Programmation Système

Fiche 4 : Signaux

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/progsys/

Les signaux unix sont décrits en section 7 du manuel unix (`man 7 signal`). Le programme `lister-signaux.c` affiche la liste numérotée des signaux à l'aide de la fonction `strsignal`.

Exercice 1.1 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 fera en sorte que seul le premier signal `SIGINT` soit traité.

Exercice 1.2 L'objectif de cet exercice est 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. Il s'agit de compléter le programme `signaux-pere-fils.c` pour :

1. Faire en sorte que le père affiche en les numérotant (dénombrant) les signaux qu'il reçoit. Tester cette commande en envoyant d'abord une seule série de signaux ($k = 1$).
2. Tester votre commande en envoyant une série de signaux ($k = 1000$ par exemple). Commenter.
3. Décommenter l'appel à `sleep (1)`. Pourquoi faut-il : (a) faire les appels à `sigaction` avant l'appel à `fork` ou bien (b) masquer les signaux dans le père avant l'appel à `fork` puis les démasquer après les appels à `sigaction` ?

Nous reprendons cet exercice un peu plus loin.

Exercice 1.3 L'objectif de cet exercice est de réaliser le traitement des processus zombies (et plus généralement le traitement des jobs) tel que le fait un shell. À cet effet nous allons modifier le programme `suivre-commandes.c` qui lance quelques processus puis exécute une boucle qui cherche à récupérer un zombie, attend une entrée sur le clavier, affiche l'état des processus de la fonction `afficher_etat`. Voici sa boucle d'interaction simplifiée :

```
while(reste_commande())
 {
   w = waitpid(0, NULL , WNOHANG);
   if(w>0)
     modifier_etat(w);
   read(0,buf,1024); // se bloquer pour simuler la lecture dans le shell
   afficher_etat();  // afficher la connaissance de l'application
 }
```

Lorsque l'on exécute le programme tel qu'il est donné on remarque que les durées d'exécution mesurées ne correspondent pas aux durées des commandes, comme le montre l'extrait de la sortie suivant :

```
...
# l'utilisateur a tapé sur la touche enter
# au bout d'environ  3, 7, 8 et 9 secondes.

20798 : sleep(0) terminé durée détectée : 9.22469s
20799 : sleep(3) terminé durée détectée : 3.41752s
20800 : sleep(4) terminé durée détectée : 8.56692s
20801 : sleep(5) terminé durée détectée : 7.54433s

# le processus exécutant sleep(0) est resté à
# l'état de zombie durant plus de 9s.
```

Notre objectif est faire en sorte que le processus père acquitte la mort de ses processus fils *au plus tôt* via le traitement du signal `SIGCHLD` qui est systématiquement envoyé au père à la mort d'un de ses fils. Nous allons voir que ce traitrement pourra même intervenir durant l'appel à `read`.

1. Créer une fonction pour traiter le signal `SIGCHLD` et y déplacer les appels à `wait` et `modifier_etat`. Définir le traitement associé au signal `SIGCHILD`. Tester votre programme.
2. Faire en sorte que l'appel système `read` soit repris automatiquement après le traitement du signal `SIGCHLD`. Tester.
3. Vérifier que votre solution marche lorsque tous les fils se terminent à peu près en même temps en lançant 10 commandes `sleep 1` par exemple. Corriger votre programme si nécessaire.
4. Faire en sorte que les processus fils terminent immédiatement (via `exit (0)` par exemple). Analyser le comportement du programme. Pourquoi est-on amené à masquer le signal `SIGCHLD` lors de la création des processus fils ? Faut-il le démasquer dans les fils ?

Exercice 1.4 Ceci est la suite de l'exercice *1.2*.

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. À la fin du dialogue le fils utilisera le signal `SIGKILL` pour tuer son père.

3. Implémenter le protocole d'acquittement en utilisant l'appel système en (1) modifiant le traitant du père pour y envoyer l'acquittemment, (2) définissant un traitant pour USR1 dans le fils et en (3) utilisant `pause` pour synchroniser le rythme d'émission du fils à la vitesse de traitement du père.
4. Tester puis montrer à force d'expériences que l'appel à `pause` n'est pas adéquat. Expliquer.
5. Proposer une solution utilisant les appels systèmes `sigsuspend` et `sigprocmask`.

Exercice 1.5

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`

```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;
}
```

Intuitivement, comment devrait se comporter le programme `trap.c` ? Exécuter le programme. Expliquer son comportement en imaginant le code machine exécuté.

Les langages modernes disposent de mécanismes pour rattraper les erreurs exceptionnelles (erreurs matélrielle, arithmétique, de segment comme dans l'exemple ci-dessus,...). Nous allons voir comment en C on peut implémenter un tel mécanisme en associant deux techniques : l'utilisation de sauts non locaux et le traitement des signaux. Tout d'abord nous allons mettre en œuvre les sauts non locaux sur un exemple trivial.

Exercice 1.6 Soit le programme suivant :

```c
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 1.7 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 (fun, p, signal)` qui en renvoie 0 si et seulement si l'exécution `fun (p)` s'est déroulée sans provoquer/subir la délivrance du signal donné en paramètre. Voici typiquement l'utilisation que l'on souhaite faire de la fonction `essayer` :

```c
#include "essayer.h"

void f(void *p)
{
  ... // code à risque
}

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

1. Donnez le code du module `essayer.c`, et en particulier le corps de la fonction `essayer`.
2. Tester cette fonction sur le programme `exemple-essayer.c`.

   Nous allons maintenant utiliser la fonction `essayer` pour limiter la durée maximale d'évaluation d'une fonction paramétrée. Pour cela on défini la fonction suivante :

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

   Cette fonction retourne 0 si et seulement si l'évaluation de `fun(parametre)` n'a pas été interrompue par une alarme.
3. En utilisant la fonction `essayer` et la fonction `alarm`, compléter le fichier `execute-avant-delai.c`.
4. Tester votre solution en utilisant le programme `oui_ou_non`.
5. 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` et `sigsetjmp` dans le module `essayer.c`.
6. De façon générale, que va-t-il se passer si l'on imbrique les appels à la fonction `essayer`, comme dans la fonction `tutu` du programme `exemple-essayer.c` ? Indiquez comment il faudrait procéder pour corriger ce problème.