Programmation Système

Signaux

Abdelkader Gouaïch

INTRODUCTION

Définition d'un signal

Un signal : un mécanisme Unix élémentaire de communication qui permet à un processus de prendre en compte des changements d' états de son environnement.

Un signal événement asynchrone: son apparition n'est pas contrôlée par le processus

Exemples:

- un utilisateur fait un Ctrl^'C'.
- une division par 0.

Les signaux sont aussi un moyen de communication entre les processus:

- communiquer avec n'importe quel processus
- l'information transmise est l'entier qui identifie le signal.

Asynchronisme

Un thread:

- une suite d'appels de fonctions.
- instruction suivante à exécuter connaissant notre état actuel est connue

Multi thread:

- juste plusieurs threads....
- même modèle : plusieurs appels de fonctions

Les signaux vont modifier ce modèle d'activité !

Asynchronisme

Les signaux vont modifier le modèle d'activité:

- ignorons à quel moment ils vont survenir
- à quel moment exécuter leurs gestionnaires.
- au moment de recevoir un signal, le processus va suspendre les activités pour lancer le gestionnaire du signal reçu.
- le gestionnaire de signal doit traiter l' événement rapidement pour rendre la main

Identifiant d'un signal

Un signal = un numéro.
Un nom symbolique sera également associé ainsi qu'une constante à utiliser dans les programmes `C`.

- nom préfixé par `SIG`,
- un mnémonique du signal.

Exemple: le nom `SIGABRT`

- `SIG`
- un événement d'abandon (abort)

Liste des signaux Linux

Les signaux sont définis dans la bibliothèque `<signal.h>`

Voici la liste non exhaustive des signaux courants:

NOM	Description	Comportement
SIGABRT	envoyé par la fonction abort()	termine le processus avec un core
SIGALRM	envoyé par alarm()	ter mine le processus
SIGCHLD	un fils a terminé	ignoré
SIGHUP	Le terminal associé au processus a été fermé	termine le processus
SIGINT	Une interruption par Ctrl^C	termine le processus
SIGSTOP	Suspend l'exécution du processus	Suspend l'exécution du processus
SIGTSTP	Suspend l'exécution du processus par l'utilisateur Ctrl^Z	Suspend l'exécution du processus
SIGKILL	Terminaison forcée du processus	termine le processus
SIGSEGV	Erreur de segmentation de la mémoire	termine le processus
SIGUSR1	Défini pour l'utilisateur	termine le processus
SIGUSR2	Défini pour l'utilisateur	termine le processus

GESTION DES SIGNAUX

Les fonctions de rappel `callback`

Un gestionnaire de signal est une fonction qui répond à la signature suivante:

```
typedef void (*sighandler_t)(int);
```

Les fonctions de rappel `callback`

Le gestionnaire sera enregistré comme un *callback* avec la fonction `signal`:

```
sighandler_t signal (int signo, sighandler_t
handler);
```

- La fonction d'enregistrement `signal` va remplacer l'ancien gestionnaire avec le nouveau.
- L'ancien gestionnaire sera retourné comme résultat de la fonction.

- Lors de la réception d'un signal, toutes les activités du processus sont suspendues avant l'exécution du gestionnaire de signal.
- Une fois que le gestionnaire termine son traitement, les activités suspendues pourront reprendre.

Les fonctions de rappel `callback`

```
sighandler_t signal (int signo,
sighandler_t handler);
```

Cas particuliers:

Si nous souhaitons remettre le gestionnaire:la constante `SIG_DFL`.

ignorer le signal: la constante `SIG_IGN`.

Les fonctions de rappel `callback`

Attention certains signaux ne peuvent pas être capturés par un simple utilisateur.

signal `SIGKILL`:

- Le comportement sera toujours de terminer le processus et l'utilisateur ne peut pas modifier ce comportement.

Attente d'un signal

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

La fonction pause va donc suspendre l'activité et retournera à la réception d'un signal.

Exemple

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
/* handler for SIGINT */
static void sigint_handler (int signo) {
    printf ("Capture de SIGINT!\n");
    exit (EXIT_SUCCESS);
    }
    int main (void) {
        /** enregistement du gestionnaire pour SIGINT. */
        if (signal (SIGINT, sigint_handler) == SIG_ERR)
        {
            fprintf (stderr, "erreur du gestionnaire pour SIGINT!\n");
            exit (EXIT_FAILURE);
        }
        while (1) {pause ();}
        return 0;
        }
```

Héritage des signaux

fork:

- le processus fils va hériter de la table des gestionnaires des signaux.
- les signaux qui sont dans la table des signaux en attente de traitement ne seront pas hérités par le fils.

`exec`:

- les gestionnaires seront réinitialisés
- les signaux en attente sont maintenus.

ENVOYER UN SIGNAL

Pour envoyer un signal nous allons utiliser la fonction kill()

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int signo);
```

Si le paramètre pid > 0 alors il désigne le PID du processus destinataire.

Si par contre il vaut 0 alors le signal est envoyé à tous les processus du groupe.

Si le pid < 0 alors le signal est envoyé au groupe de processus de GID égale à |pid|.

S’ENVOYER UN SIGNAL

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

La fonction `raise` permet d'envoyer un signal au même processus.

Elle est équivalente à:

```
kill (getpid (), signo);
```

ENVOYER UN SIGNAL À UN GROUPE DE PROCESS

```
#include <signal.h>
int killpg (int pgrp, int
signo);
```

La fonction `killpg` permet d'envoyer un signal à un groupe de processus.

Cette fonction est équivalente à

```
kill(-pgrp, signo);
```

PROBLÈME DES FONCTIONS NON RÉ-ENTRANTES

Que se passera-t-il si un autre signal intervient pendant que nous sommes déjà en train de traiter un signal?

Réponse:

- nous allons interrompre notre activité et lancer un gestionnaire de signal.

Si cela n’est pas désiré et que nous souhaitons aller au bout de notre traitement sans interruption nous devons désactiver momentanément la réception des signaux.

- Un autre problème peut arriver si le processus exécute une fonction qui n'est pas *réentrante*.
- Une fonction C non réentrante: une fonction qui ne peut pas avoir plusieurs appels concurrents
- Par précaution, il est recommandé de ne pas utiliser de fonction non réentrante dans les gestionnaires des signaux.

😱 *printf* n'est pas réentrante!

ENSEMBLE DE SIGNAUX

Une structure qui va représenter un ensemble de signaux ainsi que de fonctions pour les manipuler.

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
int sigismember (const sigset_t *set, int
signo);
```

- `sigemptyset`: va initialiser la structure sigset_t avec un ensemble vide.
- `sigfillset`: va initialiser la structure avec tous les signaux
- `sigaddset`: ajouter le signal à l'ensemble
- `sigdelset`: retire le signal de l'ensemble
- `sigismember`: vérifie si le signal est présent dans l'ensemble

BLOQUER DES SIGNAUX

```
#include <signal.h>
int sigprocmask (int how, const sigset_t
*set, sigset_t *oldset);
```

Vous pouvez bloquer/débloquer un ensemble de signaux avec la fonction `sigprocmask`.

Le comportement va aussi dépendre de la valeur du paramètre `how`

- `SIG_SETMASK`: le paramètre `set` sera le nouveau masque
- `SIG_BLOCK`: le paramètre `set` sera ajouté au masque (un ou logique)
- `SIG_UNBLOCK`: le paramètre `set` sera retiré du masque. Nous ne pouvons pas débloquer un signal qui n'est pas déjà bloqué. Si le paramètre `set` est `NULL` alors la fonction ne fait rien mais nous retourne la valeur du masque courant.

RÉCUPÉRER LES SIGNAUX BLOQUÉS/ATTENTE

```
#include <signal.h>
int sigpending (sigset_t *set);
```

Avec cette fonction nous pouvons récupérer l'ensemble des signaux qui sont en attente/bloqués pour notre processus.

ATTENTE DE SIGNAUX PARTICULIERS

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

Changer notre masque temporairement pour ensuite suspendre l'activité et attendre uniquement les signaux mis dans `set`.