UNIVERSITÉ CAEN NORMANDIE

Les signaux

31.10.2018

Auteur : Pascal Fougeray

```
Standard signals
Linux supports the standard signals listed below. Several signal numbers are architecture-dependent, as
indicated in the "Value" column. Where
three values are given, the first one is usually valid for alpha and sparc, the middle one for x86, arm
, and most other architectures, and the last
one for mips. (Values for parisc are not shown; see the Linux kernel source for signal numbering on tha
t architecture.) A dash (-) denotes that a
signal is absent on the corresponding architecture.

First the signals described in the original POSIX.1-1990 standard.

Signal Value Action Comment
──────────────────────────────────────────────────────────────────────
SIGHUP 1 Term Hangup detected on controlling terminal
or death of controlling process
SIGINT 2 Term Interrupt from keyboard
SIGQUIT 3 Core Quit from keyboard
SIGILL 4 Core Illegal Instruction
SIGABRT 6 Core Abort signal from abort(3)
SIGFPE 8 Core Floating-point exception
SIGKILL 9 Term Kill signal
SIGSEGV 11 Core Invalid memory reference
SIGPIPE 13 Term Broken pipe: write to pipe with no
readers; see pipe(7)
SIGALRM 14 Term Timer signal from alarm(2)
SIGTERM 15 Term Termination signal
SIGUSR1 30,10,16 Term User-defined signal 1
SIGUSR2 31,12,17 Term User-defined signal 2
SIGCHLD 20,17,18 Ign Child stopped or terminated
SIGCONT 19,18,25 Cont Continue if stopped
SIGSTOP 17,19,23 Stop Stop process
SIGTSTP 18,20,24 Stop Stop typed at terminal
SIGTTIN 21,21,26 Stop Terminal input for background process
SIGTTOU 22,22,27 Stop Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

1 Introduction

Dans un système, quand un processus est bloqué pour une raison ou une autre, l'utilisateur peut utiliser la commande ***kill -9 pid_de_processus*** afin de le tuer[1].

- Si un processus fait une violation de mémoire, le système ne se gène pas de faire de même sans demander l'avis de l'utilisateur. Le processus est éliminé sans aucun ménagement après avoir reçu le signal spécifique ***SIGSEGV***.
- Il en va de même lorsqu'un processus demande quelque chose d'impossible à faire au microprocesseur telle une division par zéro. Le processus reçoit alors un autre signal spécifique ***SIGFPE***.
- Lorsqu'un processus fils se termine donc devient zombi, il envoie à son père le signal ***SIGCHLD*** pour l'avertir et son père va alors demander au noyau de le retirer de la table des processus.

Nous allons voir dans ce chapitre comment on peut gérer ces signaux afin de développer des applications mieux gérées et qui se terminent de manière plus "soft".

1. Ce n'est pas une manière élégante de procéder, mais c'est la plus utilisée :-(

2 Rappel de C ☺

Comme nous allons souvent utiliser les structures, dont la manipulation n'est pas très intuitives, je commence par un petit rappel en C sur la définition et la manipulation des structures.

Une **structure** est une construction **hétérogène** d'objets (ici le terme objet n'a rien à voir avec la POO!!!, on peut parler d'éléments) regroupés séquentiellement, séquentiellement donc les uns derrières les autres!!!

Exemple :

```
1  struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)
   (int, siginfo_t *, void *);
   sigset_t sa_mask;
6  int sa_flags;
   void (*sa_restorer)(void);
   }
```

Cette construction est utilisable comme un nouveau type et l'on peut déclarer une variable de nom ***action*** et de type ***sigaction*** en écrivant la ligne de code suivante :

struct sigaction action;

Pour accéder à un champ de cette structure on saisie le nom de la variable de type structure, l'opérateur (point) **.** suivi du nom du champ :

action.sa_handler=ouille;

ouille sera donc le nom d'une fonction.

— **Différence entre une structure et un tableau**
Un tableau permet de regrouper des objets (éléments) de même type, c'est-à-dire codés sur le même nombre de bits et de la même façon. Toutefois, il est généralement utile de pouvoir rassembler des éléments de type différents tels que des **entiers** et des **chaines de caractères**.
Les structures permettent de remédier à cette lacune des tableaux, en regroupant des objets (des variables) au sein d'une entité repérée par un seul nom de variable.
Les objets contenus dans la structure sont appelés champs de la structure.

— **Déclaration d'une structure**
Lors de la déclaration de la structure, on indique les champs de la structure, c'est-à-dire le type et le nom des variables qui la composent par exemple :

```
1  struct Nom_Structure {
2      type_champ1 Nom_Champ1;
3      type_champ2 Nom_Champ2;
4      type_champ3 Nom_Champ3;
5      type_champ4 Nom_Champ4;
6      type_champ5 Nom_Champ5;
7  ... }; //  !!! La dernière accolade doit être suivie d'un point-virgule !!!
```

— **Les restrictions**
- Le nom des champs répond aux critères des noms de variable
- 2 champs ne peuvent avoir le même nom
- Les données peuvent être de n'importe quel type sauf le type de la structure dans laquelle elles se trouvent

Bonne structure

```
   struct MaStructure {
   int Age;
3  char Sexe;
   char Nom[12];
   float MoyenneUniversitaire;
   // en considérant que la structure AutreStructure est définie
   struct AutreStructure StructBis;
8  };
```

Mauvaise structure

```
struct MaStructure {
int Age;
char Age; // déjà utilisé !!!
// même nom que la structure courante !!!
struct MaStructure StructBis; };
```

— **La déclaration d'une structure**
Elle ne fait que donner l'allure de la structure, c'est-à-dire en quelque sorte une définition d'un type de variable complexe.
La déclaration ne réserve pas d'espace mémoire pour une variable structurée (variable de type structure), il faut donc définir une(ou plusieurs) variable(s) structurée(s) après avoir déclaré la structure...

— **Définition d'une variable structurée**
La définition d'une variable structurée est l'opération consistant à **créer une variable** ayant comme type celui d'une structure que l'on a précédemment déclarée, c'est-à-dire la nommer et lui réserver un emplacement en mémoire.
Elle se fait comme suit :

```
struct Nom_Structure Nom_Variable_Structuree;
```

— **Accès aux champs d'une variable structurée**
Chaque variable de type structure possède des champs repérés avec des noms uniques. **Mais le nom des champs ne suffit pas pour y accéder étant donné qu'ils n'ont de contexte qu'au sein de la variable structurée...**
Pour accéder aux champs d'une structure on utilise l'opérateur de champ (**un simple point**) placé entre le nom de la variable structurée que l'on a défini et le nom du champ :

```
Nom_Variable.Nom_Champ;

Toto.Age = 18;
Tata.Sexe = 'F';
```

— Pointer sur une structure
Voilà voilà c'était juste un petit rappel en langage C. Et un exemple simple que vous pouvez trouver sur ecampus.

```
//fichier structure-simple.c
#include <stdio.h>
#include <string.h>
typedef struct Structure Structure;

struct Structure {
        int a;
        char nom[100];
};

int main(){
        char uneChaine[]="Mme Dupond";

        // Déclaration d'une variable strucA de type Structure
        Structure strucA;
        // p un pointeur sur une variable structure ici strucA
        Structure *p;
        // le pointeur p contient l'adresse de la structure
        p=&strucA;

        // on remplit les champs de la structure
        // 2 solutions du passage des données
        // par variable ou par adresse !!!

        // passage par adresse
        // les notations p->a et (*p).a sont identiques !!!
        p->a=16; // a reçoit 16
        printf("valeur de a : %d\n", (*p).a);
        (*p).a=64; // a reçoit 64
        printf("valeur de a : %d\n", p->a);
        // passage par la variable
```

```
        strucA.a=33; // a reçoit 33
        printf("valeur de a : strucA.a = %d \n\t\t(*p).a = %d \n\t\t  p->a = %d \n"\
                        ,strucA.a, (*p).a, p->a);

        // strucA.nom="Mr Dupont"; IMPOSSIBLE car
        // Struc.nom est un tableau, donc une variable contenant
        // l'@ de l'emplacement
        // du 1er caractère de la chaine de caractère qu'il représente.
        // il faut donc utiliser la fonction strcpy
        strcpy(strucA.nom, "Mr Dupont");
        printf("Nom : %s %s %s\n",\
                        (*p).nom, p->nom, strucA.nom);
        strcpy(strucA.nom, uneChaine);
        printf("Nom : %s %s %s\n",\
                        (*p).nom, p->nom, strucA.nom);

        return  0;
}
```

résultat : rien de compliqué !

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/structure# structure-simple
valeur de a : 16
valeur de a : 64
valeur de a : strucA.a = 33
                (*p).a = 33
                  p->a = 33
Nom : Mr Dupont Mr Dupont Mr Dupont
Nom : Mme Dupond Mme Dupond Mme Dupond
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/structure#
```

3 Définition

Un signal est un **évènement asynchrone** généré par le système en réponse à certaines conditions et dont l'envoi à un processus **peut** déclencher une réaction de ce dernier.

On écrit **peut**, car cela n'est pas une obligation, le processus peut ignorer le signal qui lui est envoyé.

Ce **message** ne comporte aucune information propre, si ce n'est le nom du signal lui même qui est significatif de l'évènement rencontré.

Ce signal peut être envoyé :

— Soit par le noyau à un ou à un groupe de processus pour indiquer l'occurrence d'un évènement survenu au niveau du système,

— Soit par un autre processus, pour indiquer sa fin, exemple du fils au père : **SIGCHLD**, :

Un signal peut-être :

— **Pendant**, le signal est envoyé au processus, mais n'est pas encore pris en compte. Un seul exemplaire d'un même type de signal peut-être pendant. Donc, si un exemplaire d'un signal arrive à un processus alors qu'il en existe un exemplaire pendant, il est perdu.

— **Délivré**, lorsque le processus le prend en compte au cours de son exécution. La délivrance d'un signal à un processus a lieu lorsqu'il **passe de l'état actif noyau à l'état actif utilisateur**, le processus ne maitrise pas cette partie, c'est le noyau qui gère.

— **Pris en compte** par le processus entraine l'exécution d'une fonction particulière : l'un des éléments fournis par l'indicateur de comportement d'un type de signal est un pointeur sur le ***handler*** correspondant.

3 types d'actions peuvent alors être réalisés :

1. **Ignorer** le signal, **SIG_IGN**.
2. **Exécuter** l'action par défaut, **SIG_DFL**.
3. **Exécuter** une fonction spécifique installée par le programmeur, cette fonction est nommée ***handler***.

— **Bloqué** ou **masqué** lorsque l'on diffère volontairement sa délivrance.

Un ***handler*** ou **gestionnaire de signal** : c'est une fonction (écrite en C) attachée au code de l'utilisateur et qui sera exécutée lors de la prise en compte du signal.

Remarque : Le mode de fonctionnement des signaux présente quelques analogies avec le traitement des interruptions matérielles, et **on dit souvent que les signaux sont des interruptions logicielles.**

Sous Linux, il existe 62 signaux différents numérotés de 1 à 64[2]. On peut voir la liste en tapant la commande *shell :* ***kill -l***, dont voici un extrait.

```
 1) SIGHUP     2) SIGINT     3) SIGQUIT    4) SIGILL
 5) SIGTRAP    6) SIGABRT    7) SIGBUS     8) SIGFPE
 9) SIGKILL   10) SIGUSR1   11) SIGSEGV   12) SIGUSR2
13) SIGPIPE   14) SIGALRM   15) SIGTERM   16) SIGSTKFLT
17) SIGCHLD   18) SIGCONT   19) SIGSTOP   20) SIGTSTP
21) SIGTTIN   22) SIGTTOU   23) SIGURG    24) SIGXCPU
25) SIGXFSZ   26) SIGVTALRM    27) SIGPROF   28) SIGWINCH
29) SIGIO     30) SIGPWR    31) SIGSYS    34) SIGRTMIN
35) SIGRTMIN+1   36) SIGRTMIN+2   37) SIGRTMIN+3   38) SIGRTMIN+4
39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12
47) SIGRTMIN+13  48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14
51) SIGRTMAX-13  52) SIGRTMAX-12  53) SIGRTMAX-11  54) SIGRTMAX-10
55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6
59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

Les signaux compris entre 1 et 31 sont les signaux **classiques** que l'on peut utiliser.

Les autres signaux sont des signaux qualifiés de signaux temps réel, d'où leurs noms SIG**RT**MIN ou MAX qui respectent la norme POSIX.1B. Leur étude dépasse le cadre de ce support.

Le tableau suivant donne le nom et la description des principaux signaux que nous pourrons utiliser lors du TD et du TP.

Noms des signaux	Description	Noms des signaux	Description
SIGABORT	Abandon	***SIGKILL***	Tuer le processus
SIGALM	Alarme	*SIGPIPE*[3]	Écriture sur un tube sans lecteur
SIGFPE	Erreur arithmétique	*SIGSEGV*	Adresse mémoire invalide
SIGILL	Instruction Illégale	*SIGURS1*	Signal 1 & 2
SIGINT	Interruption CTRL-C	*SIGURS2*	définis par l'utilisateur
SIGCHLD	Terminaison d'un fils	***SIGSTOP***	Suspension du processus
SIGCONT	Reprise du processus	*SIGTERM*	Terminaison
SIGQUIT	Terminaison CTRL-\		

Remarque : **Les signaux *SIGKILL* et *SIGSTOP* sont des signaux non déroutables. Rien ne peut empêcher que l'on tue un Processus, imaginez sinon...**

L'étude du mécanisme des signaux vise à répondre aux questions suivantes :

- **Quelles informations un signal véhicule-t-il ?**
- **Comment les signaux sont-ils émis ?**
- **Quand les signaux sont-ils pris en compte ?**
- **Que fait un processus à la prise en compte d'un signal ?**

2. Les signaux Nř 32 SIGPWR Chute d'alimentation (non Posix) et 33 SIGUNUSED non utilisé...
3. *Nous l'utiliserons au prochain cours sur les tubes.*

4 Des utilisations courantes

Avant de voir plus en détails, voyons 2 utilisations courantes de l'utilité de la gestion des signaux.

On est nul en math, si si je vous l'assure. Imaginons un programme où à un moment donné on a une division par 0... et bien le programme va "planter" et renvoyer un message d'erreur car le noyau veille au grain (il n'aime pas faire une division par 0 ! ! !) et votre processus va recevoir un signal de type **SIGFPE** (FPE comme ***Floating Point Exception***).

Analysons le code suivant :

1. On positionne le ***handler***, si division par 0 la fonction **handFPE** sera lancée si la personne commet une 1ière fois l'erreur de faire une division par 0 !
2. Dans cette fonction (***handler***) **handFPE** on positionne un nouveau ***handler*** rienCompris qui sera lancée si la personne commet une 2ième fois l'erreur de faire une division par 0 !
3. Dans cette fonction (***handler***) **rienCompris** on positionne le ***handler*** par défaut **SIG_DFL** qui sera lancé si la personne commet une 3ième fois l'erreur de faire une division par 0 !
4. Comme la personne (le programme) fait 3 fois une division par zéro, la 3ième fois le noyau se "fâche" et le processus est tué !

   L'exécution du code renvoie ceci

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ./division-par-zero
Vous avez fait une division par 0 !!!
Je viens de vous dire que le dénominateur ne doit pas être nul !!!
Exception en point flottant
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <string.h> //pour memset

struct sigaction action;
sigset_t ens;

void rienCompris() { //
  printf("Je viens de vous dire que le dénominateur ne doit pas être nul !!! \n");
  action.sa_handler=SIG_DFL; // la prochaine fois on jette !
  sigaction(SIGFPE, &action, NULL);
}

void handFPE() { // Le handler ou fonction lancée !
  printf("Vous avez fait une division par 0 !!!\n");
  action.sa_handler=rienCompris; // en permanence
  sigaction(SIGFPE, &action, NULL);
}

int main() {
  int a =12;
  int b = 0;
  int c;

  action.sa_handler = handFPE;
  action.sa_flags=0;
  //vide l'ensemble de signaux fourni
  sigemptyset(&action.sa_mask);
  sigaction(SIGFPE, &action, NULL);
  sigaddset(&action.sa_mask,SIGFPE);

  c = a/b; // la divison par 0 ...
  printf("encore une fois \n");
  c = a/b;
  printf("encore une fois et c'est la dernière ! \n");
  c = a/b;
  printf("Je ne serais jamais affiché sniff !!!\n");
  return 0;
}
```

Exécuter l'action par défaut, SIG_DFL.

3

2

1

4

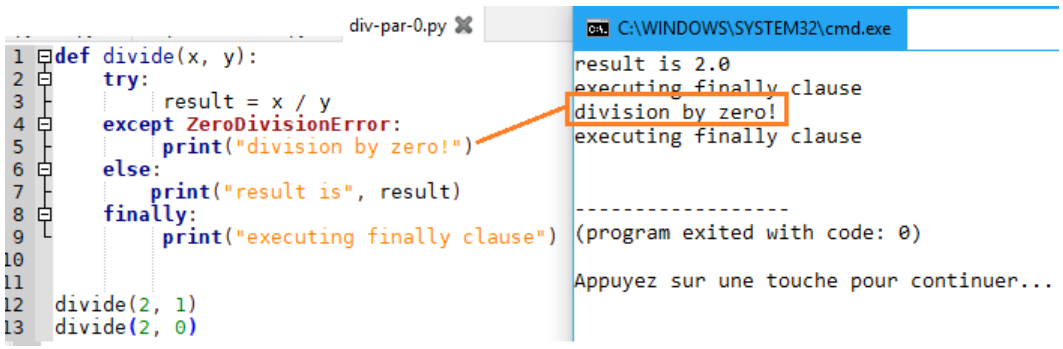
```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ./ex3
Vous avez fait une division par 0 !!!
Je viens de vous dire que le dénominateur ne doit pas être nul !!!
Exception en point flottant
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

Le même ou presque en python et testé sur un autre système d'exploitation que l'on jette trop souvent par la fenêtre ... :

https://docs.python.org/fr/3.7/tutorial/errors.html#user-defined-exceptions

```python
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")

divide(2, 1)
divide(2, 0)
```



https://docs.python.org/fr/3.7/tutorial/errors.html

Le même en java : (pas testé mais vous allez le faire ☺ !

Java pour vous c'est de la rigolade.

```
public static void main(String[] args) {
    int pay=8,payda=0;
3   try {
                // if I cast any of the two variables,
                // program does not recognize the catch block, why is it so?
        double result=pay/(double)payda;
        System.out.println(result);
8       System.out.println("inside-try");
    } catch (Exception e) {
        System.out.println("division by zero exception");
        System.out.println("inside-catch");
    }
13  }
```

4.1 Utiliser trap en bash pour piéger des signaux

Le shell Bash (et les autres aussi !) est capable d'intercepter les signaux envoyés par certains raccourcis claviers (Ctrl-C etc...) et de changer le comportement par défaut de ces raccourcis. Il suffit pour cela d'utiliser la commande **trap**.

Elle prend en premier argument la commande à exécuter puis les signaux sur lesquels elle doit réagir.

```
trap "echo j\'aime pas le CTRL-C :p \n" 2
```

ce qui donne à l'exécution

```
root@debian95-Rx-Sys-Fougeray:~# trap "echo j\'aime pas le CTRL-C :p    "2
root@debian95-Rx-Sys-Fougeray:~# ^Cj'aime pas le CTRL-C :p
                                                              ou INT
root@debian95-Rx-Sys-Fougeray:~#
```

Question : Quel est le rôle du **back-slash ** devant l'**apostrophe '** ?

5 Programmation des signaux

Bon et bien il va bien falloir s'y mettre et en langage C en plus...

5.1 Envoyer un signal

Un processus envoie un signal à un autre processus en appelant la primitive ***kill()***

#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);

L'interprétation du résultat de la fonction diffère en fonction de la valeur présente dans le champ **pid**.

— si **pid>0**, le signal est envoyé au processus identifié par le **pid** ;

— si **pid=0**, le signal est envoyé à tous les processus appartenant au même groupe que le processus appelant ;
— si **pid=-1**, le signal est envoyé à tous les processus du système sauf INIT ;
— si **pid<-1**, le signal est envoyé à tous les processus du groupe ***-pid***.

L'exemple ci-contre montre qu'un processus peut s'envoyer des signaux. Si si c'est possible... il peut même se faire un harakiri ... s'il envoie le bon Nř de signal ☺

```
// Fichier autokill.c
/* utilisation : autokill Num. signal */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

int main(int argc, char *argv[]){
  if(argc !=2){
    fprintf(stderr, "Usage:%s signal\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  if(kill(getpid(), atoi(argv[1])) ==-1)
    perror(NULL);
  fflush(stdout); //vide tampon
  return EXIT_SUCCESS;
}
```

5.2 Attacher un *handler* à un signal

Si on n'attache pas de *handler* à un signal, c'est le *handler* par défaut **SIG_DFL** qui est installé et le processus est alors interruptible.

Il y a 2 primitives pour installer un *handler* :

1. La primitive **signal**(). Son utilisation est très simple, mais déconseillée[4], car elle peut poser des problèmes de compatibilité entre les différents systèmes Unix.

   En TP, par manque de temps on pourra l'utiliser.

2. L'appel système **sigaction**(). Son utilisation est plus complexe, mais son comportement est fiable et elle est normalisé par la norme ***POSIX*** (*Portable Operating System Interface* et X pour Unix).

5.2.1 Signal()

La primitive **signal**(), son prototype est

#include <signal.h>
void (*signal(int sig, void (*p_handler)(int)))(int);

Typiquement, le code d'un tel *handler* sera de la forme

void handler(int sig) {
.....
signal(sig, handler); ou ***signal(sig, SIG_DFL);***
.....}

Le code suivant donne un exemple.

La fonction *main* intercepte le signal **SIGINT** (CTRL-C). Le reste du temps elle affiche en boucle un message toutes les secondes. Le 1^{er} CTRL-C provoque une réaction du programme, qui lance son *handler ouille* et affiche la chaîne de caractères *OUILLE*...

Le handler repositionne le signal par défaut et au second CTRL-C, le processus est arrêté.

Si on remplace ***SIG_DFL*** par le *handler ouille*, le processus ne devient plus interruptible par le CTRL-C, car on **réarme le signal** à chaque fois.

4. Nous l'utiliserons quand même dans un premier temps, pour la compréhension des mécanismes.

```c
/* fichier ctrlc.c */
/* utilisation : ctrlc */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void ouille(int sig){
  printf("\n\t\tOUILLE!-Signal reçu %d\n",sig);
  //On réarme le signal par défaut
  (void) signal(SIGINT, ouille);
}

int main(){
  // handler ouille CTRL-C
  (void) signal(SIGINT, ouille);
  while(1) {
    printf("Hello World \n");
    sleep(1);}
  return EXIT_SUCCESS;
}
```

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ctrlc
Hello World
Hello World
Hello World
^C
                OUILLE!-Signal reçu 2
Hello World
Hello World
Hello World
Hello World
^C
                OUILLE!-Signal reçu 2
Hello World
Hello World
```

Le processus continue !!!

5.2.2 Sigaction()

C'est cette primitive qu'il faut utiliser ! ! !

La primitive **sigaction**(), son prototype est

#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

signum indique le signal concerné, à l'exception de SIGKILL et SIGSTOP.

Si ***act*** est non nul, la nouvelle action pour le signal *signum* est définie par *act*.

Si ***oldact*** est non nul, l'ancienne action est sauvegardée dans *oldact*.

Elle prend en argument 3 paramètres

1. La constante symbolique représentant le nř du signal ;
2. 2 pointeurs sur des structures **sigaction**.

Ce sont des structures qui définissent la gestion d'un signal lors de sa réception. La première structure définit le comportement à adopter et la seconde sert de sauvegarde pour une ancienne structure.

- **sa_handler** : de type ***sighandler_t***, il s'agit d'un pointeur vers la fonction de gestion du signal.
- **sa_mask** : fournit un masque des signaux à bloquer pendant l'exécution du gestionnaire. Il n'est pas nécessaire de bloquer explicitement le signal qui déclenche la routine de gestion, il est automatiquement bloqué.
- **sa_flags** : permet de choisir des options quant aux comportements du gestionnaire de signal. C'est un OU binaire entre différentes options, telle celle utilisée dans l'exemple : **SA_RESETHAND**.

Il est également possible de modifier de manière "dynamique" le masque des signaux bloqués. Pour cela, on utilise la primitive :

sigprocmask()

#include <signal.h>
int sigprocmask(int opt, const sigset_t *new, sigset_t old);

Cette primitive accepte 3 arguments :

1. Une constante déterminant la manière de modifier le masque,
   — SIG_SETMASK redéfinit un nouveau masque en partant de rien,
   — SIG_BLOCK ajoute un ou des signaux à bloquer,
   — SIG_UNBLOCK retire un ou des signaux.
2. Un argument de type *sigset_t* qui est l'ensemble des signaux à définir, à ajouter ou à retirer,
3. Un argument de type *sigset_t* qui permet de sauvegarder l'ancien ensemble de signaux.
   — **sigemptyset**() vide l'ensemble de signaux fourni,
   — **sigaddset**() et **sigdelset**() ajoutent ou suppriment respectivement le signal.

```
/* fichier ctrlc2.c */
/* utilisation : ctrlc2 */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void ouille(int sig){
  printf("\n\t\tOUILLE!-Signal reçu %d\n",sig);
}

int main(){
  struct sigaction action;
  action.sa_handler = ouille;
  //on ne bloque aucun signal
  sigemptyset(&action.sa_mask);
  action.sa_flags = SA_RESETHAND;
  //On détourne le signal Ctrl-C Num. 2
  sigaction(SIGINT, &action, 0);
  while(1) {
    printf("Hello World \n");
    sleep(1);}
  return EXIT_SUCCESS;
}
```

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ./ctrlc2
Hello World
Hello World
Hello World
^C
                OUILLE!-Signal reçu 2
Hello World
Hello World
Hello World
Hello World
^C
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

Le processus stoppé au second CTRL ^c !!!

Bizarre bizarre, à l'exemple précédent le processus continuait à tourner et se contentait d'afficher OUILLE !

Dans l'exemple suivant :
— Le signal Nř3, CTRL-\ (touches CTRL-ALTGR-\) est toujours bloqué
— Le signal Nř2, CTRL-C lui est bloqué une première fois et est débloqué dans le *handler*.

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# gcc sigaction_exemple.c -o sigaction_exemple
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ./sigaction_exemple
^C
 On entre dans le handler avec le signal: 2
Signaux bloqués : * 2 ** 3 *
On quitte le handler
^C
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

```
 1  /* fichier sigaction_exemple.c */
    #include <stdio.h>
    #include <signal.h>
    #include <unistd.h>

 6  sigset_t ens;
    struct sigaction action;
    int sig; // variable globale pour main et handler

    void handler(int sig) {
11    int i;
      printf("\n On entre dans le handler avec le signal: %d\n",sig);
      // sigprocmask fournit l'ensemble des signaux masqués
      sigprocmask(SIG_BLOCK,(sigset_t *)0,&ens);
      printf("Signaux bloqués : ");
16    for(i=1;i<NSIG;i++){
        if(sigismember(&ens,i))
        printf("* %d *",i);
        }
      putchar('\n'); // pour aller à la ligne et vider le tampon.
21    if(sig == SIGINT){
        action.sa_handler=SIG_DFL;
        // Comportement standard (SIG_DFL)du signal SIGINT rétabli.
        sigaction(SIGINT,&action,NULL);
      }
26    printf("On quitte le handler\n");
    }

    int main(void) {
      action.sa_handler=handler; // quelle fonction lancer
31    sigemptyset(&action.sa_mask); //vide l'ensemble de signaux fournis
      sigaction(SIGQUIT,&action,NULL);
        // seul SIGQUIT est masqué pendant l'exécution du handler.
      sigaddset(&action.sa_mask,SIGQUIT);
        // SIGINT et SIGQUIT masqués pendant l'exécution du handler.
36    sigaction(SIGINT,&action,(struct sigaction *)0);
      while(1) //boucle sans fin
        sleep(1);
     printf("Vous avez envoyé un second CTRl–c \n", sig);
     return 0;
41  }
```

Remarque : On aurait pu utiliser un autre signal par exemple SIGTSTP qui correspond au CTRL-Z cela aurait donné

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# gcc sigaction_exemple-SIGTSTP.c -o sigaction_exemple
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ./sigaction_exemple-SIGTSTP
^C
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ./sigaction_exemple-SIGTSTP
^Z
 On entre dans le handler avec le signal: 20
Signaux bloqués : * 3 ** 20 *
On quitte le handler
^C
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

Question : L'avant dernière ligne de code : ***printf("Vous avez envoyé un second CTRl-c \n", sig)*** ; sera-t'elle exécutée ?
— **Réponse** NON car... à vous !

6 Attendre un signal

Un processus peut se mettre en attente de la délivrance d'un signal au moyen des primitives ***pause***() et ***sigsuspend***().

6.1 pause()

La primitive **pause**() permet à un processus de se mettre en attente de la délivrance d'un signal.
Son prototype est :
#include <unistd.h>
int pause(void);
Elle ne permet pas de spécifier ni l'attente d'un signal particulier, ni de savoir quel signal a réveillé le processus.
Elle renvoie toujours la valeur -1 en positionnant *errno* à la valeur *EINTR*.
Dans l'exemple suivant :
— Le processus s'endort à l'aide de la primitive ***pause*** jusqu'à recevoir le signal *SIGINT* (CTRL-C) qui le réveillera.
— Le processus exécute alors le *handler* correspondant à ce signal et se met donc au travail.

```
/* Utilisation de la primitive pause() */
/* fichier travail_pause.c */
/* utilisation : travail_pause puis CTRL-C*/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

void au_boulot();

int main(){
  /* Processus arme le signal et l'attend */
  if (signal(SIGINT, au_boulot)==SIG_ERR)
    { perror("signal");
      exit(1);
    }
  pause(); // attente d'un signal quelconque !
  return 0;
}

void au_boulot(){
  char chaine[10];
  int fp; //descripteur de fichier
  printf("\n\n Appuyez sur CTRL^C sinon ça peut durer longtemps... \n");
  printf("\nOui, oui je sais il faut travailler :)\n");
  printf("Alors, donnez-moi une chaine de caractères \n");
  scanf("%s",chaine);
  if((fp = open("fichier", O_WRONLY))==-1){
    perror("Avez-vous créé le fichier fichier ?");
    exit(1); // Cela s'est mal passé :-(
  }
  write(fp, chaine, strlen(chaine));
  close(fp); // on ferme le fichier...
```

```
  exit(0); // tout s'est bien passé :-)
}
```

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# travail_pause


 Appuyez sur CTRL^C sinon ça peut durer longtemps...
^C
Oui, oui je sais il faut travailler :)
Alors, donnez-moi une chaine de caractères
Bonjour les L3, réveillez vous je continue le cours !!!
Avez-vous créé le fichier fichier ?: No such file or directory
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# touch fichier
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# travail_pause


 Appuyez sur CTRL^C sinon ça peut durer longtemps...
^C
Oui, oui je sais il faut travailler :)
Alors, donnez-moi une chaine de caractères
Bonjour les L3, réveillez vous je continue le cours !!!
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# cat fichier
Bonjourroot@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

L'inconvénient de la primitive *pause*, c'est quelle ne permet pas de programmer l'attente d'un signal particulier.

On pourrait[5], **imaginez** le scénario suivant :

1. **Associer** un ***handler h*** à la délivrance de tous les signaux récupérables ;
2. **Tester** dans ce *handler* si le signal reçu est le signal attendu et alors **positionner un booléen à vrai** ;
3. **Programmer** une boucle itérant l'appel de la primitive *pause*, le test de sortie de boucle portant sur la variable booléenne.

Mais cela ne fonctionne pas, car il faudrait que l'exécution des points 1 et 2 soit **atomique**, c'est-à-dire que l'on ne puisse pas l'interrompre, et rien n'interdit à l'ordonnanceur "d'arrêter" un processus entre ces 2 points.

6.2 Sigsuspend()

Il existe pour cela la primitive **,** qui permet **d'endormir le processus de manière atomiqu**e, car cette fonction restaure le masque de signaux installés au moment de l'appel. Il est ainsi possible de bloquer certains signaux et d'endormir le processus de manière atomique. Cela ne fonctionne pas pour tous les signaux ! ! ! En effet, il n'est pas possible de bloquer **SIGKILL** ou **SIGSTOP** ; spécifier ces signaux dans ***mask*** n'a aucun effet sur le masque de signaux du processus.

Cette primitive réalise automatiquement :

— l'installation du masque de signaux ;

— la mise en sommeil jusqu'à l'arrivée d'un signal non masqué qui va provoquer la mort du processus ou l'exécution du **handler** installé pour ce signal.

```
// Cet exemple remplace le masque de signal, puis suspend l'exécution.
// fichier : exemple-sigsuspend.c
#define _POSIX_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>

void catcher(int signum) {
  switch (signum) {
```

5. j'emploie bien le conditionnel ! ! !

```
    case SIGUSR1: puts("l'attrapeur attrape le signal SIGUSR1");
                  break;
    case SIGUSR2: puts("l'attrapeur attrape le signal SIGUSR2");
                  break;
    default:      printf("catcher caught unexpected signal %d\n",signum);
  }
}

int main() {
  sigset_t sigset;
  struct sigaction sact;
  time_t   t;

  if (fork() == 0) { // on est dans le fils
    sleep(10); // il dort 10s
    puts("Le fils envoie le signal SIGUSR2 - qui devrait être bloqué");
    kill(getppid(), SIGUSR2); // le fils envoie le signal SIGUSR2 à son père
    sleep(5); // il dort 5s
    puts("Le fils envoie le signal SIGUSR2 - qui devrait être attrapé");
    kill(getppid(), SIGUSR1); // le fils envoie le signal SIGUSR1 à son père
    exit(0); //  le fils quitte le programme
  }

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  if (sigaction(SIGUSR1, &sact, NULL) != 0)
    perror("1st sigaction() error");
  else if (sigaction(SIGUSR2, &sact, NULL) != 0)
    perror("2nd sigaction() error");
  else {
    sigfillset(&sigset);
    sigdelset(&sigset, SIGUSR1);
    time(&t);
    printf("le père attend son fils qui va lui envoyer le signal SIGUSR1
               à la date %s", ctime(&t));
    if (sigsuspend(&sigset) == -1)
      perror("sigsuspend() returned -1 as expected");
    time(&t);
    printf("sigsuspend est terminé à %s", ctime(&t));
  }
}
```

l'exécution de ce programme renvoie :

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# gcc exemple-sigsuspend.c -o exemple-sigsuspend
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# ./exemple-sigsuspend
le père attend son fils qui va lui envoyer le signal SIGUSR1 à la date Sat Oct 27 12:35:33 2018
Le fils envoie le signal SIGUSR2 - qui devrait être bloqué
Le fils envoie le signal SIGUSR2 - qui devrait être attrapé
l'attrapeur attrape le signal SIGUSR1
l'attrapeur attrape le signal SIGUSR2
sigsuspend() returned -1 as expected: Interrupted system call
sigsuspend est terminé à Sat Oct 27 12:35:48 2018
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

7 Et avec des threads ?

Oui oui, je sais nous n'avons pas encore vu les threads, patience cela va venir ☺

Juste un petit code pour voir que cela est possible. Je reprends l'exemple de la division par 0 avec 2 cas différents, 2 processus et 3 threads.

7.1 2 Processus et 1 SIGFPE

Scénario : le père crée un fils qui fait un division par 0, le fils et seul le fils reçoit ce signal, il est “tué” et pas le père

```
// fichier SIGFPE-2-Processus.c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h> //pour memset

struct sigaction action;
sigset_t ens;

void rienCompris() { //
        printf("Je viens de vous dire que le dénominateur ne doit pas être nul !!! \n");
        action.sa_handler=SIG_DFL; // la prochaine fois on jette !
        sigaction(SIGFPE, &action, NULL);
}

void handFPE() { // Le handler ou fonction lancée !
        printf("Vous avez fait une division par 0 !!!\n");
        action.sa_handler=rienCompris; // en permanence
        sigaction(SIGFPE, &action, NULL);
}

int main() {
  int a =12;
  int b = 0;
  int c;
  int pid;

  action.sa_handler = handFPE;
  action.sa_flags=0;
  //vide l'ensemble de signaux fournis
  sigemptyset(&action.sa_mask);
  sigaction(SIGFPE, &action, NULL);
  sigaddset(&action.sa_mask,SIGFPE);

  pid = fork();
  if (pid==0){ // Toujours le fils qui fait des c...
        printf("\n\t\t Je suis le fils et nul en math, mon PID : \
%d et mon père %d \n",getpid(), getppid());
        c = a/b; // la divison par 0 ...
        printf("\t\t encore une fois \n");
        c = a/b;
        printf("\t\t encore une fois et c'est la dernière ! \n");
        c = a/b;
        printf("\t\t Je ne serais jamais affiché sniff !!!\n");
  }
  else { // Le père en rigole encore ;-)
        sleep(1);
        printf("\nBin... il est mort mon fils... \
juste pour erreur de math, ouf pas pour les étudiants \n");
        printf("Moi je suis encore bien en vie, et mon PID %d \n\n", getpid());
        return 0;
```

```
        }
}
```

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# gcc SIGFPE-2-Processus.c -o SIGFPE-2-P
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux# SIGFPE-2-Processus

                Je suis le fils et nul en math, mon PID : 19262 et mon père 19261
Vous avez fait une division par 0 !!!
Je viens de vous dire que le dénominateur ne doit pas être nul !!!

Bin... il est mort mon fils... juste pour erreur de math, ouf pas pour les étudiants
Moi je suis encore bien en vie, et mon PID 19261

root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/signaux#
```

7.2 3 Threads et 1 SIGFPE

Imaginons le cas suivant, un thread qui déclenche un signal par exemple de type FPE que se passe-t-il ?

Scénario : le Thread Principal crée 2 Thread, l'un d'eux (et non deux !) fait un division par 0, et bien tout le monde meurt, c'est **ballot**

```
//fichier signal-fpe.c dans thread !

void* jedivisepar0(void *arg){
        int a,b,c;
        a=0; b=10;
        sleep(2); // dort 2s pour que l'autre thread puisse W un peu
        c=b/a;     // La division par 0
        pthread_exit(NULL); // Fermeture du thread
}
void* jaffiche(void *arg){
        while(1){
                printf("jaffiche \n");
                usleep(250000); // dort 0,25s
        }
        pthread_exit(NULL); // Fermeture du thread
}
int main(int argc, char *argv[]){
        int i;
        pthread_t thread_id1, thread_id2;
        pthread_create(&thread_id1, NULL, &jaffiche, NULL);
        pthread_create(&thread_id2, NULL, &jedivisepar0, NULL);

        //sleep(1);

        for(i=0; i<2; i++)
                printf("Thread P j'ai que 2 fois à écrire ouf \n");

        // Thread P attend autres threads
        pthread_join(thread_id1, NULL);
        pthread_join(thread_id2, NULL);
        return 0;
}
```

Le résultat est : ... catastrophique ...☹Les 3 threads vont être tués.. !

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# gccp signal-fpe.c -o signal-fpe
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# signal-fpe
Thread P j'ai que 2 fois à écrire ouf
Thread P j'ai que 2 fois à écrire ouf
jaffiche
  PID  SPID TTY          TIME CMD
 1260  1260 pts/0    00:00:01 bash
14866 14866 pts/0    00:00:00 signal-fpe
14866 14867 pts/0    00:00:00 signal-fpe
14866 14868 pts/0    00:00:00 signal-fpe
14869 14869 pts/0    00:00:00 sh
14870 14870 pts/0    00:00:00 ps
jaffiche
jaffiche
jaffiche
jaffiche
jaffiche
jaffiche
jaffiche
Exception en point flottant
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# ps -T
  PID  SPID TTY          TIME CMD
 1260  1260 pts/0    00:00:01 bash
14871 14871 pts/0    00:00:00 ps
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread#
```

PS -T montre 3 threads

3 threads terminés pour 1 thread qui a fait une division par 0

Alors que faire ?
Suite au cours sur les Thread ! ! !

8 Conclusion

La connaissance du mécanisme des signaux dans un système d'exploitation tel *Linux*, permet de pouvoir écrire des applications plus robustes. En effet on peut gérer les interruptions dues aux aléas dynamiques pendant l'exécution d'un programme.

Un processus ne doit pas être interrompu ou stoppé dès qu'il y a une "erreur" qui peut être gérée par le programmeur lorsqu'il développe son application. Par exemple, l'erreur la plus courante dans les applications de calculs est la division par zéro ou bien le débordement, vous avez réservé un emplacement de 32 bits (***int***) pour une variable alors que cette dernière va dépasser une fois cette valeur.

On pourrait faire une synthèse des primitives : ***sleep***(), ***wait***() ou ***waitpid***(), ***pause***() et ***sigsusend***() ... ou on pourrait pour le CT par exemple, non ? ☺