Module n°2

PROGRAMMATION SYSTÈME

Auteur : Singodiwirjo Hermantino
Version 1.0 – 6 février 2004
Nombre de pages : 17

Paris Academy of Computer Science SUPINFO Paris Ecole Supérieure d'Informatique

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com - herman@supinfo.com

Table des matières

1. GESTION CLASSIQUE DES SIGNAUX 3

1.1. PRESENTATION DES SIGNAUX 3
1.2. EMISSION D'UN SIGNAL 4
1.3. RECEPTION DES SIGNAUX AVEC L'APPEL SIGNAL() 5

2. GESTION DES SIGNAUX POSIX.1 7

2.1. RECEPTION DES SIGNAUX AVEC L'APPEL SYSTEME SIGACTION() 7
2.2. CONFIGURATION DES ENSEMBLES DE SIGNAUX 8
2.3. BLOCAGE DE SIGNAUX 10
2.3.1. Blocage de signaux dans le processus 10
2.3.2. Blocage de signaux dans le gestionnaire 12
2.4. UTILISER DES SAUTS NON LOCAUX 13

3. LE SIGNAL D'ALARME 16

1. Gestion classique des signaux

1.1.Présentation des signaux

Vous avez sûrement du en tant qu'utilisateur usé au moins une fois d'un signal. N'avez-vous jamais saisie la fameuse commande :

```
kill -9 <pid>
```

Le résultat de cette commande est l'envoi du signal 9 ou encore SIGKILL au processus de pid <pid>. En saisissant cette commande vous avez en quelque sorte communiqué avec le processus de pid <pid> en lui imposant son arrêt immédiat.
Dès lors l'intérêt des signaux dans la programmation système est évident, il s'agit là d'un nouveau moyen de communication entre processus. Le principe est à priori simple, un processus peut envoyer sous certaines conditions, un signal à un autre. Lorsqu'un processus reçoit un signal, il est obligé à prendre une mesure spécifique :

- Ignorer le signal (quand cela est possible)
- Détourner (quand cela est possible) provisoirement son exécution vers une routine particulière appelée gestionnaire de signal
- Laisser le système traiter le signal avec un comportement par défaut.

Nous avons précisé "quand cela est possible" car il existe comme nous allons le voir des signaux qui ne peuvent être ni capturés ni stoppés.
La liste de tous les signaux disponibles sur votre système est obtenue avec :

```
kill -l
```

Voici la signification de quelques uns, ceux que l'on va éventuellement être amenés à utiliser :

- SIGSEGV : Indique une violation de segmentation (adresse correcte mais située en dehors de l'espace d'adressage affecté à un processus), ce signal survient typiquement lors de l'emploi d'un pointeur mal initialisé, c'est lui qui déclenche le fameux "Segmentation Fault". Par défaut, un processus recevant ce signal s'arrête est un fichier core est créé.
- SIGCHLD : Ce signal est émis par le noyau vers un processus dont un fils vient de se terminer, ou d'être stoppé.
- SIGINT : Ce signal est émis vers tous les processus en avant-plan lors de la frappe de la touche d'interruption d'un terminal (habituellement Contrôle-C). Le comportement par défaut d'un processus recevant ce signal est de s'arrêter.
- SIGKILL : C'est l'un des seul signaux avec SIGSTOP qui ne puisse être ni capturé ni ignoré par un processus. A sa réception tout processus est immédiatement arrêté.
- SIGQUIT : C'est presque le même signal que SIGINT, à la différence près qu'il crée un fichier core après l'arrêt du processus ciblé. Il peut être activé au clavier avec Contrôle-\.
- SIGSTOP,SIGSTP,SIGCONT : Les deux premiers signaux permettent d'arrêter temporairement un processus. La différence entre les deux étant que SIGSTOP ne peut être capturé ou ignoré. SIGCONT permet au processus qui le reçoit de reprendre sans exécution si il était arrêté.
- SIGTERM : Il s'agit d'une demande "gentille" de terminaison de processus.
- SIGUSR1 et SIGUSR2 : Ces deux signaux sont mis à la disposition du programmeur pour ses application. Par défaut un processus recevant ce signal se termine.*

Pour obtenir le libellé d'un signal dont vous connaissez le numéro, vous pouvez utiliser :

```
char *strsignal (int sig_num);
```

ou

```
int psignal(int sig_num, const char* prefix);
```

qui prend en argument le numéro de signal à décrire et la chaîne à afficher avant la description, un peut à la manière de perror. Notez que psignal affiche son résultat sur la sortie d'erreur standard.

Enfin le dernier moyen consiste à parcourir le tableau global de chaîne de caractères sys_siglist et d'afficher la description indexé au numéro de signal voulu:

```
char * sys_siglist[sig_num]
```

Enfin sachez que la constante NSIG (ou __NSIG sous certains systèmes) contient le nombre total de signaux. Voici un petit programme qui affiche tous les signaux disponibles sur votre système :

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

int main ()
{
     int i;

     fprintf(stdout, "Fonction strsignal() :\n");

     for(i=1;i<NSIG;i++)
     {
            fprintf(stdout, "%s\n", strsignal(i);
     }

     fprintf(stdout, "Utilisation de sys_siglist[] :\n");

     for(i=1;i<NSIG;i++)
     {
            fprintf(stdout, "%s\n", sys_siglist[i]);
     }

     return 0;

}
```

Que pouvez vous remarquer comme différence entre les deux sorties ?

1.2.Emission d'un signal

Pour envoyer un signal à un processus on utilise l'appel système :

```
int kill(pid_t pid, int sig_num);
```

Si le premier argument est positif, il correspond au pid du processus visé, si il est nul, le signal sera envoyé à tous les processus du groupe auquel appartient le processus appelant et si il est négatif (sauf -1), le signal est envoyé à tous les processus du groupe spécifié par la valeur absolue du pid.
Enfin si vous fournissez -1 en premier argument, Posix indique un comportement indéfini, sous Linux le signal est envoyé à tous les processus sauf init.

Le second argument correspond au numéro de signal que vous souhaitez envoyer. Attention à ne jamais utiliser un nombre pour le second argument mais plutôt les constantes symboliques associées

(SIGTERM, SIGCONT ..). Cela aura comme premier effet d'améliorer considérablement la lisibilité de votre code et assurer de plus une portabilité accrue.

Il existe une fonction de bibliothèque nommé **raise** :

```
void raise(int sig_num);
```

qui est équivalent à l'appel :

```
kill(getpid(), sig_num);
```

En bref elle permet à un processus d'envoyer un signal à lui-même.
Bien entendu un processus ne peut envoyer de signaux à tout va, voici les quelques règles simples qui régissent l'envoi d'un signal :

- Un processus peut envoyer un signal à un autre processus si l'UID réel ou effectif est égal à l'UID réel ou sauvé de la cible.
- Un processus dont l'UID effectif est 0 (root) peut envoyer des signaux à tout processus.
- Si le signal envoyé est SIGCONT, il suffit que le processus émetteur appartienne à la même session que le processus visé.

1.3.Réception des signaux avec l'appel signal()

Un processus peut demander au noyau d'installer un gestionnaire pour un signal particulier, c'est-à-dire une routine spécifique qui sera invoquée lors de l'arrivée de ce signal. Ceci peut se faire grâce à l'appel système signal :

```
gestion_t signal(int sig_num,  gestion_t gestionnaire);
```

Le premier argument est le numéro du signal que l'on veut traiter. Le second argument permet de passer le nom de la routine que vous souhaitez installer comme gestionnaire de signal.
Si vous passez en second argument SIG_IGN, le processus demande au noyau d'ignorer le signal.

Le gestionnaire de signal est une routine comme les autres qui prend un argument de type entier et qui ne renvoie rien (void). L'argument transmis correspond au numéro de signal ayant déclanché le gestionnaire. Il est donc possible d'écrire un seul gestionnaire pour plusieurs signaux en déterminant les actions à effectuer à l'aide d'un switch/case.
Voici un exemple de la mise en place d'un gestionnaire :

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

void gestionnaire(int sig_num);

int main ()
{
     int i;

     for(i=1;i<NSIG;i++)
     {
            if(signal(i, gestionnaire) == SIG_ERR);
            {
                   fprintf(stderr,"Signal %d non capturé\n", i);
            }
     }
     while (1)
     {
```

```
            fprintf(stderr, "En attente d'un signal … \n");
            pause();
      }

      return 0;
}

void gestionnaire(int sig_num)
{
      fprintf("Dans le processus %u\n", getpid());
     fprintf(stderr, "Signal recu : %d ⇔ %s\n", sig_num,
                                        sys_siglist[sig_num]);
}
```

Le seul moyen d'arrêter se programme est d'envoyer le signal SIGKILL au processus depuis un autre terminal.

2. Gestion des signaux Posix.1

La gestion des signaux à la manière Posix.1 n'est pas beaucoup plus compliquée que ce que nous venons de voire. Cette méthode d'implémentation de la gestion des signaux permet un paramétrage plus souple du comportement du programme. L'appel signal() doit être réservé aux programmes destinés à être portés sur des systèmes non Posix, dans tous les autres cas on lui préférera sigaction() que nous allons voir dans ce chapitre.

2.1. Réception des signaux avec l'appel système sigaction()

La mise en place d'un gestionnaire de signal Posix.1 passe par l'appel à **sigaction**, défini comme tel :

```
  int sigaction (int sig_num, const struct sigaction *new, struct
sigaction *old);
```

Si le pointeur sur la nouvelle structure est NULL, aucune modification n'a lieu, seul l'ancien comportement est sauvegardé dans la structure old. De même si le second pointeur est NULL, aucune sauvegarde n'aura lieu. Le premier argument permet comme pour signal de préciser le signal pour lequel on souhaite installer le gestionnaire.
Comme vous pouvez le constater aucun des deux arguments suivants ne permettent de passer directement le nom de la routine qui va être appelée à la réception du signal.
Regardons plus en détail la composition de la structure sigaction :

Champ	Type	Description
sa_handler	sighandler_t	Pointeur sur le gestionnaire de signal
sa_mask	sigset_t	Ensemble de signaux qui seront bloqués pendant l'exécution du gestionnaire.
sa_flags	int	OU binaire entre les différentes constantes ci-dessous.

- SA_NOCLDSTOP : Il s'agit de la seule constante réellement définie par Posix, lorsqu'elle est définie, le gestionnaire n'est pas invoqué à la réception d'un signal SIGCHLD déclenché par l'arrêt temporaire d'un processus fils. Par contre il sera appelé si le signal SIGCHLD est engendré par l'arrêt définitif d'un processus enfant.
- SA_RESTART : Lorsque cette constante est définie les appels systèmes lents interrompus par le signal concerné sont automatiquement relancés.
- SA_ONESHOT : Lorsqu'un gestionnaire de signal est invoqué et que cette constante est passée au travers de la structure sigaction, le comportement par défaut est réinstallé pour le signal concerné. Cela vous permet par exemple d'obtenir des comportements du type :

```
  (Vous avez saisi Ctrl-C, saisissez encore une fois cette combinaison
pour terminer le programme)
```

- SA_SIGINFO : Cette constante n'est définie que depuis Linux 2.2. Elle est surtout utilisée pour les signaux temps réel mais peut également être utilisée pour les signaux classiques. Un gestionnaire installé avec cette option recevra des informations supplémentaires en plus du numéro de signal qui l'a déclenché. Le gestionnaire doit alors accepter 3 arguments :
  1. Le numéro du signal reçu.
  2. Un pointeur sur une structure siginfo_t.
  3. Un argument du type void *, nous détaillerons ses possibilité plus tard.

- SA_ONSTACK : Egalement défini depuis Linux 2.2 cette constante permet de préciser au gestionnaire du signal en question d'utiliser une pile différente de celle du programme.

2.2.Configuration des ensembles de signaux

Voyons maintenant comment manipuler des ensembles de signaux, qui sont en fait le type sigset_t. Il ne faut jamais manipuler ce type "à la main" mais toujours préférer utiliser l'ensemble des routines que nous allons voir.

1. Pour vider un ensemble, ce qui vaut à l'initialiser avec aucun signal :
```
int sigemptyset (sigset_t * ensemble);
```

2. Pour remplir un ensemble avec tous les signaux connus sur le système :
```
int sigfillset(sigset_t * ensemble);
```

3. Pour ajouter un signal à un ensemble :
```
int sigaddset(sigset_t *ensemble, int sig_num);
```

4. Pour supprimer un signal dans un ensemble :
```
int sigdelset(sigset_t *ensemble, int sig_num);
```

5. Pour savoir si un signal fait parti d'un ensemble on utilisera :
```
int sigismember(const sigset_t *ensemble, int sig_num);
```

Les routines 3 et 4 renvoie 0 en cas de succès et -1 en cas d'échec. La routine 5 elle renvoie 1 si le signal fait parti de l'ensemble et 0 si ce n'est pas le cas.

Bien à présent que nous savons manipuler le type sigset_t, voyons comment installer un gestionnaire de signal simple à la manière Posix.1 :

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

void gestionnaire(int sig_num);

int main ()
{
     struct sigaction action;
     int i;

     action.sa_handler = gestionnaire;
     sigemptyset(&(action.sa_mask));
     action.sa_flags = 0;

     for(i=1;i<NSIG;i++)
     {
            if(sigaction(i, &action,NULL) := 0);
            {
                   fprintf(stderr,"Signal %d non capturé\n", i);
            }
     }
     while (1)
     {
            fprintf(stderr, "En attente d'un signal … \n");
            pause();
     }
```

```
     return 0;
}

void gestionnaire(int sig_num)
{
     fprintf("Dans le processus %u\n", getpid());
     fprintf(stderr, "Signal recu : %d ⇔ %s\n", sig_num,
                                        sys_siglist[sig_num]);
}
```

Essayons maintenant de n'installer le gestionnaire que pour SIGQUIT et SIGINT, nous préciserons en plus dans le cas de SIGINT de relancer automatiquement les appels systèmes lent (ici read) et de restituer son comportement par défaut après une utilisation :

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

void gestionnaire(int sig_num);

int main ()
{
     struct sigaction action;
     int i;

     /*Pour SIGQUIT*/
     action.sa_handler = gestionnaire;
     sigemptyset(&(action.sa_mask));
     action.sa_flags = 0;

     if(sigaction(SIGQUIT, &action,NULL) := 0);
     {
            fprintf(stderr,"Signal %d ne peut être capturé\n", i);
            exit(0);
     }

     /*Pour SIGINT*/
     action.sa_handler = gestionnaire;
     sigemptyset(&(action.sa_mask));
     action.sa_flags = SA_RESTART|SA_ONESHOT;

     if(sigaction(SIGINT, &action,NULL) := 0);
     {
            fprintf(stderr,"Signal %d ne peut être capturé\n", i);
            exit (0);
     }

     while (1)
     {
            fprintf(stderr, "Appel à read()\n");

            if(read(0, &i, sizeof(int))<0)
            {
                   if(errno == EINTR)
                          printf("EINTR\n");
            }
     }

     return 0;
}
```

```
void gestionnaire(int sig_num)
{
     switch(sig_num)
     {
            case SIGQUIT:
                  fprintf(stdout, "\nSIGQUIT reçu\n);
                  fflush(stdout);
                  break;

            case SIGINT:
                  fprintf(stdout, "\nSIGINT reçu\n");
                  fflush(stdout);
                  break;
     }
}
```

2.3.Blocage de signaux

2.3.1. Blocage de signaux dans le processus

Comme nous l'avons spécifié à plusieurs reprises, vous pouvez choisir de bloquer un ensemble de signaux (sauf SIGKILL et SIGSTOP). Pour cela vous devez utiliser la routine :

```
int sigprocmask(int methode, const sigset_t *ensemble, sigset_t
*ancien);
```

Celle-ci est très complète et vous permettra de :

- Bloquer/débloquer des signaux
- Fixer un nouveau masque complet de signaux à bloquer
- Consulter l'ancien masque de blocage

Le premier argument peut prendre l'une des valeurs suivantes :

- SIG_BLOCK : Ajoute la liste de signaux transmis en second argument au masque de blocage des signaux.
- SIG_UNBLOCK : Retire la liste de signaux transmis en second argument au masque de blocage des signaux.
- SIG_SETMASK : Le second argument est utilisé directement comme masque de blocage pour les signaux, comme pour SIG_UNBLOCK la modification du masque peut entraîner le déblocage de un ou plusieurs signaux en attente.

L'intérêt principal du blocage de signaux est la protection de portion critique du code.
Les signaux bloqués par le masque ne sont cependant pas détruits, ils restent en attente derrière le masque jusqu'à que celui-ci soit retiré. Il est possible à tout moment de récupérer dans une variable **sigset_t** l'ensemble des signaux en attente avec :

```
      int sigpending (sigset_t *ensemble) ;
```

Puis pour tester l'existence d'un certain signal dans l'ensemble on utilisera la fonction **sigismember** présentée plus haut. Voici un exemple qui illustre l'utilisation de sigprocmask, sigpending et sigismember dans un petit programme qui dans un premier temps bloque tous les signaux possible, puis consulte tous ceux qui sont en attente derrière le masque et enfin les libère et les regarde arriver :

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

```
/*Definition du gestionnaire de signaux*/
void gestionnaire(int sig_num)
{
     char message[1204];
     sprintf(message,">>Signal recu : %s\n", sys_siglist[sig_num]);
     write(1, message, strlen(message));
}

int main()
{
     int i;
     struct sigaction action;
     sigset_t ensemble;

     /*On affiche le pid pour savoir à qui envoyer des signaux*/
     fprintf(stdout,"Envoyez moi des signaux !! [%u]\n", getpid());
     fflush(stdout);

     /*On met en place le gestionnaire qui capturera tous les
       signaux*/
     action.sa_handler = gestionnaire;
     sigemptyset(&(action.sa_mask));
     action.sa_flags = 0;

     for(i=1; i<NSIG; i++)
     {
           if(sigaction(i, &action, NULL) != 0)
                 perror("Signal non capturé");

     }

     /*On met en place le masque*/
     sigfillset(&ensemble);
     sigdelset(&ensemble, SIGINT);
     sigprocmask(SIG_BLOCK, &ensemble, NULL);

     /*Lecture bloquante avec read pour attendre des signaux*/
     read(0, &i, sizeof(int));


     /*Si le read à débloqué, (l'utilisateur à appuyé sur ENTRÉE)
       on commence par consulter l'ensemble des signaux en attente*/
     sigpending(&ensemble);

     for(i=1; i<NSIG; i++)
     {
           if(sigismember(&ensemble, i))
           {
                 fprintf(stdout,"<<signal    %s    en    attente\n",
sys_siglist[i]);
                 fflush(stdout);
           }
     }

     printf("\n");

     /*Puis on retire le masque*/
     sigemptyset(&ensemble);
     sigprocmask(SIG_SETMASK, &ensemble, NULL);
     return 0;
}
```

2.3.2. Blocage de signaux dans le gestionnaire

Il est des cas où l'on ne souhaite pas mettre en place un masque général au processus car nous avons besoin de recevoir un ensemble de signaux assez important. Cependant la réception d'un de ces signaux durant l'exécution du gestionnaire serait intolérable car celui-ci peut par exemple être entrain de modifier une structure globale et ne doit en aucun cas être interrompu. Pour cela nous avons à notre disposition un "mini masque" de signaux valable que lors de l'exécution du gestionnaire, nous l'avons déjà présenté comme étant membre de la structure sigaction, il s'agit de **sa_mask**. Celui-ci s'utilise exactement comme le masque général mais n'existera que lors de l'exécution du gestionnaire. Voyons un petit exemple où nous mettrons en place deux gestionnaires :

- Le premier se déclenchera à la réception du signal SIGUSR1 et simulera une tâche longue à réaliser en bouclant 'n' fois et s'endormant 1seconde à chaque fin de tour de boucle. Ce gestionnaire sera installé avec un sa_mask le protégeant de tous les signaux possible.
- Le second sera un gestionnaire permettant de capturer tout type de signaux et en affichera le nom.

Notre application elle devra créer un fils qui lui enverra tout d'abord le signal SIGUSR1, ceci aura comme conséquence de faire entrer le père dans le premier gestionnaire "long". Puis le fils enverra une série de signaux à son père. Comme on peut s'y attendre tous ces signaux seront ignorés, de plus à la fin de l'exécution du gestionnaire "long" les signaux en attente seront capturés par le second gestionnaire.

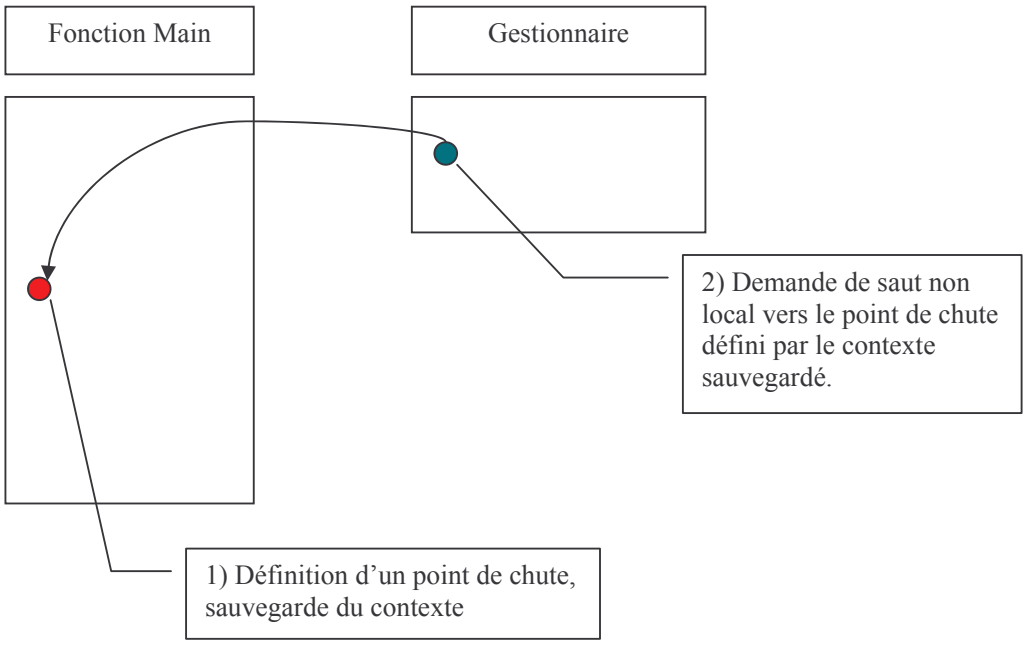
```
  #include <signal.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <unistd.h>
  #include <errno.h>


  void gestionnaire2(int sig_num)
  {
       char message[1024];
       sprintf(message, "Signal reçu : %s\n", sys_siglist[sig_num]);
       write(1, message, strlen(message));
  }

  void gestionnaire(int sig_num)
  {
       int i;
       fprintf(stdout, "\nrecu %s\n", sys_siglist[sig_num]);
       fflush(stdout);

       for(i=0;i<6;i++)
       {
             printf("dans     le     gestionnaire     par     %s\n",
sys_siglist[sig_num]);
             sleep(1);
       }
  }

  int main()
  {
       int i;
       struct sigaction action;

       action.sa_handler = gestionnaire;

       action.sa_flags = 0;

       /*Dans le premier gestionnaire, on bloque tout */
       sigfillset(&(action.sa_mask));

       if(sigaction(SIGUSR1, &action, NULL) != 0)
             perror("Signal SIGUSR1 non capturé");
```

```
	/*Durans le second gestionnaire on ne permet l'arrivée
	  d'aucun signal*/
	sigfillset(&(action.sa_mask));
	action.sa_handler = gestionnaire2;

	for(i=0;i<NSIG;i++)
	{
		if(i!=SIGUSR1 && i!=SIGINT)
		{
			sigaction(i, &action, NULL);
		}
	}

	switch(fork())
	{
		case -1:
			fprintf(stderr, "Erreur dans le fork ..\n");
			return 1;
		break;

		case 0:

			/*Dans le fils en envoie tous les signaux*/
			sleep(1);
			kill(getppid(), SIGUSR1);
			sleep(1);

			printf("Envoie de SIGFPE\n");
			kill(getppid(), SIGFPE);

			sleep(1);
			printf("Envoie de SIGUSR2\n");
			kill(getppid(), SIGUSR2);

			sleep(1);
			printf("Envoie de SIGQUIT\n");
			kill(getppid(), SIGQUIT);


			sleep(1);
			printf("Envoie de SIGXCPU\n");
			kill(getppid(), SIGXCPU);


			return 0;

		break;

		default:
			/*Dans le père on attend
			-> A ce moment on devrait déjà être dans
			   le gestionnaire long*/
			read(0, &i, sizeof(int));
		break;
	}

	return 0;
}
```

2.4.Utiliser des sauts non locaux

Il peut arriver de vouloir traiter d'une manière très particulière la réception d'un signal, en exigeant par exemple de reprendre l'exécution du programme dans un contexte différent qui à été sauvegardé auparavant. Cette méthode permet entre autre de reprendre le contrôle d'un programme ayant effectué

une instruction illégale, comme une division par zéro par exemple. Voici un schéma qui montre le mécanisme que nous allons mettre en place :



En fait ce mécanisme est tout à fait semblable à un *goto*, à la différence près que dans ce cas le saut peut se faire d'une fonction à une autre en restaurant le contexte (états des variables ..) sauvegardé au moment de la définition du point de chute.
La définition d'un point de chute se fait avec la fonction :

```
int sigsetjmp(sigjmp_buf contexte, int sauver_signaux) ;
```

Le premier argument est la variable qui va contenir le contexte à sauvegarder, celle-ci doit être déclarée comme globale pour permettre à toutes les fonctions effectuant un saut d'y accéder. Si le second argument est non nul, la sauvegarde du contexte comprendra la sauvegarde des masques de blocages.
Si la valeur de retour est nulle, cela signifie que le point de chute viens d'être mis en place, sinon elle indique la valeur passée en second argument à **siglongjmp** que nous allons voire maintenant. Ceci permet de savoir d'où on vient lorsque nous effectuons des sauts locaux depuis plusieurs gestionnaires de signaux par exemple.
Voici le prototype de siglongjmp qui nous permet d'effectuer un saut non local :

```
void siglongjmp(sigjmp_buf contexte, int valeur) ;
```

Voici un exemple de saut non local au travers l'implémentation d'une calculatrice qui ne fait que diviser deux nombres mais qui permet de contrôler une demande de division par zéro qui engendre l'émission du signal SIGFPE (*floating point exception*).

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

/*Variable pour sauvegarder le contexte*/
sigjmp_buf contexte;
```

```
  /*Le gestionnaire qui effectue le saut*/
  void gestionnaire_fpe(int sig_num)
  {
       /*On saute vers le point de chute !!*/
       siglongjmp(contexte, 1);

       /*Le saut à raté, il faut sortir !!*/
       signal(sig_num, SIG_DFL);
       raise(sig_num);
  }

  /*Un gestionnaire permettant de quitter avec Ctrl-C*/
  void gestionnaire_int(int sig_num)
  {
       fprintf(stdout,  "Le  signal  %s  est  détecté,  on  quitte.\n",
sys_siglist[sig_num]);
       exit(0);
  }

  int main ()
  {
       int p,q,r;
       signal(SIGFPE, gestionnaire_fpe);
       signal(SIGINT, gestionnaire_int);

       for(;;)
       {

              /*Définition du point de chute*/
              if(sigsetjmp(contexte, 1) != 0)
              {
                     /*On est arrivé ici par siglongjmp..*/
                     fprintf(stdout, "Erreur mathématique, division par
0 ?\n");
                     fflush (stdout);
              }
              while(1)
              {
                     fprintf(stdout,"Entrer le dividende :\n");
                     fflush(stdout);
                     if(fscanf(stdin, "%d", &p) == 1)
                            break;
              }

              while(1)
              {
                     fprintf(stdout,"Entrer le diviseur :\n");
                     fflush(stdout);
                     if(fscanf(stdin, "%d", &q) == 1)
                            break;
              }
              r = p/q;
              fprintf(stdout,  "Le  resultat  de  la  division  est  %d\n",
r);
              fflush (stdout);
       }
       return 0;
  }
```

3. Le signal d'alarme

Le signal SIGALRM vous permet de mettre en place, avec l'appel alarm() une temporisation, en fait un délai maximal à ne pas dépasser. Lorsque ce délai expire le signal SIGALRM est envoyé, celui-ci fera échouer tout appel bloquant en remplissant la variable errno avec la valeur EINTR. Pour ce faire le signal doit être capturé par un gestionnaire de signale installé sans l'option SA_RESTART.
Ce mécanisme permet donc de placer des délais ("timeouts") sur des appels système bloquants. Attention cependant, sur des systèmes très chargé, le délai programmé peut expiré avant même le début de l'appel système lui-même, celui-ci pourra alors rester bloqué indéfiniment. Pour être sur d'imposé notre délai à un appel système utilisera les saut non locaux comme l'illustre le programme ci-dessous. Pour simplifier la syntaxe on utilisera l'appel à signal plutôt que sigaction.

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

sigjmp_buf contexte;

void gestionnaire_alarme(int sig_num)
{
     siglongjmp(contexte, 1);

     /*Le saut à raté si on en est la, on préviens et on quitte*/
     fprintf(stdout, "Le saut à échoué on quitte !!\n");
     fflush(stdout);
     raise(SIGKILL);
}

int main()
{
     int i;
     char ligne[80];

     signal(SIGALRM, gestionnaire_alarme);

     fprintf(stdout, "Entrez un nombre, vous avez 8 secondes\n");
     fflush(stdout);

     if( sigsetjmp(contexte,1) == 0)
     {
            alarm(8);

            while(1)
            {
                  if(fgets(ligne, 1024, stdin)!=NULL)
                  {
                        if(sscanf(ligne, "%d", &i) == 1)
                              break;
                  }

                  fprintf(stdout, "Entre un entier SVP !!!!!\n");

                  fflush(stdout);
            }
            /*Si on est là, tout s'est bien passé .. on arrete
l'alarme */
            alarm(0);
            fprintf(stdout, "Merci\n");
            fflush(stdout);

     }
```

```
        else
        {
                fprintf(stdout, "Il est trop tard ...\n");
                fflush(stdout);
                exit(1);
        }
  return 0;
  }
```

Ceci représente une gestion de délai fiable fonctionnant avec n'importe quelle fonction de bibliothèque ou appel système risquant de bloquer indéfiniment.

NOTE : Le dernier risque, qui n'à pas été traité ici est que SIGALRM se produise pendant l'exécution du gestionnaire, on se préviendra facilement de ce cas en implémenta,t le tous à la manière Posix.1 et en plaçant SIGALRM dans le sa_mask du gestionnaire.