Système d'exploitation et programmation système

Cours 3

EISTI

2008/2009

Département informatique

Communication inter processus

Les signaux

Définition

- Un signal est une information atomique envoyée à un processus ou à un groupe de processus par le système d'exploitation ou par un autre processus
- Lorsqu'un processus reçoit un signal, le systeme d'exploitation l'informe: "Tu as reçu un signal" sans plus
- Un signal ne transporte aucune autre information utile
- Le processus pourra alors mettre en oeuvre une réponse décidée et pré-définie à l'avance (***handler***)

Définition (suite I)

- Les signaux sont un mécanisme asynchrone de communication inter-processus
- Il est assimilable à une sonnerie indiquant des évenements différents **pouvant** donner lieu à une réaction
- Chaque signal correspond à un événement spécifique

Rappel :

- Intruption matérielle (IRQ) traitement synchrone (modalité)
- Intéruption logicielle traitement asynchrone

Définition (suite II)

Ce mécanisme est implanté par un **moniteur**, qui scrute en permanence l'occurrence des signaux. C'est par ce mécanisme que le système communique avec les processus utilisateurs :

- provenance interne en cas d'erreur du processus
  - violation mémoire
  - erreur d'E/S
  - *segmentation fault (core dumped)*
- à la demande de l'utilisateur lui-même via le clavier...,
- lorsque vous tapez la commande kill,
- ou encore pour la déconnection de la ligne/terminal provenance externe)

Origine des signaux

- **Causes internes au processus**
  - Erreur d'adressage → SIGSEVG (*segmentation violation*)
  - Division par zéro → SIGFPE (*Floating Point Exception*)
- **Terminal : grâce aux caractères speciaux**
  - Intr → SIGINT "CTRL C" (*interruption*)
  - Quit →SIGQUIT "CTRL \"
- **Déconnection du terminal**
  - envoie à l'ensemble des processus de son groupe → SIGHUP (Hangrup=décrochage)

L'envoi d'un signal

- La primitive kill permet au systeme d'envoyer un signal à un processus :

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

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

  - la primitive renvoie 0 en cas de succès, -1 sinon
  - *signal* est un numéro compris entre 1 et NSIG (défini dans <signal.h>)
  - *pid* numero du processus

0 : Tous les processus du groupe du processus réalisant l'appel **kill**

1 : En système V.4 tous les processus du système sauf les processus de pid 0 et 1

pid positif : le processus du pid indiqué

pid négatif : tous les processus du groupe |pid|

L’envoi d’un signal (suite)

Remarque 1:

La fonction raise(int *signal*) est un raccourci pour kill(getpid(), *signal*)

(envoi à lui même un signal)

Remarque 2:

On peut réecrire kill(0, *signal*) par kill(-getpid(), *signal*)

(les PIDs sont toujours positifs)

Remarque 3:

Un processus ne peut envoyer un signal qu’à un processus de même propriétaire

La commande shell **kill**

Pour envoyer un signal à un processus, on utilise la commande appelée **kill**. Celle-ci prend en option le numéro du signal à envoyer et en argument le numéro du (ou des) processus destinataire(s).

- Exemple 1 :

$> **kill** 36 : par défaut le signal 15 (SIGTERM) est envoyé au processus de pid 36.

- Exemple 2 :

$> **kill** 0 : Envoie le signal 15 à tous les processus fils, petits-fils... tous ceux lancés depuis ce terminal.

- Exemple 3 :

$> **kill** -9 36 : Envoie le signal de numéro 9 (SIGKILL) au processus de pid 36.

- Exemple 4 :

$> **kill** -s SIGKILL 36 : Envoie le signal SIGKILL au processus de pid 36

Comportements possibles du processus

- A chaque type de signal est associé à un *handler* par défaut **SIG_DFL**
- Ce *handler* définit le comportement par défaut du processus ou la procédure à exécuter à la réception du signal donné.
- Les différents comportements gérés par ce *handler s*ont :
  - terminaison du processus, avec ou sans une image mémoire (fichier core),
  - signal ignoré, (SIGKILL et SIGSTOP ne peuvent être ignorés)
  - suspension (SIGSTOP) du processus,
  - continuation (SIGCONT) : reprise du processus stoppé et ignoré sinon.

Comportements par défaut des signaux

- Rien :
  - SIGCHLD(*Terminaison d'un process fils*),
  - SIGPWR,
  - SIGCONT…
- Fin :
  - SIGHUP(*Fin de session*),
  - SIGINT,
  - SIGKILL…
- Génération d'une image mémoire (CORE) :
  - SIGQUIT,
  - SIGILL(*Instruction illégale*),
  - SIGSEGV(*Violation de mémoire*)…
- Arrêt :
  - SIGSTOP,
  - SIGSTP(*Demande de suspension depuis le terminal*)…

Détournement

- Pour certains signaux, on peut détourner l'action par défaut.
- Le caractère non modifiable de certains signaux assure la stabilité du système (SIGKILL, SIGCONT, SIGSTOP).

Gestion interne des signaux

- La table de gestion des signaux de chaque processus se trouve dans le bloc de contrôle (BCP)
- Chaque table contient pour chaque signal défini sur la machine une structure

```
struct sigvec {
                    bit pendant;
                    void (*traitement)(int)
                }
```

- Le drapeau **pendant** indique que le processus a reçu un signal, mais ne l'a pas encore pris en compte
- Dans la norme POSIX, on a un champ supplémentaire : bit **masque**

Etats possibles du signal

- Pendant : le signal a été envoyé à un processus mais il n'a pas encore été pris en compte
- Délivré : le processus à pris en compte le signal reçu (il est passé à l'état actif)
- Bloqué ou masqué : La délivrance du signal a été modifiée par le processus (différée ou ignorée)

Remarque 1:

Si un signal est à l'état **pendant** pour un processus et que le même signal arrive une seconde fois, ce dernier est perdu → prise en charge d'un seul et même signal.

Remarque 2:

Cette interface est rendue obsolète par sigaction.

L'association signaux/handlers

- Tout processus peut installer pour les autres signaux un nouveau handler de la façàn suivante :

```
#include<signal.h>

void  p_handler (int signal)
```

- Il existe deux interfaces de manipulation permettant l'installation d'un handler :
  - L'une, historique (ATT) est simplifiée ("**signal ( )**"), mais avec un comportement incertain,
  - ("**sigaction ( )**"), POSIX, plus complexe que la première garantit un comportement plus sûr et des programmes plus portables.

La gestion simplifiée avec la fonction signal()

La fonction **signal** permet de spécifier ou de connaître le comportement du processus à la récéption du signal donné.

- ancien C : (***signal** (sig, function))()

```
int sig;
int (*function)();
```

- ANSI C : void (***signal**(int sig, void(action)(int)))(int);

  avec 3 possibilités pour le paramètre action :

  - HANDLER vers votre fonction de traitement du signal
  - SIG_IGN pour ignorer le signal (SIGKILL et SIGSTOP ne peuvent être ignorés)
  - SIG_DFL pour l'action par défaut (termine, ignore, pause, continue)

L'association “sigaction”

- Elle repose sur la fonction “**sigaction( )**” qui associe un signal à un contexte de déroutement, représenté par une structure “**struct sigaction**”.

- **int sigaction (int *num_sig*,** // *le signal concerné*
  const struct sigaction * *nouv_action*,
  const struct sigaction * *anc_action*) ;

- La primitive d'association “**sigaction( )**” récupère en paramètre l'ancien contexte de déroutement dans lequel elle peut basculer, dans certaines circonstances, qui seront détaillées dans les différentes fonctions associée à cette interface.
- La valeur renvoyée par sigaction() est
  - 0 si tout s’est bien passé
  - -1 si une erreur est survenue, l’appel à sigaction() est ignoré

Structures sigaction

- struct sigaction {

```
// pointeur sur handler (fonction de traitement)
void (*sa_handler)(int);
// pointeur sur une fonction de traitement
void (*sa_sigaction)(int, siginfo_t *, void *);
// indique au système comment réagir sur reception d'un signal
sigset_t sa_mask;
// différents options
int sa_flags;
// pointeur sur une fonction de restauration
void (*sa_restorer)(void);
}
```

- On utilise un Type Union pour appeler soit :
  - la fonction sa_handler
  - la fonction sa_sigaction

Structures sigaction (suite)

- « **void(*sa_handler)()** » est le handler associé au signal mentionné à l 'appel à la fonction « sigaction ( ) », qui peut être
  - Un pointeur vers la fonction de traitement du signal
  - SIG_IGN pour ignorer le signal
  - SIG_DFL pour restaurer la réaction par défaut
- Pointeur sur une structure de type **sigaction**, structure qui sera remplie par la fonction selon l'ancienne configuration du traitement du signal (pointeur null)
- « **sa_mask** » est un ensemble/vecteur de signaux qui seront bloqués avec celui passé à l'appel de « sigaction ( ) », lors de l'exécution du handler associé
- « **sa_flag** » permet de passer des drapeaux, et indique les option liées à la gestion du signal.
- L'élément ***sa_restorer*** est obsolète et ne doit pas être utilisé, POSIX ne mentionne pas de membre *sa_restorer*.

Manipulation du mask

L'interface de manipulation du masque (« **sa_mask** ») est la suivante :

/* vide l'ensemble des signaux du masque */

- **int sigemptyset (sigset_t * *ens_signaux*);**

/* ajout d'un signal au masque */

- **int sigaddset (sigset_t * *ens_signaux*, int *num_sig*);**

/* retire un signal du masque */

- **int sigdelset (sigset_t * *ens_signaux*, int *num_sig*);**

/* bloque ou débloque l'ensemble des signaux du masque */

- **int sigprocmask (int *action*, const sigset_t *ens_signaux*, sigset_t * *ancien_signaux*);**

/* voir si le signal appartient à *ens_signaux* */

- **int sigismember(sigset_t * *ens_signaux*, int *num_sig*);**

Flags possibles

- SA_NOCLDSTOP : le signal n'est pas traité si le fils est arrêté (signal de type SIGSTOP, SIGTSTP, SIGTTIN ou SIGTTOU reçu).
- SA_ONESHOT : retour de l'action par défaut après réception d'un signal (action par défaut de la primitive signal()).

Signaux et héritage

- Un fils n'hérite pas des signaux pendants du père, mais bien des associations signaux-handler faites par le père.

- Lors d'un « fork () », suivi par un « exec () » dans le fils , toutes les associations signaux-handler sont réinitialisées dans le fils avec les handlers par défaut.

Exemple

- Ignorer tous les signaux sauf SIGQUIT

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

void main(void){
    int i;
    struct sigaction action;
    action.sa_handler = SIG_IGN; //ignore le signal
    for(i=1;i<NSIG;i++)
            sigaction(i, &action, NULL);
    action.sa_handler = SIG_DFL; //remise à la valeur
                                 //par defaut du signal
    sigaction(SIGQUIT, &action, NULL);
}
```

Quelques signaux

En fait il existe 64 signaux différents numérotés à partir de 1, dont plus de la moitié pour le temps-réel. Ces signaux portent également des noms « normalisés »

- On les trouve :
  - dans "/usr/include/signal.h" ou
  - en tapant la commande shell : **$ kill -l**

- **SIGHUP (1)** il est envoyé lorsque la connexion physique de la ligne est interrompue ou en cas de terminaison du processus leader de la session (« CTRL D » interruption clavier) ;
- **SIGINT (2)** frappe du caractère **intr** (« CTRL C » interruption clavier) sur le clavier du terminal de contrôle ;
- **SIGQUIT (3)** frappe du caractère **quit** (« CTRL \ » interruption clavier avec sauvegarde de l'image mémoire dans le fichier de nom **core**) sur le clavier du terminal de contrôle ;

Quelques signaux (suite I)

- **SIGKILL (9)** signal de terminaison (non déroutable).
- **SIGILL (4)** Instruction illégale.
- **SIGFPE (8)** Erreur arithmétique.
- **SIGUSR1 (10)** Signal 1 défini par l'utilisateur.
- **SIGSEGV (11)** Adressage mémoire invalide.
- **SIGUSR2 (12)** Signal 2 défini par l'utilisateur.
- **SIGPIPE (13)** Écriture sur un tube sans lecteur.
- **SIGALRM (14)** Permet de gérer un timer(Alarme)
- **SIGTERM (15)** signal de terminaison, il est envoyé à tous les processus actifs par le programme **shutdown**, qui permet d'arrêter proprement un système UNIX. Terminaison normale. d'état de l'un des ses fils

Quelques signaux (suite II)

- **SIGCHLD (17)** Reveille le processus dont le fils vient de mourir .
- **SIGCONT (18)** Reprise du processus (non déroutable)..
- **SIGSTOP (19)** Suspension du processus (non déroutable).
- **SIGTSTP (20)** Émission vers le terminal du caractère de suspension (“CTRL Z”).
- **SIGTTIN (21)** Lecture du terminal pour un processus d'arrière-plan.
- **SIGTTOU (22)** Écriture vers le terminal pour un processus d'arrière-plan.

FIN