Chapitre 4 - Les Processus et les signaux

Table des matières

1 - Introduction

1.1 - Les signaux

Un signal est une information qui peut être envoyé à un processus, et modifier l'exécution de ce dernier.

Sur un système donné, on dispose de **NSIG** signaux numérotés de 1 à **NSIG**. La constante **NSIG**, ainsi que les différents signaux et les prototypes des fonctions qui les manipulent, sont définis dans le fichier <signal.h>.

Sous Linux, ce fichier « /usr/include/signal.h » utilise un autre fichier d'entête se nommant « /usr/include/asm/signal.h »

1.2 - Le comportement à la réception d'un signal

Terminologie des signaux

Un signal envoyé par le noyau ou par un autre processus est un signal **pendant** : cet envoi est mémorisé dans le BCP du processus.

Un signal est **délivré** (ou pris en compte) lorsque le processus concerné réalise l'action qui lui est associée dans son BCP , c'est à dire, au choix :

- l'action par défaut : en général la mort du processus,
- ignorer le signal,
- l'action définie par l'utilisateur (handler) : le signal est dit **capté**.

Un signal peut également être **masqué** (ou bloqué) : sa prise en compte sera différée jusqu'à ce que le signal ne soit plus masqué.

Limites des signaux

Lorsqu'un processus est en sommeil et qu'il reçoit plusieurs signaux :

- Aucune mémorisation du nombre de signaux reçus : 10 signaux SIGINT ≡ 1 signal SIGINT.
- Aucune mémorisation de la date de réception d'un signal : les signaux seront traités ultérieurement par ordre de numéro.
- Aucun moyen de connaître le PID du processus émetteur du signal.

La délivrance des signaux

Attention, les signaux sont asynchrones : la délivrance des signaux non masqués a lieu un « certain temps » après leur envoi, quand le processus récepteur passe de l'état actif noyau à l'état actif utilisateur. Cela explique pourquoi un processus n'est pas « interruptible » lorsqu'il exécute un appel système.

2 - L'envoi des signaux

2.1 - La primitive kill

Il est possible d'envoyer un signal à un processus en utilisant la primitive « kill() » :

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

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

Où « pid » référence le processus destinataire, et « sig » le numéro du signal. Les processus émetteur et récepteur doivent avoir le même propriétaire. Le « faux » signal 0 peut être envoyé pour tester l'existence d'un processus.

Remarque

La liste des signaux peut s'obtenir sur la ligne de commande en utilisant la commande « kill -l ».

2.2 - Exemple d'utilisation de la primitive kill

```
/* signal1.c */
int main(void) {
  pid_t pid,pidBis;
  int status;
  switch(pid = fork()) {
    case (pid_t)0 :
      while(1) sleep(1);
    default:
      sleep(60);
      if(kill(pid,0) == -1) {
        printf("processus fils %d inexistant\n",pid);
        exit(1);
      } else {
        printf("envoi de SIGUSR1 au fils %d\n",pid);
        kill(pid,SIGUSR1);
      }
      pidBis=wait(&status);
      printf("Mort de %d, status=%d\n",pidBis,status);
  }
  exit(0);
}
```

3 - Le masquage des signaux

3.1 - La primitive sigprocmask

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

Cette primitive permet l'installation manuelle d'un masque à partir de l'ensemble pointé par new et éventuellement du masque antérieur que l'on récupère au retour de la primitive à l'adresse old si le troisième paramètre n'est pas le pointeur nul. Le type sigset_t correspond à un ensemble de signaux.

Le paramètre opt précise ce que l'on fait avec ces ensembles :

Valeur du paramètre opt	*Nouveau masque*
SIG_SETMASK	*new
SIG_BLOCK	*new U *old
SIG_UNBLOCK	*old - *new

3.2 - Les primitive supplémentaires

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

Ecrit à l'adresse « ens » la liste des signaux pendants qui sont masqués.

```
int sigemptyset(sigset_t *set);
```

Vide l'ensemble de signaux donné par « set ».

```
int sigaddset(sigset_t *set, int signum);
```

Ajoute le signal « signul » à l'ensemble « set ».

```
int sigismember(const sigset_t *set, int signum);
```

Teste si le signal « signum » est membre de l'ensemble « set »

Remarque : Il existe aussi les fonctions sigfillset(...) et sigdelset(...).

3.3 - Exemple de masquage d'un signal

```
/* signal2.c */
int main(void) {
  sigset_t ens1,ens2; int sig;

  sigemptyset(&ens1);sigaddset(&ens1,SIGINT);
  sigaddset(&ens1,SIGQUIT);sigaddset(&ens1,SIGUSR1);

  sigprocmask(SIG_SETMASK,&ens1,NULL);
  printf("Masquage en place pour 60 secondes...\n");
  sleep(60);

  sigpending(&ens2);
  printf("Signaux pendants:\n");
  for(sig=1;sig<NSIG;sig++)
    if(sigismember(&ens2,sig))  printf("%d \n",sig);
  sigemptyset(&ens1);
  printf("Déblocage des signaux...\n");
  sigprocmask(SIG_SETMASK,&ens1,NULL);
  sleep(15);
  printf("Fin normale du processus\n");
  exit(0);
}
```

4 - Le captage des signaux

4.1 - La structure sigaction

Le comportement général d'un processus lors de la délivrance d'un signal correspond à la structure « *sigaction* » :

```
struct sigaction {
    void        (*sa_handler)() ;
    sigset_t    sa_mask ;
    int         sa_flags ;
}
```

« *sa_handler* » peut être « SIG_DFL », « SIG_IGN », ou un pointeur sur la fonction chargée de gérer le signal envoyé.

Le champ *sa_mask* correspond à une liste de signaux qui doivent être ajoutés, pendant l'exécution du handler à ceux déjà masqués.

Si on utilise la primitive « *sigaction()* », le signal en cours de délivrance sera automatiquement ajouté à cette liste.

4.2 - La primitive sigaction

La fonction « *sigaction()* » permet d'installer le captage d'un signal tout en masquant un ensemble de signaux (y compris, par défaut, le signal capté).

```
#include <signal.h>
int sigaction(int sig, struct sigaction *p_action,
              struct sigaction *p_action_anc) ;
```

La délivrance du signal « *sig* » entraîne l'exécution du handler de « *p_action* ». « *P_action_anc* » permet de retrouver l'ancien comportement du signal.

4.3 - Exemple de capture d'un signal

```
/* signal3.c */
sigset_t ens;
struct sigaction action;

void handler(int sig);

int main(void) {
  action.sa_handler=handler;
  action.sa_flags=0;
  sigemptyset(&action.sa_mask);
  sigaction(SIGQUIT,&action,NULL);

/* Seul SIGQUIT sera masqué pdt l'exécution du handler.*/
  sigaddset(&action.sa_mask,SIGQUIT);
  sigaction(SIGINT,&action,NULL);

/*SIGINT et SIGQUIT masqués pendant l'exécution du handler*/
  while(1) sleep(1);
}
```

```
void handler(int sig) {
  int i;

  printf("Entrée dans le handler ");
  printf("avec le signal: %d\n",sig);
  sigprocmask(SIG_BLOCK,NULL,&ens);
  printf("Signaux masqués: ");
  for(i=1;i<NSIG;i++) {
    if(sigismember(&ens,i))  printf("%d ",i);
  }
  putchar('\n');

  if(sig == SIGINT) {
    action.sa_handler=SIG_DFL;
    sigaction(SIGINT,&action,NULL);
    /* Comportement standard de SIGINT rétabli. */
  }
  printf("Sortie du handler\n");
}
```

5 - L'attente d'un signal.

```
#include  <signal.h>
int sigsuspend(const  sigset_t  *ens);
```

Cette primitive réalise atomiquement :

- l'installation du masque de signaux pointé par *ens*,
- 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.

6 - Nom d'un signal

Il est possible d'obtenir le nom d'un signal à partir de son numéro par l'appel :

```
#include <string.h>
char *strsignal(int sig);
```

7 - Les signaux temps-réel

7.1 - Introduction

Linux supporte les signaux temps-réel POSIX. Ces signaux n'ont pas de significations particulières et leurs valeurs s'étale de SIGRTMIN à SIGRTMAX. Ils sont donc théoriquement complètement disponibles pour l'application, toutefois Linux utilise parfois les premiers d'entre eux avec certaines bibliothèques.

Les différences des signaux temps-réel avec les signaux standards sont globalement les suivantes :

- Les signaux temps-réel sont « empilés » et tous distribués,
- Il est possible de passer une valeur en même temps que le signal,
- Il est possible de connaitre le PID de l'émetteur du signal,
- Ils possèdent une priorité (numéro du signal) si plusieurs sont à délivrer,

7.2 - Envoi d'un signal

Il faut pour cela activer le flag « SA_SIGINFO » de la structure « sigaction » et utiliser la primitive suivante :

```
int sigqueue(pid_t pid, int sig, const union sigval valeur);
```

Avec « pid » pour le pid du processus destinataire, « sig » pour le numéro du signal, et « valeur » pour le paramètre supplémentaire POSIX définit de la manière suivante :

```
union sigval {
  int   sival_int;
  void *sival_ptr;
};
```

```
/* signal4.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void gestionnaire(int sig, siginfo_t *info, void *inutile) ;

int main()
{
  pid_t pid_fils;
  int info_fin;
  union sigval valeur;
  valeur.sival_int=10;
  struct sigaction action;

  switch ( pid_fils = (long)fork() ){
  case -1:
    perror("Erreur lors de fork");
    return EXIT_FAILURE;
```

```
  case 0:
    action.sa_sigaction = gestionnaire;
    action.sa_flags = SA_SIGINFO;
    sigemptyset(&action.sa_mask);
    sigaction(SIGUSR1, &action, NULL);
    while (1) {
      printf("Le processus fils est vivant\n");
      sleep(2);
    }
  default:
    sleep(5);
    printf("Père envoie SIGUSR1 à son fils\n");
    if ( kill(pid_fils, SIGUSR1)==-1 ) {
      perror("kill "); exit(EXIT_FAILURE); }
    sleep(2);
    if ( sigqueue(pid_fils, SIGUSR1, valeur)==-1 ) {
      perror("sigqueue "); exit(EXIT_FAILURE); }
    sleep(2);
    if ( kill(pid_fils, SIGKILL)==-1 ) {
      perror("kill "); exit(EXIT_FAILURE); }
  }
}
```

```
void gestionnaire(int sig, siginfo_t *info, void *inutile)
{
  char *origine;

  printf("Reception du signal n° %d ", sig);
  switch ( info->si_code ){
    case SI_USER:
      printf("kill() ou raise(), processus %ld\n",
              info->si_pid);
      break;
    case SI_QUEUE:
      printf("sigqueue(), processus %ld, valeur %d\n",
              (long)info->si_pid, info->si_value.sival_int);
      break;
    default:
      printf("\n");
      break;
  }
}
```