Système Unix

TP minishell1

B-PSU-050 b-psu-050@epitech.eu

Abstract:
Ce TP a pour objectif de vous présenter le fonctionnement général des processus dans le système.
Alors que le cours aborde de façon assez complète les informations relatives à un processus, ce TP a pour vocation de vous préparer au mini-projet associé : le ***minishell-1***

Table des matières

.1 Introduction 2
.2 Etape 1 : fork 3
.3 Etape 2 : exec* 6
.4 Etape 3 : wait* 9

.1 Introduction

Le projet minishell-1 a pour objectif de vous permettre d'appréhender la problématique de gestion des processus au sein d'un système d'exploitation de type Unix.

Ce TP s'articulera autour de 3 grandes étapes.

- fork
- exec
- wait

Pour mettre en valeur ces 3 mécanismes essentiels, nous utiliserons des fonctions relatives aux **pid**, **pgid** et **sid**.

Les propriétés, les limites et les droits ne seront pas abordés dans ce TP.
Notez cependant que les fonctions relatives à ces propriétés fonctionnent toutes sur le même principe simple :

- une fonction pour obtenir l'information
- une fonction pour changer l'information

Passons donc sans plus attendre à l'étape 1 !

.2 Etape 1 : fork

C'est parti ! Nous allons en plusieurs étapes mettre en évidence l'utilisation et le comportement de `fork`.

L'appel système `fork` permet de créer un nouveau processus depuis le processus appelant. Ce nouveau processus possèdera alors un lien de parenté avec le processus appelant. On parle alors de "processus fils" et de "processus père".

1. Exercice 1

   (a) Lire le man de `fork` (2) ;

   (b) Ecrire un programme **"my_fork"** qui affiche son **pid**.

`man 2 getpid`
`Cette fonction renvoie un pid_t. Cherchez donc à quoi correspond ce type afin de pouvoir l'afficher.`

L'exécution de votre programme devrait ressembler à ceci :

```
1 (gdx@tls)./my_fork
2 42
3 (gdx@tls)
```

2. Exercice 2

   (a) Ajoutez à votre programme, après l'affichage de votre pid, un appel à la fonction `fork` ;

   (b) Affichez à nouveau le pid (pensez à rappeler `getpid`).

`(a) Combien de pid s'affiche(nt) à l'écran ?`

`(b) Comment expliquez vous cela ?`

L'affichage peut être ''mélangé''. Nous mettrons en évidence, plus tard dans ce tp, la raison de ce phénomène.

3. Exercice 3

   (a) Ajoutez en fin de programme une boucle infinie (du type `while(42)`) ;

   (b) Exécutez à nouveau votre programme ;

   (c) Dans un autre terminal, exécutez le programme **top** pour bien voir l'existence des 2 processus.

Vous remarquerez que les 2 processus font la même taille en mémoire. En effet, c'est le même programme qui s'exécute pour chacun d'eux. De plus, les 2 processus semblent continuer leur exécution à partir de l'instruction qui suit l'appel à fork (le nouveau processus ne s'exécute pas depuis le début, comme cela a été le cas pour le processus père.

4. Exercice 4

   (a) Relisez plus en détail le man de `fork` ;

   (b) Dans votre programme, déterminez si vous êtes :

      i. dans le processus père ;

      ii. dans le processus fils.

   (c) Dans le processus père, affichez "pid pere : " suivi du pid du processus père ;

   (d) Dans le processus fils, affichez "pid fils : " suivi du pid du fils.

   (e) Dans le processus fils, contrôlez que le **ppid** est correct.

`man 2 getppid`

5. Exercice 5

(a) Grące à `getsid` et à `getpgid`, vérifiez que les processus père et fils issus du `fork` appartiennent au même groupe et à la même session.

Voilà, nous en avons fini avec `fork`, passons maintenant à la suite ...

.3 Etape 2 : exec*

L'appel système `exec` (et toutes ses variantes) permet de remplacer le code exécutable du programme appelant par celui d'un autre programme.

Il n'y a **EN AUCUN CAS** création d'un nouveau processus (c'est le travail de fork). Le processus reste le même (même **pid**).

Cela signifie que le reste du programme initial n'a aucune chance de continuer à s'éxécuter après l'appel à `exec` (puisque son code exécutable est remplacé par le nouveau).

Mettons maintenant en évidence ce mécanisme...

1. Exercice 1

   (a) Ecrire un programme "my_exec1" contenant la ligne suivante

```
1 execlp(``ls'', ``ls'', 0);
```

   (b) Afficher un message avant et après cet appel.

Que se passe-t-il lors de l'exécution ?

La commande **ls** a été exécutée, exactement comme si vous l'aviez tapée sur le shell. Le message AVANT l'appel à exec s'est affiché, mais pas celui après.
Lorsque **ls** se termine, le processus s'est arrêté. Notre programme ``my_exec1'' ne ressurgit pas de nulle part.

2. Exercice 2
   Vérifions qu'il n'y ait bien pas de création de nouveau processus, mais bel et bien un remplacement du programme exécuté par un autre.

   (a) Ecrire un programme "my_exec2" qui affiche son **pid** ;

(b) Ajoutez en fin de programme une boucle infinie (du type `while(42)`).

Ce programme “my_exec2” va être exécuté par notre programme “my_exec1”.

(a) Remplacez l’exécution de **“ls”** par **“my_exec2”** ;

(b) Faites afficher à “my_exec1” sont **pid** et faites lui faire une pause (avec **sleep(10)**) avant l’appel à **exec** ;

(c) Surveiller avec la commande **top** qu’il n’y a à aucun moment 2 processus.

Il est probable que vous deviez exécuter ‘‘./my_exec2’’ pour que cela fonctionne.

Dans le cas où ‘‘my_exec1’’ ne peut exécuter ‘‘my_exec2’’, votre message situé après l’appel à exec est exécuté

L’exécution de tout cela devrait afficher le même pid

3. Exercice 3 : Les arguments
   Voyons ensuite comment passer des arguments à nos programmes exécutés.
   Reprenez pour cela la première version de notre programme “my_exec1” et modifiez la pour obtenir :

```
1 execlp(‘‘ls’’, ‘‘ls’’, ‘‘-la’’, 0);
```

Vous voyez, ce n’est pas compliqué. Les arguments de **execlp**, à compter du deuxième, sont les arguments passés au programme exécuté. L’ensemble de ces arguments vont se retrouver dans le paramètre **argv** de la fonction main du programme exécuté.

4. Exercice 4 : La gestion du `path`

(a) Changez l’appel à **execlp** par un appel à **execl** ;

(b) Testez votre programme ;

(c) Ca fonctionne moins bien ? Essayez alors d'éxécuter "/bin/ls" au lieu de "ls" ;

(d) Testez à nouveau votre programme.

Dans toute la collection des fonctions en **exec***, celles qui contiennent la lettre ''p'' symbolisent une gestion du path. Cela signifie que ces fonctions vont parcourir tous les répertoires inclus dans la variable d'environnement PATH pour y chercher le programme à exécuter. Pour les autres fonctions (sans le ''p''), il faudra alors spécifier le chemin complet vers le programme à exécuter.

Retenez pour le moment qu'il y a toute une série de fonctions (lisez attentivement le man) qui diffèrent sur le format des arguments et certaines propriétés. Elles utilisent en fait toute la même fonction principale d'exécution : **execve**

.

.4 Etape 3 : wait*

Revenons à nos histoires de **fork**, avec les processus père et fils, en abordant une nouvelle famille de fonctions, les fonctions **wait***.
wait, comme son nom l'indique, permet à un processus père d'attendre la fin du processus fils.
Plutôt que de longs discours, passons immédiatement à sa mise en pratique pour mieux comprendre son fonctionnement.

1. Exercice 1

   (a) Créez un programme "my_wait" qui fait un **fork** ;

   (b) Dans le processus père, faites un appel à la fonction **wait** ;

   (c) Dans le processus fils, patientez 5 ou 10 secondes, et terminez le processus avec l'aide de **exit** ;

   (d) Testez votre programme. Vérifiez bien que l'exécution du père est suspendue tant que le fils existe.

`N'hésitez pas à mettre de l'affichage un peu partout pour bien comprendre ce qu'il se passe.`

2. Exercice 2 : le statut

   (a) Changez la valeur du paramètre de **exit** dans le fils et retrouvez votre valeur dans le père.

`Le paramètre` **`status`** `de` **`wait`** `permet de récupérer le code de sortie du processus fils (le man parle d'une macro ... si vous cherchiez un peu de ce côté?!?!?!).`

3. Exercice 3
   La fonction **wait4** est la plus complète de toutes les fonctions **wait*** disponibles.

   (a) Effectuez la boucle suivante dans le père après le **fork**.

```
while (fils_pid != wait4(fils_pid, &status, WNOHANG, 0))
{
  sleep(1);
}
```

Comment récupérer le pid du fils dans le père ? Il semblerait que **fork** puisse répondre à cette question ...

Notez ici l'utilisation de **wait4** en mode non bloquant, grâce à l'option **WNOHANG**.
On est alors obligé de le rappeler de temps en temps pour savoir si le processus fils n'est pas mort.
Cela permet alors au processus père de faire autre chose, tout en surveillant l'état du processus fils. Il suffit de remplacer l'appel à **sleep** par ce que vous voulez faire réaliser au processus père.

Le cours sur les signaux apportera une facilité de traitement pour ce genre de situation. Pour les plus curieux, le père recoit un signal SIGCHLD lors d'un évènement lié à un processus fils, il sait alors quand faire appel à **wait**.

Merci d'avoir suivi ce TP jusqu'au bout et bon courage pour votre projet.