

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 !
signal SIGUSR1 reçu !
signal SIGUSR1 reçu !
signal SIGUSR1 reçu !
signal SIGUSR1 reçu !
signal SIGUSR1 reçu !
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, c  d. les 32 premiers (pour afficher la liste des signaux d  finis sur votre syst  me, tapez la commande `kill -l`) et affichez un message d'erreur si un signal n'a pas pu   tre arm  .

Une fois les signaux arm  s, le traitement de votre programme se limitera    une boucle infinie consistant en l'  criture 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'ex  cution du `write`.

Testez votre programme avec diff  rents signaux (commande `kill` ex  cut  e depuis un autre terminal). Essayez notamment d'arr  ter le processus avec Ctrl-C et Ctrl-Z.

Voici un exemple d'ex  cution :

```
[853] Hello, I am TIMER! ;)
Signal 0 ((null)) non capture
Signal 9 (Killed) non capture
Signal 19 (Stopped (signal)) non capture

.....^C Signal 'Interrupt' re  u .....^Z Signal 'Stopped' re  u ..
..... Signal 'User defined signal 1' re  u .....
..... Signal 'Broken pipe' re  u .....Killed
```

```
$ kill -10 853
$ kill -13 853
$ kill -9 853
```

4.3. Appels syst  me : pipe & sigaction

Reprenez la solution de l'exercice 3.A sur les pipes et modifiez-la de sorte que le fils ferme l'extr  mit   en lecture du pipe. L'ex  cution du programme s'arr  tera lorsque le p  re tentera d'  crire 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 $?` apr  s avoir ex  cut   votre programme.

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

4.4. Nohup

Ecrivez un programme qui sera une version simplifi  e de la commande linux *nohup*. Elle aura la sp  cification suivante d'apr  s Wikipedia (20/03/2023) :

***nohup** est une commande Unix permettant de lancer un processus qui restera actif m  me apr  s la d  connexion de l'utilisateur l'ayant initi  e. Combin      l'esperluette qui permet le lancement en arri  re-plan, nohup permet donc de cr  er des processus s'ex  cutant de mani  re transparente sans   tre d  pendants de l'utilisateur.*

N'h  sitez pas    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>