Processus et threads

Dans ce TP vous allez devoir utiliser différents appels systèmes. En pratique, un programme utilise rarement directement un appel système mais passe par des fonctions de la biliothèque `libc` qui se charge de le réaliser. Cela permet à la `libc` de proposer des interfaces plus agréables à l'utilisateur que celles souvent complexes proposées par l'OS.

Toutes les fonctions sont documentées dans les sections 2 et 3 des pages de manuel. Dans votre terminal, la commande "`man 3 exec`" vous permet par exemple de consulter la documentation des fonctions permettant de réaliser l'appel système exec. On peut lire que différentes interfaces sont proposées pour celui-ci ainsi que la manière de l'utiliser correctement et le fichier d'en-tête à inclure.

Pour les problèmes de ce TP vous aurez besoin des appels systèmes suivants, il vous est donc conseillé de consulter leur documentation :

`fork, exec, sleep, usleep, wait`

On rappel aussi qu'un programme peut prendre des arguments sur la ligne de commande. Afin d'accéder à ceux-ci, votre fonction `main` doit avoir le prototype suivant :

```
int main(int argc, char *argv[])
```

Le paramètre `argc` indique le nombre d'arguments donné au programme et le paramètre `argv` est un tableau de chaines de caractères contenant les arguments. Le premier argument donné au programme est son propre nom.

Exercice 1. Questions de cours

Les questions de cours sont à destinées à vous permettre de vérifier votre compréhension du cours. Elles sont à travailler à l'avance et ne seront pas traitées en TD ou TP.

1. Quelle est la définition d'un processus ?
2. De quoi se compose un processus ? Expliquez.
3. Peut-on avoir plusieurs processus prêts dans le système ?
4. Le système d'exploitation est-il un processus ?

Exercice 2. Création de processus

Dans cet exercice et les suivant, vous devez utiliser les appels systèmes vus en cours ainsi que d'autres qui sont documentés dans les pages de manuels des systèmes UNIX. (Dans un terminal utilisez la commande `man` suivie du nom de l'appel système)

1. Écrivez en C un programme qui affiche "Bonjour", attend 5 secondes et affiche "Au revoir" avant de se terminer.
2. Modifiez votre programme de manière à ce qu'il effectue un `fork` et que les deux processus préfixent leurs affichages par le numéro de processus. Le processus père attendra 5 secondes tandis que le fils attendra seulement 2 secondes avant de se terminer.

Exercice 3. Explosion

On considère le programme suivant : (qui n'est bien sûr pas un exemple de bonne programmation)

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 int main(void) {
4     fork();
5     fork();
6     fork();
7     fork();
8     return EXIT_SUCCESS;
9 }
```

1. Combien de processus sont créés par l'exécution de se programme ?

2. Que ce passe-t-il si l'on remplace le premier appel à `fork` par un appel à `exec` exécutant le programme lui-même ?

3. Même question mais pour le deuxième appel à `fork` ?

Exercice 4. Recherche parallèle

La recherche des occurences d'une valeur dans un tableau non trié nécessite de parcourir tout le tableau. Si le tableau est grand ou si la comparaison des valeurs est coûteuse, cette recherche peut être longue. Si l'ordinateur exécutant le programme dispose de plusieurs processeurs ou cœurs, il est possible de réaliser la recherche simultanement sur chacun d'entre eux afin de l'accelérer.

Dans ce problème, vous devez écrire un programme qui alloue un grand tableau d'entiers aléatoires, puis recherche toutes les occurences d'une valeur et affiche leurs indices en utilisant deux processus.

Les ordinateurs modernes sont très rapides et si l'on souhaite pouvoir observé des choses intéressantes ici il faudrait travailler sur un tableau vraiment immense ou utiliser une fonction de comparaison complexe. Pour garder un programme simple, le plus facile est d'utiliser un tableau de taille 1000 contenant des entiers inférieurs à 100 mais d'ajouter un appel à `usleep(10000);` *à l'interieur de la boucle de recherche pour simuler une fonction de comparaison coûteuse.*

1. Commencez par écrire un premier programme non-parallèle qui alloue un grand tableau, le remplis d'entiers aléatoires et recherche toutes les occurences de la valeur 0. Pour chaque occurence trouvée, affichez un message indiquant son indice.

2. Il s'agit maintenant de parallèliser ce programme sur deux processus, pour cela le programme doit réaliser un `fork`, puis un des processus doit chercher les valeurs 0 dans la première moitié du tableau pendant que l'autre cherche dans la deuxième moitié.

3. Comparez les temps d'exécution et les affichages des programmes des deux questions précédentes. Que remarquez-vous ?

Exercice 5. Signaux

Les signaux permettent à l'OS d'informer immédiatement un processus d'un évennement. Ils sont utilisés principalement pour les situations d'erreurs mais peuvent aussi être utilisés dans d'autre circonstances telles qu'une alarme ou le redimensionement du terminal. Dans cet exercice, nous allons utiliser le signal `SIGINT` qui à l'avantage de pouvoir être envoyé simplement à un processus par la combinaison de touches `ctrl-C`.

1. Écrivez un programme qui simplement attend 10 secondes avant de quitter. Lorsqu'il reçoit le signal `SIGINT` affiche "Au revoir" sur le terminal et quitte sans attendre la fin des 10 secondes.

2. Modifiez votre programme afin qu'avant de patienter 10s il crée un processus fils. Ce processus fils devra attendre 5s puis envoyer le signal `SIGINT` à son processus père.

Exercice 6. Commande shell

Le shell de base sous UNIX, `sh`, offre une commande `if` permettant d'exécuter une commande de manière conditionelle en fonction du résultat de l'exécution d'une autre commande. Dans cet exercice, ils vous est demandé d'implémenter une version simplifiée de cette commande qui s'utilise de la manière suivante :

```
si <commande1> alors <commande2> sinon <commande3>
```

Le programme `si` exécute la première commande, attend qu'elle se termine et si son exécution s'est bien passé exécute la deuxième commande, sinon il exécute la troisième commande. Chaque commande peut avoir des arguments et la troisième est optionelle. (si elle n'est pas présente le mot clé `sinon` ne doit pas être présent non plus)

Afin de simplifier les aspects purement programation en C et se concentrer sur les aspects système, on considère que l'utilisateur utilise correctement la commande, vous n'avez pas à gérer les cas ou les arguments donnés sont faux. Il n'est pas, par exemple, nécessaire de gérer le cas ou le mot clé `alors` *est absent ou le cas ou le mot clé* `sinon` *apparait avant le mot clé* `alors`.

1. Commencez par une version basique du programme qui ne prend que deux arguments, la commande à exécuter pour le test et celle à exécuter si la première se termine correctement. (Avec un code de retour de 0) Les deux commandes n'acceptent donc pas d'arguments et sont respectivement dans `argv[1]` et `argv[2]`.

2. Modifier votre programme afin qu'il accepte des commandes avec arguments. Le mot clé `alors` doit apparaitre parmi les arguments du programme, tout ce qui est avant constitue le test et tout ce qui est après constitue la commande à exécuter.

3. Enfin, ajoutez la gestion du mot clé `sinon` afin d'obtenir la commande complète.

Exercice 7. Redirections

Attention, cet exercice utilise le concept des descripteurs de fichiers qui seront étudiés plus tard dans le cours. Il demande peu de code mais plus de réflexion que les autres, vous pourrez y revenir une fois le cours sur les fichiers assimilé.

Contrairement à d'autre systèmes tels que Windows, les systèmes UNIX décomposent l'exécution d'un programme en deux étapes :

- la création du processus par duplication d'un processus existant ;
- son recouvrement par le programme à exécuter.

Cette séparation permet au processus qui souhaite exécuter un autre programme de réaliser certaines opérations de configuration du processus entre les deux étapes. C'est par exemple à ce moment qu'il est possible de mettre en place la redirection des entrées et sorties standard comme le fait le shell. La sortie standard est par exemple fournie par le système sous la forme d'un fichier virtuel dont le descripteur est donné par la constante `STDOUT_FILENO`. Il est possible de changer ce descripteur afin de diriger la sortie vers un fichier ou vers un autre processus.

Dans cet exercice, vous devez écrire un programme exécutant l'équivalent de la commande shell "`ls . > stdout.txt`". C'est-à-dire l'exécution de la commande `ls` afin de lister le répertoire courant et d'écrire le résultat dans le fichier `stdout.txt`.

1. Commencez par écrire un programme simple qui execute la commande `ls` dans le répertoire courant à l'aide de `fork+exec`. Le processus père devra attendre la fin de l'exécution du processus fils puis afficher si l'exécution s'est bien passée.

2. Maintenant modifier le code exécuté par le processus fils afin qu'il ouvre en écriture le fichier `stdout.txt` et le réassigne à la sortie standard. Pour cela, vous aurez besoin des appels système open et dup2.

Le shell utilise ce méchanisme pour les redirections vers des fichiers. Pour la connection de deux programmes à l'aide d'un tube comme "`ls | more`" les choses sont un peu plus complexes. L'appel système `pipe` demande au système d'allouer deux descripteurs de fichiers connectés de telle manière que ce qui est écris dans le premier peut être lus dan sle deuxième. Le shell l'utilise puis réalise deux fork afin de créer les deux processus fils. Ensuite, chaque fils va modifier son entrée ou sa sortie standard avant de réaliser l'appel à `exec`.