Programmation Système

Signaux

Abdelkader Gouaïch

2022

Introduction

Définitions

Définition d'un signal

Les signaux d'Unix offrent un mécanisme élémentaire de communication qui permet à un processus de prendre en compte des changements d'états de son environnement. Ces événements sont asynchrones dans le sens où leur apparition n'est pas contrôlée par le processus lui même. Par exemple, un utilisateur peut décider, à n'importe quel moment, d'envoyer un signal d'interruption à un processus en cours d'exécution en appuyant sur les touches Contrôle et 'C'. Un autre exemple est celui de la réception d'un signal de la CPU indiquant qu'une opération arithmétique illégale est exécutée comme la division par 0. Les signaux sont aussi un moyen de communication entre les processus. En effet, pour l'instant nous avons vu que les processus ne partagent pas leur mémoire virtuelle et nous avons utilisé qu'un seul mécanisme rudimentaire de communication entre un processus fils et son père : il s'agit du statu de terminaison. Avec les signaux nous pouvons communiquer avec n'importe quel processus et cela pendant notre activité. L'information transmise est certes rudimentaire, car il s'agit pour l'instant que d'un entier qui identifie le signal. Mais ceci nous ouvre la voie vers la communication entre processus que nous appelons l'Inter Process Communication (IPC)

Asynchronisme

Un processus va exécuter un ou plusieurs threads. Nous pouvons voir que chaque traitement, même s'il est parallèle, revient à sera une suite d'appels de fonctions. De plus, nous pouvons déterminer exactement l'instruction suivante à exécuter connaissant notre état actuel. Les signaux vont modifier ce modèle d'activité car nous ignorons à quel moment il vont survenir et par conséquent à quel moment

exécuter leurs gestionnaires. Les fonctions de traitement des signaux ne seront pas appelées par le programmeur mais seront évaluées au moment de la réception de l'événement. Cela veut dire qu'au moment de recevoir un signal, le processus va suspendre toutes ses activités pour se consacrer uniquement à l'exécution de la fonction de gestion du signal reçu. Une fois le gestionnaire terminé, alors les activités pourront reprendre. Ceci nous montre le caractère spécial et urgent réservé à la gestion des signaux. Il donne aussi au gestionnaire de signal une certaine responsabilité car il doit traiter l'événement rapidement pour rendre la main aux activités suspendues.

Identifiant d'un signal

Pour identifier les signaux le système d'exploitation va leurs associer des numéros. Un nom symbolique sera également associé ainsi qu'une constante à utiliser dans les programmes `C`. Le nom des constantes est préfixé par `SIG`, vient ensuite le mnémonique du signal. Par exemple, le nom `SIGABRT` nous informe qu'il s'agit d'un signal (`SIG`) et que ce signal désigne un événement d'abandon (abort) `SIGKILL` est aussi un signal (`SIG`) qui désigne la terminaison forcée du processus (`KILL`)

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

NOM	Description	Comportement
SIGUSR2	Défini pour l'utilisateur	termine le processus

Gestion simple des signaux

Gestion des signaux

Les fonctions de rappel `callback`

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal (int signo, sighandler_t handler);
```

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

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

C'est donc une fonction prend un paramètre entier, qui sera le numéro de signal reçu, et qui ne retourne aucun résultat.

Ce 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.

Attention certains signaux ne peuvent pas être capturés par un simple utilisateur. C'est le cas par exemple du signal `SIGKILL`. Le comportement sera toujours de terminer le processus et l'utilisateur ne peut pas modifier ce comportement.

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.

Cas particuliers: Si nous souhaitons remettre le gestionnaire par défaut pour un signal, alors nous allons utiliser la constante `SIG_DFL` comme paramètre de la fonction `signal`.

Si par contre, nous souhaitons ignorer le signal alors nous devrions utiliser la valeur `SIG_IGN`. Attention, certains signaux ne peuvent pas être ignorés (par exemple du signal `SIGKILL`).

Attente d un signal

Un processus peut recevoir un signal à n'importe quel moment de son activité.

Mais si nous n'avons aucun traintement à réaliser en attendant un signal, alors nous devons suspendre notre activité avec la fonction `pause()`

```
#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

Après un fork le processus fils va hériter de la table des gestionnaires des signaux. Cela veut dire qu'il aura exactement le même comportement que son père lors de la réception d'un signal.

Cependant, les signaux qui sont dans la table des signaux en attente de traitement ne seront pas hérités par le fils.

Cela parait logique car un signal est envoyé à destination d'un processus particulier en utilisant son PID et le fils a un PID différent de celui du père.

Le comportement est différent avec `exec`. Dans ce cas tous les gestionnaires seront réinitialisés avec les gestionnaires par défaut.

Par contre, la table des signaux en attente de traitement est maintenue pour le nouveau progamme.

Encore une fois, cela parait logique car ces signaux ont été envoyés au processus identifié par son PID et après l'`exec` le processus conserve son PID.

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 $|pid|$.

Gestion des permissions

Les signaux vont induire l'exécution d'un comportement chez le processus receveur.

La question des droits et des permissions se pose alors. En effet, que se passera-t-il si un processus utilisateur envoie un signal de terminaison à un processus système important ? Est ce que ce processus va arrêter son activité ?

Linux va définir des *capacités* pour les processus. Si un processus possède la capacité `CAP_KILL` alors aura le droit d'envoyer des signaux de terminaison à tous les autres processus. Cette capacité est octroyée automatiquement aux processus du root.

Sans cette capacité, un processus à le droit d'envoyer des signaux uniquement au processus qui partagent son effective user id soit au niveau de leur propre effective user id ou leur saved user id. L'échange des signaux est donc restreint aux processus d'un même utilisateur.

Pour vérifier les droits d'envoi des signaux, sans nécessairement envoyer un signal, nous pouvons utiliser la valeur spéciale 0 comme numéro de signal. Avec cette valeur, la vérification des droits sera réalisée mais aucun signal ne sera reçu par le processus récepteur.

Exemple:

```
int ret; ret = kill (2334, 0);
if (ret) { /* problème de de permissions */}
else { ; /* nous avons la permission */}
```

L'exemple ci dessus teste si nous avons la permission d'envoyer un signal au processus 2334. Selon le code retour nous allons pouvoir confirmer notre droit.

Envoyer un signal à soi

```
#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. Il nous faut simplement spécifier l'identifiant du groupe de processus.

Cette fonction est équivalente à

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

Problème des fonctions non ré-entrantes

Un signal peut survenir à n'importe quel moment et le processus va interrompre son activité pour lancer le gestionnaire et revenir ensuite continuer son flow d'exécution.

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

Et bien la procédure est la même, 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 est une fonction qui ne peut pas avoir plusieurs appels au même temps soit par récursion ou par des threads différents.

Souvent ces fonctions utilisent des variables globales, qui ne sont pas dans la pile des appels.

La même variable globale se trouve alors modifiée par plusieurs instances d'appel de la fonction, ce qui des problèmes de cohérence.

Examinons un scénario ou un premier appel est en cours d'exécution par le processus et le gestionnaire du signal réalise aussi un appel à la même fonction.

Si la fonction n'est pas réentrante alors cela causera un problème, car les variables globales seront modifiées sans protection. De plus, le gestionnaire du signal ne peut pas avoir d'information pour déduire quelle fonction était en cours d'exécution par le processus avant son appel.

Par précaution il est recommandé de ne pas utiliser de fonction non réentrante dans les gestionnaires des signaux.

Ensemble de signaux

Il est souvent pratique de manipuler un ensemble de signaux au lieu de les traiter un par un. Pour cela nous allons disposer d'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);
```

- `siemptyset`: va initialiser la structure sigset_t avec un ensmble 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`. On parle alors d'un masque qui va empêcher les signaux désignés

de vous atteindre. 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 paramère `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);
```

Cette fonction nous permet de changer notre masque temporairement pour ensuite suspendre l'activité et attendre uniquement les signaux mis dans `set`.