R3.05 Programmation Système
Séance Pratique 1 : processus, forks, exec

Pour vous aider tout au long de ce TP, il est recommandé de vous référer aux supports de cours, ainsi qu'au man de Linux.

Dans les codes fournis, les inclusions de librairies ont été omises

Exercice 1

Dessinez la hiérarchie de processus créée par le code suivant :

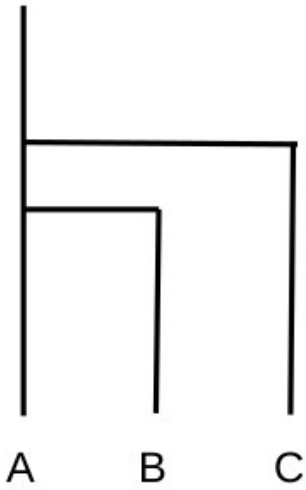
```
int main() {

      int retour1 = fork();
      if (retour1 == 0) {
            if (fork() > 0) {
                  printf("Processus A\n");
            } else {
                  printf("Processus B\n");
            }
      } else {
            printf("Processus C\n");
      }
      return 0;
}
```

Vérifiez maintenant cette hiérarchie en exécutant ce code, et en affichant pour chaque processus son **pid** et son **ppid**.
Pour cela, vous utiliserez les fonctions **getpid()** et **getppid()**.
Donnez la nouvelle hiérarchie corrigée, si votre précédente était fausse.

Exercice 2

Modifiez le code précédent afin d'obtenir la hiérarchie suivante :



À l'aide de la fonction **sleep** (voir le man pour son utilisation, attention c'est une fonction, sa documentation est donc dans la section 3 du man), mettez les trois processus en pause pendant 10 secondes (par exemple juste après le printf). Lancez une exécution de votre programme, et pendant les 10 secondes pendant lesquelles les processus sont en pause, ouvrez un autre terminal, et retrouvez vos trois processus dans l'arbre des processus grâce à la commande **pstree**.

Exercice 3

On propose d'écrire un programme qui va compter le nombre de nombres premiers inférieurs à une certaine borne N donnée en paramètre, grâce à deux processus parallèles. Pour cela, le processus principal va créer deux processus fils qui se chargeront de compter le nombre de nombres premiers entre 0 et n/2 pour le premier, et entre n/2 et n pour le second. Ils retourneront leur résultat via le return de leur main, que le processus père récupèrera. On vous donne le squelette dans le fichier nb_primes.c.

Complétez le programme fourni, et testez le avec la commande time. Que remarquez-vous ?

Exercice 4

Écrivez un programme **`shell`** implémentant les fonction basiques d'un interpréteur de commande (« shell »). Ce programme boucle sur les trois actions suivantes :

1. affichage d'un message d'invite,
2. saisie d'une instruction par l'utilisateur
3. exécution de l'instruction saisie

jusqu'à ce que l'utilisateur saisisse le caractère fin de fichier (CTRL+D).

Remarques :

- On vous fournit un fichier **ligne_commande.c** (et son header **ligne_commande.h**), dans lequel il y a notamment la fonction **lis_ligne()**. Celle-ci attend que l'utilisateur saisisse une commande, puis retourne un tableau de chaîne de caractères avec cette commande

- Pour compiler tout ceci :
  gcc -o shell shell.c ligne_commande.c

- Si l'utilisateur saisit "CTRL+D", alors le tableau retourné par **lis_ligne()** vaut NULL
- Si l'utilisateur ne saisit rien (il fait un retour à la ligne), alors la première case du tableau retourné par **lis_ligne()** vaut NULL

- Une fois ce tableau récupéré et valide, votre processus doit se dupliquer (avec fork), puis :
- Le père attend la fin du processus fils (avec **wait** ou **waitpid**), puis il continue la boucle pour attendre une nouvelle instruction
- Le fils va exécuter la commande saisie. Pour cela on utilisera la fonction **execvp** de la librairie **unistd.h**, qui prend en entrée ce que retourne la fonction **lis_ligne()**, plus précisément :
  - le premier paramètre de **execvp** est la première case du tableau retourné par **lis_ligne()**
  - le second paramètre de **execvp** est le tableau retourné par **lis_ligne()**
- Si la commande saisie par l'utilisateur est valide, alors ce qui suit l'appel à **execvp** n'est jamais exécuté, car le processus exécute dorénavant le nouveau programme, et non plus votre shell
- Donc si l'instruction qui suit l'appel à **execvp** est exécutée, cela signifie que l'utilisateur a saisi une commande qui n'existe pas.