TP 4 – Signaux, Création de processus

Vous devez faire au moins les exercices 1 à 5.
Un exemple de sujet d'examen concernant cette partie est donné dans l'exercice 6. Toutefois, pour la dernière question, il faudra attendre le prochain cours avant de pouvoir y répondre.

Exercice 1 - `kill`, `signal`

Cet exercice porte sur les programmes `sigsend` et `sigcatch` dont les sources sont disponibles dans `/pub/FISE_OSSE11/syscall/sigsend.c` et `/pub/FISE_OSSE11/syscall/sigcatch.c`.

1. Complétez `sigsend.c` et compilez le.

```
sh$ gcc -Wall -Wextra -o sigsend sigsend.c
```

2. Décrivez le comportement du programme `sigcatch` :

   main

   lignes 24 à 29 attache le ..... à tous les signaux de ... à ... .
   lignes 26 à 27 affiche un message d'erreur pour les signaux qui ne sont pas ..... .
   lignes 33 à 36 boucle infinie : affichage d'un ..... puis attente d'un ..... .

   callback (gestionnaire de signal)

   lignes 15 à 16 affiche le .....

3. a. Compilez le.

```
sh$ gcc -Wall -Wextra -o sigcatch sigcatch.c
```

   b. Lancez deux fois le programme `sigcatch`, dans deux terminaux différents.

```
sh$ xterm -e ./sigcatch &
sh$ xterm -e ./sigcatch &
```

   c. Dans quel état sont les deux processus `sigcatch` ?

4. Envoyez quelques signaux à chacun des deux processus.

```
sh$ ./sigsend hup <pid1>
sh$ ./sigsend 10 <pid2>
```

5. a. Tapez <CTRL-C> dans les terminaux des deux processus `sigcatch`.
   b. Se sont-ils terminés ?
   c. Que se passe-t-il quand <CTRL-C> est tapé dans un terminal ?

6. a. Tapez <CTRL-Z> dans les terminaux des processus `sigcatch`.
   b. Se sont-ils mis en pause ?
   c. Que se passe-t-il quand <CTRL-Z> est tapé dans un terminal ?

7. a. Envoyez le signal `SIGSTOP` à un des processus `sigcatch`.

```
sh$ ./sigsend STOP <pid1>
```

   b. A-t-il été reçu par le processus ?
   c. Envoyez lui maintenant le signal `SIGUSR1`. L'a-t-il reçu ?
   d. Tapez <CTRL-C> dans le terminal du processus. A-t-il reçu le signal `SIGINT` ?
   e. Envoyez lui maintenant le signal `SIGCONT`. L'a-t-il reçu ?
   f. Qu'en est il des signaux `SIGUSR1` et `SIGINT` précédents ?

8. Recommencez l'expérimentation précédente en envoyant plusieurs fois le signal `SIGUSR1`.

   Combien de réceptions du signal `SIGUSR1` sont traitées ?

9. Terminez les deux processus en utilisant la commande shell `kill` (qui est une version améliorée du programme `sigsend`).

Exercice 2 - `pause`, `alarm`

L'appel système

```
int pause()
```

met le processus courant en pause, et ce jusqu'à réception d'un signal.

L'appel système

```
int alarm(int nb)
```

indique au système d'envoyer un signal `SIGALRM` au processus courant dans `nb` secondes.

1. Copiez le fichier `/pub/FISE_OSSE11/syscall/mysleep.c`, et écrivez le code de la fonction `mysleep`. Cette fonction prend en argument un entier positif `nb`, ne renvoie rien, et a pour effet de suspendre l'exécution du processus courant pendant `nb` secondes.

2. Compilez votre code et testez le.

   note : Votre programme doit s'arrêter au bout de `nb` secondes, et afficher le message `done`. Si ce n'est pas le cas, vous avez probablement un problème dans le traitement des signaux.

Exercice 3 - `fork`, `wait`

Le but de cet exercice est d'écrire le programme `hello`, dont le comportement est le suivant.

— Le programme principal (père) :
  1. crée un premier fils grâce à un appel à `fork`,
  2. crée un deuxième fils (toujours à l'aide de `fork`),
  3. écrit `"hello"` sur le flux standard de sortie,
  4. se termine.

— Le premier fils :
  1. attend 2 secondes,
  2. écrit `"\n"` sur le flux standard de sortie,
  3. se termine.

— Le deuxième fils :
  1. attend 1 seconde
  2. écrit `" world"` sur le flux standard de sortie,
  3. se termine.

1. Écrivez le programme `hello`, et testez-le.

2. Modifiez le programme `hello` pour que le père attende l'arrêt de ses deux fils avant de se terminer.

Exercice 4 - `fork`, `wait`, `sleep`

Le but de cet exercice est d'écrire le programme `sleepsort`, de sorte que

sleepsort n_1 n_2 ... n_k

écrive les entiers n_i (supposés positifs) dans l'ordre croissant sur le flux standard de sortie.

Pour cela, nous allons procéder de la façon suivante :

— Le programme principal (père) va créer un fils par entier n_i, puis attendre l'arrêt de tous ses fils.

— Le fils numéro i va attendre n_i secondes, puis écrire l'entier n_i sur le flux standard de sortie standard, et enfin se terminer.

1. Écrivez le fichier `sleepsort.c`.

2. Compilez et testez votre programme.

```
sh$  sleepsort 3 1 5 4 3
1
3
3
4
5
sh$
```

Exercice 5 - `fork`, `wait`

L'objectif de cet exercice est la réalisation d'une version parallélisée de la commande `wc -l`.

1. Écrivez le fichier `wc-par-v1.c`, selon l'approche suivante :
   - Le père calcule la taille `sz` du fichier f passé en argument (`argv[1]`), puis crée 4 fils et attend que ses fils s'arrêtent avant de se terminer.
   - Le premier fils ouvre le fichier f, lit les octets 0 (inclus) à $sz/4$ (exclu) de f, compte le nombre de sauts de ligne (caractère `'\n'`) de cette partie du fichier, et affiche le résultat sur le flux standard de sortie. Il traite ainsi la plage $[\![0,\ \frac{sz}{4}[\![$ du fichier f.
   - Le deuxième fils fait la même chose, mais sur la plage $[\![\frac{sz}{4},\ \frac{sz}{2}[\![$ du fichier.
   - Le troisième fils fait la même chose, mais sur la plage $[\![\frac{sz}{2},\ \frac{3\,sz}{4}[\![$ du fichier.
   - Le quatrième fils fait la même chose, mais sur la plage $[\![\frac{3\,sz}{4},\ sz[\![$ du fichier.

2. Testez votre implémentation et vérifiez que vos résultats sont cohérents avec ceux produits par la commande `wc -l`.

3. Faites un essai avec un fichier de 200 Mo placé dans `/tmp`.

```
sh$ dd if=/dev/urandom of=/tmp/bigfile.txt bs=1M count=200
sh$ wc -l /tmp/bigfile.txt ; ./wc-par-v1 /tmp/bigfile.txt
```

4. Ecrivez la variante `wc-par-v2.c`, dans laquelle les fils s'arrêtent avec comme code de retour le nombre de sauts de lignes lus. Le père pourra alors récupérer ces codes et en faire la somme.

> **aide :** Consultez le `man` de `wait` pour voir comment utiliser la macro `WEXITSTATUS`.

5. Quels sont les défauts de cette approche ?

Exercice 6 - Processus **(examen 2022-2023)**

On considère dans cet exercice le programme `abc`, dont le comportement est le suivant.

Le processus principal, qu'on appellera *père* dans la suite :

- crée un nouveau processus *fils*,
- écrit "aaa " sur le flux standard de sortie,
- attend la terminaison de *fils*,
- écrit "\n" sur le flux standard de sortie.

Le processus *fils* :

- crée un nouveau processus *petit-fils*,
- écrit "bbb " sur le flux standard de sortie,
- attend la terminaison de *petit-fils*.

Enfin, le processus *petit-fils* :

- écrit "ccc " sur le flux standard de sortie.

1. Écrivez le code C complet du programme `abc`. Vous utiliserez des fonctions de **flux noyau** pour les écritures mentionnées ci-dessus.

2. Que pourra-t-on obtenir à l'écran si on exécute le programme `abc` ?

3. Expliquez comment faire pour que le processus *père* récupère le numéro (PID) du processus *petit-fils* à l'aide d'un *pipe*.