I2180 : Exercices de Programmation Système (5)

5.1. Appels système : `pipe & sigaction`

Reprenez la solution de l'exercice 3.1 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. 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.

5.2. Appel système : `sigaction & alarm`

Soient X et R deux entiers naturels, une suite arithmétique est une suite infinie de la forme :

<X, X + R, X + 2*R, X + 3*R, …, X + n*R, …>

Par exemple, si X = 4 et R = 2, la suite <4, 6, 8, 10, 12, …> est une suite arithmétique.

Écrivez un programme qui calcule les valeurs successives d'une suite arithmétique. Ce programme crée un fils qui reçoit les deux valeurs X et R sur la ligne de commande et calcule les valeurs successives de la suite.

On souhaite que ce programme réagisse à différents signaux clavier :

- Lorsque **Ctrl-\ (SIGQUIT)** est utilisé, le père ordonne à son fils d'afficher l'état du calcul, c'est-à-dire la valeur de n et celle du dernier élément « X + n * R » de la suite calculé.
- Lorsque **Ctrl-Z (SIGTSTP)** est entré, le processus père endort son fils s'il est en exécution et le redémarre s'il est endormi.
- Le **Ctrl-C (SIGINT)** est utilisé pour terminer le programme. Faites en sorte qu'entrer une seule fois Ctrl-C n'ait aucun effet mais qu'entrer deux fois Ctrl-C en moins d'une seconde provoque la fin du programme (utilisez l'appel système *alarm*). De plus, assurez-vous que le processus père tue son fils avant de se terminer. Le fils doit quant à lui afficher une dernière fois la valeur de n et celle de l'élément « X + n * R ».

Chacun des deux processus doit définir un unique handler pour traiter les différents signaux qu'il reçoit.

Vous pouvez vérifier l'état du processus fils en entrant une de ces commandes dans un autre terminal :

ps -a
ps -A | grep *nom_programme*

Conseil : Afin d'éviter que les calculs soient trop rapides, introduisez une pause de 1/20 seconde dans la boucle grâce à l'appel système *nanosleep* (en incluant time.h) :

```
struct timespec t;
t.tv_sec = 0;
t.tv_nsec = 50000000;
nanosleep(&t,NULL);    // veille de 1/20 seconde
```

5.3. Appels système : `sigprocmask & alarm`

Ecrivez un programme qui synchronise deux processus à l'aide du signal SIGUSR1. Le processus fils attend la réception du signal SIGUSR1 envoyé par son père. À la réception du signal, il envoie SIGUSR1 à son père et retourne en attente du signal. Le processus père effectue une boucle infinie où il envoie SIGUSR1 à son fils puis se met en attente du même signal.

Faites en sorte que le processus père se termine après 5 secondes. Avant de se terminer, il affiche le nombre de signaux reçus et envoyés, puis envoie un signal SIGUSR2 à son fils pour qu'il se termine à son tour en affichant le nombre de signaux reçus et envoyés.

Pour que les deux processus se mettent en attente d'un signal, utilisez les instructions suivantes au lieu de l'appel système *pause* :

```
sigset_t empty;
int res = sigemptyset(&empty);
checkNeg(res,"sigemptyset error");
...
sigsuspend(&empty);
```

Afin de vérifier que tout se passe correctement, affichez le nombre de messages reçus par le fils dans son handler de SIGUSR2. Au bout de 5 secondes, les processus père et fils devraient avoir échangé plusieurs centaines de milliers de signaux !

Si vous avez bien compris le fonctionnement des signaux, expliquez pourquoi il faut absolument utiliser `sigprocmask` dans ce programme ?

5.4. Appels système : `sigprocmask & sigsuspend`

Ecrivez un programme qui va faire dialoguer un père et son fils à l'aide de signaux.

Pour cela, on va concevoir une commande :

```
./sequence k s1 s2 ... sn
```

où `si` est un entier correspondant à un signal et `k` indique le nombre de fois que la séquence de signaux devra être émise.

Le programme *sequence.c* doit effectuer les actions suivantes :

1. Le processus fils va émettre vers son père `k` fois la séquence de signaux donnée en paramètre.
2. Le père affiche les signaux qu'il reçoit en les numérotant.

   Testez d'abord cette commande en envoyant d'abord une seule série de signaux (k=1). Assurez-vous que le père a bien armé les signaux fournis en paramètre avant leur réception.

   Testez ensuite la commande en envoyant une série de signaux (k=1000 par exemple).

3. Afin de ne pas perdre de signaux, on va mettre en œuvre le protocole suivant : le père devra envoyer au fils le signal SIGUSR1 pour acquitter chaque réception, de son côté le fils devra attendre l'acquittement du père pour poursuivre l'émission. À la fin du dialogue le fils utilisera le signal SIGKILL pour tuer son père.

   Implémentez ce protocole d'acquittement en :

   a) modifiant le traitant du père pour y envoyer l'acquittement,

   b) définissant dans le fils un traitement pour SIGUSR1,

   c) utilisant `pause()` pour synchroniser le rythme d'émission du fils à la vitesse de traitement du père.

   Testez puis montrez à force d'expériences que l'appel à `pause()` n'est pas adéquat. Pourquoi ?

4. [BONUS] Proposez une solution au problème précédent en utilisant les appels systèmes `sigsuspend()` et `sigprocmask()`.