TD5 –Gestion de processus

1 Objectif

Ces exercices introduisent aux propriétés et à l'utilisation des primitives de gestion de processus de POSIX, en particulier `fork()`, `execXX()` et `wait()`.

2 Propriétés de `fork()`

Ecrire un programme `fork_prop.exe` qui mette en évidence les propriétés suivantes :

- un processus a le même propriétaire réel et effectif, ainsi que le même groupe réel et effectif que son père ;
- un processus hérite des descripteurs des fichiers ouverts par son père, et partage les pointeurs d'E/S avec ce dernier ;
- un processus fils hérite des répertoires ouverts de son père ;
- un processus orphelin est adopté par le processus `init` (processus de `pid = 1`).

Indication Outre la primitive `fork()`, vous aurez besoin des primitives d'entrée-sortie, des primitives de manipulation de répertoires, des primitives `getpid()/getppid()`, `getuid()/geteuid()`, `getgid()/getegid()` (voir leurs pages de man), ainsi que de la primitive `sleep()` qui permet de suspendre un processus pendant un certain temps (man 3 `sleep`).

3 Créations multiples de processus fils

Ecrire un programme `multiple_fork.exe` qui crée `P` processus fils (`P=10`) s'exécutant en parallèle. Chaque processus fils exécute `N` fois (`N=10`) une boucle où il affiche sur la sortie standard une ligne réduite à son numéro d'ordre (de 0 à `P-1`). Le père attend la fin de ses fils.

Vérifier à l'aide de la commande shell `wc -c` que l'exécution de `multiple_fork.exe` produit bien le nombre de caractères attendus (200 avec les valeurs choisies pour `P` et `N`).

Vérifiez que les processus s'exécutent bien en parallèle en insérant dans la boucle de chaque processus, juste après l'affichage de son numéro, un appel à `sleep(1)`.

4 Propriétés des primitives de la famille `execXX()`

Ecrire un programme `exec_prop.exe` qui mette en évidence les propriétés suivantes :

- après un `fork()`, le fils hérite des buffers d'E/S (ceux de la bibliothèque standard `stdio`) du père, alors qu'après un `execXX()`, ces buffers sont perdus, car écrasés par ceux (initialement vides) du nouveau programme ;
- après un `execXX()`, le programme exécuté change, mais pas le `pid` du processus.

Indication Pour montrer le premier point, vous aurez besoin d'exécuter des E/S, par exemple des écritures sur la sortie standard, sans *flusher* le buffer. Pour ce faire, il suffit d'utiliser `printf()` sans terminer la chaîne à afficher par une fin de ligne (`'\n'`) :

```
printf("ceci ne flushe pas");
```

A l'inverse, pour forcer le *flush*, on peut écrire une fin de ligne ou appeler `fflush(stdout)`.

5 Le shell et la fonction `system()`

5.1 Principe d'exécution du shell

Lorsque le shell doit exécuter une commande, il crée un processus fils pour le faire. Ce fils exécute (par `execXX()`) le fichier binaire correspondant à la commande.

Ecrire un programme `shell-exec.exe` qui reproduit le comportement du shell lorsqu'il exécute les deux commandes successives suivantes :

```
who
ls -ls
```

Ce programme doit être écrit directement avec les primitives `fork()` et `execXX()`.

Modifier légèrement ce programme afin qu'il reproduise le comportement du shell lors de l'exécution des trois commandes successives :

```
who
cd
ls -ls
```

Rappel : `cd` sans paramètre renvoie le shell dans le répertoire initial (Home Directory).

Vous constaterez qu'il y a un problème avec l'exécution de `cd`. Essayez de trouver une solution. Puis comparez le résultat de votre programme à celui de l'exécution directe de la commande au terminal et aussi à celle d'un script shell contenant la commande en question. Dans quels répertoires êtes-vous après ces exécutions ? Pourquoi ?

5.2 La fonction `system()`

Ecrire une fonction `system()` telle que vue en cours et, grâce à elle, un programme `shell-system.exe` qui simule la boucle principale du shell (cette boucle doit lire les commandes sur l'entrée standard). Ce programme doit pouvoir exécuter correctement la suite des trois commandes précédente.

Note Il n'est pas demandé de supporter toute la syntaxe du shell ! Ignorez les alias, les variables, la *backquote*, les *jokers* dans les noms de fichiers, etc. Une commande sera de la forme :

```
nom param1 param2 param3...
```

où `nom` est le nom de la commande (identifiant en général son fichier binaire) et `parami` ses paramètres. Tous ces éléments sont des chaînes de caractères fixes (sans caractères spéciaux).

Indication La fonction `strtok()` permet de découper une chaîne de caractères en mots.

5.3 Utilisation du « set user bit » (`setuid`)

Ecrire un programme `fcap.exe` qui permette à un autre utilisateur (à choisir !) de lire ses fichiers personnels, même s'ils sont protégés en lecture pour le groupe et les autres. Le nom du fichier à lire sera passé en argument à ce programme.

Attention Pour des raisons de sécurité, les programmes « *setuid* » ne sont pas autorisés sur les partitions utilisateurs des serveurs de l'école. Pour tester ce programme, il faudra vous mettre dans un répertoire autorisant l'exécution de tels programmes (par exemple `/tmp`). Lisez le man de la commande shell `chmod` afin de savoir comment installer un tel programme.