Labo 5

Objectif : les appels système liés à l'envoi de signaux

Introduction[1]

Dans le labo 3, afin de comprendre les commandes liées à la gestion des processus, la notion de flèche a été rappelée.

En effet, les flèches ou *interruptions logicielles* sont le mécanisme qui est utilisé par le système d'exploitation pour aviser les processus utilisateurs de l'occurrence d'un événement important.

De nombreuses erreurs détectées par le matériel comme l'exécution d'une instruction non autorisée, par exemple une division par 0, ou l'emploi d'une adresse non valide, sont converties en flèches qui sont envoyées au processus fautif. Un mécanisme permet à un processus de réagir à cet événement sans être obligé d'en tester en permanence l'arrivée, pour ce faire, on lui associe un gestionnaire de flèche…

Les processus peuvent indiquer au système ce qui doit se passer à la réception d'une flèche : l'ignorer, la prendre en compte, ou encore la laisser tuer le processus. Certaines flèches ne peuvent être ni ignorées, ni capturées. Si un processus choisit de prendre en compte les flèches qu'il reçoit, il doit alors spécifier la procédure de gestion de flèche ou *handler*. Quand une flèche arrive, la procédure associée est exécutée. À la fin de l'exécution de la procédure, le processus s'exécute à partir de l'instruction qui suit celle durant laquelle l'interruption a eu lieu.

En résumé, les flèches sont utilisées pour établir une communication minimale entre processus, une communication avec le monde extérieur et permettre la gestion des erreurs. Ainsi, un processus peut :

- ignorer la flèche.
- appeler une routine de traitement fournie par le noyau.
- appeler une procédure spécifique créée par le programmeur. Tous les signaux ne permettent pas ce type d'action.

Lever son bouclier (ou se protéger d'une flèche)

L'appel système permettant la gestion des flèches est `signal`.

signal	
Librairie	#include <signal.h>
Prototype	`void (*signal(int signum, void (*handler)(int)))(int);`
Commentaires	Installe un nouveau gestionnaire `handler` pour la flèche numéro `signum`. Le gestionnaire de flèche peut être soit une fonction spécifique de l'utilisateur, soit l'une des constantes suivantes : • `SIG_IGN` ignorer la flèche • `SIG_DFL` reprendre le comportement par défaut pour la flèche L'argument entier qui est passé au gestionnaire de flèche est le numéro de la flèche. Ceci permet d'utiliser un même gestionnaire pour plusieurs flèches. Renvoie la valeur précédente du gestionnaire de signaux, ou `SIG_ERR` en cas d'erreur.

[1] Inspiré de "*INF3600, Systèmes d'exploitation, de Hanifa Boucheneb et Juan Manuel Torres-Moreno, au Département de génie informatique de l'École Polytechnique de Montréal, version 3.90*"

Exemple

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

void traitement(int signum) {
   printf ("On a recu une fleche SIGTERM\n");
   fflush(stdout);
}

int main(void) {
   char buffer [256];
   if (signal(SIGTERM, &traitement) == SIG_ERR) {
      printf("Ne peut pas manipuler le signal\n");
      exit(EXIT_FAILURE);
   }
   while (1) {
      printf("Tapez votre message : ");
      fgets(buffer ,sizeof(buffer), stdin);
      printf("Message : %s", buffer);
   }
   exit(EXIT_SUCCESS);
}
```

Pour tester cet exemple, il convient de le lancer, de rentrer quelques messages puis de le mettre en arrière-plan `[CTRL+Z]`, de faire un `kill` (après avoir obtenu son `pid`) et de le remettre en avant plan `fg`. On remarque que lorsqu'on envoie une flèche SIGTERM avec `[CTRL+Z]`, la flèche est interceptée par `signal` et le gestionnaire `traitement` est appelé au lieu du gestionnaire fourni par le noyau. Pour tuer le processus, faites [CTRL+C] …

En voici une illustration, en Figure 1.

```
piroc@svr24:~$ ./ex
Tapez votre message : Hello
Message : Hello
Tapez votre message : world
Message : world
Tapez votre message : ^Z
[1]+  Stoppé                 ./ex
piroc@svr24:~$ ps
  PID TTY          TIME CMD
28798 pts/2    00:00:00 bash
29087 pts/2    00:00:00 ex
29088 pts/2    00:00:00 ps
piroc@svr24:~$ kill 29087
piroc@svr24:~$ fg
./ex
On a recu une fleche SIGTERM
La question est avant le [ctrl+Z]
Message : La question est avant le [ctrl+Z]
Tapez votre message : ^C
```

Figure 1 – Exemple de gestionnaire de flèche

Envoyer une flèche

L'envoi d'une flèche peut également se faire avec l'appel système `kill`.

kill	
Librairies	#include <sys/types.h> #include <signal.h>
Prototype	`int kill(pid_t pid, int signal);`
Commentaires	Envoi la flèche numéro `signal` à un processus identifié par son `pid`. En cas d'erreur il retourne -1, et 0 autrement. Selon la valeur de `pid`, les cibles seront les suivantes : • > 0 le processus `pid` • = 0 les processus faisant partie du groupe de `pid` • = -1 tous les processus (seul `root` peut le faire) • < 0 le groupe `gid` = |`pid`|

Exemple

```
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <stdio.h>

void traitement (int signum) {
   printf("Signal %d, d'ou ", signum);
   switch (signum) {
      case SIGTSTP :  printf("le processus s'endort...\n");
               kill(getpid(), SIGSTOP);
               /* auto-endormissement */
               signal(SIGTSTP, traitement);
               /* repositionnement */
               printf("le processus est reveillé !\n");
               break;
      case SIGINT : case SIGTERM :
               printf("Fin du programme.\n");
               exit(EXIT_SUCCESS);
               break;
   }
}

int main(void) {
   signal(SIGTSTP, traitement);      /* si on reçoit contrôle-Z */
   signal(SIGINT, traitement);       /* si contrôle-C */
   signal(SIGTERM, traitement);      /* si kill processus */

   while (1) {
      sleep(1);
      printf(".");
      fflush(stdout);
   }

   printf("Fin");
   exit(EXIT_SUCCESS);
}
```

Pour le tester, procéder comme pour l'exemple précédent : lancer le programme, tapez CTRL-Z, fg puis CTRL-C.

Autres fonctions utiles…

alarm	
Librairie	#include < unistd.h>
Prototype	`long alarm(long delai);`
Commentaires	Envoi la flèche `SIGALRM` au processus dans un `delai` fixé en secondes Si une alarme était déjà positionnée, elle est remplacée. Un délai nul supprime l'alarme existante et renvoie le nombre de secondes restantes.

pause	
Librairie	#include < unistd.h>
Prototype	`int pause (void);`
Commentaires	Bloque le processus courant jusqu'à ce qu'il reçoive un signal.

Exemple complet

Énoncé

Écrire un programme où un processus père lance 2 fils ayant la même image que lui.
Chaque processus commence par afficher un message de début et se termine par un message de fin.

Après que chaque processus ait affiché son message de début selon l'ordre père – fils1 – fils2, les fils affichent un message personnel dix fois chacun. Les messages envoyés par les fils doivent apparaître en alternance (fils1, fils2, fils1...). Le premier message doit être envoyé par le fils2.

Aucun processus ne peut se terminer avant que les 2 fils n'aient envoyé leur dernier message. Les messages de fin respectent l'ordre fils2 – fils1 – père.

Utilisez SIGUSR1 pour la gestion des flèches.

Le résultat attendu est présenté à la Figure 2.

```
piroc@svr24:~$ ./ex
PID 5150 - Debut du pere
PID 5151 - Debut du fils 1 [PPID - 5150]
PID 5152 - Debut du fils 2 [PPID - 5150]
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Message du fils 2
PID 5151 - Message du fils 1
PID 5152 - Fin du fils 2
PID 5151 - Fin du fils 1
PID 5150 - Fin du pere
```

Figure 2 - Résultat attendu

Une solution…

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

// déclaration des fonctions
// 1) gestion des processus
void pere(void);
void fils1(void);
void fils2(void);

// 2) gestion des signaux
void killReceiver(int signum);
void extendedPause(void);

//-------------------------------------------------------------------------------------------
// déclaration : variables globales
// données utilisables dans tous les processus créés pour ne pas devoir les transporter en
// paramètre de fonction.
//-------------------------------------------------------------------------------------------

// variable utilisée par plusieurs fonctions de manière asynchrone il ne faudra cependant
// pas oublier d'initialiser celle-ci dans chaque processus
int killReceived;

pid_t pidP, pidF1, pidF2;

//-------------------------------------------------------------------------------------------
// 1) Gestion des processus
//-------------------------------------------------------------------------------------------
int main (int ac, char **av) {
   // par réflexe et sécurité pour la gestion des kill/pause
   killReceived = 0;
   signal(SIGUSR1, killReceiver);

   pidP = getpid();
   printf("PID %d - Debut du pere\n", pidP);
   fflush(stdout);

   pere();
}

void pere(void) {
   int i, status;
   pid_t finishedPid;

   //-------------------------------------------------------------------------------------
   // phase de création des processus
   //-------------------------------------------------------------------------------------
   // création du fils 1
   if ((pidF1 = fork()) < 0) {
      perror("erreur creation du fils 1 !");
      exit(EXIT_FAILURE);
   }
   if (pidF1 == 0) {
      fils1();
   }

   // attendre que la création et l'envoi du message de début du fils 1
   extendedPause();
```

```
   // création du fils 2
   if ((pidF2 = fork()) < 0) {
      perror("erreur creation du fils 4!");
      exit(EXIT_FAILURE);
   }
   if (pidF2 == 0) {
      fils2();
   }
   // attendre que la création et l'envoi du message de début du fils 2
   extendedPause();

   //-----------------------------------------------------------------------------------------
   // partie principale
   //-----------------------------------------------------------------------------------------
   for(i = 1; i <= 10; i++) {
      kill(pidF2, SIGUSR1);
      extendedPause();
   }

   //-----------------------------------------------------------------------------------------
   // partie terminaison des processus
   //-----------------------------------------------------------------------------------------
   // on demande au fils 2 d'afficher son message de fin et de se terminer
   kill(pidF2, SIGUSR1);
   // on attend la fin du fils 2
   finishedPid = wait(&status);
   if (finishedPid != pidF2) {
      printf("===== OUPS ===== %d est mort alors qu'on "
            "attendait la mort de %d\n", finishedPid, pidF2);
   }
   if (status != 0)   {
      printf("===== OUPS ===== %d est mort par accident "
            "(status=%04x)\n", finishedPid, status);
   }
   // on demande au fils 1 d'afficher son message de fin et de se terminer
   kill(pidF1, SIGUSR1);
   // on attend la fin du fils 1
   finishedPid = wait(&status);
   if (finishedPid != pidF1) {
      printf("===== OUPS ===== %d est mort alors qu'on "
            "attendait la mort de %d\n", finishedPid, pidF1);
   }
   if (status != 0) {
      printf("===== OUPS ===== %d est mort par accident "
            "(status=%04x)\n", finishedPid, status);
   }

   // et on termine
   printf("PID %d - Fin du pere\n", pidP);
   exit(EXIT_SUCCESS);
}
```

```
void fils1(void) {
   int i;

   // par reflexe et sécurité pour la gestion des kill/pause
   killReceived = 0;
   signal(SIGUSR1, killReceiver);

   pidF1 = getpid();
   //-------------------------------------------------------------------------------------
   // phase de création des processus
   //-------------------------------------------------------------------------------------
   printf("PID %d - Debut du fils 1 [PPID - %d]\n", pidF1, pidP);
   fflush(stdout);
   // prévient le père que le message de début a été affiché
   kill(pidP, SIGUSR1);
   //-------------------------------------------------------------------------------------
   // partie principale
   //-------------------------------------------------------------------------------------
   for(i = 1; i <= 10; i++) {
      extendedPause();
      printf("PID %d - Message du fils 1\n", pidF1);
      fflush(stdout);
      kill(pidP, SIGUSR1);
   }
   //-------------------------------------------------------------------------------------
   // partie terminaison des processus
   //-------------------------------------------------------------------------------------
   // on attend que le père confirme la fin du fils 2
   extendedPause();

   printf("PID %d - Fin du fils 1\n", pidF1);
   exit(EXIT_SUCCESS);
}

void fils2(void) {
   int i;
   // par reflexe et sécurité pour la gestion des kill/pause
   killReceived = 0;
   signal(SIGUSR1, killReceiver);

   pidF2 = getpid();
   //-------------------------------------------------------------------------------------
   // phase de création des processus
   //-------------------------------------------------------------------------------------
   printf("PID %d - Debut du fils 2 [PPID - %d]\n", pidF2, pidP);
   fflush(stdout) ;
   // signale au père que le message de début a été affiché
   kill(pidP, SIGUSR1);
   //-------------------------------------------------------------------------------------
   // partie principale
   //-------------------------------------------------------------------------------------
   for(i = 1; i <= 10; i++) {
      extendedPause();
      printf("PID %d - Message du fils 2\n", pidF2);
      fflush(stdout);
      kill(pidF1, SIGUSR1);
   }
   //-------------------------------------------------------------------------------------
   // partie terminaison des processus
   //-------------------------------------------------------------------------------------
   printf("PID %d - Fin du fils 2\n", pidF2);
   exit(EXIT_SUCCESS);
}
```

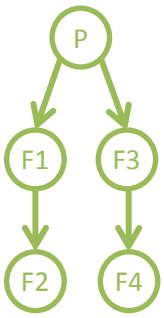
```
//-----------------------------------------------------------------------------------------
// 2) gestion des signaux
//-----------------------------------------------------------------------------------------
void killReceiver(int signum) {
   killReceived++;
}

void extendedPause(void) {
   if (killReceived == 0) {
      pause();
   }
   killReceived--;
}
```

À vous de jouer !

Utilisez les appels `fork`, `getpid`, `signal`, `kill`, `pause`…

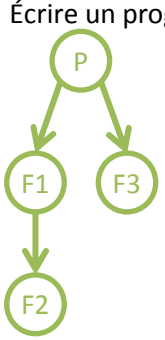
1. Écrire un programme où un processus père lance deux fils qui ont chacun un fils.



Chaque processus commence par afficher un message de début et se termine par un message de fin. Le père attend la fin de l'exécution de tous les fils avant d'afficher son message de fin. Les messages de début respectent l'ordre père - fils1 – fils2 – fils 3 – fils4, et les messages de fin s'affichent dans l'ordre inverse.

Les fils affichent un message personnel dix fois chacun. Les messages envoyés par les fils doivent apparaître en alternance dans l'ordre suivant : fils4, fils3, fils2, fils1.

2. Écrire un programme où un processus père lance deux fils. Le fils 1 a également un fils.



Chaque processus commence par afficher un message de début et se termine par un message de fin.

Les fils affichent un message personnel dix fois chacun. Les messages envoyés par les fils doivent apparaître en alternance dans l'ordre suivant : fils3, fils2, fils1.

Le père attend la fin de l'exécution de tous les fils avant d'afficher son message de fin.

3. Écrire un programme où un processus père lance X fils, X ∈ [1, 10], ayant la même image que lui. Chaque processus commence par afficher un message de début et se termine par un message de fin.

Les fils affichent un message personnel dix fois chacun. Les messages envoyés par les fils doivent apparaître en alternance (fils1, fils2… filsX, fils1...). Le premier message doit être envoyé par le fils1.

Le père attend la fin de l'exécution de ses deux fils avant d'afficher son message de fin.