
Systeme et réseau : module n° 2

Séminaire n° 2

Processus

Samir EL KHATTABI *et* Christian VERCAUTER

Version éditée le 14 mars 2020

Table des matières

Introduction	3
I Introduction	4
1 Composants d'un système d'exploitation multi-tâche	4
2 Caractéristiques d'un processus	4
2.1 Identifiant	4
2.2 Espace d'adressage	5
2.3 Bloc de contrôle de processus	7
2.4 États d'un processus	7
3 Ordonnancement	9
3.1 Rôle	9
3.2 Commutation de contexte et élection	10
3.3 Politiques d'ordonnancement	10
II Création de processus	13
1 Notion d'appel système	13
1.1 Qu'est-ce-qu'un appel système ?	13
1.2 Exemple de traitement d'appel système	13
1.3 En pratique	16
2 Appels système <code>getpid()</code> et <code>getppid()</code>	17
3 Appels système <code>fork()</code> et <code>exit()</code>	18
3.1 Appel système <code>fork()</code>	18
3.2 Appel système <code>exit()</code>	19
3.3 Modèle de code	19
III Applications et compléments	21
1 Création d'un processus fils	21
2 Génération d'un processus zombie	23
3 Appel système <code>wait()</code>	24
4 Création de <code>n</code> processus	25
5 Changement de code d'un processus	28
5.1 Description et principe	28
5.2 Exemple d'utilisation	29
5.3 Exercice	30

Introduction

Ce document est un recueil d'exercices, réalisé à partir de polycopiés de cours et d'exercices créés par Étienne Craye, Samir El Khattabi et Christian Vercauter pour plusieurs enseignements effectués à EC-Lille et à IG2I Lens.

Il concerne le concept de processus comme composant de base d'un système d'exploitation multi-tâche.

Les exemples pris et les exercices proposés sont adaptés au système **LINUX** utilisé sur un ordinateur d'architecture **x86-64** bits.

Ils font appel à :

- des compétences **SHELL** acquises dans le module 1 de cet électif;
- un minimum de connaissances sur la programmation en langage **C**;
- des compétences sur les outils de développement associés.



Le compilateur **Gcc**



L'éditeur de textes **Emacs**



L'éditeur de textes **vim**

Dans la version avec réponses aux questions, de ce document, les solutions présentées ont été obtenues à partir de plusieurs ordinateurs dans différentes configurations :



Ubuntu 18.04 LTS en mode virtuel avec VirtualBox



Ubuntu 18.04 LTS en mode natif



Debian 8.10 en mode natif

Introduction

1. Composants d'un système d'exploitation multi-tâche

UNIX est un système multi-utilisateur et multi-tâche. L'espace de mémoire centrale — et l'espace disque en cas de *swap* — est partagé entre toutes les tâches internes du système et les tâches des utilisateurs.

L'utilisation du processeur, des processeurs le cas échéant, et de ses cœurs, est également partagée entre toutes les tâches en cours d'exécution, donnant l'illusion d'un véritable parallélisme d'exécution.

C'est la responsabilité d'un composant du noyau du système d'exploitation, appelé **ordonnanceur** (*scheduler*), d'attribuer et de reprendre le processeur à une tâche en cours.

La gestion des espaces mémoires (physique et virtuel) est confiée à l'unité **MMU** (*memory management unit*) qui contrôle les accès aux espaces de mémoire affectés aux tâches en cours d'exécution.

Un processus est un programme en cours d'exécution dans un espace de mémoire qui lui est propre et qui utilise le processeur et ses registres. C'est en quelque sorte, l'instance d'un programme : un programme peut être lancé plusieurs fois, mais chaque exécution donne naissance à un processus distinct.

Sous UNIX et dérivés, les processus sont organisés en une arborescence de processus dont la racine est le processus `init`.

Bien qu'ils aient des espaces de mémoire indépendants, les processus peuvent interagir par des mécanismes du noyau de communication inter-processus (IPC pour *Inter Processus Communication*).

Chaque processus peut être composé de plusieurs unités d'exécution, appelées *threads* qui partagent l'espace d'adressage du processus conteneur.

L'ordonnancement des tâches sous **LINUX** gère en réalité des threads, considérant qu'un processus est composé d'au moins un thread, le *thread principal*.

2. Caractéristiques d'un processus

2.1 Identifiant

Chaque processus est identifié par son numéro unique appelé **PID** (*Process IDentifier*).

La numérotation des processus est séquentielle ; la racine de l'arbre des processus a par conséquent le n° 1. Lorsque le dernier numéro est atteint, l'ordonnanceur affectera aux nouveaux processus le plus petit des PIDs des processus terminés.

Question 1.

- a) Consulter le fichier `/proc/sys/kernel/pid_max` afin de déterminer le plus grand numéro de processus.
- b) Consulter le fichier `/proc/sys/kernel/threads-max` afin de déterminer le plus grand numéro de thread.
- c) Quelle est la commande permettant d'afficher dans cet ordre ...
 - le PID ;
 - le nom d'utilisateur ;
 - le temps écoulé depuis le lancement de l'exécution ;
 - l'état ;
 - le nom de la commande ...
 du processus n° 1 ?
- d) Dans quel répertoire se situe-t-elle et que représente-t-il ?

Réponse à la question 1.

- a) La commande `cat /proc/sys/kernel/pid_max` produit le résultat 32768
- b) La commande `cat /proc/sys/kernel/threads-max` produit le résultat 31558
- c) La commande `ps -opid,user,etime,state,cmd -p1` produit le résultat :

PID	USER	ELAPSED	S	CMD
1	root	07:49:48	S	/sbin/init

- d) La commande correspondante se nomme `init` et se situe dans le répertoire `/sbin` qui est le dossier des commandes destinées à l'administrateur : ce dossier n'est généralement pas dans le PATH d'un utilisateur ordinaire, mais est présent dans celui de l'utilisateur `root`.

On y trouve par exemples, les commandes :

- `ifconfig` pour accéder à la configuration du réseau ;
- `fdisk` pour accéder à la table des partitions ;
- `lsmod` pour afficher la liste des modules intégrés dans le système ;
- `mksf` pour créer un système de fichiers.

2.2 Espace d'adressage

L'espace d'adressage d'un processus se compose de plusieurs régions ou segments comme le montre la figure ci-dessous :

- le segment **DATA** qui contient les variables globales ainsi que les variables statiques, initialisées explicitement dans le code source ;
- le segment **BSS** qui contient les variables globales ainsi que les variables statiques, non initialisées explicitement dans le code source.

Ces variables sont initialisées à zéro pour les variables de type caractère ou numérique, ou à **NULL** pour les pointeurs ;

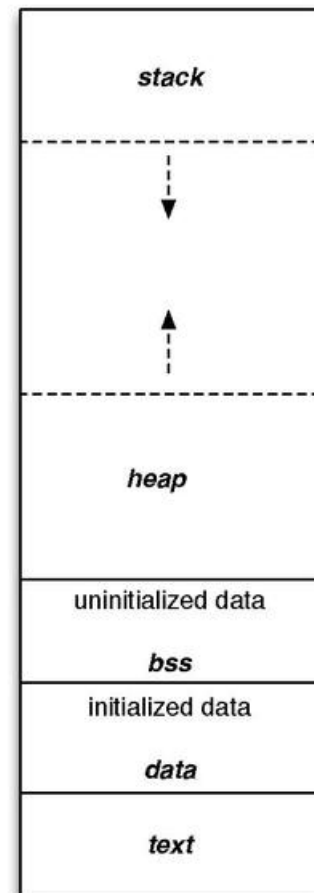
- le **tas** (*heap*), qui représente l'espace de mémoire dynamique, alloué et restitué lorsque des appels aux fonctions **malloc()**, **calloc()**, **realloc()** et **free()**, sont effectués ;
- la **pile** utilisateur (*stack*), gérée en FIFO, sur laquelle sont empilés :
 - les paramètres transmis à la fonction à exécuter
 - les valeurs de plusieurs registres du processeur : **rip**, registre des flags, **rbp**, **rsi**, **rdi** etc. pour des processeurs de la famille **x86-64** ;

C'est aussi sur la pile qu'est réservé l'espace mémoire utilisé par les variables de la fonction à exécuter.

La fin d'exécution réalise les opérations inverses :

- suppression de l'espace mémoire alloué aux variables locales ;
- restauration des registres sauvegardés sur la pile ;
- retour dans le programme appelant ;
- suppression des valeurs des paramètres

Le tas et la pile se partagent en fait une même zone mémoire, mais se développent en sens opposé ; lorsque le pointeur de pile rencontre le pointeur de tas, c'est qu'il n'y a plus d'espace mémoire, ni pour exécuter une fonction ni pour allouer dynamiquement de l'espace mémoire.



Question 2.

Après avoir généré les deux exécutables **testStat** et **testStat_ic** de la séance précédente, à l'aide de l'outil **make** en spécifiant correctement les cibles, exécuter la commande **size** sur chacun des deux exécutables.

Qu'obtient-on comme résultats ?

Réponse à la question 2. On exécute **make testStat testStat_ic** pour produire les deux fichiers exécutables.

Après avoir exécuté **size testStat; echo; size testStat_ic** on obtient le résultat ci-dessous :

text	data	bss	dec	hex	filename
------	------	-----	-----	-----	----------

4799	632	8	5439	153f	testStat
text	data	bss	dec	hex	filename
753778	7572	8824	770174	bc07e	testStat_ic

2.3 Bloc de contrôle de processus

Le noyau utilise et maintient une liste doublement chaînée de descripteurs de processus contenant entre autres :

- les identifiants du processus, du processus parent, de l'utilisateur ;
- son état ;
- sa priorité ;
- une sauvegarde de la valeur du compteur ordinal (adresse de la prochaine instruction du processus à exécuter, lorsqu'il disposera à nouveau du processeur ;
- une sauvegarde du pointeur de pile ;
- des informations relatives à l'espace d'adressage du processus ;
- une table de descripteurs de fichiers ;
- une liste de gestion de signaux,
- plusieurs compteurs, notamment ceux de temps d'exécution, ...

Sous LINUX, la structure d'un descripteur de processus, de type `struct task_struct` est définie dans `<linux/sched.h>`, représente environ 2 Kio¹.

2.4 États d'un processus

Sur un système d'exploitation multi-tâche, toutes les tâches lancées ne peuvent simultanément être exécutées ; le nombre de processeurs chargés de leur exécution étant généralement inférieur au nombre de ces tâches.

Une tâche, un processus peut être dans les états suivants :

Créé : c'est l'état initial d'un nouveau processus, qui attend que l'ordonnanceur le place dans l'état prêt : c'est un état transitoire ;

Prêt : (*ready* ou *runnable*). c'est l'état d'un processus qui dispose de toutes les ressources pour pouvoir s'exécuter ; il ne lui manque que le processeur !

Les processus prêts sont rangés dans une file d'attente gérée par l'ordonnanceur ;

Exécution : (*running*) état d'un processus auparavant dans l'état prêt à qui l'ordonnanceur en réalisant une *commutation de contexte*, vient d'attribuer le processeur ;

1. Le kibioctet équivaut à 2^{10} octets, cf. [Multiples normalisés](#)

En attente : état bloqué ou endormi (*blocked* ou *waiting*) C'est l'état d'un processus auparavant en exécution, qui a été préempté² ou qui attend un événement (fin d'une opération d'entrée/sortie, fin d'un délai d'attente, réception d'un signal, ...)

Terminé : le processus a terminé son exécution en fournissant un code de terminaison, ou a été forcé à s'arrêter.

Sous UNIX et dérivés, un processus prêt ou en attente peut être en mémoire ou être *swappé* sur le disque dur.

Le processus en exécution interagit avec le noyau du système en effectuant des appels-systèmes qui s'exécutent dans l'espace mémoire du noyau : les opérations qu'il réalise s'exécutent par conséquent en mode utilisateur ou en mode noyau.

Le code de terminaison d'un processus doit être acquitté par son processus père, afin que son descripteur disparaisse de la liste des descripteurs de processus en cours. Tant que cette opération n'est pas réalisée, le processus terminé est dans l'état *zombie* : l'espace d'adressage qu'il utilisait est libéré mais les 2 Kio environ de son descripteur sont toujours présents dans le système.

Question 3.

Utiliser la commande `top` afin d'observer la liste des processus avec leur état et autres informations.

- a) Combien y-a-t-il de processus lancés ?
- b) Combien y-a-t-il de processus en exécution ?
- c) Que désigne le terme KiB qui apparaît en haut de page ?
- d) Y-a-t-il des processus zombies ?
- e) Que désigne la colonne **S** ?
- f) Que désigne la colonne **RES** ?
- g) Que désigne la colonne **SHR** ?

Réponse à la question 3. Sur la capture d'écran ci-dessous :

2. périodiquement l'ordonnanceur compare la priorité du processus en exécution à celle du processus en attente le plus prioritaire


```

top - 12:57:18 up 21 min, 3 users, load average: 0.00, 0.07, 0.15
Tasks: 167 total, 1 running, 166 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.0 us, 0.8 sy, 0.0 ni, 98.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 4057412 total, 1374768 used, 2682644 free, 98440 buffers
KiB Swap: 6166524 total, 0 used, 6166524 free. 488128 cached Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
  573 root        20   0  252164  81872  52544 S   5.0   2.0   0:27.14 Xorg
 1538 christi+  20   0  304344  41220  34588 S   1.7   1.0   0:00.39 ksnapshot
  941 christi+  20   0  3135872  92208  67056 S   0.7   2.3   0:14.06 kwin
 1524 root        20   0      0      0      0 S   0.7   0.0   0:00.84 kworker/3:0
  951 christi+  20   0  3290832 160384  76612 S   0.3   4.0   0:23.07 plasma-desktop
    1 root        20   0   28684   5000   3100 S   0.0   0.1   0:00.67 systemd
    2 root        20   0      0      0      0 S   0.0   0.0   0:00.00 kthreadd
    3 root        20   0      0      0      0 S   0.0   0.0   0:09.55 ksoftirqd/0
    5 root        0 -20      0      0      0 S   0.0   0.0   0:00.00 kworker/0:0H
    7 root        20   0      0      0      0 S   0.0   0.0   0:06.08 rcu_sched
    8 root        20   0      0      0      0 S   0.0   0.0   0:00.00 rcu_bh
    9 root        rt    0      0      0      0 S   0.0   0.0   0:00.28 migration/0

```

a) La capture d'écran ci-dessus indique qu'il y a 167 processus ...

b) ... dont un seul en exécution (1 **running**)

c) L'espace de mémoire physique est de 4057 412 kibioctets.

KiB est le symbole du kibibyte et équivaut à 2^{10} bytes. C'est un standard de IEC (*International Electrotechnical Commission*) adopté comme norme internationale ;

d) La colonne **S** décrit l'état (*Status*) des différents processus et l'état **S** désigne l'état *en sommeil* d'un processus ;

e) La colonne **RES** indique la taille de mémoire résidente, c'est-à-dire la taille de la mémoire physique qu'une tâche utilise.

f) La colonne **SHR** donne la taille de mémoire disponible pour une tâche, pouvant être partagée avec d'autres processus.

3. Ordonnancement

3.1 Rôle

Le rôle de l'ordonnanceur du noyau, est de gérer l'ensemble des processus lancés, de façon à répartir au mieux l'usage du processeur pour leur exécution.

Par exemple, lorsque le processus en cours effectue une opération bloquante, comme une demande de lecture d'information saisie au clavier, ou issue d'un fichier, plutôt que d'attendre la fin de cette opération en monopolisant le processeur, l'ordonnanceur va préempter le processus en cours et réaliser une commutation de contexte pour l'attribuer à un processus prêt.

L'ordonnanceur est également invoqué à intervalles réguliers de l'ordre de 1 ms à quelques ms. Il est

alors chargé d'élire le prochain processus à exécuter et d'effectuer, si nécessaire, une commutation de contexte.

3.2 Commutation de contexte et élection

Le contexte d'un processus représente l'ensemble des informations qui lui permettront de reprendre son exécution après qu'il ait été interrompu. Cela inclut :

- son état
- une sauvegarde des principaux registres du processeur : mot d'état, compteur ordinal, son pointeur de pile, etc.
- les adresses des zones de code et de données
- son entrée dans la table des descripteurs de processus
- ...

Cet ensemble d'informations forme le bloc de contrôle de processus ou PCB (*process control block*)

Lors d'un changement de contexte, l'ordonnanceur ...

1. arrête le processus en cours d'exécution, met à jour son PCB et le sauvegarde ; ce processus passe dans l'état prêt ou en attente selon les circonstances ;
2. l'ancien PCB du processus élu est restauré, devenant ainsi le PCB de la tâche en cours : les registres du CPU reçoivent ainsi leurs valeurs sauvegardées, ce qui permet à ce processus de reprendre son exécution dans l'état précis là où il avait été interrompu

3.3 Politiques d'ordonnancement

Plusieurs stratégies d'ordonnancement existent pour l'élection du prochain processus à exécuter.

La stratégie du tourniquet (*round-robin*) est régulièrement utilisée dans les systèmes d'exploitation travaillant en temps partagés.

Un quantum de temps est attribué à chaque processus et l'ordonnanceur gère une file circulaire de processus éligibles. Le processus en tête de cette liste est le processus en cours d'exécution.

À la fin de chaque quantum, le processus en cours est interrompu, un décalage circulaire de la file est réalisé, le replaçant en queue de liste et permettant l'exécution du processus arrivé en tête de file.

Si cette liste est constituée de n tâches, alors la tâche interrompue, doit attendre au plus $n-1 \times \text{quantum}$ pour à nouveau passer dans l'état d'exécution ; un processus peut rendre le processeur avant la fin de son quantum de temps, soit de façon volontaire soit parce qu'il passe dans l'état en attente.

L'efficacité de cette stratégie est dépendante du quantum : la durée du changement de contexte doit être négligeable devant la valeur du quantum. Par exemple, si le quantum est fixé à 1 ms et qu'un changement de contexte prend 100 ns, alors l'*overhead* est de l'ordre de 10% : l'intervalle de temps entre l'exécution de deux processus est de 1,1 ms.

Un quantum trop long a un impact sur le temps de réponse des applications des utilisateurs.

UNIX et dérivés prennent en charge plusieurs politiques d'ordonnancement qui peuvent coexister au sein d'un même système :

1. des politiques d'ordonnancement pour processus *temps réel* ;
2. des politiques d'ordonnancement en temps partagés, pour des processus ordinaires ;

Une valeur de priorité statique est assignée à chaque processus, et l'ordonnanceur gère une file de processus éligibles pour chacune de ces valeurs.

Sous LINUX les priorités statiques vont de 0 (plus faible priorité) à 99.

La politique d'ordonnancement en temps partagé s'appliquent aux processus ordinaires des utilisateurs qui doivent avoir une priorité statique nulle. Elle utilise une priorité dynamique basée sur l'indice de courtoisie des processus, leur ancienneté en attente dans la file et sur leur aptitude à rendre rapidement le processeur, ce qui est l'une des caractéristiques des applications interactives.

Question 4.

En consultant le manuel du programmeur LINUX, à l'aide de la commande `man 7 sched`

- a) Quel est, depuis Linux 2.6.23, l'ordonnanceur utilisé sous LINUX ?
- b) qu'est-ce que l'indice `nice` et sous LINUX, dans quel intervalle prend-t-il ses valeurs ?
- c) quels sont les appels système permettant d'accéder à sa valeur ?
- d) quelle est la commande permettant de définir cet indice pour l'exécution d'un processus ?
- e) Est-il nécessaire d'être un utilisateur privilégié pour modifier cet indice ?

Réponse à la question 4.

- a) Depuis la version 2.6.23 du noyau LINUX, l'ordonnanceur conçu par Ingo Molnár (Red Hat, Inc.), est baptisé CFS (*Completely Fair Scheduler*).
- b) L'indice `nice` est un indice de courtoisie. Sous LINUX les valeurs de cet indice sont comprises entre -19 (moins courtois, donc plus *prioritaire*) et 20 (plus courtois, processus le moins favorisé) ; Par défaut, cet indice de courtoisie vaut 0.
- c) Les appels système permettant d'accéder à cet indice de courtoisie sont :
 - `int nice(int inc);`
 - `int getpriority(int which, int who);`
 - `int setpriority(int which, int who, int prio);`
- d) C'est la commande `nice` qui permet de définir l'indice de courtoisie d'un programme ou d'une commande à exécuter.

Exemple : `nice -n 10 sleep 15` pour exécuter la commande `sleep` avec un indice de courtoisie de 10, comme indiqué ci-dessous :

```
$ nice -n 10 sleep 15
^Z
[1]+  Stoppé                  nice -n 10 sleep 15
$ ps -l
F S  UID    PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   1274   1272  0  80   0  -  5890  -          pts/1        00:00:00 bash
0 T  1000   2263   1274  0  90  10  -  1453  -          pts/1        00:00:00 sleep
0 R  1000   2264   1274  0  80   0  -  2674  -          pts/1        00:00:00 pss
```

- e) Un utilisateur ordinaire peut augmenter l'indice de courtoisie des programmes et commandes qu'il lance, mais il ne peut le réduire.

Exemple : la commande `nice -n -10 sleep 15` produit le message suivant :

```
nice: impossible de définir le niveau de priorité: Permission non accordée
```

Création de processus

1. Notion d'appel système

1.1 Qu'est-ce-qu'un appel système ?

L'appel système est le moyen utilisé dans une application pour activer un service offert par le noyau du système.

Leur description est donnée dans le volume 2 des manuels UNIX.

On consultera notamment :

- l'introduction de ce volume, en exécutant la commande `man 2 intro`
- la présentation générale, en exécutant la commande `man 2 syscalls`

Le système UNIX comportait à l'origine environ 80 appels systèmes, qui assuraient divers services en matières de :

- gestion de fichiers, comme `open`, `read`, `write`, `unlink` `stat`, ... ;
- gestion de processus, comme `fork`, `wait`, `execve`, `getpid`, `kill`, ... ;
- gestion des communications inter-processus, comme `pipe`, `semget` `shmget`,

Il y a environ 380 appels système, permettant d'accéder aux services inclus dans un noyau récent du système `LINUX` ; certains d'entre-eux sont devenus obsolètes, mais restent présents pour assurer la comptabilité ascendante, d'autres offrent des services similaires.

D'un point de vue pratique, un appel système correspond à un appel d'une fonction C, qui après avoir rangé ses arguments dans certains registres du processeur, active une interruption logicielle qui fait basculer le processeur en mode superviseur, et qui, une fois le service réalisé, retourne la valeur -1 pour signaler son échec ou toute autre valeur, généralement 0, en cas de succès. En cas d'erreur, un code d'erreur est rangé dans une variable globale nommée `errno`

1.2 Exemple de traitement d'appel système

Soit le fichier `appelSys.c` suivant, contenant deux appels système :

- l'appel système `read()` qui lit des octets depuis un dispositif d'entrée (clavier, fichier, socket, etc.)

- l'appel système `write()` qui écrit des octets sur un dispositif de sortie (écran de la console, fichier, socket, etc.)

```
#include <unistd.h>

#define MAXLEN 80

int main (void) {
    int nblus, nbEcrits;
    char buff [MAXLEN+1];

    nblus = read(0, buff, MAXLEN);
    nbEcrits = write (1, buff, nblus);

    return 0;
}
```

La génération du code assembleur est obtenue en exécutant la commande `gcc appelSys.c -S`

```
1  .file "appelSys.c"
2  .text
3  .globl main
4  .type main, @function
5  main:
6  .LFB0:
7  .cfi_startproc
8
9  pushq %rbp
10 .cfi_def_cfa_offset 16
11 .cfi_offset 6, -16
12 movq %rsp, %rbp
13 .cfi_def_cfa_register 6
14 subq $96, %rsp
15 leaq -96(%rbp), %rax
16 movl $80, %edx
17 movq %rax, %rsi
18 movl $0, %edi
19 call read
20 movl %eax, -4(%rbp)
21 movl -4(%rbp), %eax
22 movslq %eax, %rdx
23 leaq -96(%rbp), %rax
24 movq %rax, %rsi
25 movl $1, %edi
26 call write
27 movl %eax, -8(%rbp)
28 movl $0, %eax
29 leave
30 .cfi_def_cfa 7, 8
31 ret
32 .cfi_endproc
33 .LFE0:
34 .size main, .-main
35 .ident "GCC: (Debian 4.9.2-10+deb8u2) 4.9.2"
36 .section .note.GNU-stack,"",@progbits
37
```

On y trouve :

- à la ligne 14, l'opération qui range l'adresse de la variable locale `buff` dans le registre `rax`
- à la ligne 15, le rangement de 80 (`MAXLEN`) dans le registre `rdx`
- le rangement dans le registre `rsi` du contenu de `rax`, c'est-à-dire l'adresse du buffer, à la ligne 16;

- à la ligne 17, le rangement de la valeur 0, qui désigne le dispositif standard d'entrée, dans le registre `rdi`
- l'appel de la fonction `read` à la ligne 18
- le rangement du résultat présent dans le registre `rax` dans la variable locale `nbLus`
- le rangement du nombre de caractères lus, présent dans `rax` dans le registre `rdx`, ...

Après avoir généré le code exécutable en mode statique par la commande

`gcc appelSys.c -static -o appelSys`, on génère dans un fichier texte, le code désassemblé du programme à l'aide la commande `objdump appelSys -D > appelSys.txt`

Une recherche dans le fichier `appelSys.txt` permet d'atteindre le début du code de la fonction principale.

```

1 0000000000400f8e <main>:
2   400f8e: 55                push    %rbp
3   400f8f: 48 89 e5         mov     %rsp,%rbp
4   400f92: 48 83 ec 60      sub     $0x60,%rsp
5   400f96: 48 8d 45 a0      lea     -0x60(%rbp),%rax
6   400f9a: ba 50 00 00 00   mov     $0x50,%edx
7   400f9f: 48 89 c6         mov     %rax,%rsi
8   400fa2: bf 00 00 00 00   mov     $0x0,%edi
9   400fa7: e8 e4 00 03 00   callq   431090 <__libc_read>
10  400fac: 89 45 fc         mov     %eax,-0x4(%rbp)
11  400faf: 8b 45 fc         mov     -0x4(%rbp),%eax
12  400fb2: 48 63 d0         movslq  %eax,%rdx
13  400fb5: 48 8d 45 a0      lea     -0x60(%rbp),%rax
14  400fb9: 48 89 c6         mov     %rax,%rsi
15  400fbc: bf 01 00 00 00   mov     $0x1,%edi
16  400fc1: e8 2a 01 03 00   callq   4310f0 <__libc_write>
17  400fc6: 89 45 f8         mov     %eax,-0x8(%rbp)
18  400fc9: b8 00 00 00 00   mov     $0x0,%eax
19  400fce: c9              leaveq  %eax
20  400fcf: c3              retq

```

La recherche du code de la fonction `__libc_read` nous conduit à cette portion de code :

```

1 0000000000431090 <__libc_read>:
2   431090: 83 3d 05 6c 28 00 00  cml     $0x0,0x286c05(%rip)
3   431097: 75 14             jne     4310ad <__read_nocancel+0x14>
4
5 0000000000431099 <__read_nocancel>:
6   431099: b8 00 00 00 00     mov     $0x0,%eax
7   43109e: 0f 05             syscall
8   4310a0: 48 3d 01 f0 ff ff   cmp     $0xffffffffffff001,%rax
9   4310a6: 0f 83 84 30 00 00   jae     434130 <__syscall_error>
10  4310ac: c3              retq
11

```

De même pour le code de la fonction `__libc_write` qui réalise les opérations ci-dessous :

```

1 00000000004310f0 <__libc_write>:
2   4310f0: 83 3d a5 6b 28 00 00  cml     $0x0,0x286ba5(%rip)
3   4310f7: 75 14             jne     43110d <__write_nocancel+0x14>
4
5 00000000004310f9 <__write_nocancel>:
6   4310f9: b8 01 00 00 00     mov     $0x1,%eax
7   4310fe: 0f 05             syscall
8   431100: 48 3d 01 f0 ff ff   cmp     $0xffffffffffff001,%rax
9   431106: 0f 83 24 30 00 00   jae     434130 <__syscall_error>
10  43110c: c3              retq
11

```

Ces deux fonctions font appel à l'instruction `syscall` dont le paramètre préalablement rangé dans le registre `rax` définit le service à réaliser :

- 0 pour une demande de lecture
- 1 pour une demande d'écriture

Cette instruction `syscall` remplace l'instruction d'interruption logicielle `int 0x80` utilisée sur les architectures 32 bits.

En cas d'erreur, l'une et l'autre se terminent en rangeant le code d'erreur dans une variable (probablement `errno` à ligne 5 ci-dessous) puis en réinitialisant le registre `rax` à -1 sur la ligne 7.

```

1 0000000000434120 <__syscall_error>:
2   434120: 48 f7 d8          neg     %rax
3
4 0000000000434123 <__syscall_error_1>:
5   434123: 64 89 04 25 d0 ff ff mov     %eax,%fs:0xffffffffffffffd0
6   43412a: ff
7   43412b: 48 83 c8 ff      or      $0xffffffffffffff,%rax
8   43412f: c3              retq
9
10
```

1.3 En pratique ...

Pour utiliser un appel système, il faut inclure le fichier d'en-tête `unistd.h`

L'utilisation directe dans un programme de la variable `errno` nécessite l'inclusion du fichier d'en-tête `errno.h`. La fonction `perror()` utilise la valeur de cette variable, afin d'afficher le message d'erreur correspondant, complété par un message spécifique défini par le programmeur.

Il faut tester le résultat d'un appel système pour s'assurer que le service demandé a été accompli correctement. Cela peut, par exemple, être réalisé à l'aide de la macro-fonction définie comme suit ...

```

#define CHECK(sts, msg)      \
    if (-1 == (sts)) {      \
        perror(msg);        \
        exit(EXIT_FAILURE); \
    }

```

... et utilisée comme dans cet exemple, adapté de celui présenté dans le paragraphe 1.2, page 13.

```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXLEN 80

#define CHECK(sts, msg)      \
    if (-1 == (sts)) {      \
        perror(msg);        \
        exit(EXIT_FAILURE); \
    }

int main (void)
{
    int nblus, nbEcrits;
    char buff [MAXLEN+1];

```



```

CHECK(nblus = read(0, buff, MAXLEN), "read()");
CHECK(nbEcrits = write(1, buff, nblus), "write()");

return EXIT_SUCCESS;
}

```

2. Appels système `getpid()` et `getppid()`

L'appel système `getpid()` renvoie l'identifiant du processus appelant.

L'appel système `getppid()` renvoie l'identifiant du processus père de l'appelant.

Leurs prototypes sont définis comme suit, dans le fichier `unistd.h` qu'il faut inclure en tête de programme : `#include <unistd.h>`

```
pid_t getpid(void);
```

```
pid_t getppid(void);
```

Elles retournent un résultat d'un type `pid_t` dont la définition est présente dans `sys/types.h` qu'il faut également inclure en tête de programme : `#include <sys/types.h>`

L'examen du contenu du fichier `/usr/include/x86_64-linux-gnu/sys/types.h` utilisé sur un ordinateur d'architecture `x86_64` sous `LINUX`, réalisé à l'aide de la commande :

```
cat /usr/include/x86_64-linux-gnu/sys/types.h | grep ' pid_t'
```

indique que `pid_t` est un alias de `__pid_t` :

```
typedef __pid_t pid_t;
```

Le type `__pid_t` est lui même défini dans le fichier

`/usr/include/x86_64-linux-gnu/bits/types.h` comme étant un alias pour le type `int` :

```
typedef int __pid_t;
```

Question 5.

- a) Écrire et tester le programme permettant d'afficher l'identité du processus en cours ainsi que celle de son processus père, comme dans l'exemple ci-dessous.

```

Je suis le processus ..... n°6275
Mon père est le processus ..... n°5717

```

- b) Que désigne le processus père, lorsque le programme est lancé en ligne de commande ?
- c) Que note-t-on lorsqu'on exécute plusieurs fois ce programme ?

Réponse à la question 5.

```

a) _____
#include <stdio.h>

#include <sys/types.h>

```

```
#include <unistd.h>

int main(void)
{
    printf("Je suis le processus ..... n°%d\n", getpid());
    printf("Mon père est le processus ..... n°%d\n", getppid());
    return 0;
}
```

- b) Le processus père est l'interpréteur de commandes (le *shell*). Ainsi, pour l'exemple précédent, la commande `ps -o pid,cmd 5717` produit le résultat suivant :

```
PID CMD
5717 bash
```

- c) À chaque exécution, le numéro du processus en cours est différent et la numérotation est séquentielle.

3. Appels système `fork()` et `exit()`

3.1 Appel système `fork()`

L'appel système `fork()` permet de créer un nouveau processus, appelé processus fils, qui est une copie conforme de son processus père. Il a cependant son propre identifiant et d'un descripteur spécifique dans la table des processus.

Schématiquement, l'espace d'adressage (*text*, *data*, *tas* et *pile*) du processus père est dupliqué; le processus fils a alors accès :

- à une copie des variables globales utilisées par son père (segment *data* et *bss*;
- à une copie des variables locales de la fonction qui a exécuté le `fork()` (segment *pile*);
- aux descripteurs de fichiers, de sockets qu'avait ouverts le processus père;
- aux variables d'environnement de son père

Il y a également duplication du bloc de contrôle du père; seules quelques informations du bloc de contrôle du processus fils sont mises à jour, comme :

- son pid et celui de son père;
- la consommation en temps CPU ...

Lorsque l'appel système `fork()` se termine deux processus en attendent le résultat :

1. le processus père qui reçoit le pid de son fils en cas de succès ou -1 si l'opération échoue;
2. le processus fils qui reçoit la valeur 0;

Lorsque l'opération échoue, un code d'erreur est rangé dans la variable globale `errno`. Les valeurs les plus fréquentes sont définies par les constantes symboliques :

ENOMEM il n'y a plus suffisamment d'espace mémoire pour créer le processus fils ;

EAGAIN le nombre maximal de processus en cours de l'utilisateur est atteint.

Le fichier à inclure et le prototype de cet appel système sont les suivants :

```
#include <unistd.h>

pid_t fork(void);
```

3.2 Appel système `exit()`

L'appel système `exit()` termine un processus en retournant un status destiné à son processus père.

Par convention la valeur 0 retournée comme status de terminaison, indique que le processus s'est terminé normalement ; une autre valeur peut être utilisée pour signaler un mauvais usage du programme, un cas anormal prévu et traité dans le programme, ...

Son fichier d'en-tête à inclure et son prototype sont les suivants :

```
#include <stdlib.h>

void exit(int status);
```

Sa présence dans un programme n'est pas toujours nécessaire ; ainsi dans la fonction principale d'un processus père, il suffit d'exécuter l'instruction `return 0` ; pour terminer le processus père en fournissant la valeur 0 comme code de terminaison.

Il faut cependant s'assurer qu'un processus fils se termine effectivement pas l'exécution d'une opération `exit()`

3.3 Modèle de code

L'usage de la macro `CHECK()` présentée précédemment n'est pas des mieux adaptés, pour le contrôle de l'appel système `fork()`, on lui préférera la structure ci-dessous :

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

#include <unistd.h>
#include <sys/types.h>
#include <errno.h>

int main(void)
{
    pid_t pidFils;

    pidFils = fork();    /* Demande de création d'un processus */
    switch (pidFils) {
        case -1 :
            perror("Échec de la création d'un processus fils");
            exit (pidFils);

        case 0 :
            /* On met ici le code spécifique du processus fils */
            exit (0); /* Fin normale du fils */

        default :
```

```
        /* On met ici la suite du code du processus père */  
        break;  
    }  
    return 0; /* Fin normale du processus père */  
}
```

Applications et compléments

1. Création d'un processus fils

Question 6.

En utilisant le modèle présenté dans le paragraphe 3.3, page 19, écrire le programme permettant à un processus père de créer un processus fils.

Le processus père affiche son identité et celle de son père, puis affiche l'identité du processus fils qu'il a créé; ce dernier affiche également son identité et celle de son père, de façon à produire comparable à ceci :

```
Je suis le processus ..... n°1655
Mon père est le processus ..... n°1620
Je suis le père du processus ..... n°1656
Je suis le processus ..... n°1656
Mon père est le processus ..... n°1655
```

Réponse à la question 6.

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

#include <unistd.h>
#include <sys/types.h>

void displayWhoami (void);

int main(void)
{
    pid_t pidFils;

    displayWhoami();
    pidFils = fork(); /* Demande de création d'un processus */
    switch (pidFils) {
        case -1 :
            perror("Échec de la création d'un processus fils");
            exit (pidFils);

        case 0 :
            /* On met ici le code spécifique du processus fils */
            displayWhoami();
            exit (0); /* Fin normale du fils */

        default :
            /* On met ici la suite du code du processus père */
            printf("Je suis le père du processus ..... n°%d\n", pidFils);
            break;
    }
    return 0; /* Fin normale du processus père */
}
```

```

void displayWhoami (void)
{
    printf("Je suis le processus ..... n°d\n", getpid());
    printf("Mon père est le processus ..... n°d\n", getppid());
}

```

Question 7.

Combien de processus seront créés lors de l'exécution du programme `fork2` dont le code source est le suivant :

```

void displayWhoami (void);

int main(void)
{
    pid_t pidFils1, pidFils2, pidFils3;

    pidFils1 = fork();    /* Demande de création d'un processus */
    pidFils2 = fork();    /* Demande de création d'un processus */
    pidFils3 = fork();    /* Demande de création d'un processus */

    displayWhoami();
    return 0; /* Fin normale du processus père */
}

void displayWhoami (void)
{
    printf("Je suis le processus ..... n°d\n", getpid());
    printf("Mon père est le processus ..... n°d\n", getppid());
}

```

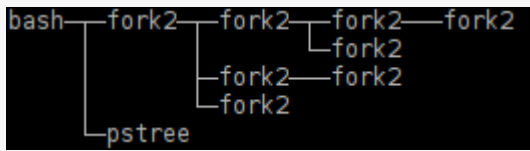
Réponse à la question 7. L'exécution du programme ci-dessus crée 8 processus. Voici un exemple de résultat produit par son exécution :

```

Je suis le processus ..... n°1899
Je suis le processus ..... n°1901
Mon père est le processus ..... n°1899
Mon père est le processus ..... n°1620
Je suis le processus ..... n°1900
Je suis le processus ..... n°1904
Mon père est le processus ..... n°1901
Mon père est le processus ..... n°1899
Je suis le processus ..... n°1905
Mon père est le processus ..... n°1900
Je suis le processus ..... n°1902
Mon père est le processus ..... n°1899
Je suis le processus ..... n°1903
Mon père est le processus ..... n°1900
Je suis le processus ..... n°1906
Mon père est le processus ..... n°1903

```

Si on insère l'instruction `pause()` juste avant l'instruction `return`, alors on est capable d'exécuter ce programme en arrière-plan, et d'exécuter la commande `ps tree` pour afficher l'arborescence des processus en cours. Pour l'exemple d'exécution ci-dessus, la commande `ps tree 1620` produit le résultat suivant :



2. Génération d'un processus zombie

Reprendre le code la fonction de la question 6 et ajouter l'appel de fonction `sleep(10)` juste avant l'instruction `return 0;` de la fonction principale.

Question 8.

- Dans quel volume, trouve-t-on la documentation de la fonction `sleep()` ?
Est-ce un appel système ?
- Que réalise l'instruction `sleep(10)` ; et dans quel état passe le processus qui l'exécute ?

Réponse à la *question 8*.

- a) La documentation de cette fonction se trouve dans le volume 3, celui des fonctions de bibliothèques. Ce n'est donc pas un appel système.
- Les informations sur l'appel système `sleep()` s'obtiennent à l'aide de la commande `man 3 sleep`
- b) L'appel `sleep(10);` place le processus qui l'exécute dans l'état endormi, pour une durée de 10 secondes.

Question 9.

Lancer le programme en arrière-plan et exécuter la commande `ps -l`

Que constate-t-on ? Expliquer cet état.

Réponse à la question 9. On lance le programme `fork3` en arrière-plan à l'aide de la commande `./fork3` ce qui produit le message

Je suis le processus n°2923
 Mon père est le processus n°1620
 Je suis le père du processus n°2924
 Je suis le processus n°2924
 Mon père est le processus n°2923

puis en exécute `ps -l` pour obtenir ce résultat :

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1000	1620	1618	0	80	0	-	5893	-	pts/1	00:00:00	bash
0	S	1000	2923	1620	0	80	0	-	1021	-	pts/1	00:00:00	fork3
1	Z	1000	2924	2923	0	80	0	-	0	-	pts/1	00:00:00	fork3 <defunct>
0	R	1000	2925	1620	0	80	0	-	2674	-	pts/1	00:00:00	ps

On note que le processus fis (n° 2924) est passé dans l'état zombie (état = Z) et il restera dans cet état tant que son processus père n'aura pas été acquitté son code de terminaison.

Si son processus père se termine sans acquitter, c'est la tâche `init` (PID = 1) qui effectuera aussitôt cet acquittement.

3. Appel système `wait()`

L'espace d'adressage du processus fils devenu zombie est libéré (voir valeur dans la colonne **SZ** lors de l'exécution de la commande `ps -l`), mais son descripteur occupe toujours quelques Kio dans la table des processus.

Le plus pénalisant concernant la présence de processus zombie, est qu'ils encombrant inutilement la table des processus, pouvant même conduire à la saturation de cette table car le nombre de processus d'un utilisateur est limité. Lorsque cette limite est atteinte, toute nouvelle demande de création de processus échoue. C'est une situation qui peut apparaître dans des services applicatifs mal conçus, destinés à fonctionner 24h/24 pendant de longues périodes, où pour chaque client ou application cliente qui s'y connecte, plusieurs processus sont créés et ensuite arrêtés lorsque le client se déconnecte.

Une bonne pratique consiste à faire en sorte le processus père attende la terminaison des fils qu'il a créés, afin d'acquitter leurs codes de fin d'exécution.

Ceci est réalisé à l'aide de l'appel système `wait()`, dont le synopsis est le suivant :

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

L'appel système `wait()` est suspensif et lorsqu'il se termine, il retourne le pid du fils qui s'est terminé et range à l'adresse transmise en paramètre, le code terminaison du fils, comme dans l'exemple ci-dessous, qui fait appel à la macro `CHECK()` présentée précédemment :

```
int main(void)
{
    pid_t pid;
    int status;
    /* ... */
    CHECK(pid = wait(&status), "wait()");
    printf("Ce code de terminaison du processus n°%d = %d\n",
           pid, status);
    return 0; /* Fin normale du processus père */
}
```

Lorsqu'il n'est pas utile de récupérer le code de terminaison cet appel système peut être effectué avec la valeur `NULL` en paramètre.

L'exemple présenté ci-dessus, peut être amélioré pour identifier correctement les différents causes possibles de terminaison d'un processus :

- fin consécutive à l'exécution d'un appel système `exit()`
- fin sur réception d'un signal non intercepté, ...

Un ensemble de macro-fonctions est mis à disposition dans le fichier d'en-tête `<sys/wait.h>`, qui lui-même inclut le fichier `<bits/waitstatus.h>`.

Dans le cas d'un ordinateur d'architecture x86-64, équipé d'une distribution LINUX adaptée, ces fichiers sont :

- `/usr/include/x86_64-linux-gnu/sys/wait.h`
- `/usr/include/x86_64-linux-gnu/bits/waitstatus.h`

Parmi les macros les plus utiles, on trouve :

- `WIFEXITED (status)` qui renvoie vrai si le statut provient d'un processus fils terminé par un `exit()` ;
- `WEXITSTATUS (status)` qui renvoie le code de terminaison du processus fils passé à `exit()` si `WIFEXITED (status)` est vrai
- `WIFSIGNALED (status)` qui renvoie vrai si le statut provient d'un processus fils terminé sur réception d'un signal non intercepté ;
- `WTERMSIG (status)` qui retourne le numéro du signal ayant provoqué la terminaison du processus fils, lorsque `WIFSIGNALED (status)` renvoie vrai.

4. Création de *n* processus

Question 10.

- a) Écrire et tester le programme qui crée cinq processus fils. Chaque processus fils, après avoir attendu pendant une durée tirée aléatoirement dans l'intervalle $[0, 5]$, affiche son identité, celle de son père, puis attend une seconde avant de se terminer en retournant son numéro d'ordre de création (1 pour le premier fils créé, etc.)

Le processus père ne se termine qu'après avoir affiché le code de terminaison de chaque fils créé.

La génération d'une valeur entière tirée au sort entre deux bornes sera réalisée comme suit :

```
int alea (int borneInf, int borneSup){
    srand(getpid());
    return rand() % (borneSup - borneInf) + borneInf;
}
```

Dans cette une version très imparfaite mais suffisante pour son usage dans cet exercice, la première opération permet de réinitialiser le générateur de nombres pseudo-aléatoires en utilisant comme *germe*, le numéro du processus en cours

Indication. Consulter l'aide en ligne de l'appel système `wait()` et notamment le paragraphe concernant les macros `Wxxx()`

- b) Modifier le programme précédent de façon à ce que le nombre de processus créés soit fourni sur la ligne de commande. Par défaut ce nombre vaut 5.

Réponse à la question 10.

```

a)
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

#define BORNEINF    0
#define BORNESUP    5
#define NBFILS      5

#define CHECK(sts, msg)          \
    if (-1 == (sts) )           \
    {                           \
        perror(msg);            \
        exit(EXIT_FAILURE);    \
    }

void displayWhoami (void);
void processusFils(int i);
int alea (int borneInf, int borneSup);

int main(void)
{
    pid_t pidFils, pid;
    int i, status;

    displayWhoami();
    for (i = 0; i < NBFILS; i++) {
        pidFils = fork(); /* Demande de création d'un processus */
        switch (pidFils) {
            case -1 :
                perror("Échec de la création d'un processus fils");
                exit (pidFils);

            case 0 :
                processusFils(i);
                exit (0); /* n'est en principe, plus utile */

            default :
                /* On met ici la suite du code du processus père */
                printf("Je suis le père du processus ..... n°%d\n", pidFils);
                break;
        }
    }

    /* Attente de terminaison des fils */
    for (i = 0; i < NBFILS; i++) {
        CHECK(pid = wait(&status), "wait()");
        printf("Code de terminaison du processus .. n°%d = %d\n",
               pid, WEXITSTATUS(status));
    }
    return 0; /* Fin normale du processus père */
}

```

Le processus fils est implémenté par la fonction ci-dessous :

```

void processusFils(int i)
{
    sleep(alea(BORNEINF, BORNESUP));
    displayWhoami();
    sleep(1);
    exit(i + 1);
}

```

b) Voici un exemple de résultat obtenu :

```

Je suis le processus ..... n°14186
Mon père est le processus ..... n°12023
Je suis le père du processus ..... n°14187
Je suis le père du processus ..... n°14188
Je suis le père du processus ..... n°14189
Je suis le père du processus ..... n°14190
Je suis le processus ..... n°14189
Je suis le père du processus ..... n°14191
Mon père est le processus ..... n°14186
Je suis le processus ..... n°14188
Mon père est le processus ..... n°14186
Code de terminaison du processus .. n°14189 = 3
Je suis le processus ..... n°14190
Mon père est le processus ..... n°14186
Code de terminaison du processus .. n°14188 = 2
Code de terminaison du processus .. n°14190 = 4
Je suis le processus ..... n°14187
Mon père est le processus ..... n°14186
Je suis le processus ..... n°14191
Mon père est le processus ..... n°14186
Code de terminaison du processus .. n°14187 = 1
Code de terminaison du processus .. n°14191 = 5

```

c) Les modifications à apporter au programme précédent sont mineures :

- 1) on déclare et initialise la variable `int nbFils = 0;`
- 2) on contrôle la présence d'argument de la ligne de commande et on convertit le premier argument en entier
- 3) s'il n'y avait pas d'argument ou si la conversion en entier échoue, alors on utilise la valeur par défaut.

Cela se traduit par les opérations ci-dessous :

```

if (argc > 1)
    nbFils = atoi(argv[1]);

if (nbFils == 0)
    nbFils = NBFILS;

```

d) On remplace dans la suite du code, tous les `for (i = 0; i < NBFILS; i++)`
par des `for (i = 0; i < nbFils; i++)`

5. Changement de code d'un processus

5.1 Description et principe

L'appel système `execve()` dont le fichier d'en-tête à inclure et le prototype ci-dessous, permet d'exécuter un programme.

```
#include <unistd.h>
```

```
int execve (char *filename, char *argv[], char *envp[]);
```

permet d'exécuter un programme.

Il n'y a pas création d'un nouveau processus mais remplacement de l'image mémoire du processus en cours par celle du programme à exécuter, autrement dit, les segments `text`, `data`, `bss` et la pile du processus appelant sont remplacés par ceux du programme chargé.

Le programme à exécuter, désigné par le premier argument de cet appel système, doit être un fichier exécutable ou un script commençant par une ligne `#! <interpréteur> [<arg. optionnel>]`

Le paramètre `argv` désigne un tableau de chaînes d'arguments transmises au nouveau programme : c'est la liste d'arguments du nouveau processus, où `argv[0]` désigne le nom du programme à exécuter.

Le paramètre `envp` est un tableau de chaînes de la définition de variables d'environnement de la forme `nom=valeur`, transmises au nouveau programme. Ces deux tableaux de pointeurs se terminent par la valeur `NULL`

En cas d'échec, cet appel système retourne la valeur -1 et range dans `errno` un code d'identification de l'erreur.

Il existe d'autres fonctions, dérivées de l'appel système `execve()` qui s'en différencient par la manière dont les paramètres leur sont transmis.

```
#include <unistd.h>

int execl (const char *filename, const char *arg, ...) ;
int execlp (const char *cmdname, const char *arg, ...) ;
int execl_e (const char *filename, char *arg, ..., char* const envp[]) ;
int execv (const char *filename, char *const argv[]) ;
int execvp (const char *cmdname, char *const argv[]) ;
```

Elles permettent de :

- spécifier les arguments du nouveau processus sous forme d'une liste d'arguments (paramètres `arg`) et non d'un tableau de chaînes de caractères (paramètre `argv`) ;
- rechercher le programme à exécuter parmi les chemins de recherches (variable d'environnement `$PATH`, paramètre `cmdname`) au lieu de spécifier complètement son chemin (paramètre `filename`)
- ...

Par exemple, la fonction `execv()` est la version simplifiée de `execve()`, n'ayant que deux paramètres :

- le chemin vers la commande ou le script à exécuter

- l'adresse d'une table de chaînes de caractères représentant les arguments fournis à la commande ou au script à exécuter. Cette table se termine par `NULL`.

On ne fournit pas l'adresse d'une table contenant les définitions de variables d'environnement.

5.2 Exemple d'utilisation

Soit le programme suivant qui affiche l'état des processus de l'utilisateur avant et après que l'appel à `execve()` ne soit réalisé :

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

#define CHECK(sts, msg) \
    if (-1 == (sts) ) \
    { \
        perror(msg); \
        exit(EXIT_FAILURE); \
    }

int main(int argc, char *argv[], char *envp[]) {
    char *args[] = {"ps", "-l", NULL};
    printf("Je suis le processus %d, fils du processus %d\n",
        getpid(), getppid());

    printf("\nAvant lancement de execve()\n");
    system("ps -l");

    printf("\nAprès lancement de execve()\n");
    CHECK(execve("/bin/ps", args, NULL), "execve()");
    printf("Fin de %s\n", argv[0]);

    return EXIT_SUCCESS;
}
```

L'exécution de ce programme nommé `execve1` produit le résultat ci-dessous :

Je suis le processus 16221, fils du processus 12023

Avant lancement de `execve()`

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1000	12023	12014	0	80	0	-	6492	wait	pts/1	00:00:00	bash
0	S	1000	16221	12023	0	80	0	-	1127	wait	pts/1	00:00:00	execve1
0	S	1000	16222	16221	0	80	0	-	1157	wait	pts/1	00:00:00	sh
4	R	1000	16223	16222	0	80	0	-	7558	-	pts/1	00:00:00	ps

Après lancement de `execve()`

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1000	12023	12014	0	80	0	-	6492	wait	pts/1	00:00:00	bash
4	R	1000	16221	12023	0	80	0	-	6486	-	pts/1	00:00:00	ps

On remarque alors que :

1. l'appel de la fonction `system()` a créé un interpréteur de commande (`pid=16222`) qui qui-même a créé un processus (`pid=16223`) permettant l'exécution de la commande `ps`

2. dans la seconde partie des résultats, le processus initial n'est pas associé à la commande `execve1`, mais à la commande `ps`, et il s'agit du même processus (`pid=16221`)

5.3 Exercice

Il s'agit d'écrire un programme qui crée un processus fils chargé d'exécuter le programme `testExecve` dont le code est présenté ci-dessous, en utilisant l'appel système `execve()` ou la fonction `execv()` :

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

int main (int argc, char *argv[], char *envp[])
{
    int i;
    int somme = 0;

    printf("\nJe suis le processus ..... n°%d\n", getpid());
    printf("Mon père est le processus ..... n°%d\n\n", getppid());

    for (i = 0; argv[i] != NULL; i++)
        somme += atoi(argv[i]);
    return somme;
}
```

On note que le programme ci-dessus, récupère une liste de valeurs présentes en argument de ligne de commande. Chaque valeur représentée par une chaîne de caractères est convertie en valeur numérique entière. Leur somme est effectuée et le résultat utilisé comme code de terminaison du programme.

Avant de lancer ce programme, le processus fils a créé un tableau de 10 nombres aléatoires compris entre 0 et 20, qu'il lui transmet en argument.

Voici un exemple de résultat généré par le programme à écrire :

```
Je suis le processus ..... n°2457
Mon père est le processus ..... n°1723
Je suis le père du processus ..... n°2458
Je suis le processus ..... n°2458
Mon père est le processus ..... n°2457
```

```
Je suis le processus ..... n°2458
Mon père est le processus ..... n°2457
```

```
Code de terminaison du processus .. n°2458 = 110
```

Question 11.

Écrire et tester le programme répondant aux spécifications présentées ci-dessus.

Indication. Le tableau d'arguments transmis au programme `testExecve` peut être déclaré et initialisé comme suit : `char *argv [11] = {NULL };`

Les 10 composantes utiles sont initialisées à `NULL` ainsi que la 11^e, marquant ainsi la fin de la table des arguments.

Indication. Utiliser la fonction `malloc()` pour allouer de l'espace mémoire pour une chaîne de caractères.

Indication. Utiliser la fonction `sprintf()` pour convertir une valeur numérique entière en une chaîne de caractères.

Réponse à la question 11.

```

/* ----- */
#include <stdio.h>
#include <stdlib.h>

/* ----- */
#include <unistd.h>
#include <errno.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

/* ----- */
#define CHECK(sts, msg) \
    if (-1 == (sts) ) \
    { \
        perror(msg); \
        exit(EXIT_FAILURE); \
    }

/* ----- */
void displayWhoami (void);
void processusFils(void);

/* ----- */
int main(void)
{
    pid_t pidFils, pid;
    int status;

    srand(time(NULL));
    displayWhoami();

    pidFils = fork(); /* Demande de création d'un processus */
    switch (pidFils) {
        case -1 :
            perror("Échec de la création d'un processus fils");
            exit (pidFils);

        case 0 :
            /* On met ici le code spécifique du processus fils */
            processusFils();
            exit (0); /* n'est en principe, plus utile */

        default :
            /* On met ici la suite du code du processus père */
            printf("Je suis le père du processus ..... n°%d\n", pidFils);
            break;
    }

    CHECK(pid = wait(&status), "wait()");
    printf("Code de terminaison du processus .. n°%d = %d\n",
        pid, WEXITSTATUS(status));
    return 0; /* Fin normale du processus père */
}

/* ----- */
void displayWhoami (void)
{
    printf("Je suis le processus ..... n°%d\n", getpid());
    printf("Mon père est le processus ..... n°%d\n", getppid());
}

/* ----- */
void processusFils(void)
{
    char *argv[11] = {NULL};
    int i;

```

```
for (i = 0; i < 10; i++) {  
    argv[i] = (char *) malloc (3 * sizeof(char));  
    sprintf(argv[i], "%02d", rand() % 21);  
}  
displayWhoami();  
  
CHECK(execve("testExecve", argv, NULL), "execve()");  
}  
/* ----- */
```


Bibliographie

- [1] W. Richard STEVENS, Stephen A. RAGO *Advanced Programming in the UNIX Environment*, Addison-Wesley, Third Edition, 2013.
- [2] Robert LOVE *Linux System Programming* O'Reilly, 2007.
- [3] Christophe BLAESS *Programmation système en C sous Linux* Eyrolles, 2^e édition, 2005.
- [4] Patrick CEGIELSKI *Conception de systèmes d'exploitation : Le cas Linux* Eyrolles, 2^e édition, 2004.
- [5] Vincent LOZANO *UNIX : Pour aller plus loin avec la ligne de commande* In Libro Veritas, Framabook, 2010 ([Téléchargeable ici](#))
- [6] Mendel COOPER, Linux Documentation Project *Advanced Bash-Scripting Guide*
 - [Version anglaise en ligne](#) ;
 - [Traduction en français en ligne](#).
- [7] Robert MECKLENBURG *Managing Projects with GNU Make* O'Reilly, 3rd edition, 2004 ([en ligne sur le projet Open Books](#))
- [8] Richard M. STALLMAN and the GCC Developer Community *Using the GNU Compiler Collection* GNU Press
 - [Version anglaise à lire en ligne](#)
 - [Version anglaise au format pdf](#)