La gestion des signaux

Les signaux constituent un mécanisme à la fois simple et puissant de contrôle des processus. Le plan de ce cours est le suivant :

1. Définition des signaux
2. L'envoi de signaux
3. La programmation des signaux

Qu'est qu'un signal ?

Un signal est un message envoyé à un processus. Le corps du message est en fait très simple puisqu'il est constitué en tout et pour tout d'un entier indiquant le type du signal. Typiquement, l'arrivée d'un signal interrompt, plus ou moins brutalement l'exécution du processus qui le reçoit.

Pour la plupart des signaux, vous pourrez redéfinir la procédure de réponse par défaut en installant un gestionnaire. Toutefois certains signaux ne peuvent être déviés de leur signification première. Tout processus peut envoyer un signal à un autre processus, à partir du moment où il connaît son PID. Toutefois, il faut bien se rendre compte que la plupart des signaux sont émis par le système pour signaler tel ou tel évènement, par exemple, une segmentation violation.

Comment émettre un signal vers un autre processus

Sans le savoir, vous avez certainement déjà utilisé les signaux ... en effet, tapper `^Z` au clavier pour interrompre un processus ne fait qu'envoyer à celui-ci un signal d'interruption, de la même manière que les commandes de *job control* `fg` et `bg` envoient au processus concerné un signal de reprise d'exécution.

Plus trivial encore, la commande `kill` que vous utilisez pour tuer un programme récalcitrant est en fait destinée à envoyer *tout type de signal*. Vous voyez, je vous l'avais bien dit, les signaux font partie de la vie quotidienne de l'utilisateur Unix.

Dans la suite de ce texte, je vais vous demander de bien différencier la commande `kill` que vous tappez dans une fenêtre shell ou incluez dans un script de la fonction C `kill`, destinée elle à la programmation.

La commande `kill`

Nous parlons ici de la commande shell `kill` et non pas de la fonction C traitée dans une section ultérieure.

La commande `kill` a deux fonctions principales :

1. Obtenir la liste des signaux disponibles sur le système
2. Envoyer un signal à un ou plusieurs processus

Nous détaillons maintenant ces deux fonctionnalités.

Obtenir la liste des signaux disponibles

La commande `kill -l` permet d'obtenir la liste complète des signaux disponibles sur la machine.

```
bipro: kill -l HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM URG STOP TSTP CONT CHLD TTIN TTOU IO XCPU
XFSZ MSG WINCH PWR USR1 USR2 PROF DANGER VTALRM MIGRATE PRE GRANT RETRACT SOUND SAK
```

Dans le cas de cet ordinateur (un biprocesseur tournant sous AIX), la liste des signaux disponibles est pour le moins impressionnante. Vous remarquerez que `kill -l` renvoie une liste de constantes symboliques. Ces constantes pourront être réutilisées avec la deuxième forme de `kill` destinée, elle, à envoyer un signal à un ou plusieurs processus.

Pour terminer avec cette liste, ajoutons que certains noms de signaux sont normalisés POSIX. Ce sont les signaux les plus courament utilisés.

Envoyer un signal

La syntaxe générale pour envoyer un signal à un processus est la suivante :

```
kill -identificateur de signal liste de processus
```

Ce qui a pour effet d'envoyer le signal identifié, soit par une constante symbolique (celle-là même renvoyée par la commande `kill -l`), soit par sa valeur numérique, à la liste des processus spéficiés.

Les processus peuvent être désignés, soit par leur PID soit par leur numéro de job précédé du caractère `%`.

Quelques signaux usuels

Nous donnons ici quelques indications sur certains des signaux les plus souvent utilisés. Tous les noms présents dans le tableau sont normalisés POSIX. Insistons également sur le fait que si les noms des constantes sont normalisés, les valeurs associées, elles, ne le sont pas. Plus que jamais, il convient d'utiliser les noms des constantes symboliques et non pas leur valeur numérique.

TERM	Terminaison propre d'un processus. Avant la mort effective du processus, les tampons sont vidés et les fonctions déclarées à l'aides de `atexit` sont appelées.	CONT	Envoyé à un processus suspendu pour qu'il reprenne l'exécution (non interceptable)
HUP	Envoyé par un shell à tous les programmes qu'il a lancé pour les prévenir de sa mort. Selon le type de shell, la réponse sera différente. En particulier avec les shells de la filiation Bourne Shell, les processus fils sont sensés mourir par défaut. Du coup, il sera sans doute nécessaire de lancer à l'aide de la commande `nohup` les processus en tâche de fond qui doivent perdurer après la mort du shell qui les a lancés.	STOP	Envoyé à un processus pour suspendre son exécution (non interceptable). Le statut du programme devient « suspended »
KILL	Envoyé à un processus pour le tuer sans sommation. La mort du processus est des plus violentes : aucun tampon n'est vidé, aucune procédure de terminaison (déclarée par `atexit`) n'est appelée. Ce signal est non interceptable.	SEV	Envoyé à un processus qui vient de commettre une faute mémoire. La réponse habituelle est la sortie du programme.
INTR	Frappe du caractère interruption ^C. D'ordinaire, un programme qui reçoit ce signal est sensé mourir proprement.	QUIT	Frappe du caractère « quitter » non présent sur tous les claviers

Quelques signaux usuels

Il est toujours possible d'obtenir la liste de tous les signaux reconnus par un système à l'aide de la commande `kill -l`.

Vous aurez remarqué que certains messages sont marqués *non interceptables*. Ceci signifie que vous ne pourrez détourner la réponse par défaut à ce signal ni même le bloquer (voir les sections blocage des signaux, gestionnaires de signaux). En particulier, le signal `KILL`, plus connu sous sa constante numérique toujours égale à 9 (l'odieux `kill -9` vous êtes sur que ca ne vous dit rien ?) ne peut être intercepté, ce qui peut avoir des conséquences

dramatiques sur la stabilité du système.

La fonction `kill`

Nous parlons ici de la fonction C `kill`, celle que vous rencontrerez dans des programmes, et non pas de la commande `kill` invoquée dans une fenêtre ou un script shell, laquelle a déjà été commentée précédemment.

Le prototype de la fonction `kill` (d'après le fichier `signal.h`) est le suivant :

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

... lequel se passe de commentaires !

Bien entendu, les signaux peuvent être désignés par une constante numérique, laquelle, à l'instar de toutes les fonctions travaillant avec des signaux, est définie dans `signal.h`. L'on s'attendrait à ce que le nom soit identique à celui utilisé par la version en ligne de commande de `kill`. Et bien, c'est presque le cas ! Toutefois, comme les noms des signaux sont très usuels, la constante programmatique a été prefixée par les lettres `SIG`. Par exemple, si `kill -l` vous a renvoyé les noms de signaux `KILL, BUS, CONT`, les constantes programmatiques associées seront respectivement `SIGKILL, SIGBUS, SIGCONT`. Quoi qu'il arrive, rien ne remplace le zieutage approfondi du fichier `signal.h` de votre système.

La programmation des signaux

Hormis la fonction `kill`, permettant d'envoyer un signal vers un processus dont on connaît le PID, il existe de nombreuses autres fonctions permettant de manipuler les signaux. Les opérations disponibles sont les suivantes :

1. Introduction
2. Blocage (et levée de blocage) de signaux
    1. Définition du type `sigset_t`
    2. Blocage de signaux
    3. Levée de blocage
3. Association d'un gestionnaire à un signal
    1. Définition du prototype d'un gestionnaire
    2. La structure sigaction

3. Mise en place et suppression d'un gestionnaire
4. Attente d'un signal

Mécanisme des signaux et vocabulaire

La prise en compte d'un signal (on parle de *délivrance*) ne peut avoir lieu que dans une circonstance bien particulière : la bascule du mode système au mode utilisateur. Lorsqu'un signal est envoyé à un processus, plusieurs cas peuvent se produire :

- Le processus est en mode utilisateur. La délivrance devra alors attendre d'abord le passage du processus en mode système puis son retour au mode utilisateur. Pendant tout ce temps, le signal serant *pendant*, c'est à dire en attente de délivrance.
- Le processus est en mode système, par définition non interruptible. Le signal sera délivré dès que le processus reviendra au mode utilisateur. Un signal arrivé sur un processus mais non encore délivré est dit *pendant*. Lorsque le signal est *délivré*, la procédure qui lui est associée (son *gestionnaire* ou *handler*) est appelée
- Le signal peut être *bloqué* (on dit également *masqué*). Ceci signifie que la délivrance du signal est différée jusqu'à ce que le blocage soit levé. Lorsqu'un signal bloqué arrive sur le processus, il n'est pas supprimé, il est juste mis en attente : il devient donc pendant. L'ensemble des signaux bloqués constitue un masque.

La structure de données interne (une par processus) gérant les signaux est un vecteur indexé sur les numéros de signaux et dont chaque case comporte 3 informations :

1. *Un booléen* indiquant si le signal est pendant. Notons que cette information est un booléen unique. Ce qui signifie que si un processus a déjà un signal d'un certain type pendant, il est inutile de lui envoyer à nouveau un signal du même type, celui-ci sera ignoré.
2. *Un booléen* indiquant si les signaux de ce type sont bloqués.
3. Un pointeur désignant le gestionnaire.

Blocage de signaux

Certaines parties de code sont trop critiques pour qu'elles puissent être interrompues par un signal usuel. Aussi, nous allons les protéger en les rendant non interruptibles. Signalons au possage que toutes les instructions priviligiées (c'est-à-dire, les instructions s'exécutant en mode système) sont, par définition, non interruptibles.

Le type `sigset_t`

Toutes les manipulations de blocage de signaux passent par la création d'un masque de signaux, c'est à dire d'un ensemble de signaux codés dans le type `sigset_t`. Bien entendu, ce fameux type n'est habituellement rien d'autre qu'un entier long.

Les fonctions suivantes permettent de le manipuler facilement, en masquant (désolé, je n'ai pas pu vous épargner ce jeu de mot marécageux) l'usage d'opérateurs logiques binaires.

`int sigemptyset(sigset_t *);`	Crée un ensemble de signaux vide
`int sigaddset(sigset_t *, const int);`	Ajouter le signal `signal` à l'ensemble de sigaux `ens`. Théoriquement, il n'est pas dangereux de tenter d'ajouter à nouveau un signal déjà présent. Toutefois, je vous recommande néanmoins d'utiliser la fonction `sigismember` afin de tester cette éventualité.
`int sigdelset(sigset_t *ens, const int signal);`	Retire le signal `signal` de l'ensemble de signaux `ens`. Cette opération peut être relativement catastrophique si le signal spécifié n'était pas dans l'ensemble. Aussi, avant de vouloir retirer un signal d'un ensemble, il faut toujours préalablement s'assurer de sa présence à l'aide de la fonction `sigismember`
`int sigismember (const sigset_t *ens , int signal);`	Renvoie 1 si le signal `signal` est présent dans l'ensemble de sigaux `ens` et 0 sinon
`int sigfillset(sigset_t *ens);`	Ajoute tous les signaux possibles à l'ensemble `ens`

La primitive `sigprocmask`

La primitive permettant de mettre en place un masque de signaux, c'est à dire, un ensemble de signaux bloqués, s'appelle `sigprocmask`, nous donnons ici son prototype :

```
int sigprocmask(int mode, const sigset_t *ens, sigset_t *anciens)
```

Passons en revue les trois paramètres de cette fonction :

Code de retour

0 si tout s'est bien passé, -1 sinon

`mode`

Ce paramètre définit le type de mise en place du masque spécifié par `ens`. En effet, il est possible de :

- Remplacer le masque existant : `mode=SIG_SETMASK`
- Ajouter au masque existant les signaux de `ens` : `mode=SIG_BLOCK`. Une utilisation classique de cette opération est la récupération dans le paramètre `anciens` de la liste des signaux actuellement bloqués, sans pour autant les débloquer, en ajoutant un masque vide.
- Retirer au masque existant les signaux de `ens` : `mode=SIG_UNBLOCK`. Attention ! il est dangereux d'essayer de retirer des signaux non présents !

`ens`

Ensemble des signaux définissant un nouveau masque

`anciens`

A moins que ce paramètre ne soit nul, la structure `sigset_t` pointée contiendra au retour la liste des signaux qui étaient bloqués avant l'opération. Cette fonctionnalité offerte par `sigprocmask` est très intéressante car elle permettra, par exemple, de remettre en place l'ancien masque à la sortie de votre section critique.

Récupérer la liste des signaux pendants bloqués

La primitive suivante permet de connaître la liste des signaux pendants et bloqués. N'oubliez pas que si vous les débloqués, la primitive de gestion sera immédiatement appelée !

```
int sigpending(sigset_t *ens)
```

A l'instar des autres primitives opérant sur les signaux, `sigpending` renvoie 0 si tout s'est déroulé convenablement — auquel cas, la structure pointée par `ens` contient la liste des signaux bloqués et pendants, liste qu'il conviendra de traiter, par exemple, avec `sigismenber` — et -1 dans tous les autres cas.

Nous donnerons un exemple de blocage de signaux après avoir traité la mise en place des gestionnaires dans la section suivante.

Mise en place de gestionnaires de signaux

Dans cette section, vous allez apprendre à redéfinir le comportement par défaut des signaux, c'est à dire, leur associer un gestionnaire.

Rappelons qu'un gestionnaire est une fonction qui est appelée dès la délivrance du signal. Lors de l'appel du gestionnaire, le contexte d'exécution est sauvegardé de manière à ce que le programme reprenne à la bonne instruction dès la fin du gestionnaire (à moins que celui-ci ne stoppe le programme).

Rappelons que certains signaux ne sont ni blocables ni détournables (bien que cela serait particulièrement agréable, surtout pour KILL), ce qui concerne, en particulier KILL, STOP et CONT.

Prototype d'un gestionnaire

Le protype d'un gestionnaire est très simple :

```
void gestionnaire(int sig)
```

Le paramètre entier passé est le numéro du signal. Ceci vous permet de pouvoir traiter plusieurs évènements à l'aide du même gestionnaire.

La structure `struct sigaction`

La primitive sigaction chargée d'associer un gestionnaire à un signal repose sur l'utilisation de la structure `struct sigaction` définie, vous vous y attendez, dans le fichier `<sys/signal.h>`.

Voici la déclaration (commentée) de cette structure :

```
struct sigaction
{
  void     (*sa_handler)(int); /* Adresse du gestionnaire */
  sigset_t sa_mask;            /* Masque des signaux bloques pendant */
                               /* l'exécution du gestionnaire */
  int      sa_flags;           /* Ignore pour l'instant */
};
```

Pour l'instant, la norme POSIX n'a pas encore défini d'utilisation précise du champ `sa_flags`, aussi, sa valeur est ignorée.

Le champ `sa_mask` permet de définir un ensemble de signaux `supplémentaires` bloqués durant l'exécution du gestionnaire. Ces signaux sont *ajoutés* au masque courant, *ils ne le remplacent pas*. En outre, si le gestionnaire indiqué est `SIG_IGN` ou `SIG_DFL`, ce masque est ignoré.

Le champ le plus intéressant reste néanmoins `sa_handler` qui désigne la fonction gestionnaire à installer. Le plus souvent, ce sera l'adresse d'une fonction gestionnaire, mais nous pourrons également trouver deux constantes prédéfinies :

SIG_IGN

Constante spéciale indiquant que le signal doit être ignoré.

SIG_DFL
: Constante spéciale indiquant que le gestionnaire par défaut du signal doit être réinstallé.

Ces deux constantes ne correspondent pas à des adresses réelles mais sont reconnues comme le système qui les prend en compte directement au niveau du noyau.

La primitive `sigaction`

Nous allons maitenant décrire la primitive qui permet d'installer un gestionnaire. Son fonctionnement repose sur l'utilisation de la structure `struct sigaction` définie ci-dessus. Son prototype est le suivant :

```
int sigaction(int sig, const struct sigaction *gestion,
                                               struct sigaction *ancien
```

Examinons ici les paramètres de `sigaction`

sig
: Il s'agit là, bien entendu, du numéro du signal auquel nous allons associer un gestionnaire.

gestion
: Pointeur vers une structure de type `struct sigaction` désignant le gestionnaire ainsi qu'un éventuel masque de signaux à bloquer

anciens
: Si ce paramètre est différent de 0 (`NULL`), alors la structure pointée est renseignée avec l'ancien gestionnaire présent, ce qui permet, par exemple, de le réinstaller ensuite.

Exemple (naïf) de mise en place d'un gestionnaire de signaux

Le code suivant met en place un gestionnaire de signaux très simple : Celui ci se contente d'émettre un message indiquant le numéro du signal reçu. Les signaux `SIGINT`, `SIGQUIT` et `SIGTERM` étant interceptés, il sera possible de tuer ce processus avec un signal `SIGHUP`. Vous devez résister à la tentation du `kill -KILL` ...

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

```
#include <signal.h>


/* Gestionnaire naif se contentant d'indiquer qu'un signal
   a été reçu accompagné de son numéro */
void handler(int theSignal)
{
  printf("Je receptionne le signal %d\n",theSignal);
  fflush(stdout);
}


int main(void)
{
  /* Déclaration d'une structure pour la mise en place des gestionnaires */
  struct sigaction prepaSignal;

  /* Remplissage de la structure pour la mise en place des gestionnaires */
  /* adresse du gestionnaire */
  prepaSignal.sa_handler=&handler;
  /* Mise a zero du champ sa_flags theoriquement ignoré
  prepaSignal.sa_flags=0;
  /* On ne bloque pas de signaux spécifiques */
  sigemptyset(&prepaSignal.sa_mask);

  /* Mise en place du gestionnaire bidon pour trois signaux */
  sigaction(SIGINT,&prepaSignal,0);
  sigaction(SIGQUIT,&prepaSignal,0);
  sigaction(SIGTERM,&prepaSignal,0);

  /* Le programme tourne jusqu'à la Saint GlinGlin : il faudra la tuer
    avec un kill -HUP*/
  while (1) ;

  return 0;
}
```

Voici un exemple de session de travail avec ce programme que nous avons appelé `catcher` :

```
bipro: ./catcher
```

```
^CReception du signal 2

^ZSuspended
bipro: jobs
[1]  + Suspended     ./catcher
bipro: kill -TERM %
bipro: Reception du signal 15

bipro: jobs
[1]  + Suspended     ./catcher
bipro: kill -HUP %
bipro:
[1]    Hangup        ./catcher

bipro: jobs
bipro:
```

Il y a plusieurs choses intéressantes à noter dans ce résultat de programme :

1. La frappe du caractère `^C` n'arrête pas le programme mais renvoie bien au gestionnaire de signal, Ouf !
2. Un programme interrompu (par exemple, par la frappe du caractère `^Z` gère encore correctement les signaux qui lui sont envoyés. Il repasse néanmoins en mode interrompu dès la fin d'exécution du gestionnaire comme le prouvent les exécutions successives de la commande `jobs`.
3. Le signal HUP est correctement exécuté malgrès l'interception de TERM, QUIT, et INT comme en témoigne la mort du processus !

Exemple d'installation de gestionnaire de signaux et de blocage

L'exemple suivant bloque un certain nombre de signaux (lesquels sont placés dans un tableau pour plus de commodité), vérifie lesquels sont pendants, et leur affecte le gestionnaire spécial `SIG_IGN` (permettant d'ignorer un signal) au moment de leur déblocage, évitant ainsi l'appel du gestionnaire par défaut, qui, dans ce cas, aurait mis fin au processus. Une fois le déblocage terminé, les gestionnaires précédents, qui avaient été sauvegardés, sont remis en place. Notez que dans le cas présent, nous aurions pu passer directement `SIG_DEF` pour remettre en place les gestionnaires par défaut, mais nous avons voulu être didactiques !

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

```
/* Définition d'un ensemble de trois signaux */
#define NB_SIGNAUX 3
int signaux[NB_SIGNAUX]={SIGINT,SIGTERM,SIGHUP};

/* Tableaux de structure sigaction pour la sauvegarde
   des gestionnaires en place */
struct sigaction sauvegardes[NB_SIGNAUX];

int main(void)
{
  /* Masques de blocage de signaux */
  sigset_t masque;
  sigset_t anciens;
  sigset_t pendants;

  int i;

  struct sigaction pourIgnorer;

  /* Creation du masque contenant les trois signaux a bloquer
     On commence par créer un masque vide avec sigemptyset
     que l'on remplit ensuite avec sigaddset
  */
  sigemptyset(&masque);
  for (i=0;i<NB_SIGNAUX;i++)
    sigaddset(&masque,signaux[i]);

  /* Mise en place du masque avec sauvegarde de l'ancien masque
     dans la variable anciens */

  sigprocmask(SIG_SETMASK,&masque,&anciens);

  /* On roupille 15 secondes, histoire de faire des misères au processus
     Tapper ^C, envoyer des signaux TERM et HUP ... */
  puts("Delai de grace 15 secondes");
  fflush(stdout);
  sleep(15);
  puts("Fin du delai de grace");
  fflush(stdout);
```

```
  /* On recupere la liste des signaux pendants */
  sigpending(&pendants);

  /* Decodage (bestial) des signaux pendants */
  for (i=1;i<NSIG;i++)
    if (sigismember(&pendants,i))
      printf("Signal %d pendant bloque\n",i);

  /* On installe des gestionnaires << ignorer >> sur notre masque avant le deblocage
       La première étape consiste à remplir les structures sigaction*/

  /* Pas de signaux particulier a bloquer pendant SIG_IGN
     Theoriquement, ce parametre n'est pas pris en compte pour SIG_IGN
     mais on ne sait jamais */
  sigemptyset(&pourIgnorer.sa_mask);

  /* Mise a zero du champ sa_flags, theoriquement il est ignore,
     mais on ne sait jamais */
  pourIgnorer.sa_flags=0;

  /* Pour chaque signal du tableau, on met en place un gestionnaire SIG_IGN
     Sans oublier de sauvergarder l'ancien gestionnaire dans le tableau de
     structures prevu a cet egard. */
  for (i=0;i<NB_SIGNAUX;i++)
    sigaction(signaux[i],&pourIgnorer,sauvegardes+i);

  /* L'ancien masque est remis en place */
  sigprocmask(SIG_SETMASK,&anciens,0);

  /* Ainsi que les anciens gestionnaires */
  for (i=0;i<NB_SIGNAUX;i++)
  {
    pourIgnorer.sa_handler=SIG_IGN;
    sigaction(signaux[i],sauvegardes+i,0);
  }

  return 0;
}
```

L'exécution de ce programme n'est guère spectaculaire, dans le cas présent nous utilisons deux terminaux. L'un pour exécuter le programme et lui envoyer

des caractères `^C`, l'autre pour tenter de lui envoyer des signaux `TERM` ou `HUP`.

Attente de signaux

Nous complètons notre étude des signaux par la possibilité de mettre un processus en attente jusqu'à réception d'un certain signal par la primitive `sigsuspend`.

Autres primitives en relation avec les signaux

monstyle