Linux – Principes et Programmation

7. Signaux

CNAM NSY103 2009/2010
Yann GAUTHERON

Extraits :
http://fr.wikipedia.org/,
LINUX Programmation système et réseau (Joëlle DELACROIX)
http://www.cnam.fr/

La gestion des signaux sous Linux

- Le traitement des signaux par le noyau
  - Définition d'un signal sous Linux
  - Comment le noyau interprète-t-il les signaux ?
- La programmation avec signaux
  - Envoyer, Bloquer, Recevoir avec un *handler*
  - Temporisation, Attente de signal
- Les signaux temps-réel
  - Définition
  - Gestionnaire de signal temps réel

La gestion des signaux sous Linux

- Le traitement des signaux par le noyau
  - Un signal est :
    - un message envoyé par le noyau à :
      - un processus ; ou
      - un groupe de processus
    - pour indiquer une occurrence d'un évènement système
  - Seule information remontée : le nom du signal
  - Fonction gestionnaire de signal : *handler de signal*
  - Exemples :
    - `SIGCHLD` permet au noyau d'avertir un processus père de la mort de son fils
    - `SIGSEGV` est associé à la trappe levée en cas de violation de mémoire, et termine le processus fautif

La gestion des signaux sous Linux

- Liste des signaux
  - Le noyau Linux 2.2 admet 64 signaux différents et ont un numéro et nom différent.
    - 0 : seul signal qui n'a pas de nom
    - 1 à 31 : signaux classiques
    - 32 à 63 : signaux « temps réels »

Liste des signaux « classiques »

SIGHUP (1) Connexion physique interrompue ou terminaison du processus leader de la session
SIGINT (2) frappe du caractère intr (interruption clavier : Ctrl C)
SIGQUIT (3) frappe du caractère quit (interruption clavier avec sauvegarde de l'image mémoire dans le fichier de nom core) sur le clavier du terminal de contrôle (Ctrl \);
SIGILL (4) Instruction illégale
SIGTRAP (5) Point d'arrêt pour le debug mode (non POSIX)
SIGIOT/**SIGABRT** (6) Terminaison anormale
SIGBUS (7) Erreur de bus
SIGFPE (8) Erreur arithmétique
SIGKILL (9) signal de terminaison non déroutable
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) 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.
SIGSTKFLT (16) Erreur de pile du coprocesseur
SIGCHLD (17) Terminaison d'un fils.
SIGCONT (18) Reprise du processus.
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.
SIGURG (23) Données urgentes sur une socket
SIGXCPU (24) Limite de temps CPU dépassé
SIGXFSZ (25) Limite de taille de fichier dépassé
SIGVTALARM (26) Alarme virtuelle (non POSIX)
SIGPROF (27) Alarme du profileur (non POSIX)
SIGWINCH (28) Fenêtre redimensionnée (non POSIX)
SIGIO (29) Arrivée de caractère à lire (non POSIX)
SIGPOLL (30) Equivalent à SIGIO (non POSIX)
SIGPWR (31) Chute d'alimentation (non POSIX)
SIGUNUSED (32) Non utilisé

La gestion des signaux sous Linux

- ## Champs du PCB associés aux signaux
  - Contient plusieurs champs lui permettant de gérer les signaux :
    - **`signal`** : Stocke les signaux envoyés au processus. Deux entiers 32 bits (stucture `sigset_t`), chaque bit représente un signal. « `1` = reçu ».
    - **`blocked`** : idem, mais stocke les signaux bloqués (retardés).
    - **`sigpending`** : indique si au moins un signal non bloqué en attente
    - **`gsig`** : pointeur vers l'action qui est associée à chaque signal

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Le traitement déclenche plusieurs mécanismes :
    - Envoi du signal à un processus
    - Prise en compte du signal reçu par le processus et action résultante
    - Interactions entre certains appels systèmes et les signaux

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Envoi du signal à un processus
    - Un processus peut recevoir un signal :
      - Si un signal lui est envoyé par un autre processus via `kill()`.
        - Exemple :
          Le shell envoi `SIGKILL` si on tape « `kill -9 pid` »
      - Si l'exécution du processus a levé une trappe et le gestionnaire d'exception associé positionne un signal associé à l'erreur.
        - Exemple :
          Une division par zéro amène le gestionnaire d'exception `divide_error()` à positionner `SIGFPE`.

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Envoi du signal à un processus
    - Lors de l'envoi d'un signal, `send_sig_info()` positionne à 1 le bit du signal reçu dans le PCB (`signal`) du processus destinataire.
      La fonction se termine immédiatement dans les deux cas suivant :
      - le numéro du signal émis est 0, le noyau retourne immédiatement une valeur d'erreur ;
      - ou lorsque le processus destinataire est dans l'état « *zombie* ».
    - Si délivré (`signal`) mais non traité (*handler*), le signal est dit « **pendant** »
    - Attention à la mémorisation limitée (un seul drapeau "bit").
      - Exemple :
        `SIGCHLD` pourra être reçu une nouvelle fois alors qu'il est déjà « pendant »

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Prise en compte d'un signal
    - Un processus prend en compte le signal lors de sa commutation en sortant du mode noyau : le signal est « délivré » au processus par `do_signal()`.
    - Suite à cela 3 types d'actions peuvent être réalisés :
      (valeur de `gsig.sa_handler` du PCB)
      - Ignorer le signal (`SIG_IGN`)
      - Exécuter l'action par défaut (`SIG_DFL`)
      - Exécuter une fonction définie par l'utilisateur (`&fct()`)
    - `do_signal()` ne traitera pas les signaux bloqués par le processus

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Prise en compte d'un signal : IGNORER
    - Si le signal est ignoré, aucune action ne sera effectuée.
    - L'exception `SIGCHLD` force le processus à contrôler la mort de ses fils afin que les blocs de contrôles associés soient détruits.
  - Prise en compte d'un signal : ACTION PAR DEFAUT
    - fin du processus,
    - fin du processus et création (si l'utilisateur l'a décidé) d'un fichier `core` dans le répertoire courant, qui est un fichier contenant l'image mémoire du processus, et qui peut être analysé par un débogueur,
    - le signal est ignoré,
    - le processus est stoppé (il passe dans l'état `TASK_STOPPED`),
    - le processus reprend son exécution (l'état est positionné à `TASK_RUNNING`).

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Voici un tableau qui indique les actions par défaut liées aux différents signaux :

Actions par défaut	Nom du signal
Fin du process	`SIGHUP, SIGINT, SIGBUS, SIGKILL, SIGUSR1, SIGUSR2, SIGPIPE, SIGALRM, SIGTERM, SIGSTKFLT, SIGXCOU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGIO, SIGPOLL, SIGPWR, SIGUNUSED`
Fin du process et création `core`	`SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGIOT, SIGFPE, SIGSEGV`
Signal ignoré	`SIGCHLD, SIGURG, SIGWINCH`
Processus stoppé	`SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU`
Processus redémarré	`SIGCONT`

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Prise en compte d'un signal : FONCTION UTILISATEUR
    - Tout processus peut installer une fonction spécifique pour chaque signal hormis `SIGKILL`.
    - Ce traitement peut être de deux natures :
      - Ignorer le signal en prenant la valeur `SIG_IGN`
      - Définir une action particulière définie dans une fonction C attachée au code source : **c'est le *handler* ou gestionnaire du signal**.
    - L'exécution d'une fonction utilisateur implique une double gestion des commutations de contextes.
      - Traitement initial (mode noyau) => Signal
      - Fonction utilisateur (mode utilisateur) => Handler
      - Reprise du traitement (mode noyau) => `restore_sigcontext()`

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Signaux et appels systèmes
    - Appel système bloquant, processus placé dans l'état `TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE` :
      - si `TASK_INTERRUPTIBLE`, le processus peut être réveillé par le système lorsqu'il reçoit un signal ;
      - inversement, le noyau met en état `TASK_UNINTERRUPTIBLE` si le processus ne doit pas être réveillé par un signal.
    - Lorsqu'un processus en état « `TASK_INTERRUPTIBLE` » est réveillé (reçoit un signal), une fois le handler terminé, le système le positionne en état « `TASK_RUNNING` », et il positionne la variable `errno` à `EINTR`.

La gestion des signaux sous Linux

- Gestion des signaux par le noyau
  - Signaux et héritage
    - **Remarque importante** : un processus fils n'hérite pas des signaux pendants de son père. De plus, en cas de recouvrement du code hérité du père, les gestionnaires par défaut sont réinstallés pour tous les signaux du fils.

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Envoi d'un signal à n'importe quel processus

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

    - Si `pid` > 0, le signal `sig` est envoyé au processus pid.
    - Si `pid` = 0, alors le signal `sig` est envoyé à tous les processus appartenant au même groupe que le processus appelant.
    - Si `pid` = -1, alors le signal `sig` est envoyé à tous les processus sauf le premier (`init`) dans l'ordre décroissant des numéros dans la table des processus (par ex: `shutdown` envoie le signal `SIGTERM` à tous les processus).
    - Si `pid` < -1, alors le signal `sig` est envoyé à tous les processus du groupe -pid.
    - Erreurs : **`EINVAL`**, **`ESRCH`**, **`EPERM`**

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Bloquer les signaux
    - Les variables de type `sigset_t` (vecteur de signaux), ne sont manipulables que par ces primitives :

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

    - Erreurs : **`EINVAL`** si signum est invalide

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Bloquer les signaux
    - La fonction `sigprocmask()` permet à un processus de bloquer (ou de débloquer) un ensemble de signaux, à l'exception des signaux `SIGKILL` et `SIGCONT`. En pratique, cette fonction manipule le champ « `blocked` » du PCB.

```
#include <signal.h>
int sigprocmask(
    int how, -- SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
    const sigset_t *set, -- Valeur de remplacement
    sigset_t *oldset -- Ancienne valeur
);
```

    - Erreurs : **`EFAULT`** si un pointeur invalide, **`EINVAL`** si `how` est invalide

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Bloquer les signaux
    - `sigpending()` renvoie l'ensemble des signaux en attente de livraison au thread appelant (c'est-à-dire les signaux qui se sont déclenchés en étant bloqués).

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

      Erreurs : **EFAULT** si un pointeur invalide
    - La primitive `sigsuspend()` permet de façon atomique de modifier le masque des signaux et de se bloquer en attente. Une fois un signal non bloqué délivré, la primitive se termine en restaurant le masque antérieur :

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

      Erreurs : **EFAULT** si un pointeur invalide, **EINTR** si appel interrompu

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Exemple d'utilisation avec le courant « Ctrl-C »

```
sigset_t   ens, ens1;

printf("Blocage du CTRL+C\n");
sigemptyset(&ens);
sigaddset(&ens, SIGINT);
sigprocmask(SIG_SETMASK, &ens, 0);
printf("Le CTRL+C est bloqué\n");

printf("Début du travail...\n");
sleep(10); // On attend 10 secondes
printf("Fin du travail\n");

sigpending(&ens1);
if (sigismember(&ens1, SIGINT))
   printf("CTRL+C a été tapé au moins 1 fois.\n");

printf("Déblocage du CTRL+C\n");
sigemptyset(&ens);
sigprocmask(SIG_SETMASK, &ens, 0);
printf("Le CTRL+C est débloqué\n");
```

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Attacher un *handler* à un signal
    - Deux primitives différentes permettent d'attacher un gestionnaire de traitement à un signal. Ce sont :
      - D'une part, l'appel système `signal()`. L'emploi de cette primitive est simple mais son utilisation peut poser des problèmes de compatibilité avec les différents systèmes UNIX ;

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

      - D'autre part, l'appel système `sigaction()`. D'un emploi plus complexe, elle présente un comportement fiable, normalisé par POSIX.

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

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Attacher un *handler* à un signal : Exemple avec `sigaction()`
    - Applicable à tous les signaux sauf `SIGKILL, SIGSTOP` et `SIGCONT`
    - La structure :

```
struct sigaction {
   void (*sa_handler)() ; -- gestionnaire appliqué
   sigset_t sa_mask ; -- signaux masqués en plus
   int sa_flags ;
}
```

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Attacher un *handler* à un signal : Exemple avec `sigaction()`

```
void sig_hand(int sig){
   printf ("signal reçu %d \n",sig);
}

int main(int argc, char **argv) {
   sigset_t sig_proc;
   struct sigaction action;
   sigemptyset(&sig_proc);

   /* changer le traitement */
   action.sa_mask=sig_proc;
   action.sa_flags=0;
   action.sa_handler = sig_hand;
   sigaction(SIGINT, &action,0);

   /* traitement quelconque durant lequel un Ctrl-C est prévu */
   sleep(10);

   return EXIT_SUCCESS;
}
```

La gestion des signaux sous Linux

- ## Programmation avec signaux en C
  - Traiter les appels systèmes interrompus
    - Cette suite d'évènements doit être traitée avec attention :
      - Un processus est placé dans l'état `TASK_INTERRUPTIBLE`
      - Puis reçoit un signal
      - Est réveillé par le noyau
      - Le *handler* correspondant s'exécute
      - Celui-ci terminé, le processus passe en état `TASK_RUNNING`, et il positionne la variable `errno` à `EINTR`.
    - Or, il ne faut pas reprendre l'exécution d'un processus qui est bloqué après un appel système. Deux solutions permettent de résoudre ce problème :
      - Relancer « manuellement » l'exécution de l'appel système lorsque celui-ci échoue avec un code d'erreur `errno` valant `EINTR`. Par exemple :

```
do {
    nb_lus=read(desc_fic, buffer, nb_a_lire);
} while ((nb_lus == -1)&&(errno == EINTR));
```

      - Sinon, utiliser `int siginterrupt(int sig, int flag);`
        - Cette fonction est appelée après l'installation du gestionnaire de signal associé au signal `sig`.

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Attendre un signal
    - L'appel à la fonction `pause()` bloque un processus dans l'attente de la délivrance d'un signal :

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

    - Note : `pause()` renvoie toujours -1, et `errno` est positionné à la valeur `EINTR`.

La gestion des signaux sous Linux

- Programmation avec signaux en C
  - Armer une temporisation
    - La primitive `alarm()` permet d'armer une temporisation. A la fin de cette temporisation, le signal `SIGALRM` est délivré au processus.

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

    - Note : `nb_sec = 0` permet d'annuler l'alarme.

La gestion des signaux sous Linux

- Les signaux temps réels
  - Des signaux avancés
    - Linux propose aussi un mécanisme de signaux « *temps réels* », qui respectent la norme POSIX 1b.
    - 32 signaux, de 32 à 63 (valeurs variables selon configuration de l'OS), aussi il vaut mieux utiliser les constantes `SIGRTMIN` et `SIGRTMAX`.
    - Les signaux temps réels ont 3 propriétés générales :
      - **Empilement** des occurrences de chaque signal (jusqu'à 1024).
      - Lorsque plusieurs signaux temps réels doivent être délivrés à un processus, le noyau délivre toujours les signaux temps réels de plus petit numéro avant les signaux temps réels de plus grand numéro. Permet un ordre de **priorités** entre les signaux temps réels.
      - Peut **transporter** avec lui, une petite quantité d'informations, délivrées au handler qui lui est attaché.

La gestion des signaux sous Linux

- Les signaux temps réels
  - Envoyer un signal temps réel
    - Peut être réalisé avec la fonction `kill()`, mais cette façon de faire est déconseillée : elle ne permet pas de délivrer d'informations complémentaires au numéro du signal reçu.
    - `sigqueue()` permet de délivrer des informations complémentaires au signal :

```
#include <signal.h>
int sigqueue(
    pid_t pid,
    int sig,
    const union sigval valeur
);
```

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

    - La valeur peut prendre deux formes :
      - Un entier `int` si le membre `sival_int` de l'union est utilisé
      - Un pointeur `void*` si le membre `sival_ptr` de l'union est utilisé

La gestion des signaux sous Linux

- Les signaux temps réels
  - Attacher un *handler* à un signal temps réel
    - Comme un signal temps réel peut transporter une information, un gestionnaire doit être créé avec des arguments spécifiquement définis :

```
void gestionnaire(
    int num_sig,
    struct siginfo *info,
    void *rien
);
```

    - La structure de type `struct siginfo` contient notamment les champs suivants, conformes à la norme POSIX :
      - `int si_signo` : le numéro du signal
      - `int si_code` : information dépendant du signal. Pour les signaux temps réels, cette information indique l'origine du signal
      - `int si_value.sival_int` correspond au champ `sival_int` dans l'appel système `sigqueue()`
      - `void *si_value.sival_ptr` correspond au champ `sival_ptr` dans l'appel système `sigqueue()`

La gestion des signaux sous Linux

- Les signaux temps réels
  - Exécution du gestionnaire de signal temps réel
    - Rappel signaux classiques : plusieurs commutations de contextes
    - Signaux temps réels : commutations trop pénalisantes pouvant être évitées grâces à des primitives d'attentes de signaux :
      - Evite au noyau d'activer le gestionnaire du signal, soit deux commutations.

```
#include <signal.h>
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(
    const sigset_t *set,
    siginfo_t *info,
    const struct timespec timeout
);
```

      - La primitive `sigwaitinfo()`, reste bloquée jusqu'à ce qu'au moins un signal fourni en paramètre lui ait été délivré.
      - La primitive retourne le numéro du signal reçu, et éventuellement les informations associées. Le processus peut alors invoquer le gestionnaire comme un simple appel de procédure, ce qui est **beaucoup moins coûteux en terme de commutations de contexte**.