Linux 2: Appels Systèmes - BINV2181 (tp04- signaux)

4.1. Signaux entre processus père et fils : Le chat aux 7 vies

a) Concevez un processus père qui engendre un fils exécutant une boucle semi-active (i.e. une boucle infinie appelant *sleep*). Faites un ps pour observer les processus actifs. Que se passe-t-il ?

A l'aide du shell, faites un kill -SIGUSR1 au processus et refaites un ps.

- b) Modifiez votre programme pour que le père envoie le signal SIGUSR1 à son fils et que le fils affiche le message « signal SIGUSR1 reçu! » à la réception de celui-ci, pour ensuite s'arrêter.
 - 1. Attention cette semaine, le signal doit être armé avant la création du fils. Si ce n'est pas fait, le père risque d'envoyer le signal SIGUSR1 à son fils avant que celui-ci n'ait eu le temps d'armer son handler, provoquant ainsi la mort du fils. La semaine prochaine nous verrons une autre manière de faire.
 - 2. Attention également que la fonction « printf » n'est pas « signal-safe » et donc ne peut pas être utilisée dans un signal handler.
- c) Processus « chat aux 7 vies » : modifiez votre code pour que le père envoie toutes les secondes un signal SIGUSR1 à son fils. Le fils ne devra se terminer que lorsqu'il aura reçu 7 fois le signal de son père.
 - 1. Attention, des signaux peuvent être perdus. Le père devra donc peut-être envoyer plus de 7 signaux à son fils.
 - 2. Attention, vous devez bien faire attention à ce que votre fils ne se termine pas avant la réception des 7 signaux.
 - 3. Notez que lorsque le fils est terminé un signal SIGCHLD sera automatiquement envoyé au père. Utilisez ce signal pour provoquer l'arrêt du processus père.

Voici un exemple d'exécution :

```
signal SIGUSR1 reçu! Fin du père
```

4.2. Programme qui affiche le signal reçu

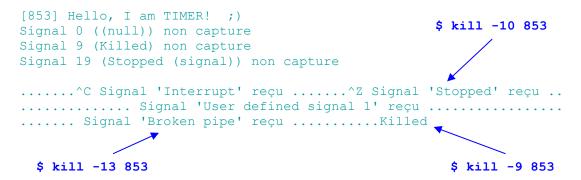
Ecrivez un programme « *timer* » qui, à la réception d'un signal, affichera simplement le signal reçu avant de reprendre son traitement. Pour afficher le signal reçu, utilisez la fonction strsignal (int sig) définie dans <string.h> et ajoutez -D_DEFAULT_SOURCE aux flags de compilation.

Limitez-vous aux signaux classiques, càd. les 32 premiers (pour afficher la liste des signaux définis sur votre système, tapez la commande kill -1) et affichez un message d'erreur si un signal n'a pas pu être armé.

Une fois les signaux armés, le traitement de votre programme se limitera à une boucle infinie consistant en l'écriture d'un point suivi d'un sleep(1). Affichez un message indiquant si une erreur autre que l'interruption par un signal (errno!=EINTR) s'est produite pendant l'exécution du write.

Testez votre programme avec différents signaux (commande kill exécutée depuis un autre terminal). Essayez notamment d'arrêter le processus avec Ctrl-C et Ctrl-Z.

Voici un exemple d'exécution :



4.3. Appels système : pipe & sigaction

Reprenez la solution de l'exercice 3.A sur les pipes et modifiez-la de sorte que le fils ferme l'extrémité en lecture du pipe. L'exécution du programme s'arrêtera lorsque le père tentera d'écrire sur le pipe. Pourquoi ?

Le code d'erreur de votre programme pourrait vous mettre sur la piste¹. Pour l'obtenir, entrez la commande bash **echo \$?** après avoir exécuté votre programme.

Modifiez votre programme en armant un gestionnaire de signal afin que le père affiche un message explicatif avant de se terminer (si possible avec l'exit code correspondant au signal reçu).

4.4. Nohup

Ecrivez un programme qui sera une version simplifiée de la commande linux *nohup*. Elle aura la spécification suivante d'après Wikipedia (20/03/2023) :

nohup est une commande Unix permettant de lancer un processus qui restera actif même après la déconnexion de l'utilisateur l'ayant initiée. Combiné à l'esperluette qui permet le lancement en arrière-plan, nohup permet donc de créer des processus s'exécutant de manière transparente sans être dépendants de l'utilisateur.

N'hésitez pas à consulter la page suivante pour avoir un peu plus d'informations : https://fr.wikipedia.org/wiki/Nohup

¹ https://tldp.org/LDP/abs/html/exitcodes.html