Notes sur les signaux Unix

Licence 3^{ieme} anne, Bruno Jacob

23 Septembre 2004

1 Introduction

Ces notes portent sur les interruptions logicielles qui peuvent être réalisées par les signaux Unix. Nous ne présentons ici que les informations relatives à la version *System V.4* d'Unix que nous avons à l'IUP. Le mécanisme de traitement de ses interruptions permet, pour un processus, de choisir un comportement spécifique à chaque signal qu'il reçoît.

2 Liste des signaux

On peut la trouver dans le fichier `/usr/include/sys/signal.h`. Le tableau 1 montre les signaux les plus utilisés avec leur comportement qui leur sont attachés par défaut .

N^o	Nom	Évènement	Comportement par défaut
2	SIGINT	interruption : <int>	terminaison
3	SIGQUIT	interruption : <quit>	"core dumped"
9	SIGKILL	terminaison	terminaison
10	SIGBUS	bus error	"core dumped"
11	SIGSEGV	violation mémoire	"core dumped"
14	SIGALRM	alarme horloge	terminaison
16	SIGUSR1	signal 1 pour les utilisateurs	terminaison "User signal 1"
17	SIGUSR2	signal 2 pour les utilisateurs	terminaison "User signal 2"
23	SIGSTOP	demande de suspension	suspension
25	SIGCONT	demande de reprise de processus suspendu	ignorance

TAB. 1 – liste de quelques signaux

Avec, pour le comportement par défaut :

- terminaison : le processus est arrêté

- "core dumped" : le processus est arrêté et une image mémoire (fichier core) est créée
- ignorance : le signal est sans effet
- suspension : le processus est mis en sommeil

Pour des raisons de lisibilité et de portabilité de vos applications, il est recommandé de désigner les signaux par leur nom plutôt que leur numéro.

On peut également savoir la liste des signaux disponibles sur une machine en appelant la commande shell `kill -l`. Par exemple, sur les stations Sun de l'IUP nous avons :

```
prompt> kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM
TERM USR1 USR2 CLD PWR WINCH URG POLL STOP TSTP CONT TTIN TTOU
VTALRM PROF XCPU XFSZ WAITING LWP FREEZE THAW LOST RTMIN
RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
```

3 Mécanisme

3.1 Fonctionnement général

Le mécanisme de prise en compte des signaux se situe dans le bloc de contrôle d'un processus. Il comporte 3 vecteurs de `NSIG` bits si le système peut disposer de `NSIG` signaux (figure 1). Le bloc de contrôle permet de prendre en compte un signal délivré au processus et d'associer cette réception à une action.

Délivrance d'un signal : cela correspond à avoir le bit correspondant à ce signal mis à 1 dans le vecteur des signaux reçus.

Association d'une action à un signal : l'action à réaliser lors de la réception d'un signal est définie dans le vecteur des "handlers". Il contient l'adresse d'une fonction. On appelle aussi cette fonction *fonction de déroutement* ou *handler*. L'association de l'action et du handler est réalisée par le fait qu'un signal à le même indice dans le vecteur des signaux reçus et dans le vecteur des handlers.

Masquage d'un signal : cela correspond à positionner un 1 dans le bit correspondant à ce signal dans le vecteur des signaux bloqués (ou masque). Pour prendre en compte un signal, on effectue d'abord un "ET logique" entre le vecteur des signaux reçus et le masque. Si le résultat est à zéro, alors on n'exécute pas la fonction de déroutement.

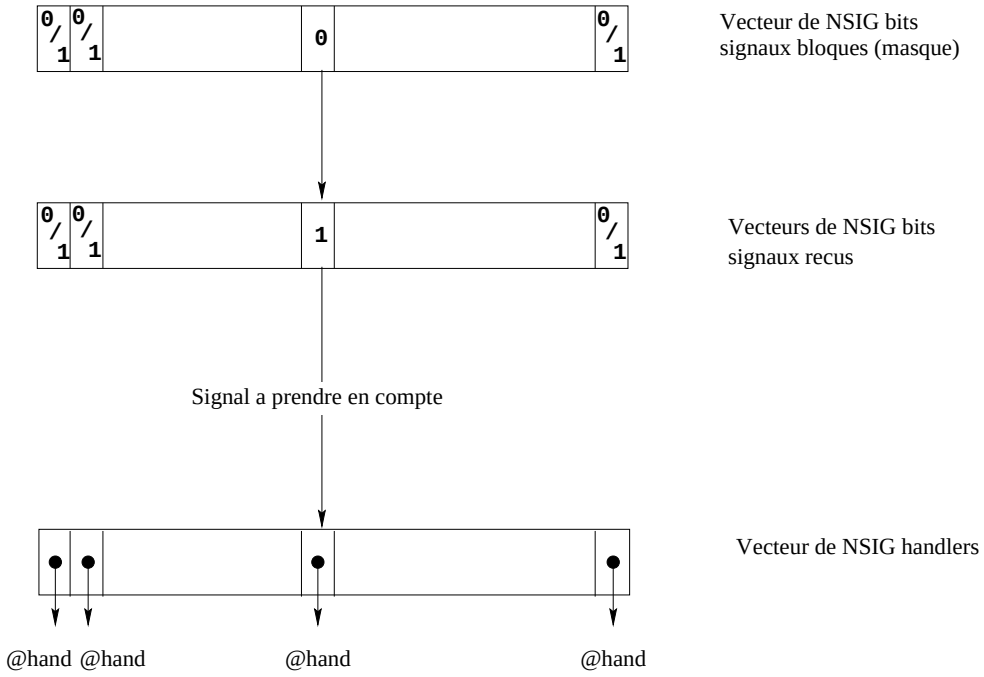

FIG. 1 – Bloc de contrôle d'un processus

Le comportement "normal" de ce mécanisme est d'exécuter le handler qui correspond au signal puis, au retour du handler, l'exécution du processus "principal" reprend au point où il a été interrompu.

3.2 Terminologie

Il y a deux étapes bien distinctes dans une communication par signaux entre deux processus

- l'envoi du signal par le processus émetteur
- la prise en compte du signal par le processus récepteur

La durée séparant ces deux évènements n'est pas prévisible. Un processus prend en compte les signaux qu'il a reçu au moment où il passe en mode "élu" ou s'il est "en sommeil".

Un signal peut être :

pendant : un signal a été envoyé au processus mais pas encore pris en compte. Il est mémorisé dans le bloc de contrôle du processus.

délivré : le signal est pris en compte. Le processus réalise l'action correspondant à ce signal.

bloqué : la prise en compte du signal est différée jusqu'à ce que le signal ne soit plus bloqué.
NB : les signaux `SIGKILL`, `SIGSTOP` et `SIGCONT` ne peuvent pas être bloqués.

ignoré : le signal est délivré mais le bit correspondant dans le bloc de contrôle est remis à zéro.

masqué : c'est la même chose que "bloqué", le bit correspondant au signal est à 1 dans le masque.

4 Les fonctions de déroutement ou handlers

Ce sont les fonctions à réaliser à la délivrance d'un signal. Ce sont obligatoirement des fonctions qui ont la déclaration `C` suivante :

```
void handler(int signal)
```

Une telle fonction ne renvoie rien et le système lui affecte automatiquement en paramètre le numéro du signal qui l'a activée.
Deux handlers prédéfinis ont un rôle particulier :

1. `SIG_DFL` : il associe le traitement par défaut au signal (voir le tableau 1)
2. `SIG_IGN` : il permet d'ignorer le signal correspondant

Il est possible à un utilisateur de définir ses propres handlers. Dans ce cas on dit que le signal est **capté** par l'utilisateur.

5 Primitives d'envoi de signal

La primitive la plus simple est `kill`. Elle a en principe le même comportement dans les différentes versions d'Unix. Elle permet d'envoyer un signal à un processus particulier.

```
#include <sys/types.h>
#include <signal.h>

int kill( pid_t pid,  /* identificateur du processus */
          int   sig); /* numero du signal            */
```

La fonction retourne 0 en cas de réussite et -1 en cas d'échec.

L'utilisateur peut récupérer l'identificateur du processus qui exécute un programme en mettant dans son code la primitive :

```
#include <unistd.h>

pid_t getpid(void);
```

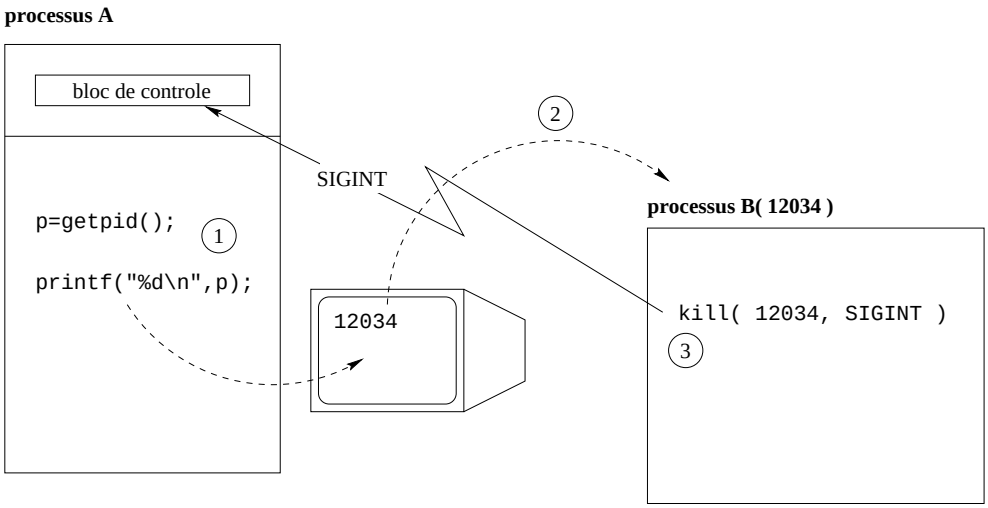

FIG. 2 – Utilisation de kill et getpid

La figure 2 montre comment utiliser ces deux primitives. Il existe d'autres primitives qui sont davantage adaptées à l'envoi d'un signal à un ensemble de processus telles que `sigsend` ou `sigsendset`. Si vous en avez besoin, faire un `man sigsend` ou demandez moi des informations supplémentaires.

6 Primitives de blocage des signaux

Il est possible, dans la version du système que nous avons à l'IUP, de masquer des signaux, c'est à dire d'en différer leur prise en compte. Si un tel mécanisme n'existait pas, alors l'exécution du handler d'un signal `S` pourrait être interrompue par ce même signal `S`. Avec le mécanisme dont nous disposons, cette protection peut être demandée explicitement ou implicitement.

Il est clair que les primitives qui bloquent les signaux ne font que manipuler le masque situé dans le bloc de contrôle du processus. Elles manipulent donc un (ou des) ensemble(s) de signaux. Nous allons donc tout d'abord nous pencher sur les primitives de création des ensembles de signaux.

6.1 Les ensembles de signaux

Un ensemble de signaux est représenté par le type `sigset_t`. Dans notre système cela correspond à une suite de 4 entiers pour coder 44 signaux. La présence d'un signal donné dans un ensemble est codée par 1 : dans la suite

de bits, celui qui correspond à ce signal est mis à "1". Les primitives de manipulation des ensembles de signaux sont regroupées dans le tableau 2.

Fonction	Effet
`sigmask(n)`	Donne le masque du signal n
`sigemptyset(S)`	$S = \emptyset$
`sigaddset(S,n)`	$S = S \cup n$
`sigdelset(S,n)`	$S = S - n$
`sigismember(S,n)`	Vrai ssi $n \in S$
`sigorset(S1,S2)`	$S1 = S1 \cup S2$
`sigandset(S)`	$S1 = S1 \cap S2$
`sigdiffset(S)`	$S1 = S1 - S2$

TAB. 2 – Macro-définitions des fonctions de construction des masques

6.2 Primitives de manipulation des masques

6.2.1 Primitive générale

Définition :

```
#include <signal.h>

int  sigprocmask( int              op,
                  const sigset_t   * p_ens,
                  sigset_t         * p_ens_ancien);
```

Paramètres :

- *Données* :
  - `op` : operation à faire sur `M` = masque du processus appelant
    - `SIG_BLOCK` → `M` = `M` + `p_ens`
    - `SIG_UNBLOCK` → `M` = `M` - `p_ens`
    - `SIG_SETMASK` → `M` = `p_ens`
  - `p_ens` : pointeur sur une structure qui contient le masque à prendre en compte. Si ce pointeur est `NULL`, alors la primitive est sans effet.
- *Résultat* :
  - `p_ens_ancien` : pointeur sur une structure dans laquelle la primitive affecte le masque qui existait avant l'appel. Si ce pointeur était à `NULL` avant l'appel, on ne récupère pas la valeur antérieure du masque.

Retour :

- 0 en cas de succès
- -1 en cas d'échec

6.2.2 Les autres

Il existe d'autres primitives telles que :
- `sighold` : bloque un signal donné
- `sigrelse` : débloque un signal donné
- `sigpause` : débloque un signal donné et met le processus en attente de réception d'un signal
- `sigsuspend` : installe un nouveau masque puis attend l'arrivée d'un signal n'appartenant pas aux signaux masqués. Le masque initial est restauré au retour.
- `sigpending` : donne l'ensemble des signaux pendants.

Faites un `man` pour obtenir plus d'informations sur ces primitives.

7 Primitives de manipulation des handlers

7.1 La primitive "`signal`"

7.1.1 Définition

À l'origine du système, la manipulation des handlers se faisait par la seule primitive `signal`

```
#include <signal.h>

void (*signal ( int sig,           /* numero du signal       */
                void (*hand)(int) /* handler a lui associer */
              ))(int);
```

Cette primitive fait de la fonction `void hand(int)` le nouveau handler du signal `sig` pour le processus appelant.

7.1.2 Remarques

- le signal n'est pas masqué, c'est à dire que le handler est interruptible par le même signal
- le handler `SIG_DFL` est réinstallé avant l'exécution du handler. Si on veut que le handler soit permanent, alors il faut le réinstaller à l'intérieur du handler
- les appels systèmes sont interrompus (retour -1 et `errno = EINTR`).

7.1.3 Exemple

LISTING

Le listing suivant montre le fonctionnement de `signal`.

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

static
void hand( int sig )
{
      static int n = 0 ;

      n++ ;
      printf("Entree dans handler (appel %d)\n",n);
      printf("   Signal recu (appel %d) = %d\n",n,sig);
      sleep(2);
      printf("Sortie de handler (appel %d)\n",n);
}

int
main( )
{
      int n = 0 ;
      char c ;


      signal(SIGINT,hand);

      printf("Entrez n\n");
      n = read(0,&c,1);
      printf("Main : n=%d\n",n);

      printf("Entrez n\n");
      n = read(0,&c,1);
      printf("Main : n=%d\n",n);

      exit(0);
}
```

EXÉCUTION 1

```
prompt> exo_sig1
Entrez n                        <Control-C>
Entree dans handler (appel 1)
   Signal recu (appel 1) = 2  <Control-C>
```

```
prompt>
```

Explications :

- au premier `Control-C` (envoie de `SIGINT`) le signal est capté, le handler par défaut est réinstallé puis le handler est exécuté
- le deuxième `Control-C`, qui arrive pendant l'exécution du handler, active le handler qui est rattaché à ce signal = le handler par défaut qui arrête le processus appelant

EXÉCUTION 2

```
prompt> exo_sig1
Entrez n                  <Control-C>
Entree dans handler (appel 1)
   Signal recu (appel 1) = 2
Sortie de handler (appel 1)
Main : n=-1
Entrez n                            <Control-C>
prompt>
```

Explications :

- au premier `Control-C` le signal est capté et l'exécution du handler n'est pas interrompue. L'appel système `read` est lui aussi interrompu et renvoie -1.
- au deuxième `Control-C`, étant donné que le signal n'a pas été réinstallé dans le handler, le signal n'est plus capté et c'est son handler par défaut qui est exécuté, c'est à dire l'arrêt du processus appelant.

7.2 La primitive "sigset"

7.2.1 Définition

Cette primitive fonctionne comme `signal` mais, en plus, elle masque le signal capté.

```
void (*sigset( int sig,
               void (*disp)(int)
             ))(int);
```

7.2.2 Exemple

Nous reprenons le listing précédent, en ne change que
- "`signal(SIGINT,hand) ;`" par
- "`sigset(SIGINT,hand) ;`"

dans le `main`.

EXÉCUTION

```
prompt> exo_sig2
Entrez n                          <Control-C>
Entree dans handler (appel 1)
   Signal recu (appel 1) = 2      <Control-C>
Sortie de handler (appel 1)
Entree dans handler (appel 2)
   Signal recu (appel 2) = 2
Sortie de handler (appel 2)
Main : n=-1
Entrez n                          <Control-C>
Entree dans handler (appel 3)
   Signal recu (appel 3) = 2      <Control-C>
Sortie de handler (appel 3)
Entree dans handler (appel 4)
   Signal recu (appel 4) = 2
Sortie de handler (appel 4)
Main : n=-1
prompt>
```

Explications

1. au premier `Control-C` le signal est capté et masqué.
2. le deuxième `Control-C` n'est pas pris en compte car `SIGINT` est masqué : l'exécution de son handler est différée
3. la première exécution du handler terminée, le signal est débloqué, ce qui permet à la deuxième exécution du handler de démarrer.
4. une fois la deuxième exécution du handler effectuée, l'appel système `read`, qui était interrompu, renvoie -1
5. la suite veut montrer que le handler de l'utilisateur est toujours installé.

7.3 La primitive générale

7.3.1 Parenthèse sur la norme POSIX

Pour atteindre une portabilité maximum au niveau des systèmes temps-rèel, une norme a été créée : POSIX ou *Portable Operating System Interface*. Cette norme propose la spécification d'un ensemble de fonctions permettant de solliciter les services de base d'un système d'exploitation. L'objectif est de garantir le développement d'applications portables au niveau du code source, entre les systèmes d'exploitation conformes à la norme.

Le but de cette norme est de masquer les spécificités des systèmes sous-jacents.

POSIX fournit des moyens, sous forme de bibliothèques, pour rendre portables des applications écrites pour un système d'exploitation donné, et de rendre l'écriture d'un programme plus simple.

7.3.2 Définition de la primitive `sigaction`

La norme POSIX propose la primitive `sigaction` regroupant l'ensemble des fonctionnalités des différentes primitives de manipulation des handlers.

```
#include <signal.h>

int sigaction( int                          sig,
               const struct sigaction  * p_action,
               struct sigaction        * p_action_ancienne );
```

Définition de la structure `struct sigaction` :

```
struct sigaction
{
  int sa_flags;         /* options pour prise en compte           */
  void (*_handler)();   /* handler de signal                      */
  sigset_t sa_mask;     /* signaux a masquer a la prise en compte */
};
```

Le champ `sa_flags` est une combinaison de valeurs parmi lesquelles on trouve :

- `SA_RESETHAND` : le handler par défaut est réinstallé à la prise en compte
- `SA_RESTART` : appels systèmes repris après l'interruption
- `SA_NODEFER` : signal non bloqué automatiquement pendant l'exécution du handler

7.3.3 Exemple

L'exemple suivant illustre les différentes utilisations de la primitive générale de manipulation des signaux et des handlers.

LISTING :

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

static
void hand_int( int sig )
{
     static int n = 0 ;

     n++ ;
     printf("Entree dans handler INT (appel %d)\n",n);
     printf("   Signal recu (appel %d) = %d\n",n,sig);
     sleep(2);
     kill( getpid() , SIGUSR1 );
     kill( getpid() , SIGUSR2 );
     sleep(3);
     printf("Sortie de handler INT (appel %d)\n",n);
}

static
void hand_usr1( int sig )
{
     static int n = 0 ;

     n++ ;
     printf("Entree dans handler USR1 (appel %d)\n",n);
     printf("   Signal recu (appel %d) = %d\n",n,sig);
     sleep(3);
     printf("Sortie de handler USR1(appel %d)\n",n);
}

static
void hand_usr2( int sig )
{
     static int n = 0 ;

     n++ ;
     printf("Entree dans handler USR2 (appel %d)\n",n);
     printf("   Signal recu (appel %d) = %d\n",n,sig);
     sleep(5);
```

```
      printf("Sortie de handler USR2(appel %d)\n",n);
}

static
void hand_quit( int sig )
{
      static int n = 0 ;

      n++ ;
      printf("Entree dans handler QUIT (appel %d)\n",n);
      printf("   Signal recu (appel %d) = %d\n",n,sig);
      sleep(3);
      printf("Sortie de handler QUIT(appel %d)\n",n);
}

int
main( )
{
      int n = 0 ;
      char c ;
      struct sigaction action ;


      /* Capture du signal INT */
      action.sa_flags = SA_RESTART ;
      action.sa_handler = hand_int ;
      sigemptyset( &action.sa_mask ) ;
      sigaddset( &action.sa_mask , SIGUSR1 );
      sigaction(SIGINT, &action , NULL );

      /* Capture du signal USR1 */
      action.sa_flags = SA_RESETHAND | SA_NODEFER ;
      action.sa_handler = hand_usr1 ;
      sigemptyset( &action.sa_mask ) ;
      sigaction(SIGUSR1, &action , NULL );

      /* Capture du signal USR2 */
      action.sa_flags = 0 ;
      action.sa_handler = hand_usr2 ;
      sigemptyset( &action.sa_mask ) ;
      sigaction(SIGUSR2, &action , NULL );

      /* Capture signal QUIT */
      action.sa_flags = 0 ;
      action.sa_handler = hand_quit ;
      sigemptyset( &action.sa_mask ) ;
      sigaction(SIGQUIT, &action , NULL );

      printf("Entrez n\n");
```

```
    n = read(0,&c,1);
    printf("Main : retour de read, n=%d\n",n);

    kill( getpid() , SIGUSR1 );

    while(1) ;
    exit(0);
}
```

EXÉCUTION

```
prompt> exo_sigaction                (numero de pid 4079)
Entrez n                             <Control-C>
Entree dans handler INT (appel 1)
   Signal recu (appel 1) = 2
Entree dans handler USR2 (appel 1)
   Signal recu (appel 1) = 17
Sortie de handler USR2(appel 1)
Sortie de handler INT (appel 1)
Entree dans handler USR1 (appel 1)
   Signal recu (appel 1) = 16
Sortie de handler USR1(appel 1)
Entree dans handler QUIT (appel 1)
   Signal recu (appel 1) = 3
Sortie de handler QUIT(appel 1)
Main : retour de read, n=-1         <kill -QUIT 4079>
User signal 1
prompt>
```

Explications

1. Les signaux `SIGINT`, `SIGQUIT`, `SIGUSR1`, `SIGUSR2` sont captés
2. Le handler de `SIGINT` est activé par le premier `Control-C` au cours de l'exécution de l'appel système `read`. L'appel système est repris au retour du handler (option `SA_RESTART`) ;
3. Dans le handler de `SIGINT`, le processus s'envoie les signaux `SIGUSR1` et `SIGUSR2` ;
   - `SIGUSR1` est masqué dans `hand_int` : il n'interrompt pas `hand_int`, l'exécution de `hand_usr1` est différée à la fin de `hand_int`
   - `SIGUSR2` n'est pas masqué dans `hand_int` : il interrompt `hand_int` et exécute `hand_usr2`
4. `hand_usr2` se termine → reprise là où on s'était arrêté dans `hand_int`.

Puis `hand_int` se termine → `hand_usr1` peut alors s'exécuter. À la sortie de `hand_usr1`,
   – le handler `SIG_DFL` de `SIGUSR1` est réinstallé (option `SA_RESETHAND`).
   – l'appel système `read` est repris

5. À la réception du signal `SIGQUIT` (commande shell "`kill -QUIT 4079`"), `hand_quit` est activé. Au retour de `hand_quit`, l'appel système n'est pas repris (pas d'utilisation de `SA_RESTART`), d'où la valeur de retour de `read` à -1 ;
6. Pour finir, le processus s'envoie le signal `SIGUSR1` qui active son handler par défaut (terminaison du processus avec affichage du message "User signal 1").

8 Primitives de reprises d'exécution

Dans le cas général, nous avons dit qu'au retour du handler, l'exécution du processus reprenait au point où le processus avait été interrompu.

Dans certains cas, l'exécution doit être reprise en un point différent. Les étiquettes du `C` ayant une portée locale aux fonctions dans lesquelles elles ont été définies, la bibliothèque standard `C` permet la gestion d'étiquettes globales.

Ces étiquettes globales sont réalisées par le type `jmp_buf`. Ce type contient un contexte d'exécution d'un processus.

8.1 Création d'une étiquette globale

Pour affecter le contexte d'exécution du processus courant à une variable de type `jmp_buf`, on fait appel à la fonction :

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

8.2 Branchement sur une étiquette globale

La fonction `longjmp` permet de reprendre l'exécution du processus à partir d'un contexte sauvegardé dans une variable de type `jmp_buf`.

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

Cette fonction restitue le contexte sauvegardé dans `env` et renvoie `val`. Les variables auront donc les valeurs qu'elles avaient au moment de `setjmp`.

8.3 Précaution d'utilisation

Si `longjmp` est dans un handler et si des signaux sont bloqués lors de son exécution, alors ces signaux ne sont pas débloqués automatiquement quand on sort du handler. Il est nécessaire, avant l'appel à `longjmp`, de débloquer les signaux manuellement (par `sigrelse(sig)`).

8.4 Exemple 1

Cet exemple montre l'uitilisation de `setjmp/longjmp` dans le cas de signaux non bloqués (utilisation de `signal` pour capter `SIGINT` et `SIGQUIT`). `longjmp` est située dans un handler et renvoie le numéro du signal qui a activé le handler. `setjmp` renvoie alors :

- `0` à l'initialisation, quand on créer l'étiquette globale
- `SIGINT` quand on sort du handler si celui ci est activé par le signal `SIGINT`.
- `SIGQUIT` quand on sort du handler si celui ci est activé par le signal `SIGQUIT`.

8.4.1 Listing

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

jmp_buf env ;
static int n ;

static
int f( int * pi )
{
      n++ ;
      (*pi)++ ;
      return(0) ;
}
```

```
static
void hand( int sig )
{
     signal(SIGINT,hand);
     longjmp(env,sig);
}

int
main( )
{
     int i = 0 ;

     signal(SIGINT,hand);
     signal(SIGQUIT,hand);
     while(1)
     {
          switch(setjmp(env))
          {
             case 0 :
                  while(1)
                  {
                       f(&i);
                       sleep(5);
                  }

             case SIGINT :
                  printf("Signal SIGINT recu : n=%d i=%d\n", n , i );
                  break;

             case SIGQUIT :
                  printf("Signal SIGQUIT recu : n=%d i=%d\n", n , i );
                  exit(0);

             default :
                  printf("Signal recu inconnu : n=%d i=%d\n", n , i );
                  exit(-1);
          }
     }
}
```

8.4.2 Exécution

```
prompt> exo_jmp &
[1]                              <kill -INT %1>
Signal SIGINT recu : n=1 i=1     <kill -INT %1>
Signal SIGINT recu : n=2 i=2     <kill -INT %1>
```

```
Signal SIGINT recu : n=3 i=3   <kill -QUIT %1>
Signal SIGINT recu : n=5 i=5
```

8.5 Exemple 2

Cet exemple montre que la sortie du handler par `longjmp` n'est pas un vrai retour de fonction : les signaux initialement masqués pour juste l'exécution du handler ne sont pas débloqués avec une sortie par `longjmp`.

Le code suivant reprend le listing de l'exemple 1 mais utilise `sigset` pour bloquer les signaux. Le premier envoi du signal `SIGINT` est pris en compte mais pas les suivants car `SIGINT` reste masqué.

8.5.1 Listing

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

jmp_buf env ;
static int n ;

static
void affiche_masque( sigset_t masque )
{
     int i ;

     for( i=0 ; i<4 ; i++ )
          printf("%u" , masque.__sigbits[i] );
}

static
int f( int * pi )
{
     n++ ;
     (*pi)++ ;
     return(0) ;
}

static
void hand( int sig )
{
     /* sigrelse(sig); */
     longjmp(env,sig);
}
```

```
int
main( )
{
      int i = 0 ;
      sigset_t set ;


      sigset(SIGINT,hand);
      sigset(SIGQUIT,hand);

      sigprocmask(SIG_BLOCK, NULL, &set);
      printf("Masque initial du processus %ld = ", (long)getpid() );
      affiche_masque(set);
      printf("\n");

      while(1)
      {
            switch(setjmp(env))
            {
               case 0 :
                     while(1)
                     {
                           f(&i);
                           sleep(5);
                     }

               case SIGINT :
                     printf("Signal SIGINT recu : n=%d i=%d\n", n , i );

                     sigprocmask(SIG_BLOCK, NULL, &set);
                     printf("Masque du processus %ld apres envoi SIGINT = ", (long)getpi
                     affiche_masque(set);
                     printf("\n");

                     break;

               case SIGQUIT :
                     printf("Signal SIGQUIT recu : n=%d i=%d\n", n , i );
                     exit(0);

               default :
                     printf("Signal recu inconnu : n=%d i=%d\n", n , i );
                     exit(-1);
            }
      }
}
```

8.5.2 Exécution

```
promt> exo_jmp2 &
[1] 21463
Masque initial du processus 21463 = 0000
                                                <kill -INT %1>
Signal SIGINT recu : n=3 i=3
Masque du processus 21463 apres envoi SIGINT = 2000
                                                <kill -INT %1>
                                                <kill -INT %1>
                                                <kill -INT %1>
                                                <kill -INT %1>
                                                <kill -QUIT %1>
Signal SIGQUIT recu : n=7 i=7
```

À part le premier, la série de "`kill -INT %1`" reste sans effet, il vaut mieux débloquer le signal par `sigrelse` avant `longjmp`.