpid());
189     printf("PPID (id du process à l'origine de la création du
        ↪ programme) = %d\n", getppid());
```

```
190     // La somme est bien de 13 et non 20 puisque la somme fut
        ↪ faite par le fils, mais uniquement avec ses propres
        ↪ variables et non celles du père
191     printf("a + b = %d.\n", a + b);
192     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");
193     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");
194
195     if (unlock_memory(dataLock, dataSize) == -1)
196             perror("Error with locking memory\n");
197     else
198             printf ("Memory unlocked in RAM\n");
199
200     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 et leurs états. Pour le
        ↪ terminer, faites un 'kill %d' dans une autre fenêtre
        ↪ de terminal ou faites un CTRL + C ici\n", getpid());
201
202     pthread_join(tid, NULL);
203     printf("After Thread\n");
204
205     while(1){} // Simplement pour faire attendre le père. Un
        ↪ simple 'ps -aux' montrera son état
206
207     exit(0);
208 }
```

7.1.2 forkMoreProcesses.c

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>
#include <mm_malloc.h>
#include <spawn.h>
#include <sys/mman.h>
#include <signal.h>
#include <ctype.h>
#include <string.h>
#include <sys/wait.h>
#include <math.h>
#include <sys/stat.h>
#include <fcntl.h>


// Pour compiler avec les threads et que les fonctions mathématiques soient reconnues par le compilateur (grâce à l'option '-lm'):
// gcc -pthread -o fork fork.c -lm


// Nombre de duplication qui seront effectuées via 'fork'
int nbForks = 5;

/**
 Cette fonction permet à l'utilisateur de choisir à quel moment reprendre
 l'exécution du programme pour lui laisser le temps de faire les manipulations qu'il désire
```

```
29  */
30 void continueProgram()
31 {
32     printf("Pour continuer le programme, entrez 'continue' ou
       ↪  'c' : ");
33     int c, numberCounter = 0, letterCounter = 0;
34     while ((c = getchar()) != 'c')
35         if (isalpha(c))
36             letterCounter++;
37         else if (isdigit(c))
38             numberCounter++;
39 }
40
41 /**
42  Crée un fichier text de près de 10Mo
43  */
44 void create10MiBFile()
45 {
46     FILE * fp=fopen("BigFile.txt", "w");
47     for (unsigned int i = 0; i < 550000; i++) // 550000 to
       ↪  get to 10Mo with the string to put in
48     {
49         fprintf(fp, "forkMoreProcesses\n");
50     }
51     fclose(fp);
52 }
53
54 /**
55  Cette fonction permet de charger le contenu d'un fichier en
   ↪  RAM.
56
57  @param fileName Pour le nom du fichier à lire
58  */
59 void read_file(char fileName[])
```

```
60 {
61     FILE* ptr;
62     char ch;
63 
64     // Ouvre le fichier en mode lecture
65     ptr = fopen(fileName, "r");
66 
67     if (NULL == ptr) {
68         printf("Le fichier ne peut être ouvert \n");
69     }
70 
71     // Charge ce qui est dans le fichier
72     // caractère par caractère
73     do {
74         ch = fgetc(ptr);
75         // Ceci permet d'afficher le contenu du fichier
76         //printf("%c", ch);
77 
78         // Regarde si la fin du fichier n'est pas atteinte
79         // Si la fin est atteinte, alors, la lecture s'arrête
80     } while (ch != EOF);
81 
82     // Ferme le fichier
83     fclose(ptr);
84 }
85 
86 /**
87  * Le FORK duplique l'aspace d'adressage.
88  * Donc, le fils fera les additions de son côté, dans son
    ↪  propre espace d'adressage
89  *
90  * Du côté du père, l'addition ne sera pas faite puisqu'il
    ↪  exécutera seulement le code suivant le "if" qui vérifie
    ↪  le bon code de retour de l'appel à fork()
```

```
91   *
92   * Vu que l'espace d'adressage est dupliqué lors du FORK, les
     ↪  variables ne seront modifiées que dans
93   *  l'espace d'adressage du FILS.
94   *  Donc, les variablses du père ne sont pas modifiées
95   */
96  int main(int argc, char **argv) {
97
98      printf("\n\n\n\nCODES D'ÉTAT DE PROCESSUS \nVoici 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 :\n\n"
99
100             "D    en sommeil non interruptible (normalement
                ↪  entrées et sorties) ;\n"
101             "R    s'exécutant ou pouvant s'exécuter (dans la
                ↪  file d'exécution) ;\n"
102             "S    en sommeil interruptible (en attente d'un
                ↪  événement pour finir) ;\n"
103             "T    arrêté, par un signal de contrôle des tâches
                ↪  ou parce qu'il a été tracé ;\n"
104             "W    pagination (non valable depuis le noyau
                ↪  2.6.xx) ;\n"
105             "X    tué (ne devrait jamais être vu) ;\n"
106             "Z    processus zombie (<defunct>), terminé mais
                ↪  pas détruit par son parent.\n\n"
107
108             "Pour les formats BSD et quand le mot-clé stat est
                ↪  utilisé, les caractères supplémentaires
                ↪  suivants peuvent être affichés :\n\n"
109
110             "<    haute priorité (non poli pour les autres
                ↪  utilisateurs) ;\n"
```

```
111             "N    basse priorité (poli pour les autres
                ↪  utilisateurs) ;\n"
112             "L    les pages du processus sont verrouillées en
                ↪  mémoire;\n"
113             "s    meneur de session ;\n"
114             "l    possède plusieurs processus légers («
                ↪  multi-thread », utilisant CLONE_THREAD comme
                ↪  NPTL pthreads le fait) ;\n"
115             "+    dans le groupe de processus au premier
                ↪  plan.\n\n\n\n\n\n\n");
116
117
118
119     printf("\nDans une autre fenêtre de terminal, entrez la
        ↪  commande 'top' pour voir quels process sont en cours
        ↪  et plsu d'informations, dont leur utilisation de la
        ↪  mémoire et ce, en temps réel !\n Ceci permettra
        ↪  d'observer que 5 lignes seront créées dans le tableau
        ↪  du résultat de la commande car fork() crée des
        ↪  process à part entière.\n\n");
120     continueProgram();
121
122     create10MiBFile();
123
124     // Récupérer la valeur de retour des nbForks fork dans un
        ↪  tableau
125     int forkRetNums[nbForks];
126
127     printf("PID du père = %d\n", getpid());
128     for (unsigned int i = 0; i < nbForks; i++)
129     {
130         read_file("BigFile.txt");
131         forkRetNums[i] = fork();
132
```

```
133         if(forkRetNums[i] == 0)
134         {  // Est-ce le process fils ?
135             printf("Ceci est le process fils et le PID est :
                ↪  %d\n", getpid());
136 
137             //continueProgram();
138             while(1){} // Faire en sorte que le fils attende,
                ↪  mais en étant en état d'exécution. Un simple
                ↪  'ps -aux' le montrera
139             exit(0);
140         }
141         else if (forkRetNums[i] > 0)
142         {  // Est-ce le process parent ?
143             printf("Ceci est le process parent et le PID est
                ↪  : %d\n", getpid());
144         }
145         else
146         { // Y a-t-il eu une erreur lors de la création du
            ↪  process fils ?
147             printf("Problème durant la duplication\n");
148             exit(EXIT_FAILURE);
149         }
150         if(i == nbForks - 1)
151                 printf("\n\nLes fils sont en train de tourner
                    ↪  à l'infini via un 'while(1)' pour montrer
                    ↪  la mémoire qu'ils occupent via la
                    ↪  commande 'top' (cfr 'ps -aux'). Pour les
                    ↪  arrêter, dans une autre fenêtre de
                    ↪  terminal, entrez la commande 'kill
                    ↪  {$PID_du_premier_fils..$PID_su_dernier_fils'
                    ↪  !\n\n");
152     }
153     // Code du père
```

```
154     // Cette boucle est pour éviter de créer des zombies en
        ↪ les tuant via un autre terminal
155     for (unsigned int i = 0; i < nbForks; i++)
156     {
157         wait((void*)(intptr_t) forkRetNums[i]);
158     }
159     printf("\nLes fils sont terminés\n");
160 
161     printf("PID (du père, donc) = %d\n", getpid());
162     printf("PPID (id du process à l'origine de la création du
        ↪ programme) = %d\n", getppid());
163 
164     printf("\nVu que lorsque vous tuez 1 process via un autre
        ↪ terminal, l'entrée  liée au process de l'affichage
        ↪ généré par la commande 'top' disparait, cela prouve
        ↪ que chaque process a bien son propre espace
        ↪ d'adressage.\n\n");
165 
166     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 et leurs états. Pour le
        ↪ terminer, faites un 'kill %d' dans une autre fenêtre
        ↪ de terminal ou faites un CTRL + C ici\n", getpid());
167 
168     while(1){} // Simplement pour faire attendre le père. Un
        ↪ simple 'ps -aux' montrera son état
169 
170     exit(0);
171 }
```

7.2 VFORK

7.2.1 vfork.c

```
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 crée de nouveaux
39      return NULL;
40  }
41
42  /**
43   * Le VFORK duplique l'espace d'adressage.
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
49   *  l'espace d'adressage du PERE (qui est aussi celui du
    ↪  fils)
50   *  Donc, les variables du père sont modifiées,
51   *   ce qui permet la prise en compte de la modification des
    ↪  valeurs des variables, modifications§ faites par le fils
52   */
```

```
53 int main(int argc, char **argv)
54 {
55     printf("\n\n\n\nCODES D'ÉTAT DE PROCESSUS \nVoici 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 :\n\n"
56
57                 "D    en sommeil non interruptible
                    ↪  (normalement entrées et sorties) ;\n"
58                 "R    s'exécutant ou pouvant s'exécuter (dans
                    ↪  la file d'exécution) ;\n"
59                 "S    en sommeil interruptible (en attente
                    ↪  d'un événement pour finir) ;\n"
60                 "T    arrêté, par un signal de contrôle des
                    ↪  tâches ou parce qu'il a été tracé ;\n"
61                 "W    pagination (non valable depuis le noyau
                    ↪  2.6.xx) ;\n"
62                 "X    tué (ne devrait jamais être vu) ;\n"
63                 "Z    processus zombie (<defunct>), terminé
                    ↪  mais pas détruit par son parent.\n\n"
64
65         "Pour les formats BSD et quand le mot-clé stat est
           ↪  utilisé, les caractères supplémentaires suivants
           ↪  peuvent être affichés :\n\n"
66
67                 "<    haute priorité (non poli pour les autres
                    ↪  utilisateurs) ;\n"
68                 "N    basse priorité (poli pour les autres
                    ↪  utilisateurs) ;\n"
69                 "L    les pages du processus sont verrouillées
                    ↪  en mémoire;\n"
70                 "s    meneur de session ;\n"
```

```
71                  "l    possède plusieurs processus légers («
                    ↪  multi-thread », utilisant CLONE_THREAD
                    ↪  comme NPTL pthreads le fait) ;\n"
72                  "+    dans le groupe de processus au premier
                    ↪  plan.\n\n\n\n\n\n\n");
73
74
75
76      // Entiers à augmenter dans le fils pour prouver l'espace
         ↪  d'adressage commun
77      int a = 5, b = 8;
78      // Récupérer la valeur de retour de la fonction créant le
         ↪  proocess fils
79      int vforkRetNum;
80
81      printf("PID du père = %d\n", getpid());
82      printf ("Ces 2 variables sont créées et initialisées par
         ↪  le père :\n");
83      printf("a = %d\n", a);
84      printf("b = %d\n", b);
85
86      continueProgram();
87
88      printf("\nCeci est avant que le père ne crée un
         ↪  thread\n\n");
89
90      continueProgram();
91
92      pthread_t tid;
93      // Crée 3 threads
94      for (unsigned int i = 0; i < 3; i++)
95          pthread_create(&tid, NULL, threadCreation, (void
             ↪  *)&tid);
96
```

```
97      printf("%s", "");
98
99      vforkRetNum = vfork();
100
101     if(vforkRetNum == 0)
102     { // La création du fils s'est-elle correctement produite
        ↪  ?
103         printf("Le processus fils vient d'être créé. La suite
            ↪  est affichée par le fils.\n");
104         // a = 10
105         a = a + 5;
106         printf("Maintenant, a = %d et ce, dans l'espace
            ↪  d'adressage du fils qui est aussi celui du
            ↪  père\n", a);
107
108         // b = 10
109         b = b + 2;
110         printf("Maintenant, b = %d et ce, dans l'espace
            ↪  d'adressage du fils qui est aussi celui du
            ↪  père\n", b);
111
112         printf("PID (du fils, donc) = %d\n", getpid());
113         printf("PID du père = %d\n", getppid());
114 //          printf("Value of vfork is %d.\n", vforkRetNum); //
    ↪  Indiquer la valeur de retour de la fonciton créant le
    ↪  process
115         printf("a + b = %d.\n", a + b); // line b
116         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");
```

```
117         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");
118         printf("RSS signifie Resident Set Size et montre la
            ↪  quantité de RAM utilisée au moment de la sortie
            ↪  de la commande. "
119                 "Il convient également de noter qu'il affiche
                    ↪  toute la pile de mémoire physiquement
                    ↪  allouée.\n\n");
120         printf("VSZ est l'abréviation de Virtual Memory Size.
            ↪  C'est la quantité totale de mémoire à laquelle un
            ↪  processus peut hypothétiquement accéder. "
121                 "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.\n");
122         printf("\n\nDans une autre fenêtre de terminal,
            ↪  entrez la commande 'cat /proc/$PID/status' pour
            ↪  voir les informations du process !\n");
123
124         // wait est un processus bloquant. Donc, la suite ne
            ↪  sera pas exécutée tant qu'une condition ne sera
            ↪  pas remplie. Si l'on met un pointeur d'un nombre,
            ↪  alors, on pourra récupérer le code de terminaison
            ↪  du processus enfant. Pareil pour exit
125
126         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 %d' !\n", getpid());
```

```
127         while(1){} // Faire en sorte que le fils attende,
            ↪  mais en étant en état d'exécution. Un simple 'ps
            ↪  -aux' le montrera
128         exit(0);
129     }
130     else if (vforkRetNum > 0)
131     {  // Est-ce le process parent ?
132         printf("Ceci est le process parent et le PID est :
            ↪  %d\n", getpid());
133     }
134     else
135     { // Y a-t-il eu une erreur lors de la création du
        ↪  process fils ?
136         printf("Problème durant le vfork\n");
137         exit(EXIT_FAILURE);
138     }
139
140     wait(0); // Pour éviter de faire du fils un zombie
141     printf ("Le fils est terminé\n");
142     printf("PID (du père, donc) = %d\n", getpid());
143     printf("PPID (id du process à l'origine de la création du
        ↪  programme) = %d\n", getppid());
144     // La somme est bien de 20 puisque la somme fut faite par
        ↪  le fils avec les mêmes variables que celles du père
145     printf("a + b = %d.\n", a + b);
146     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");
147     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");
148
```

```
149     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 et leurs états. Pour le
        ↪ terminer, faites un 'kill %d' dans une autre fenêtre
        ↪ de terminal ou faites un CTRL+C ici\n", getpid());
150
151     pthread_join(tid, NULL);
152     printf("Après les threads\n");
153
154     while(1){} // Simplement pour faire attendre le père que
        ↪ l'on fasse un 'ps -aux' pour pouvoir voir son état
155     exit(0);
156 }
```

7.2.2 vforkMoreProcesses.c

```
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 #include <sys/wait.h>
13 #include <math.h>
14
15
16
```

```
17 // Pour compiler avec les threads et que les fonctions
   ↪ mathématiques soient reconnues par le compilateur (grâce
   ↪ à l'option '-lm'):
18 // gcc -pthread -o fork fork.c -lm
19
20
21 // Nombre de duplication qui seront effectuées via 'fork'
22 int nbVforks = 5;
23
24 /**
25  Cette fonction permet à l'utilisateur de choisir à quel
   ↪ moment reprendre
26  l'exécution du programme pour lui laisser le temps de faire
   ↪ les manipulations qu'il désire
27  */
28 void continueProgram()
29 {
30     printf("Pour continuer le programme, entrez 'continue' ou
       ↪ 'c' : ");
31     int c, numberCounter = 0, letterCounter = 0;
32     while ((c = getchar()) != 'c')
33         if (isalpha(c))
34             letterCounter++;
35         else if (isdigit(c))
36             numberCounter++;
37 }
38
39 /**
40  Cette fonction permet de charger le contenu d'un fichier en
   ↪ RAM.
41
42  @param fileName Pour le nom du fichier à lire
43  */
44 void read_file(char fileName[])
```

```
45 {
46     FILE* ptr;
47     char ch;
48 
49     // Ouvre le fichier en mode lecture
50     ptr = fopen(fileName, "r");
51 
52     if (NULL == ptr) {
53         printf("Le fichier ne peut être ouvert \n");
54     }
55 
56     // Charge ce qui est dans le fichier
57     // caractère par caractère
58     do {
59         ch = fgetc(ptr);
60         // Ceci permet d'afficher le contenu du fichier
61         //printf("%c", ch);
62 
63         // Regarde si la fin du fichier n'est pas atteinte
64         // Si la fin est atteinte, alors, la lecture s'arrête
65     } while (ch != EOF);
66 
67     // Ferme le fichier
68     fclose(ptr);
69 }
70 
71 /**
72  Crée un fichier text de près de 10Mo
73  */
74 void create10MiBFile()
75 {
76     FILE * fp = fopen("BigFile.txt", "w");
77     for (unsigned int i = 0; i < 550000; i++) // 550000 to
       ↪  get to 10Mo with the string to put in
```

```
78      {
79          fprintf(fp, "forkMoreProcesses\n");
80      }
81      fclose(fp);
82  }
83
84  /**
85   * Le FORK duplique l'aspace d'adressage.
86   * Donc, le fils fera les additions de son côté, dans son
    ↪  propre espace d'adressage
87   *
88   * Du côté du père, l'addition ne sera pas faite puisqu'il
    ↪  exécutera seulement le code suivant le "if" qui vérifie
    ↪  le bon code de retour de l'appel à fork()
89   *
90   * Vu que l'espace d'adressage est dupliqué lors du FORK, les
    ↪  variables ne seront modifiées que dans
91   *  l'espace d'adressage du FILS.
92   *  Donc, les variablses du père ne sont pas modifiées
93   */
94  int main(int argc, char **argv) {
95
96      printf("\n\n\n\nCODES D'ÉTAT DE PROCESSUS \nVoici 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 :\n\n"
97
98              "D    en sommeil non interruptible (normalement
                ↪  entrées et sorties) ;\n"
99              "R    s'exécutant ou pouvant s'exécuter (dans la
                ↪  file d'exécution) ;\n"
100             "S    en sommeil interruptible (en attente d'un
                ↪  événement pour finir) ;\n"
```

```
101             "T    arrêté, par un signal de contrôle des tâches
                ↪  ou parce qu'il a été tracé ;\n"
102             "W    pagination (non valable depuis le noyau
                ↪  2.6.xx) ;\n"
103             "X    tué (ne devrait jamais être vu) ;\n"
104             "Z    processus zombie (<defunct>), terminé mais
                ↪  pas détruit par son parent.\n\n"
105
106             "Pour les formats BSD et quand le mot-clé stat est
                ↪  utilisé, les caractères supplémentaires
                ↪  suivants peuvent être affichés :\n\n"
107
108             "<    haute priorité (non poli pour les autres
                ↪  utilisateurs) ;\n"
109             "N    basse priorité (poli pour les autres
                ↪  utilisateurs) ;\n"
110             "L    les pages du processus sont verrouillées en
                ↪  mémoire;\n"
111             "s    meneur de session ;\n"
112             "l    possède plusieurs processus légers («
                ↪  multi-thread », utilisant CLONE_THREAD comme
                ↪  NPTL pthreads le fait) ;\n"
113             "+    dans le groupe de processus au premier
                ↪  plan.\n\n\n\n\n\n\n");
114
115
116
117
118
119
120     printf("\nDans une autre fenêtre de terminal, entrez la
        ↪  commande 'top' pour voir quels process sont en cours
        ↪  et plsu d'informations, dont leur utilisation de la
        ↪  mémoire et ce, en temps réel !\n\n");
```

```
121     continueProgram();
122 
123     create10MiBFile();
124 
125     // Récupérer la valeur de retour des nbVforks vfork
126     int vforkRetNums[nbVforks];
127 
128     printf("PID du père = %d\n", getpid());
129 
130     // Nécessaire de faire une suite de vfork en if car, lors
        ↪  d'un vfork
131     // Le père est mis en pause et l'espace d'adressage est
        ↪  partagé avec le père
132     vforkRetNums[0] = vfork();
133     if (vforkRetNums[0] == 0)
134     {
135         printf("Ceci est le process fils et le PID est :
            ↪  %d\n", getpid());
136         read_file("BigFile.txt");
137         vforkRetNums[1] = vfork();
138         if (vforkRetNums[1] == 0)
139         {
140             printf("Ceci est le process fils et le PID est :
                ↪  %d\n", getpid());
141             read_file("BigFile.txt");
142             vforkRetNums[2] = vfork();
143             if (vforkRetNums[2] == 0)
144             {
145                 printf("Ceci est le process fils et le PID
                    ↪  est : %d\n", getpid());
146                 read_file("BigFile.txt");
147                 vforkRetNums[3] = vfork();
148                 if (vforkRetNums[3] == 0)
149                 {
```

```
150                 printf("Ceci est le process fils et le
                    ↪  PID est : %d\n", getpid());
151                 read_file("BigFile.txt");
152                 vforkRetNums[4] = vfork();
153                 if (vforkRetNums[4] == 0)
154                 {
155                     read_file("BigFile.txt");
156                     printf("Ceci est le process fils et
                        ↪  le PID est : %d\n", getpid());
157                     printf("\nVous pouvez observer que,
                        ↪  dans le tableau de la commande
                        ↪  'top', une seule ligne concernant
                        ↪  ce programme a été créée. Cela
                        ↪  signifie que vfork() a créé des
                        ↪  process en créant des threads. A
                        ↪  ce stade, les fils créés par
                        ↪  vfork ne sont que des threads.
                        ↪  Ils deviendront des process
                        ↪  lorsque les fils fils appelleront
                        ↪  une fonction de la famille
                        ↪  exec().\n\n");
158                     printf("\n\nLe fils est en train de
                        ↪  tourner à l'infini via un
                        ↪  'while(1)' pour montrer la
                        ↪  mémoire utilisée par l'espace des
                        ↪  process lié à ce programme
                        ↪  (puisque l'espace d'adressage est
                        ↪  partagé entre le père et le
                        ↪  fils).\n\nPour l'arrêter, dans
                        ↪  une autre fenêtre de terminal,
                        ↪  entrez la commande 'kill %d'
                        ↪  !\n", getpid());
```

```
159                         while(1){} // Faire en sorte que le
                             ↪  fils attende, mais en étant en
                             ↪  état d'exécution. Un simple 'ps
                             ↪  -aux' le montrera
160                         exit(0);
161                     }
162                     else if (vforkRetNums[4] > 0)
163                     {  // Est-ce le process parent ?
164                         printf("Ceci est le process parent et
                             ↪  le PID est : %d\n", getpid());
165                     }
166                     else
167                     { // Y a-t-il eu une erreur lors de la
                         ↪  création du process fils ?
168                         printf("Problème durant la
                             ↪  duplication\n");
169                         exit(EXIT_FAILURE);
170                     }
171                     printf("\n\nLe fils est en train de
                         ↪  tourner à l'infini via un 'while(1)'
                         ↪  pour montrer la mémoire utilisée par
                         ↪  l'espace des process lié à ce
                         ↪  programme (puisque l'espace
                         ↪  d'adressage est partagé entre le père
                         ↪  et le fils).\n\nPour l'arrêter, dans
                         ↪  une autre fenêtre de terminal, entrez
                         ↪  la commande 'kill %d' !\n",
                         ↪  getpid());
172     wait((void*)(intptr_t) vforkRetNums[4]); // Eviter de
         ↪  faire du fils un zombie
173                     while(1){} // Faire en sorte que le fils
                         ↪  attende, mais en étant en état
                         ↪  d'exécution. Un simple 'ps -aux' le
                         ↪  montrera
```

```
174                     exit(0);
175                 }
176                 else if (vforkRetNums[3] > 0)
177                 {  // Est-ce le process parent ?
178                     printf("Ceci est le process parent et le
                        ↪  PID est : %d\n", getpid());
179                 }
180                 else
181                 { // Y a-t-il eu une erreur lors de la
                    ↪  création du process fils ?
182                     printf("Problème durant la
                        ↪  duplication\n");
183                     exit(EXIT_FAILURE);
184                 }
185                 printf("\n\nLe fils est en train de tourner à
                    ↪  l'infini via un 'while(1)' pour montrer
                    ↪  la mémoire utilisée par l'espace des
                    ↪  process lié à ce programme (puisque
                    ↪  l'espace d'adressage est partagé entre le
                    ↪  père et le fils).\n\nPour l'arrêter, dans
                    ↪  une autre fenêtre de terminal, entrez la
                    ↪  commande 'kill %d' !\n", getpid());
186     wait((void*)(intptr_t) vforkRetNums[3]); // Eviter de
        ↪  faire du fils un zombie
187                 while(1){} // Faire en sorte que le fils
                    ↪  attende, mais en étant en état
                    ↪  d'exécution. Un simple 'ps -aux' le
                    ↪  montrera
188                 exit(0);
189             }
190             else if (vforkRetNums[2] > 0)
191             {  // Est-ce le process parent ?
192                 printf("Ceci est le process parent et le PID
                    ↪  est : %d\n", getpid());
```

```
193             }
194             else
195             { // Y a-t-il eu une erreur lors de la création
                ↪  du process fils ?
196                 printf("Problème durant la duplication\n");
197                 exit(EXIT_FAILURE);
198             }
199             printf("\n\nLe fils est en train de tourner à
                ↪  l'infini via un 'while(1)' pour montrer la
                ↪  mémoire utilisée par l'espace des process lié
                ↪  à ce programme (puisque l'espace d'adressage
                ↪  est partagé entre le père et le
                ↪  fils).\n\nPour l'arrêter, dans une autre
                ↪  fenêtre de terminal, entrez la commande 'kill
                ↪  %d' !\n", getpid());
200         wait((void*)(intptr_t) vforkRetNums[2]); // Eviter de
            ↪  faire du fils un zombie
201             while(1){} // Faire en sorte que le fils attende,
                ↪  mais en étant en état d'exécution. Un simple
                ↪  'ps -aux' le montrera
202             exit(0);
203         }
204         else if (vforkRetNums[1] > 0)
205         {  // Est-ce le process parent ?
206             printf("Ceci est le process parent et le PID est
                ↪  : %d\n", getpid());
207         }
208         else
209         { // Y a-t-il eu une erreur lors de la création du
            ↪  process fils ?
210             printf("Problème durant la duplication\n");
211             exit(EXIT_FAILURE);
212         }
```

```
213         printf("\n\nLe fils est en train de tourner à
            ↪ l'infini via un 'while(1)' pour montrer la
            ↪ mémoire utilisée par l'espace des process lié à
            ↪ ce programme (puisque l'espace d'adressage est
            ↪ partagé entre le père et le fils).\n\nPour
            ↪ l'arrêter, dans une autre fenêtre de terminal,
            ↪ entrez la commande 'kill %d' !\n", getpid());
214     wait((void*)(intptr_t) vforkRetNums[1]); // Eviter de
        ↪ faire du fils un zombie
215         while(1){} // Faire en sorte que le fils attende,
            ↪ mais en étant en état d'exécution. Un simple 'ps
            ↪ -aux' le montrera
216         exit(0);
217     }
218     else if (vforkRetNums[0] > 0)
219     {  // Est-ce le process parent ?
220         printf("Ceci est le process parent et le PID est :
            ↪ %d\n", getpid());
221     }
222     else
223     { // Y a-t-il eu une erreur lors de la création du
        ↪ process fils ?
224         printf("Problème durant la duplication\n");
225         exit(EXIT_FAILURE);
226     }
227
228     // Code du père
229     wait((void*)(intptr_t) vforkRetNums[0]); // Eviter de
        ↪ faire du fils un zombie
230     printf("Les fils sont terminés\n");
```

```
231     printf("\n\nAu fur et à mesure que vous killiez les
        ↪  process un à un, vous avez pu observer que la mémoire
        ↪  occupée par le process courant n'a pas diminué, ce
        ↪  qui prouve que l'espace d'adressage est partagé entre
        ↪  un process père et un process fils si le moyen de
        ↪  duplication de process est 'vfork()' !\n");
232     printf("\n\nPID (du père, donc) = %d\n", getpid());
233     printf("\nPPID (id du process à l'origine de la création
        ↪  du programme) = %d\n", getppid());
234
235     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");
236
237     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 et leurs états. Pour le
        ↪  terminer, faites un 'kill %d' dans une autre fenêtre
        ↪  de terminal ou faites un CTRL+C ici. Vous verrez
        ↪  alors dans l'affichage généré par la commande 'top'
        ↪  que la mémoire occupée par le process courant est
        ↪  libérée\n", getpid());
238
239     while(1){} // Simplement pour faire attendre le père. Un
        ↪  simple 'ps -aux' montrera son état
240
241     exit(0);
242 }
```

7.2.3 vforkParentPAUSED.c

```
1 #include <sys/types.h>
2 #include <unistd.h>
```

```c
#include <stdio.h>
#include <stdlib.h>

/**
 vfork() affiche le contenu du 'if{} else{}' deux fois, d'abord dans l'enfant, puis dans le parent.
 Vu que les deux processus partagent le même espace d'adressage, la première sortie contient la valeur du PID correspondant au process fils.
 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.
 */
int main()
{

    printf("\n\n\nvfork() affiche le contenu du 'if{} else{}' deux fois, d'abord dans l'enfant, puis dans le parent.\n"
        "Vu que les deux processus partagent le même espace d'adressage, la première sortie contient la valeur du PID correspondant au process fils.\n"
        "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.\n\n\n");
    //pid_t pid = vfork();  // Crée le process fils

    printf("Process parent avant le 'if{} else{}': %d\n", getpid());

    pid_t pid = vfork();  // Crée le process fils

    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  }
```

7.3 CLONE

7.3.1 clone.c

```c
// Il est nécessaire de définir _GNU_SOURCE pour avoir acces à clone(2) et aux flags CLONE_*

#define _GNU_SOURCE
#include <sched.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sched.h>
#include <linux/sched.h>


/**
 child
 */
static int child_func(void* arg)
{
    char* buffer = (char*)arg;
    printf("Child sees buffer = \"%s\"\n", buffer);
    strcpy(buffer, "Hello from child");
    return 0;
}

// Cette fonction a pour but de montrer les informations liées à un même processus via son PID
void processStatus(int pid)
{
```

```
30      char parentProcessStatus[18] = "/proc/"; // 18 car si le
        ↪  PID est à 5 chiffres, alors, le tableau de char sera
        ↪  de longueur 18
31
32      char *num;
33      char buff[100];
34
35      // Ce if permet de concaténer un pid_t à un char *
36      if (asprintf(&num, "%d", pid) == -1) {
37          perror("asprintf");
38      } else {
39          strcat(strcpy(buff, parentProcessStatus), num);
40      }
41
42      strcat(buff, "/status");
43      if (fork() == 0)
44              execl("/bin/cat", "cat", buff, (char *)0);
45  }
46
47  /**
48   Ici, clone() est utilisé de deux manières : une fois avec le
    ↪  flag CLONE_VM (CLONE_VM = clone virtual memory) et une
    ↪  fois sans.
49   Un buffer est passé dans le processus enfant, et le
    ↪  processus enfant y écrit un string.
50   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').
51   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.
52
```

```
53  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.
54  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.
55  */
56  int main(int argc, char** argv)
57  {
58      printf("Selon l'ordonnanceur, les 2 lignes à propos du
        ↪  PID du père et du fils peuvent être écrites plus
        ↪  qu'une fois par ligne.");
59      // Alloue un stack pour la tâche du fils
60      const int STACK_SIZE = 65536;
61      char* stack = malloc(STACK_SIZE);
62      if (!stack) { // Si 'stack' n'a pas été correctement créé
63          perror("malloc");
64          exit(1);
65      }
66
67      // Lorsqu'il est appelé avec l'argument 'vm' en ligne de
        ↪  commande, active le flag CLONE_VM.
68      unsigned long flags = 0;
69      if (argc > 1 && !strcmp(argv[1], "vm")) {
70
71          /**
72           int clone(int (*fn)(void *), void *child_stack,
73                     int flags, void *arg, ...
74                     pid_t *ptid, struct user_desc *tls, pid_t
    ↪  *ctid );
75           */
76
```

```
77          /**
78           Lorsque le processus enfant est créé avec clone(),
    ↪  il exécute la fonction fn(arg).
79           (Cela diffère de fork(2) dans lequel l'exécution
    ↪  continue dans le fils à partir du point d'appel de
    ↪  fork(2).)
80           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.
81           */
82          /**
83           CLONE_VM (depuis Linux 2.0)
84                       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.
85                       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.
86
87                       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).
88           */
89          flags |= CLONE_VM; // 'flags' vaudra 'CLONE_VM' ou
           ↪  non en fonction du fait que l'option 'vm' soit
           ↪  spécifiée ou non.
90      }
91
```

```
92      char buffer[100];
93      strcpy(buffer, "Hello from parent"); // Ecrit 'hello from
        ↪  parent' dans le buffer
94
95      int cloneRetNum;
96      // Clone le processus père
97      // Seul appel à 'clone'. Pour avoir les différentes
        ↪  exécutions, il faut ajouter 'vm' comme argument lors
        ↪  de l'appel en ligne de commande
98      // Vu que lorsque CLONE_VM est défini, l'espace
        ↪  d'adressage mémoire est partaé,
99      // 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'
100     cloneRetNum = clone(child_func, stack + STACK_SIZE, flags
        ↪  | SIGCHLD, buffer);
101     // Selon l'ordonnanceur, les 2 lignes suivantes (à propos
        ↪  du PID du père et du fils) peuvent être écrites plus
        ↪  qu'une fois par ligne.
102     printf("PID du fils : %d\n", cloneRetNum);
103     printf("PID du père : %d\n", getpid());
104 //    processStatus(getpid());
105 //    processStatus(cloneRetNum);
106     if (cloneRetNum == -1) {
107         perror("clone");
108         exit(1);
109     }
110
111     printf("\n");
112
113     int status;
114     if (wait(&status) == -1) {
115         perror("wait");
116         exit(1);
```

```
117     }
118
119     printf("Child exited with status %d (0 = success).
        ↪  \nbuffer = \"%s\"\n", status, buffer);
120     return 0;
121 }
```

8 Sources

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
https://www.baeldung.com/linux/
fork-vfork-exec-clone
https://man7.org/linux/man-pages/man2/clone.2.html