TP 2 – Création de processus

Les exercices sont à faire dans l'ordre.

1 Utilisation de *fork* et *execl*

Voir l'énoncé du TP1 (il faut/fallait absolument avoir réalisé cet exercice avant de passer au suivant).

2 Attente d'un fils

Lors TP1, vous avez sans doute été confronté·e au (faux) problème du prompt qui ne revient pas, laissant penser que le programme n'était pas fini :

```
prompt$ ./parent 2 truc
Je suis le parent (25316)
prompt$ Je suis l'enfant (25317): 4 truc
|  <-- curseur clignotant ;-)
```

Mais en fait il était bien fini et en tapant n'importe quelle commande vous auriez eu une réponse. En effet le prompt était déjà revenu (indiquant la fin du parent) avant l'écriture du message du processus enfant ! C'est entre autres pour cela que nous allons maintenant faire attendre la fin du processus enfant par le processus parent (et aussi pour que le processus enfant ne reste pas longtemps en état *Zombie* après sa fin - voir le chapitre 1).

2.1 Avec *waitpid*

Écrire 2 programmes :

(1) **Le premier programme** est celui du parent : Il crée 2 enfants qui seront recouverts par le même programme (seuls les arguments transmis changeront).

(2) **Le programme recouvrant** fera dormir les enfants. Le temps d'attente sera passé en paramètre. La fonction *main* retournera *exceptionnellement* une valeur non nulle (le temps d'attente par exemple). Dans un premier temps le deuxième enfant sera endormi plus longtemps que le premier.

▶ Nous voulons que le parent attende son deuxième enfant : réalisez cette attente avec la primitive *waitpid* (regardez le chapitre 1 et l'encart ci-dessous). Le parent affichera alors la valeur de retour de cet enfant puis se terminera.

Aide C :

- ☞ La fonction *sleep*(n) permet d'endormir un processus pendant n secondes.
- ☞ Pour récupérer le status d'un processus fils après *waitpid*, vous utiliserez la macro WEXITSTATUS(*status*) où *status* est l'entier (`int`) passé par référence comme deuxième paramètre de la fonction *waitpid* (`waitpid(pid, &status, 0)`). *waitpid* remplit l'entier *status*. ATTENTION : *status* n'est pas directement la valeur de retour du processus. *status* contient des informations sur la terminaison du processus DONT la valeur de retour. On filtre l'information contenue dans *status* avec la macro WEXITSTATUS(status).

2.2 L'état *zombie*

- Modifiez le programme recouvrant pour qu'après *sleep*, il affiche le pid du processus et celui de son parent.
- Maintenant le deuxième enfant s'endormira moins longtemps que le premier. Observez la valeur des *pid* des processus et surtout de leur processus parent. Juste avant la fin du premier enfant (qui s'arrête donc en dernier) quel est son parent ?

Réponse : Le premier enfant finit après la fin du parent qui l'a créé. Il est normalement rattaché au processus 1 (*init*) mais ce n'est plus le cas pour les versions > Ubuntu 14.04 : Il est rattaché au processus leader de votre session.

3 Héritage

Cet exercice illustre l'impact de `fork` sur l'héritage (ou non) des données.

- Réalisez un programme qui crée un processus enfant (sans execl). Avant le `fork`, le parent créera un tableau de 4 entiers. Le tableau devra être initialisé avec des 1.
- Après le `fork`, le processus parent affichera (dans sa partie) le tableau puis remplira le tableau avec des 2 et attendra avec `wait`. De la même manière, le processus enfant affichera le tableau puis remplira le tableau avec des 4 et s'endormira 2 secondes.
- À la fin de chaque processus (après le `wait` pour le parent, et `sleep` pour l'enfant), affichez l'adresse du tableau (avec %p) ainsi que les valeurs du tableau.

Question : L'adresse affichée est identique pour les 2 processus, pourtant le contenu est différent. Comment expliquez-vous ce phénomène ?

Réponse : Il s'agit bien de 2 tableaux différents. Il s'agit de l'adresse dans l'espace (local) d'adressage de chaque processus (adressage virtuel) et non de l'adresse réelle en mémoire centrale. Cette notion sera vue dans le chapitre sur la mémoire.

4 Plus on est de fork, plus on...

Le but sera de vérifier la répartition uniforme de la fonction dite "aléatoire" du langage C. Nous allons répartir un travail sur plusieurs processus.

- Réalisez un programme qui crée un tableau de taille N. Dans un premier temps, définissez N avec la directive #define N 300. Remplissez le tableau aléatoirement avec des entiers entre 0 et 20, en utilisant la fonction *rand*. Voir l'encart ci-dessous pour la création du tableau et *rand*.

Aide C :

☞ Utilisez la fonction *malloc* pour créer le tableau :
`int * tab = (int*) malloc(N*sizeof(int));`
et n'oubliez de faire `free(tab)` à la fin de vos programmes (parent et enfant). Astuce : dès que vous mettez un `malloc`... tout de suite, écrivez le `free` et là *vous aurez tout compris !* ;-)

☞ Pour remplir le tableau avec des valeurs aléatoires : utilisez la fonction *rand*() (il faut initialiser avec la fonction *srand*()) :

```
#include <stdlib.h>
#include <time.h>
...
/* Initialisation du générateur au début du père :*/
srand(time(NULL)); /* ou srand(getpid()) si plusieurs
        processus doivent le faire en même temps*/
...
...
/* génération d'un nombre */
/* nombre entre 0 et RAND_MAX (exclus a priori
        contrairement à ce qui est écrit dans le man !) */
int i = rand();
/*nombre entre 0 et N (exclus)*/
int j = rand() % N;
/* nombre entre 0 et 1. (exclus)*/
float f = rand()/(float)RAND_MAX;
```

- Réalisez une fonction qui compte le nombre fois qu'apparaît une valeur dans un tableau. Cette fonction prend comme paramètres un tableau d'entiers, sa taille et la valeur à rechercher dans le tableau. Elle affichera le pourcentage d'apparitions (exemple : "2 : 10%")
- Dans la fonction `main`, après la création du tableau et son initialisation, créez un processus enfant pour faire la recherche sur une valeur (c'est pour tester). L'enfant ne doit pas recréer explicitement un nouveau tableau puisqu'il hérite de celui du père.

- Puis lorsque cela fonctionnera, faites en sorte que le parent crée autant de processus enfants que nécessaire pour rechercher les valeurs comprises entre 0 et 20 (chaque enfant s'occupant d'un nombre). ATTENTION : n'oubliez pas de faire finir la partie enfant par un `exit` sinon gare à la boucle + fork !!
- Pour faire attendre le parent (que ses enfants finissent), pourquoi ne pas mettre le `wait` dans la partie *default* du switch ni dans la boucle ?

  Parce que sinon les enfants ne fonctionneraient pas en parallèle, chaque fils serait créé et lancé séquentiellement après la fin de l'enfant précédent. Il faut mettre l'attente hors de la boucle et donc une nouvelle boucle pour faire `2l wait`.
- Faites tourner votre programme pour un tableau de 1 000 cases puis 10 000 cases et enfin 1 000 000 de cases.

  Questions/Réflexions :
  — La répartition est-elle bien uniforme ?
  — Au final, combien y a-t-il de tableau(x) en mémoire ? [n'auriez-vous pas oublié quelques `free` ?]
  — Comment expliquez-vous que les fils ne finissent pas dans l'ordre ?

  Réponses :
  - Répartition : normalement oui quasiment ;-)
  - Nombre de tableaux en mémoire : il y a héritage et les tableaux sont recopiés automatiquement dans l'espace enfants (il y a donc potentiellement autant de tableaux que d'enfant + celui du parent... donc il ne faut pas oublier les `free` des enfants.
  - Ordre : Les processus ne mettent pas forcément le même temps (ils se partagent le(s) processeur(s)). Cette notion sera vue plus en détails dans le chapitre sur l'ordonnancement du processeur.

5 Buffering de l'affichage

[*Exercice très court d'observation*]
Comme nous l'avons vu ou le verrons lors du chapitre 2, la sortie standard n'envoie pas directement les caractères à l'écran, car on privilégie l'exécution des commandes et ainsi on évite de perdre du temps à écrire des messages à l'écran. Ainsi `printf` écrit sur le buffer de la sortie standard (de taille 8192 octets par défaut) et une fois plein ou si `fflush(stdout)` est utilisé juste après le processus d'affichage à l'écran est déclenché. La sortie d'erreur standard elle n'est pas bufferisée (puisque normalement les erreurs sont plus rares et souvent définitives !).

Faites le test avec le programme suivant :

```
#include <stdio.h>

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

```
    char * tab = "Test";
    printf("Attention, il va y avoir une erreur de segmentation");
    tab[15] = 12;
    exit(0);
}
```

En lançant le programme : le message ne s'affiche pas ! On pourrait alors croire que l'erreur est avant non ? (et ce n'est pas le cas ! l'erreur est bien lors de l'écriture dans un case à l'extérieur de la taille du tableau).

Donc pour déboguer ou pour vous assurer l'affichage d'un message (et surtout en *Système* avec plusieurs processus), 2 solutions :

1. Ajoutez après le `printf` : `fflush(stdout)`
2. ou utilisez : `fprintf(stderr, ".....")`

Testez ces ceux solutions.

L'on peut trouver au détour de livre ou de forum une troisième solution qui consiste à simplement ajouter un `\n` à la fin du message mais cette solution ne fonctionne pas systématiquement.