Gestion des processus Unix

Auteur: Zouheir HAMROUNI

Unix : concepts fondamentaux

Unix met en oeuvre plusieurs concepts fondamentaux :

- Les processus : pour la gestion des traitements (exécutions de programmes). Ce concept encapsule l'exécution d'un programme de façon à assurer l'allocation et le contrôle des ressources nécessaires à cette exécution (processeur(s), mémoires, périphériques, fichiers, pipes, sockets…). Plusieurs processus peuvent exister en parallèle et se partager de façon optimisée ces ressources.

- Les fichiers : pour la gestion des données, ce concept permet d'assurer une rémanence et un contrôle des données des usagers du système.

- La gestion dynamique des processus et fichiers dans une structure arborescente. En effet, tout processus est un descendant d'un processus père, et appartient à une arborescence qui a pour racine le processus initial (numéro 1) ; et tout fichier est placé dans une arborescence de répertoires ayant une racine unique.

- La protection des ressources : Tout processus s'exécute pour un usager appartenant à un groupe. Tout fichier possède de la même manière un créateur identifié. L'identification d'un usager définit ainsi une capacité d'accès de cet usager aux ressources du système via les processus qu'il engendrera.

- La transparence de l'architecture matérielle : ce principe consiste à masquer les particularités des architectures supports. Le noyau système définit par exemple un système de gestion d'exceptions (déroutements et interruptions), et des signaux indépendants de tout processeur matériel.

1 Etape 1 : Les Processus

Un processus est une instance d'exécution d'un programme (un programme en cours d'exécution). Ce concept encapsule l'exécution d'un programme de manière à assurer l'allocation et le contrôle des ressources nécessaires à cette exécution (temps processeur, mémoire, périphériques, fichiers, pipes, sockets…). Plusieurs processus peuvent exister en parallèle et se partager de façon optimisée ces ressources.

Les processus sont organisés dans une structure arborescente : tout processus est un descendant d'un processus père, et appartient à une arborescence qui a pour racine le processus initial.

Chaque processus Unix possède un numéro "pid" (process id), et un processus père (son créateur, et son supérieur dans l'arborescence des processus). Le processus initial (racine de l'arborescence) porte le numéro 1.

Nous allons nous intéresser aux primitives de base de gestion des processus. Par défaut, et sauf précision contraire, ces primitives sont définies dans "unistd.h", fichier à référencer en début du code.

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

1.1 Identification : getpid() et getppid()

- pid_t getpid() : retourne le pid du processus appelant (pid_t : synonyme d'entier).
- pid_t getppid() : retourne le pid du père du processus appelant.

Chaque processus peut donc connaître son numéro (pid) et le numéro de son père (ppid).

1.2 La primitive sleep()

unsigned int sleep(int n) : endort le processus appelant pour une durée de n secondes. Cette durée peut être écourtée dès qu'un signal non ignoré est reçu.

sleep() renvoie zéro si le temps prévu s'est écoulé, ou le nombre de secondes restantes si l'appel a été interrompu.

Nous l'utiliserons pour faire durer les processus testés pendant un temps choisi.

1.3 La primitive exit()

void exit(int n) : met fin au processus appelant avec n comme code de retour.

Par convention, le code de retour est :

- est égal à 0 (EXIT_SUCCESS : constante prédéfinie) après un comportement correct
- est différent de 0 pour indiquer une erreur

1.4 Création d'un processus : fork()

Un processus peut créer un fils en appelant la primitive fork() :

pid_t fork() : crée dynamiquement un nouveau processus (fils) qui s'exécute de façon concurrente avec le processus qui le crée (processus père). Ce processus fils hérite du processus père les attributs suivants :

- même code
- une copie de la zone de données du père
- environnement d'exécution
- priorité
- descripteurs de fichiers ouverts
- traitement des signaux

Après exécution du fork(), on se trouve avec deux processus (un père et un fils) ayant le même code et la même image de données. En cas de succès, la primitive fork() renvoie deux valeurs différentes :

- dans le code du père : la valeur de retour indique le pid du fils créé (>0)
- dans le code du fils : la valeur de retour est égale à 0

En cas d'échec (nombre maximum de processus atteint) : aucun fils n'est créé et la valeur de retour du fork() est égale à -1

Même si le père et son fils ont le même code, on peut différencier les traitements effectués par l'un et par l'autre en se basant sur la valeur de retour du fork.

Le code suivant illustre ce concept : le processus principal crée 3 fils, en répétant le même traitement à l'intérieur d'une boucle "for" dans laquelle il passe 3 fois.

Au premier passage (fils=1), le processus principal exécute la primitive fork(), dont on doit traiter les 3 cas de retour :

- retour = -1 : le `fork()` a échoué et aucun fils n'a été créé. Dans ce cas, le processus principal s'arrête en faisant appel à la primitive `exit()` et en y indiquant une valeur d'arrêt différente de 0 (par convention). Il est important de prévoir le traitement du retour d'erreur de toute primitive système.

- retour >= 0 : le `fork()` a créé un processus fils qui hérite du même code que son père. La valeur de retour n'est pas la même dans les deux codes :

  - retour = 0 : on se trouve dans le code du fils, qui va exécuter la séquence à l'intérieur du "if (retour == 0)" :
    - affiche son numéro, son pid et le pid de son père
    - s'endort pendant 3 secondes
    - et s'arrête en faisant appel à la primitive `exit()`. Par conséquent, le fils, même s'il hérite de la totalité du code de son père, ne va jamais entrer de nouveau dans la boucle "for". Nous verrons plus loin ce qu'il se passe si on enlève l'appel à `exit()`.

  - retour > 0 : on se trouve dans le code du père, qui affiche le numéro et le pid du fils qu'il vient de créer, et repart dans la boucle "for".

Le même traitement se répète lors du deuxième et du troisième passage dans la boucle "for".

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Création de fils : fork et exit */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour ;
    int duree_sommeil = 2 ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
                    fils, getpid(), getppid()) ;
            sleep(duree_sommeil) ;
            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }
    sleep(duree_sommeil) ;
    return EXIT_SUCCESS ;
}
```

Pour valider cet exercice, il faut :

- Compiler le code ci-dessus : gcc -Wall -o nom_executable pere_fils.c
- lancer l'outil de validation "apisys" dans un terminal shell
- se placer sur le bon exercice (commandes p et s) et lancer son exécution
- répondre aux questions posées

Visualiser l'interprétation des résultats

1.5 Importance du exit()

Dans l'exemple dessus, le code du fils est délimité entre le "if (retour == 0)" et exit(EXIT_SUCCESS). Ce qui permet de séparer clairement le code exécuté par le père de celui du fils.

Si on reprend le même code que dessus et on supprime le exit(EXIT_SUCCESS), on abtient un comprtement très différent des fils.

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Absence du exit dans le fils, et conséquences */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour ;
    int duree_sommeil = 3 ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
                    fils, getpid(), getppid()) ;
            /* Le fils ne s'arrete pas ici */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }
    sleep(duree_sommeil) ;
    return EXIT_SUCCESS ;
}
```

Testez ce code dans l'outil "apisys" (comme indiqué plus haut), et répondez aux questions posées.

Visualiser l'interprétation des résultats

1.6 Héritage des données

Comme indiqué dessus, le processus fils hérite d'une copie de la zone de données de son père. Il possède donc une copie de toutes les variables du code. Pour une variable donnée, la valeur vue par le fils est la même que celle vue par le père tant que cette variable n'est pas modifiée. Car dès que la avriable est modifiée par le père ou par le fils, la copie du fils devient forcément différente de celle du père.

Le code suivant reprend le premier code présenté plus haut en ajoutant une variable appelée "patrimoine_fils", que le père initialise à 10000, comme cadeau de naissance qu'il réserve à chaque nouveau fils.

Chaque fils, fait fructifier son patrimoine, modifie donc la variable patrimoine_fils, et l'affiche avant de s'arrêter.

De son côté, le père essaie de calculer le patrimoine total de ses fils en lisant la variable patrimoine_fils, mais malheureusement pour lui, il n'a accès qu'à sa propore copie, et pas à celles de ses fils.

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Héritage et dupplication des données */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */

#define NB_FILS 3     /* nombre de fils */
#define DELAI 3

int main()
{
    int fils, retour ;
    int cagnotte, patrimoine_fils ;
    int duree_sommeil = 3 ;
    cagnotte = 30000 ;
    patrimoine_fils = cagnotte / NB_FILS ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
    printf("Je dispose de %d Euros, que je partage entre mes futurs fils\n", cagnotte) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            printf("\n    Processus fils numero %d : mon pere m'a offert %d Euros\n",
                    fils, patrimoine_fils) ;
            patrimoine_fils = patrimoine_fils * (fils + 1) ;
            sleep(duree_sommeil) ;
            printf("\n    Processus fils numero %d - j'ai augmente mon patrimoine a %d Euros\n",
                    fils, patrimoine_fils) ;
            exit(EXIT_SUCCESS) ;   /* Te:rminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }
    sleep(duree_sommeil+1) ;

    printf("\nProcessus Principal - le patrimoine total de mes fils est de %d\n", patrimoine_fils*NB_FIL
    return EXIT_SUCCESS ;
}
```

Testez ce code dans l'outil "apisys" (comme indiqué plus haut), et répondez aux questions posées.

Visualiser l'interprétation des résultats

1.7 Héritage des descripteurs

Chaque processus échange des informations avec son environnement (écran, clavier, fichiers, etc) sous la forme de suites d'octets (ou flots d'entrées/sorties). L'accès à chaque canal de communication (lecture/écriture) se fait par l'intermédiaire d'un descripteur (numéro >=0).

Par défaut, Unix/Linux gère trois flots d'entrée/sortie :

- l'entrée standard (flot numéro 0), `stdin`, associée au clavier,
- la sortie standard (flot numéro 1), `stdout`, associée à l'écran,
- la sortie erreur standard (flot numéro 2), `stderr`, associée aussi à l'écran.

Il est possible de modifier ces associations pour un processus : par exemple, associer la sortie standad à un fichier, de manière à ce que les messages ne s'affichent plus à l'écran, mais s'écrivent dans ce fichier.

Cette modification ne nécessite pas la modification du code, et se fait simplement avec un mécanisme appelé redirection :

- la sortie standard peut être redéfinie à l'aide du caractère `>` : dans la commande suivante, le résultat du `ls` est redirigé dans le fichier *fic*
  ```
  ls > fic
  ```
- il est possible d'ajouter à un fichier le résultat d'une commande : dans la commande suivante, le résultat du `ls -l` s'ajoute au fichier *fic*
  ```
  ls -l >> fic
  ```
- en Bourne shell, sh ou bash, la sortie erreur standard peut être redirigée par `2>` : la commande suivante écrit les messages d'erreurs éventuellement engendrés par l'exécution de `rm` dans le fichier *fic*
  ```
  rm toto 2> ficerreur
  ```
- l'entrée standard peut être redirigée par `<` : les commandes suivantes réalisent une lecture d'un caractère au clavier et son affichage, en passant par la variable c
  ```
  read c; echo $c
  ```
  Dans cette seconde version, la lecture est faite dans le fichier *fic*
  ```
  read c < fic ; echo $c
  ```

Un processus fils hérite de tous les descripteurs ouverts dans son père (entrées/sorties standard, fichiers, etc.).

- Nous avons vu lors des exécutions précédentes que les messages du père et ceux des différents fils sont affichés dans le même terminal (sortie standard).
- De même, tout descripteur de fichier ouvert par le père est hérité par ses fils, y compris lorsque ce fichier est ouvert par redirection.

Pour illustrer cela, on reprend le premier programme pere_fils.c et on l'exécute en redirigeant la sortie standard dans un fichier : (pere_fils > fic_sortie).

Après exécution, on peut remarquer :

- aucun message n'est affiché dans le terminal d'exécution
- les messages qui devaient s'afficher dans le terminal d'exécution se trouvent maintenant dans le fichier "fic_sortie"
- mais, certains messages sont en double ou en triple

L'explication de ce dernier point, se trouve dans le fonctionnement du "printf". D'après le standard ISO, le comprtement du "printf" n'est pas le même pour la sortie standard et pour un fichier ordinaire :

- dans le cas d'une sortie interactive (terminal), le flot est géré par ligne et \n provoque la vidange du tampon langage
- dans le cas d'une sortie dans un fichier ordinaire, le flot est géré par bloc (pour limiter les échanges avec le fichier) : '\n' est traité comme un caractère ordinaire et ne commande plus la vidange du buffer.

Chaque fils hérite du tampon (zone mémoire) de sortie du père, qui contient les messages déjà envoyés par le père et non vidangés, auxquels va s'ajouter le message du fils. Par conséquent, chaque fils va envoyer un flot de données contenant, en plus de son propre message, les messages antérieurs du père. C'est pour cela que certains messages se strouvent en 3 exemplaires (réaffichés par les fils 2 et 3), et d'autres en double exemplaire (réaffichés par le fils 3).

- Le fis nuéro 1 récupère un tampon qui contient le premeir message du père
- Le fis nuéro 2 récupère un tampon qui contient
  - le premeir message du père
  - le message affiché par le père lors du premier passage dans la boucle
- Le fis nuéro 3 récupère un tampon qui contient
  - le premeir message du père
  - le message affiché par le père lors du premier passage dans la boucle
  - le message affiché par le père lors du deuxième passage dans la boucle

Pour forcer la vidange du tampon de sortie, on fait un appel explicite à `fflush(stdout)` après chaque printf du père. Ainsi, le fils créé hérite d'un tampon vide et ne dupplique plus l'affichage des messages de son père.

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Redirection et fflush */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour ;
    int duree_sommeil = 2 ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    /* Vidange du tampon de sortie pour que le fils le récupère vide        */
    /* D'après le standard ISO le comportement du printf présente 2 cas :   */
    /* - sortie interactive (terminal) : flot géré par ligne et \n provoque */
    /*   la vidange du tampon langage
    /* - sortie dans un fichier : flot géré par bloc et \n est traité comme */
    /*   un caractère ordinaire. fflush(stdout) force la vidange du tampon. */

    fflush(stdout) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
```

```
31              printf("Erreur fork\n") ;
32              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
33              exit(1) ;
34          }
35
36          /* fils */
37          if (retour == 0) {
38              printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
39                      fils, getpid(), getppid()) ;
40              sleep(duree_sommeil) ;
41              /* Important : terminer un processus par exit */
42              exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
43          }
44
45          /* pere */
46          else {
47              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
48                      getpid(), fils, retour) ;
49              /* vidange du tampon de sortie pour que le fils le récupère vide */
50              fflush(stdout) ;
51          }
52      }
53      sleep(duree_sommeil) ;
54      return EXIT_SUCCESS ;
55  }
```

Testez ce code dans l'outil "apisys" et vérifiez que les messages ne sont plus duppliqués.

1.8 Fils orphelin

Lorsque le processus père se termine (disparait) avant son fils, ce dernier devient orphelin, et est rattaché (adopté) au processus initial portant le numéro 1.

On reprend le programme initial pere_fils.c, et on fait durer les fils plus longtemps que le père, en agissant sur le paramètre du sleep.

```
1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2  /* Fils orphelins */
3
4  #include <stdio.h>    /* entrées sorties */
5  #include <unistd.h>   /* pimitives de base : fork, ...*/
6  #include <stdlib.h>   /* exit */
7
8  #define NB_FILS 3     /* nombre de fils */
9
10 int main()
11 {
12     int fils, retour ;
13     int duree_sommeil = 120 ;
14
15     printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
16
17     for (fils = 1 ; fils <= NB_FILS ; fils++) {
18         retour = fork() ;
19
20         /* Bonne pratique : tester systématiquement le retour des appels système */
21         if (retour < 0) {   /* échec du fork */
22             printf("Erreur fork\n") ;
23             /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
24             exit(1) ;
25         }
26
27         /* fils */
28         if (retour == 0) {
29             printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
30                     fils, getpid(), getppid()) ;
31             sleep(duree_sommeil) ;
32             exit(EXIT_SUCCESS) ;       /* Terminaison normale */
33         }
```

```
34
35          /* pere */
36          else {
37              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
38                      getpid(), fils, retour) ;
39          }
40      }
41      sleep(1) ;
42      return EXIT_SUCCESS ;
43  }
```

Si on exécute ce nouveau code, et on tape la commande "ps l" dans un autre terminal, on peut voir dans la colonne "PPID" du tableau affiché, que les 3 fils ont pour père le processus 1.

1.9 Fils zombie

Lorsque un processus fils se termine sans que son père prenne connaissance de cette terminaison, ce fils reste présent et entre dans un état zombie.

On reprend le programme initial pere_fils.c, et on fait durer le père plus longtemps que les fils, en agissant sur le paramètre du sleep.

```
 1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
 2  /* Fils Zombie */
 3
 4  #include <stdio.h>    /* entrées sorties */
 5  #include <unistd.h>   /* pimitives de base : fork, ...*/
 6  #include <stdlib.h>   /* exit */
 7
 8  #define NB_FILS 3     /* nombre de fils */
 9
10  int main()
11  {
12      int fils, retour ;
13      int duree_sommeil = 120 ;
14
15      printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
16
17      for (fils = 1 ; fils <= NB_FILS ; fils++) {
18          retour = fork() ;
19
20          /* Bonne pratique : tester systématiquement le retour des appels système */
21          if (retour < 0) {   /* échec du fork */
22              printf("Erreur fork\n") ;
23              exit(1) ;
24          }
25
26          /* fils */
27          if (retour == 0) {
28              printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
29                      fils, getpid(), getppid()) ;
30              exit(EXIT_SUCCESS) ;   /* Terminaison normale */
31          }
32
33          /* pere */
34          else {
35              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
36                      getpid(), fils, retour) ;
37          }
38      }
39      sleep(duree_sommeil) ;
40      return EXIT_SUCCESS ;
41  }
```

Si on exécute ce nouveau code, et on tape la commande "ps l" dans un autre terminal, on peut voir dans la colonne STAT du tableau affiché, que les 3 fils sont dans l'état zombie (Z, ou Z+) et dans la dernière colonne le terme "defunct" ajouté après leur nom.

1.10 wait()

Souvent, le processus père a besoin de prendre connaissance de la terminaison des fils qu'il a créés, et d'en connaître le code ou la cause de fin. La primitive `wait()` (définie dans sys/wait.h) permet de réaliser cette opération.

`pid_t wait(int *status)` permet d'attendre (de façon bloquante) la terminaison d'un fils du processus appelant (terminé avec exit ou tué par un signal). Elle :

- renvoie le pid du fils qui s'est arrêté ou a été arrêté
- indique dans le paramètre status, passé par référence, le code de terminaison (exit) ou le numéro du signal ayant tué le fils. L'accès à ces informations est facilité par des macros :
  - WIFEXITED(status) est vrai si le fils s'est terminé avec exit
  - WEXITSTATUS(status) renvoie la valeur du exit
  - WIFSIGNALED(status) est vrai si le fils a été tué par un signal
  - WTERMSIG(status) renvoie le numéro du signal ayant tué le fils

Le code suivant reprend le code initial en y modifiant les points suivants :

- les fils 1 et 3 terminent immédiatement sans attente
- le fils 2 s'en dort pendant une durée assez longue
- les 3 fils renvoient leur numéro (respectivement 1, 2, 3) comme valeur de fin
- le processus principal (père) attend la fin de ses 3 fils, et affiche pour chacun d'eux :
  - la cause de teminaison : avec exit, ou tué par un signal :
  - la valuer du exit ou le numéro du signal

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* wait : le père attend la fin de ses fils */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */

#define NB_FILS 3     /* nombre de fils */

int main(
{
    int fils, retour, wstatus, fils_termine ;
    int duree_sommeil = 300;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
    /* Vidange du tampon de sortie pour que le fils le récupère vide        */
    fflush(stdout) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */

            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
                    fils, getpid(), getppid()) ;
            /* Le fils 2 s'endort pendant une durée asse longue */
            if (fils == 2) {
                sleep(duree_sommeil) ;
            }
            exit(fils) ;  /* normalement exit(0), mais on veut illustrer WEXITSTATUS */
        }
```

```
40
41          /* pere */
42          else {
43              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
44                      getpid(), fils, retour) ;
45          }
46      }
47
48      sleep(3) ;   /* pour les besoins de l'outil de validation automatique */
49
50      /* attendre la fin des fils */
51      for (fils = 1 ; fils <= NB_FILS ; fils++) {
52          /* attendre la fin d'un fils */
53          fils_termine = wait(&wstatus) ;
54
55          if WIFEXITED(wstatus) {   /* fils terminé avec exit */

56              printf("\nMon fils de pid %d a termine avec exit %d\n",
57                      fils_termine, WEXITSTATUS(wstatus)) ;
58          }
59          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
60              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
61                      fils_termine, WTERMSIG(wstatus)) ;
62          }
63      }
64      printf("\nProcessus Principal termine\n") ;
65      return EXIT_SUCCESS ;
66  }
```

Testez ce code dans l'outils "apisys" et répondez aux questions posées.

1.11 execl()

Les primitives `exec` constituent une famille de primitives permettant le lancement de l'exécution d'un nouveau programme par le processus appelant. Cela n'entraîne pas la création d'un nouveau processus mais seulement le chargement et le lancement d'un nouveau programme exécutable, en lieu et place du programme effectuant l'appel à la primitive `exec`.

On parle de recouvrement : le nouveau programme recouvre (remplace complètement) le processus appelant.

`int execl(char *ref, char *arg0, char *arg1,..., char *argn, NULL)`

`int execlp(char *ref, char *arg0, char *arg1,..., char *argn, NULL)`

permettent le lancement de l'exécution du fichier exécutable dont :

- le nom est indiqué par le premier paramètre "ref"
- la liste des arguments fournie dans "arg0", "arg1",…,"argn"

execlp effectue une recherche de l'exécutable dans tous les répertoires de PATH.

`ref` doit référencer un fichier qui existe et qui est exécutable. Le paramètre "arg0" est obligatoire et pointe en général sur une chaîne identique à celle pointée par "ref".

Le comportement de la primitive `execl` est le suivant :

- si l'exécutable référencé par `ref` existe, il sera chargé et lancé en en lieu et place du processus courant, sans aucun retour dans ce dernier.
- sinon, on continue à exécuter le code du processus appelant, et dans ce cas il faut traiter l'erreur.

On reprend l'exempe initial, et on :

- remplace dans les fils, l'appel à "sleep()" par l'exécution du programme "dormir" qui fait la même chose.
- introduit une petite erreur dans le nom de l'exécutable pour le fils 2
- ajoute dans le père une boucle d'attente de la terminaison des 3 fils, en affichant pour chacun le mode d'arrêt et la valeur correspondante

Ce qui donne le code ci-dessous.

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* execl */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, wstatus, fils_termine ;

    char ref_exec[]="./dormir" ; /* exécutable  */
    char arg0_exec[]="je dors" ;   /* argument0 du exec : nom donnée au processus */
    char arg1_exec[]="3" ;       /* argument0 du exec : durée de sommeil */

    printf("Je suis le processus principal de pid %d\n", getpid()) ;
    /* Vidange du tampon de sortie pour que le fils le récupère vide        */
    fflush(stdout) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {

         /* mettre un executable inexistant pour le fils 2 */
            if (fils == 2) {
                ref_exec[3] = 'a' ;
            }

            execl(ref_exec, arg0_exec, arg1_exec, NULL) ;

            /* on ne se retrouve ici que si exec échoue */
            printf("\n     Processus fils numero %d : ERREUR EXEC\n", fils) ;
            /* perror : affiche un message relatif à l'erreur du dernier appel systàme */
            perror("     exec ") ;
            exit(fils) ;   /* sortie avec le numéro di fils qui a échoué */
        }
        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
            fflush(stdout) ;
        }
    }
    sleep(3) ;   /* pour les besoins de l'outil de validation automatique */

    /* attendre la fin des fils */
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        /* attendre la fin d'un fils */
        fils_termine = wait(&wstatus) ;

        if WIFEXITED(wstatus) {   /* fils terminé avec exit */
            printf("\nMon fils de pid %d a termine avec exit %d\n",
```

```
65                      fils_termine, WEXITSTATUS(wstatus)) ;
66          }
67          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
68              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
69                      fils_termine, WTERMSIG(wstatus)) ;
70          }
71      }
72      printf("\nProcessus Principal termine\n") ;
73      return EXIT_SUCCESS ;
74  }
```

Testez ce code dans l'outil "apisys" (comme indiqué plus haut), et répondez aux questions posées.

Visualiser l'interprétation des résultats

1.12 execv()

int execv(char *ref, char *argv[])

permet le lancement de l'exécution du fichier exécutable dont le nom est "ref", avec comme arguments les chaînes pointées par argv[0],

argv[1],...,argv[n]. argv[n+1] doit être contenir "NULL".

Le code suivant reprend le code précédent en remplaçant l'appel à execl par execv, et en utilisant un tableau de pointeurs sur les arguments du exec.

Contrairement au code précédent, on uilise ici la même référence pour les paramètres "ref" et argv[0](ce qui est souvent le cas).

```
1  /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2  /* execv */
3
4  #include <stdio.h>    /* entrées sorties */
5  #include <unistd.h>   /* pimitives de base : fork, ...*/
6  #include <stdlib.h>   /* exit */
7  #include <sys/wait.h> /* wait */
8  #include <string.h>   /* opérations sur les chaines */
9
10 #define NB_FILS 3     /* nombre de fils */
11
12 int main()
13 {
14     int fils, retour, wstatus, fils_termine ;
15
16     char *argv[8] ;               /* tableau de pointeurs sur les arguments du exec */
17     char args_exec[8][16] ;         /* tableau des arguments du exec */
18     strcpy(args_exec[0], "./dormir") ;   /* arg0 */
19     argv[0] = args_exec[0] ;         /* ponteur sur arg0 */
20     strcpy(args_exec[1], "3") ;      /* arg1 : durée de sommeil */
21     argv[1] = args_exec[1] ;         /* ponteur sur arg1 */
22     argv[2] = NULL ;               /* dernier pointeur = NULL */
23
24     printf("Je suis le processus principal de pid %d\n", getpid()) ;
25     /* Vidange du tampon de sortie pour que le fils le récupère vide        */
26     fflush(stdout) ;
27
28     for (fils = 1 ; fils <= NB_FILS ; fils++) {
29         retour = fork() ;
30
31         /* Bonne pratique : tester systématiquement le retour des appels système */
32         if (retour < 0) {   /* échec du fork */
33             printf("Erreur fork\n") ;
34             /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
35             exit(1) ;
36         }
37
38         /* fils */
```

```
39          if (retour == 0) {
40
41              /* mettre un executable inexistant pour le fils 2 */
42              if (fils == 2) {
43                  args_exec[0][3] = 'a' ;
44              }
45
46              execv(argv[0], argv) ;   /* argv[0] utilisé comme nom de l'exécutable */
47
48              /* on ne se retrouve ici que si exec échoue */
49              printf("\n     Processus fils numero %d : ERREUR EXEC\n", fils) ;
50              /* perror : affiche un message relatif à l'erreur du dernier appel systàme */
51              perror("     exec ") ;
52              exit(fils) ;   /* sortie avec le numéro di fils qui a échoué */
53          }
54          /* pere */
55          else {
56              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
57                      getpid(), fils, retour) ;
58              fflush(stdout) ;
59          }
60      }
61      sleep(3) ;   /* pour les besoins de l'outil de validation automatique */
62
63      /* attendre la fin des fils */
64      for (fils = 1 ; fils <= NB_FILS ; fils++) {
65          /* attendre la fin d'un fils */
66          fils_termine = wait(&wstatus) ;
67
68          if WIFEXITED(wstatus) {   /* fils terminé avec exit */
69              printf("\nMon fils de pid %d a termine avec exit %d\n",
70                      fils_termine, WEXITSTATUS(wstatus)) ;
71          }
72          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
73              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
74                      fils_termine, WTERMSIG(wstatus)) ;
75          }
76      }
77      printf("\nProcessus Principal termine\n") ;
78      return EXIT_SUCCESS ;
79  }
```

Dans les deux versions précédentes, le fichier exécutable est trouvé grâce à son chemin complet (./dormir).

`execlp` et `execvp` sont deux variantes de respectivement execl et execv, mais avec recherche du fichier exécutable dans la liste des répertoires contenue dans la variable PATH.

1.13 Conservation des descripteurs

Les descripteurs ouverts avant l'appel à une primitive `exec` le restent, et sont donc accessibles par l'exécutable lancé avec `exec`.

On peut illustrer cela en exécutant l'avant dernier programme en redirigeant la sortie standard dans un fichier.

Mais l'appel à execl écrase le tampon de sortie ; ce qui induit un risque de perte d'information. Par exemple, si on introduit, dans le fils, l'affichage d'un message avant l'appel à execl, ce message risque de ne pas s'afficher s'il ne se termine pas avec '\n' ou si l'exécution se fait avec redirection.

Pour forcer le vidage de ce tampon avant l'appel à execl on utilise la fonction fflush() de la bibliothèque standard stdio.h.

Ce qui donne le code suivant :

```
1 /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2 /* execl et fflush */
```

```c

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */

#define NB_FILS 3     /* nombre de fils */

int main()
{
    int fils, retour, wstatus, fils_termine ;

    char ref_exec[]="./dormir" ; /* exécutable  */
    char arg0_exec[]="je dors" ;   /* argument0 du exec : nom donnée au processus */
    char arg1_exec[]="3" ;       /* argument0 du exec : durée de sommeil */

    printf("Je suis le processus principal de pid %d\n", getpid()) ;
    /* Vidange du tampon de sortie pour que le fils le récupère vide        */
    fflush(stdout) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
                    fils, getpid(), getppid()) ;

            /* Précaution en cas d'une exécution avec redirection :                 */
            /* Vidange du tampon de sortie avant qu'il ne soit écrasé par exec      */
            /* D'après le standard ISO le comportement du printf présente 2 cas :   */
            /* - sortie interactive (terminal) : flot géré par ligne et \n provoque */
            /*   la vidange du tampon langage */
            /* - sortie dans un fichier : flot géré par bloc et \n est traité comme */
            /*   un caractère ordinaire. fflush(stdout) force la vidange du tampon. */
            fflush(stdout) ;

         /* executable inexistant pour le fils 2 */
            if (fils == 2) {
                ref_exec[3] = 'a' ;
            }

            execl(ref_exec, arg0_exec, arg1_exec, NULL) ;

            /* on ne se retrouve ici que si exec échoue */
            printf("\n     Processus fils numero %d : ERREUR EXEC\n", fils) ;
            /* perror : affiche un message relatif à l'erreur du dernier appel systàme */
            perror("     exec ") ;
            exit(fils) ;   /* sortie avec le numéro di fils qui a échoué */
        }
        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
            fflush(stdout) ;
        }
    }
    sleep(3) ;   /* pour les besoins de l'outil de validation automatique */

    /* attendre la fin des fils */
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        /* attendre la fin d'un fils */
        fils_termine = wait(&wstatus) ;

        if WIFEXITED(wstatus) {   /* fils terminé avec exit */
            printf("\nMon fils de pid %d a termine avec exit %d\n",
                    fils_termine, WEXITSTATUS(wstatus)) ;
```

```
77          }
78          else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
79              printf("\nMon fils de pid %d a ete tue par le signal %d\n",
80                      fils_termine, WTERMSIG(wstatus)) ;
81          }
82      }
83      printf("\nProcessus Principal termine\n") ;
84      return EXIT_SUCCESS ;
85  }
```