TP 5
Début avec les processus (1h30)

Rappel : Un processus est un programme en cours d'exécution sur le système. L'un des rôles du système d'exploitation est de permettre à un ensemble de processus de progresser dans leur exécution en même temps sur votre système : il doit partager les ressources disponibles (processeurs, mémoire, périphériques) entre les différents processus. Ainsi, à chaque instant de nombreux processus sont soit en cours d'exécution, soit bloqués en attente d'exécution, mais vous avez toujours l'impression qu'ils s'exécutent en « même temps ».

Remarque : Dans le sujet, on fera attention à la différence entre le chiffre `1` et la lettre `l`.

EXERCICE 1 ▶ Observation des processus depuis le shell

Dans cet exercice, on va considérer un exemple de processus dont l'exécution « se voit à l'œil nu » : il s'agit du script `clock.py` qui vous est fourni, et qui affiche une horloge à aiguilles avec une trotteuse.

Si vous êtes sur votre ordinateur, et que le script `clock.py` ne fonctionne pas, c'est sûrement à cause de l'absence de la bibliothèque Python `tkinter` ; si votre système est basé sur Debian, vous pouvez installer le package `python3-tk`.

1) Par défaut, la commande `ps` liste les processus rattachés au terminal (`TTY`) dans lequel elle est lancée. Ouvrez un terminal, et entrez la commande `ps` : parmi les informations listées, interprétez les colonnes `CMD`, `PID` et `TTY`.

2) Dans votre terminal, téléchargez dans votre répertoire de travail pour ce TP le script `clock.py`, et lancez par deux fois la commande `python3 ./clock.py &` puis entrez à nouveau `ps` : qu'observez-vous ?

3) Maintenant, entrez la commande `bash`, puis lancez à nouveau une horloge à l'arrière plan avec `./clock.py &`. Utilisez maintenant la commande `ps -l` pour afficher plus d'informations. Comment interprétez-vous la colonne `PPID` ?

4) Dans le résultat de la commande `ps -l` précédente, comment interprétez-vous la colonne `UID` ? Pour vous aider, vous pouvez aussi vous intéresser au résultat de la commande `id`.

5) Vous avez dû comprendre que les processus sont organisés d'une façon arborescente. Pour la visualiser, vous pouvez utiliser `pstree` : à l'aide de `ps`, repérer le `PID` du premier processus `bash` lancé dans votre terminal ; on note n ce `PID` ; entrez la commande `pstree -p` n. Que constatez vous ?

6) Maintenant, ouvrez un *nouveau* terminal, allez dans le répertoire contenant l'exécutable `clock.py` puis entrez la commande `ps` : vous ne retrouvez pas dans la liste les processus que vous aviez lancés dans le premier terminal… Comment afficher la liste de tous les processus que vous avez lancés sur le système ?

7) Comment afficher tous les processus lancés sur le système ?

8) Fermez tous ces terminaux utilisés pour l'exercice (clic sur la croix). Cela supprime les processus `clock.py` lancés…

EXERCICE 2 ▶ Début avec fork()

La fonction `fork()` permet à un processus, dit « père », de se dupliquer : à la suite d'un appel réussi à `fork()`, le système comporte un nouveau processus, dit « fils », qui est une copie du père. **Il est important de comprendre qu'après l'appel à fork(), le fils continue d'exécuter une copie du code du père.**

1) Consultez le manuel de la fonction `fork()`. Quels « includes » devez-vous placer dans votre programme pour pouvoir utiliser cette fonction ? Quelles valeurs retourne-t-elle ?

2) Consultez le manuel de la fonction `sleep()`. Que permet de faire cette fonction ?

3) Écrivez un programme `tstfork1.cpp` dans lequel le processus père (la fonction `main()`) crée un processus fils avec `fork()`. Dans la suite du programme :

- le processus père devra afficher les 9 lettres de `'a'` à `'i'`, en s'endormant entre chaque affichage pendant 1 seconde, puis affichera « `fin du père` » avant de se terminer en retournant 0 ;
- le processus fils devra afficher les entiers de 1 à 10, en s'endormant entre chaque affichage pendant 1 seconde, puis affichera « `fin du fils` » avant de se terminer en retournant 0.

Faites suivre chaque affichage d'un `std::endl` pour forcer un affichage rapide sur la sortie standard avec (vous pouvez aussi utiliser `std::flush`). N'obligez pas le père à attendre la fin de son fils pour l'instant. Compilez votre programme avec `g++ -o tstfork1 tstfork1.cpp`, puis testez en le lançant plusieurs fois : comment constatez-vous que le système fait progresser « en même temps » l'exécution des deux processus ? Observez-vous toujours le même affichage ?

4) Mesurez, en secondes, le temps d'exécution de votre programme avec la commande
`/usr/bin/time -f "%es" ./tstfork1`
Quelle durée mesurez vous effectivement ?

5) Consultez le manuel de `waitpid()`. Quels « includes » devez-vous placer dans votre programme pour pouvoir utiliser cette fonction ? Quels arguments passer à la fonction pour permettre à un processus d'attendre un fils de PID spécifique, sans récupérer le statut du fils terminé ?

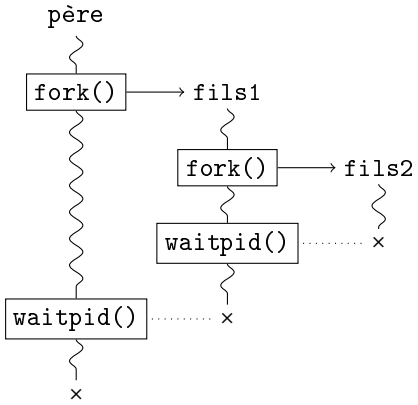
6) Ajouter dans le programme un appel à `waitpid()` permettant au père d'attendre la fin de son fils, juste avant d'afficher « `fin du père` » et de se terminer. Désormais, observez-vous à l'exécution toujours le même affichage ? Peut-on en tirer une conclusion générale sur l'ordre d'exécution de l'affichage dans les deux processus ?

7) Mesurez à nouveau le temps d'exécution de votre programme programme : quelle durée mesurez vous désormais ?

EXERCICE 3 ▶ **Une famille de processus**

1) Consultez le manuel des fonctions `getpid()` et `getppid()`. Que retournent ces fonctions ?

2) Écrivez un nouveau programme, `tstfork2.cpp`, dans le quel le processus principal crée un fils, qui lui même crée un fils, comme dans le diagramme ci-dessous. Chaque processus doit attendre la terminaison du fils crée grâce à un appel bloquant à `waitpid()` (en attendant spécifiquement le fils créé).



Le processus `fils2` doit se mettre en sommeil pendant 60 secondes. Chaque processus doit afficher, avec des messages clairs, son PID et le PID du fils qu'il a créé.

3) Testez votre programme, et vérifiez qu'il crée bien l'arborescence de processus demandée.

4) Rajoutez les instructions traitant les cas où la valeur de retour des appels `fork()` et `waitpid()` est -1.