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*) et puis se termine. Faites un `ps` pour observer les processus actifs. Que constatez-vous ?

   A l'aide du shell, faites un `kill -SIGUSR1` au processus fils 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.

   REMARQUES :

   - 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.
   - Attention également que la fonction *printf* n'est pas « signal-safe » et ne peut donc pas être utilisée dans un gestionnaire de signal.

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.

   REMARQUES :

   - Attention, des signaux peuvent être perdus. Le père devra donc peut-être envoyer plus de 7 signaux à son fils.
   - Vous devez bien faire attention à ce que votre fils ne se termine pas avant la réception des 7 signaux.
   - 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

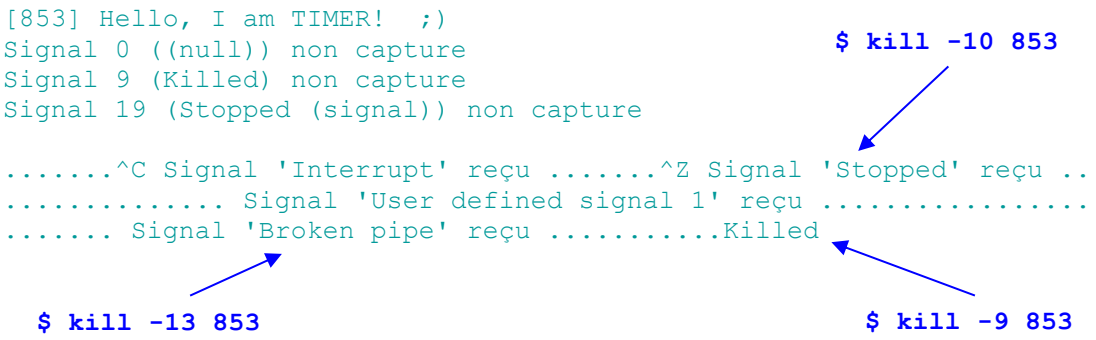
Ecrivez un programme « *timer* » qui, à la réception d'un signal, affichera simplement le signal reçu avant de reprendre son traitement. Pour afficher le signal reçu, utilisez la fonction `strsignal(int sig)` définie dans <string.h> et ajoutez -D_DEFAULT_SOURCE aux flags de compilation.

Limitez-vous aux signaux classiques, càd. les 32 premiers (pour afficher la liste des signaux définis sur votre système, tapez la commande `kill -l`) 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. Interruption d'un appel système par un signal

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 piste[1]. 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).

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

4.4. Appels système : `pipe & sigaction`

Mr. Pointilleux, un enseignant de Vinci, voudrait vérifier la validité de formules utilisées dans différents cours de programmation. Deux formules sont généralement utilisées pour générer un entier aléatoire compris entre 0 et MAX_VAL-1 :

1. Formule simple (modulo) :

   ```
   rand() % MAX_VAL
   ```

2. Formule complexe (conversion en réels) :

   ```
   (int) (rand() / (RAND_MAX+1.0) * MAX_VAL)
   ```

où `rand()` est une fonction qui génère une valeur entière pseudo-aléatoire dans l'intervalle [0,RAND_MAX] (RAND_MAX et `rand()` sont définies dans stdlib.h).

Mr. Pointilleux vous demande d'écrire un programme pour vérifier la validité des ces deux formules.

Processus PERE

Le programme crée deux fils, un pour vérifier chacune des formules. Le processus père génère ensuite des nombres aléatoires dans une boucle infinie à l'aide de la fonction `rand()`. Il transmet chaque valeur aléatoire à ses deux fils via deux pipes (un pipe pour communiquer la valeur à chaque fils).

Lorsque l'utilisateur entre Ctrl-Z (SIGTSTP), le processus libère ses ressources, attend la terminaison de ses fils et s'arrête.

Processus FILS

Chaque fils lit sur son pipe les valeurs aléatoires que le processus père lui envoie. Il applique ensuite sa formule pour restreindre cette valeur à l'intervalle [0,MAX_VAL[: le fils 1 applique la 1e formule et le fils 2 applique la 2e formule.

Chacun met à jour deux compteurs : `cnt`, le nombre de valeurs lues sur le pipe, et `nInf`, le nombre de valeurs résultantes de sa formule qui sont strictement inférieures à la constante SEUIL (voir ci-dessous).

A chaque fois que l'utilisateur entre Ctrl-C[2] (SIGINT) au clavier, le fils calcule et affiche le pourcentage de nombres générés qui sont inférieurs à SEUIL, càd. `nInf/cnt`.

Lorsque le processus père libère ses ressources (i.e. les pipes), le fils libère à son tour les siennes puis se termine.

Validation

Dans le fichier **random.c** qui vous est fourni, tout comme dans l'exemple d'exécution ci-dessous, les valeurs de différentes constantes sont fixées pour faciliter la vérification des formules :

- RAND_MAX = 2.147.483.647 (stdlib.h)
- MAX_VAL = 1.000.000.000
- SEUIL = MAX_VAL/2 = 500.000.000

[2] Pour rappel, Ctrl-C envoie un signal SIGINT au processus qui a la main sur le terminal mais aussi à tous ses fils ! (cf. concept de *Process Group ID* - PGID)

- POURCENT = SEUIL / MAX_VAL = 50%

Dans une suite uniforme, `nInf/cnt` doit tendre vers SEUIL/MAX_VAL et ce, pour toute valeur de SEUIL. Par exemple, si MAX_VAL = 1000, 30% (POURCENT) des valeurs générées doivent être inférieures à 300 (SEUIL), 50% des valeurs doivent être inférieures à 500, 80% des valeurs doivent être inférieures à 800, etc.

Considérons les constantes définies dans *random.c*. Si la suite de valeurs aléatoires générées dans l'intervalle [0,999.999.999] est uniforme, le pourcentage de valeurs inférieures au seuil 500.000.000 doit tendre vers 500.000.000/1.000.000.000 = 50%.

Voici un exemple d'exécution du programme random :

```
studinfo@lab000001:LAS$ ./random
RAND_MAX = 2147483647
MAX_VAL  = 1000000000  ==> valeurs aléatoires entre 0 et 999999999
SEUIL = MAX_VAL/2 = 500000000  ==> valeurs inférieures au seuil: 50%

Pour une suite uniformément répartie dans l'intervalle [0,MAX_VAL[, le pourcentage de valeurs
aléatoires inférieures au seuil MAX_VAL/2 devrait correspondre à 50.00%.
Vérifions si c'est bien le cas.

^C
FILS2: 556921 valeurs reçues --> 49.9523% valeurs générées inférieures au seuil

FILS1: 555957 valeurs reçues --> 53.4281% valeurs générées inférieures au seuil
^C
FILS2: 4167879 valeurs reçues --> 49.9894% valeurs générées inférieures au seuil

FILS1: 4168009 valeurs reçues --> 53.4496% valeurs générées inférieures au seuil
^C
FILS1: 11031439 valeurs reçues --> 53.4314% valeurs générées inférieures au seuil

FILS2: 11031364 valeurs reçues --> 50.0012% valeurs générées inférieures au seuil
^ZFin du fils 2
Fin du fils 1
Fin du père
studinfo@lab000001:LAS$
```

[Alerte spoiler : la première formule (fils 1) ne respecte pas une répartition uniforme des valeurs aléatoires dans l'intervalle, contrairement à la seconde (fils 2) !]

Consignes

- Complétez le fichier **random.c** fourni.
- Ecrivez un fichier **Makefile**.
- Les opérations d'entrée/sortie doivent se faire via des appels système, à l'exception des messages à destination de l'utilisateur qui peuvent faire appel à *printf*.
- Tous les appels système doivent être testés. En cas d'erreur, l'exécution du programme doit être interrompue.
- Evitez que les signaux SIGINT émis par l'utilisateur provoquent un arrêt abrupte de votre programme. Dans le cas de l'interruption d'une lecture sur un pipe, la perte d'un nombre aléatoire n'affectera pas les statistiques. Vous pouvez dès lors continuer l'exécution et passer directement à la lecture de la valeur suivante (pour ce faire, n'utilisez pas `sread()` du module utils mais bien le syscall `read()`).

Bonus : 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