Signaux

Exercice 4.1 Afficher la liste numérotée des signaux à l'aide de la fonction `strsignal()`.

Exercice 4.2 Écrire un programme qui boucle tout en affichant `ctrl-c` lorsque le processus qui l'exécute reçoit le signal `SIGINT`. Dans un second temps on faira en sorte que seul le premier signal `SIGINT` soit traité.

Exercice 4.3 Écrire une commande `recevoir-signaux` qui affiche son pid puis boucle tout en affichant en les numérotant (dénombrant) les signaux que son processus reçoit. Tester cette commande en utilisant la commande unix `kill`.

Exercice 4.4 Écrire une commande `emettre-signaux pid-cible k s1 s2 ... sn` qui émet à destination du processus cible `k` fois la séquence de signaux donnée en paramètre. Utiliser cette commande pour «bombarder» de signaux un processus exécutant la commande de l'exercice précédent.

Exercice 4.5 Il s'agit maintenant de faire dialoguer un père et son fils à l'aide de signaux. Pour cela on va écrire une commande `signaux-pere-fils k s1 s2 ... sn` où le fils va émettre vers son père `k` fois la séquence de signaux donnée en paramètre. On utilisera le signal `KILL` pour terminer le dialogue.
Afin de ne pas perdre de signaux on va mettre en oeuvre le protocole suivant : le père devra envoyer au fils le signal `USR1` pour acquitter chaque réception, de son coté le fils devra attendre l'acquittement du père pour poursuivre l'émission.

1. Reprendre le code de deux exercices précédents pour commencer ;
2. Faire en sorte que le signal `KILL` soit pris en compte ;
3. Implémenter le protocole d'acquittement en utilisant l'appel système `pause()` pour ralentir le fils ;
4. expliquer pourquoi l'appel à `pause()` n'est pas adéquat ;
5. proposer une solution utilisant l'appel système `sigsuspend()`.

Exercice 4.6 Le programme `suivre-commandes.c` permet de suivre périodiquement l'évolution d'un ensemble bien précis de commandes en affichant toutes les secondes l'état (non démarré, en cours d'exécution, stoppé, terminé) de chaque élément de l'ensemble de commandes. Par exemple si on execute 3 fois la commande sleep avec les arguments 1 2 3 on aura :

```
0s
en cours sleep 1
en cours sleep 2
en cours sleep 3

1s
terminé sleep 1
en cours sleep 2
en cours sleep 3

2s
terminé sleep 1
```

```
terminé sleep 2
en cours sleep 3

Tous les processus ont terminés !
```

1. Modifier le code du programme `suivre-commandes.c` afin de suivre périodiquement l'évolution des commandes via le traitement du signal SIGALRM ; on appelera la fonction `timer()` pour générer le signal périodique.
2. Rendre le `wait` bloquant puis observer la valeur de retour de `wait` et le contenu de la variable `errno` via la fonction `perror()`. Faire en sorte que l'appel système soit repris automatiquement après le traitement de l'alarme.
3. Proposer une solution basée sur le traitement du signal `SIGCHLD`.

Exercice 4.7 Soit le programme suivant :

```
int main(){
  int i ;
  for (i = 0; i < 10; i++)
    printf("%d\n",i)
  return 0;
}
```

1. Remplacer la boucle `for` du programme précédent par un code utilisant le mécanisme de sauts non locaux `setjmp()/longjmp()`.
2. Tester le programme en compilant le programme avec et sans optimisation (`-O3`).
3. Utiliser le mot clé `volatile` pour qualifier la variable `i`. Tester le programme.
4. Placer l'appel à `longjmp()` dans une fonction `f()`. Tester.
5. Placer l'appel à `setjmp()` dans une fonction `g()`. Tester.

Exercice 4.8 On désire pouvoir limiter, grâce à une alarme, la durée maximale d'évaluation d'une fonction paramétrée. Pour cela on défini la fonction suivante :

```
int execute_avant_delai( void (*fun)(void *), void *parametre, int delai_en_seconde);
```

Cette fonction retourne 0 si l'évaluation de `fun(parametre)` a été interrompue par l'alarme, 1 autrement (le delai 0 ne sera pas pris en compte). En utilisant le timer de la fonction `alarm()`, écrire le fichier `execute-avant-delai.c`.
Tester votre solution en utilisant le programme `oui_ou_non`.

Que se passe-t-il si l'on exécute la fonction `execute_avant_delai()` deux fois de suite dans la fonction `main()` ? Pour corriger le problème, utilisez les fonctions `siglongjmp()/sigsetjmp()`.

Exercice 4.9 Par défaut, un processus est arrêté par le système d'exploitation lorsqu'il provoque une erreur matérielle (accès mémoire erroné, tentative d'exécuter une instruction illégale, etc.) Néanmoins, il est parfois possible de laisser le processus continuer son exécution, lorsque l'application est capable de récupérer l'erreur correctement. Voici par exemple le source du `trap.c`

```
#include <signal.h>
#include <stdio.h>

volatile char *a=NULL, b='x';

void traitant(int s)
```

```
{
  printf("signal %d\n", s);
  a = &b;
}

int main()
{
  struct sigaction s;
  char x;
  s.sa_handler = traitant;
  sigemptyset(&s.sa_mask);
  s.sa_flags=0;
  sigaction(SIGSEGV,&s,NULL);
  x = *a;
  printf("fin %c\n",x);
  return 0;
}
```

Question 1 Quel est, selon votre intuition le comportement du programme `trap.c` ? Exécuter le programme et expliquer son comportement.
Nous nous proposons de fournir un outil permettant à un programme de tenter l'exécution d'une fonction à l'aide d'une fonction `essayer` qui en renvoie 0 si l'exécution s'est déroulée normalement ou -1 si l'exécution a provoqué la délivrance d'un signal donné (par exemple `SIGSEGV`).
Voici typiquement l'utilisation que l'on souhaite faire de la fonction `essayer` :

```
#include "safe_exec.h"

void f()
{
  ... // code à risque
}

int main()
{
  int r;
  r = essayer(f, SIGSEGV);
  if(r == 0)
    printf("L'exécution de f s'est déroulée sans problème\n");
  else
    printf("L'exécution de f a échoué\n");
  ...
}
```

Question 2 Donnez le code du module `safe_exec.c`, et en particulier le corps de la fonction `essayer`. On ne cherchera pas à rattraper d'autres signaux que celui passé en paramètre...

Question 3 Que va-t-il se passer si, dans l'exemple précédent, la fonction `f` appelle à son tour `essayer(g)` par exemple ? Expliquez précisément pourquoi. Indiquez comment il faudrait procéder pour corriger ce problème.