LES SIGNAUX

Guillaume Chanel

ENVOIS DE SIGNAUX

FONCTIONNEMENT DE L'ENVOIS

Un signal est un **événement asynchrone** (i.e. qui peux arriver à tout moment), **identifié par un numéro** (constantes C définies), et **envoyé par le noyau à un processus**.

Un signal est envoyé lorsque:

- le noyaux souhaite signaler un événement (e.g. `SIGCHLD`, `SIGSEGV`);
- un autre processus a demandé à envoyer le signal à travers **un appel système** (e.g. `kill`);
  - l s'agit alors de communication inter-processus;
  - c'est le noyau qui determine si le processus est autorisé à faire cet envois (`errno = EACCESS`)

Un processus recevant un signal exécute généralement une action par défaut définie pour chaque signal.

QUESTION

Quel est le signal permettant de terminer une application ?

EXEMPLES DE SIGNAUX

Les signaux suivants sont particulièrement importants (`man 7 signal`).

Identifiant	Définition	Action par défaut	Remarque
SIGKILL	Terminaison forcée	terminer	***
SIGTERM	Terminaison logicielle	terminer	
SIGABRT	Terminaison anormale	core-dump	
SIGSTOP	Demande d'attente	mise en attente	***
SIGCONT	Demande de reprise	reprendre	après SIGSTOP/SIGSTP
SIGSEGV	Référence mémoire invalide	core-dump	erreur de segmentation
SIGCHLD	Process. enfant terminé	ignoré	utilisé par wait()
SIGXCPU	Temps CPU dépassé	core-dump	c.f. limites des ressources
SIGUSR1/2	Définit par le programmeur	terminer	

***** ne peut pas être bloqué ou ignoré; n'est pas modifiable par un handler.**

ENVOYER UN SIGNAL

Pour envoyer le signal sig à un processus pid on utilise:

```c
#include <signal.h>
int kill (pid_t pid, int sig);
```

ATTENTION le nom prête à confusion: on peut envoyer tout type de signal avec cette fonction pas uniquement SIGKILL.

Pour envoyer le signal sig au processus courant on utilise:

```c
#include <signal.h>
int raise (int sig);
```

Pour envoyer le signal SIGALRM au processus courant après un délais de seconds secondes:

```c
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

MASQUAGE ET ATTENTE DE SIGNAUX

MASQUAGE DES SIGNAUX

Un processus recevant un signal est **interrompu** pour exécuter une action.

Toutefois, un signal peut être masqué, dans ce cas:

- l'action sera suspendue jusqu'à ce que le masque du signal soit retiré;
- une fois le masque retiré l'action est exécuté;
- si un signal masqué est reçu plusieurs fois l'action ne sera exécutée qu'une seule fois lorsque le masque sera retiré.

PROGRAMMATION DES MASQUES

1. Définir l'ensemble des signaux concernés avec les fonctions dédiées:

```c
#include <signal.h>
int sigemptyset (sigset_t * set); //un ensemble vide
int sigfillset (sigset_t * set); //l'ensemble de tous les signaux
int sigaddset (sigset_t * set, int signum); //signum est ajouté a set
int sigdelset (sigset_t * set, int signum); //signum est supprimé de set
int sigismember (const sigset_t * set, int signum); //signum est membre de set ?
```

PROGRAMMATION DES MASQUES

2. Appeler la fonction de masquage:

```c
int sigprocmask (int how, const sigset_t * set, sigset_t * oldset);
```

`set` contient l'ensemble de signaux à masquer et `oldset` contient l'ensemble des signaux initialement masqués (utile pour remettre le masque dans l'état initial).

`how` peut avoir pour valeur:

- `SIG_BLOCK`: les signaux de set sont ajoutés au signaux actuellement bloqués;
- `SIG_UNBLOCK`: les signaux de set sont retirés des signaux actuellement bloqués;
- `SIG_SETMASK`: les signaux actuellement bloqués sont remplacés par set.

ATTENTE DE SIGNAUX

Deux fonctions existent pour mettre un processus en attente de signaux:

```
#include <unistd.h>
int pause(void);
```

Cette fonction met le processus en mode « sleep » (TASK_ INTERRUPTIBLE) jusqu'à ce qu'un signal soit reçu.

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

Cette fonction masque temporairement les signaux de l'ensemble mask et met le processus en mode « sleep » jusqu'à réception d'un signal non masqué.

QUESTION

Le deux codes ci-dessous semblent équivalents Code 1:

```
sigfillset(&blockall);
sigprocmask (SIG_SETMASK, &blockall, &oldset);
//..
//Ici s'éxécute un code critique qui ne doit pas être interrompu...
//..
sigsuspend(&oldset);
```

Code 2:

```
sigfillset(&blockall);
sigprocmask (SIG_SETMASK, &blockall, &oldset);
//..
//Ici s'éxécute un code critique qui ne doit pas être interrompu...
//..
sigprocmask (SIG_SETMASK, &oldset, NULL);
pause();
```

Pourtant le deuxième code peut ne pas fonctionner, pourquoi ?

RECEPTION DE SIGNAUX

ACTIONS DES SIGNAUX

Un processus recevant un signal est **interrompu** pour exécuter l'action correspondante.

L'action peut-être:

- une action par défaut;
- d'ignorer le signal: dans ce cas aucune action n'est exécutée (attention cela est différent du masquage);
- **un «handler» programmé par l'utilisateur**.

Un handler est une fonction appellée à chaque recéption d'un signal.

ASSOCIER UNE ACTION À UN SIGNAL

Pour définir l'action d'un signal on utilise:

```c
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- `signum` : est le numero du signal auquel on souhaite associé une action (e.g. SIGUSR1);
- `act`: la configuration de la nouvelle action à effectuer sur récéption du signal `signum`, peut être `NULL` pour recevoir uniquement `oldact`;
- `oldact`: la configuration initial du signal `signum`.

La stucture `sigaction` a les champs suivants:

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

ASSOCIER UNE ACTION À UN SIGNAL

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

`sa_handler` peut prendre pour valuer soit:

- SIG_DFL: l'action par default est mise en place;
- SIG_IGN: le signal sera ignoré;
- l'adresse du handler à exécuter (en fait simplement le nom de la fonction). Cette fonction recevra un entier qui correpond au numéro de signal reçu.

Example:

```
void mon_handler(int signum) {};
struct sigaction sa;
sa.sa_handler = mon_handler;
```

ASSOCIER UNE ACTION À UN SIGNAL

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

Une autre manière de déclarer un handler plus complet. La fonction reçevra les informations suivantes:

- `int`: le numéro du signal reçu
- `siginfo_t`: des informations supplémentaires (e.g le pid du processus envoyant le signal);
- `void*`: un pointeur sur des données envoyées par le processus émetteur.

Example:

```
void mon_handler(int signum, sig_info_t info, void* mydata) {};
struct sigaction sa;
sa.sa_sigaction = mon_handler;
```

ASSOCIER UNE ACTION À UN SIGNAL

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

Attention: ne pas utiliser `sa_handler` et `sa_sigaction` en même temps: une union peut être impliquée !

ASSOCIER UNE ACTION À UN SIGNAL

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

- `sa_mask`: Un ensemble de signaux à masquer pendant l'exécution de l'action. **De plus le signal traité (i.e. signum) est systématiquement bloqué**.
- `sa_flags`: modifie le comportement du handler. Important: **SA_SIGINFO permet l'utilisation de sa_sigaction au lieu de sa_handler**.

RÈGLES DEPROGRAMMATION DES HANDLERS

- un handler doit garantir qu'il n'est pas appelé à nouveau lors de son exécution (signaux associés au handler bloqués et pas de récursivité);
- il faut aussi garantir que le code soit ré-entrant:
  - une fonction est ré-entrante si elle peut être interrompue sans causer d'effets de bords;
  - toute fonction appelée dans le handler doit aussi être ré-entrante.

Exemple de fonctions NON ré-entrantes:

- `malloc`, `calloc`, `free`;
- `printf`, `scanf`, ...

EXEMPLE

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

void manage_signals(int sig) {
    switch (sig)
    {
    case SIGUSR1:
        write(STDOUT_FILENO, "SIGUSR1\n", 8);
        break;
    case SIGTERM:
        write(STDOUT_FILENO, "SIGTERM\n", 8);
        break;
    default:
        break;
    }
}

void usr2_exit(int sig) {
    write(STDOUT_FILENO, "Received SIGUSR2, process exiting\n", 34);
    exit(0);
}

int main() {

    struct sigaction sa;

    printf("Pid: %d\n", getpid());

    sa.sa_handler = manage_signals;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGUSR1);
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
```