Erick Gallesio, Stéphane Lavirotte, Jean-Paul-Rigault, Hélène Collavizza
Polytech'Nice – Sophia
SI3
2019-2020

TD n° 3
Processus

Ces exercices explorent les propriétés et l'utilisation des primitives de gestion de processus de Posix, en particulier `fork()`, `exec()` et `wait()`. Ils sont tous très simples et courts (`4.2` étant toutefois un peu plus long que les autres). L'important n'est pas uniquement de faire les exercices mais de bien comprendre les mécanismes mis en jeu.

Pour réaliser ces exercices, les primitives `getpid()`/`getppid()` seront utiles (voir leurs pages de `man`), ainsi sans doute que `sleep()`, qui permet de suspendre un processus pendant un certain temps (faire `man 3 sleep`).

Pour vous faciliter la vie (et ne pas perdre de temps), nous vous proposons un environnement de travail pour Visual Studio Code ainsi qu'un `Makefile` générique qui vous permettra de compiler tous vos programmes à partir des fichiers `.c` que vous allez créer.

```
https://lms.univ-cotedazur.fr/mod/resource/view.php?id=80916
```

1 Créations multiples

Exercice n°1:

Écrire un programme `multiple_fork` qui crée P processus fils (P = `10`) s'exécutant en parallèle. Le père doit attendre la fin de tous ses fils. Chaque processus fils exécute une boucle N fois (N = `10`) où il affiche sur la sortie standard son numéro d'ordre (de `0` à P - `1`) suivi du caractère retour à la ligne '`\n`'.

Vérifiez à l'aide de la commande Shell `wc -l` que l'exécution de `multiple_fork` produit bien le nombre de lignes attendues (`100` en principe, avec les valeurs choisies).

Exercice n°2:

Vérifiez que le processus s'exécute bien en parallèle en insérant dans la boucle de chaque processus, juste après l'affichage de son numéro, un appel à la fonction `sleep(1)`. Pour finir de vous convaincre du lancement en parallèle des processus, pendant l'exécution de votre programme (qui doit prendre `10` secondes avec la pause de `1` seconde et les paramètres choisis), lancez un terminal dans lequel vous listerez tous les processus de nom `multiple_fork.exe` :

```
ps aux | grep multiple_fork.exe
```

Quels sont les pid de chacun des processus ? Que pouvez-vous en conclure ?

2 Fin de vie de processus

Pour la suite des exercices, nous n'aurons pas besoin de créer P processus. Donc vous ne devez plus avoir de boucle pour la création des processus.

2.1 Zombie

L'exécution asynchrone entre processus père et fils a certaines conséquences. Souvent, le fils d'un processus se termine, mais son père ne l'attend pas. Le processus fils devient alors un processus **zombie** Le processus fils existe toujours dans la table des processus, mais il n'utilise plus les ressources du noyau. L'exécution de l'appel système `wait()` ou `waitpid()` par le processus père élimine le fils de la table des processus.

Exercice n°3:

Ecrire un programme `zombie` qui mette en évidence cette caractéristique de processus zombie. Comme précédemment, vous utiliserez la fonction `sleep` mais cette fois-ci dans le père sans faire d'appel à `wait()`ou `waitpid()`. Toujours avec la commande `ps aux`, vérifiez les informations de votre processus `zombie`.

TD n° 3
Processus

2.2 Orphelin

Il peut y avoir aussi des terminaisons prématurées, où le processus père se termine avant ses processus fils. Dans cette situation, ses processus fils sont adoptés par le processus `init`.

Exercice n°4:

Ecrire un programme `orphelin` qui mette en évidence l'adoption par un processus. Vous ferez une boucle d'attente jusqu'au changement de pid du père dans le fils, avec un `sleep` dans la boucle pour éviter que l'ordinateur ne chauffe. Vérifier le pid du processus qui a adopté l'orphelin.

Grâce à ce pid, trouvez quel est le processus en question grâce à la commande `ps aux`.

> **Remarque** : l'adoption d'un processus orphelin peut ne pas être le processus `init` de pid `1` dont le propriétaire est le super utilisateur, mais un processus `init` lancé en tant que simple utilisateur et non super-utilisateur. A ce moment-là, le processus ayant été lancé plus tardivement dans l'initialisation du système, il a un numéro `> 1`. Ce comportement est généralement constaté avec un environnement graphique comme Gnome.

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

Exercice n°5:

Écrire un programme `exec_prop` 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.
- On ne continue le code qui est écrit après un `execXX()`.

Vous réaliserez un deuxième programme `exec_prop-aux` qui sera celui exécuté par la fonction `execXX()`. Il affiche le pid du processus ainsi que `argv[0]`. Après avoir fait fonctionner cet exercice, vous tenterez de lancer votre `execlp` avec les paramètres suivants : `execlp("./exec_prop-aux.exe", "exec_prop-aux", "coucou", NULL);`

> **Suggestion** : Pour démontrer le premier point, vous aurez besoin de réaliser des écritures sur la sortie standard, sans vider (`flush`) 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")`. À l'inverse, pour forcer le flush, on peut appeler `fflush(stdout)` ou bien de faire un `printf` avec une fin de ligne (ce qui a pour conséquence de vider le buffer).

4 Le Shell et la fonction `system()`

4.1 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.

Exercice n°6:

Écrire un programme `shell_exec` 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()`.

TD n° 3
Processus

4.2 Extension de `shell_exec`

Exercice n°7:

Modifiez légèrement le programme `shell_exec` afin qu'il reproduise le comportement du Shell lors de l'exécution des trois commandes successives :

```
who
cd
ls -ls
```

On rappelle que `cd` sans paramètre renvoie le Shell dans le répertoire initial (« Home Directory »). Vous constaterez qu'il y a un problème avec l'exécution de la commande `cd` avec un `exec`. Essayez de trouver une solution (cf `getenv` et `chdir`). 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 ?

4.3 La fonction `system()`

Exercice n°8:

Écrire un programme `shell_system` qui simule la boucle principale du Shell et utilise la fonction `system()`(`man 3 system`)pour lancer les commandes (cette boucle infinie doit afficher le prompt et lire les commandes sur l'entrée standard). Bien entendu, grâce à ce programme, vous devez pouvoir exécuter correctement la suite des trois commandes précédentes.

> **Remarque** : On ne vous demande pas de supporter toute la syntaxe du Shell ! Vous ignorerez donc pour l'instant, les alias, les variables, la backquote, les jokers dans les noms de fichiers, etc. Pour vous, 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 $\texttt{param}_i$ ses paramètres. Tous ces éléments sont ici des chaînes de caractères fixes (sans caractères spéciaux).

> **Suggestion** : Il se pourrait bien que la fonction `strtok()` qui découpe une chaîne de caractères en mots puisse vous être de quelque utilité... Attention, cependant au fait qu'elle modifie la chaîne de caractères à laquelle on l'applique.

Exercice n°9:

Écrire une fonction `my_system()` analogue à la fonction `system()` vue en cours et remplacer l'appel à `system()` dans votre programme précédent.

Comparer la différence de comportement des deux versions `shell_system` lorsqu'on essaie d'interrompre un programme qui prend du temps à l'exécution. Vous pourrez tester cette différence de comportement en tapant ^C pendant l'exécution de commandes comme `ls -lR ~` ou `cat` sans paramètre.

TD n° 3
Processus

Exercices Complémentaires

Pour réaliser ces exercices, vous avez besoin de primitives sur l'ouverture, la lecture et l'écriture dans des fichiers, ainsi que la manipulation des dossiers en Posix. Sauf à déjà connaître ces fonctions, vous pourrez réaliser ces programmes après avoir vu la gestion des entrées-sorties en Posix.

5 Propriétés de `fork()`

Exercice n°10:

Écrire le programme `fork_prop` permettant de montrer les propriétés de l'appel-système `fork()`, en particulier :

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

Suggestion : Vous aurez besoin des primitives `getuid()`/`geteuid()`, `getgid()`/`getegid()`(voir leurs pages de `man`).

6 Utilisation du set user bit

Exercice n°11:

Écrire le programme `fcat` qui permet à un ami (que vous choisissez) de lire vos fichiers, 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 *set user bit* (`setuid`) ne sont pas autorisés sur les partitions utilisateurs de certains serveurs. Par conséquent, pour tester votre programme, il faudra vous mettre dans un répertoire autorisant l'exécution de programmes `setuid` (par exemple `/tmp`). On lira avec profit le manuel de la commande chmod afin de savoir comment installer un tel programme.