Module SY5 – Systèmes d'Exploitation

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Cité
L3 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2022-2023

Signaux

Définition

mécanisme d'interruption asynchrone, *i.e.* non corrélé à une demande explicite : l'interruption peut intervenir à tout moment de l'exécution

envoyés par le système pour notifier un événement (chaque signal code un type d'événement), ou par un autre processus

le processus qui envoie un signal n'a pas de moyen de savoir quand le signal sera délivré (ni même s'il l'est)

liste des signaux disponibles : `kill -l`
liste non normalisée (et encore moins la correspondance nom-numéro)

Réactions par défaut

1. terminaison du processus

   `SIGINT`, `SIGTERM`, `SIGKILL`...

2. terminaison + génération d'un fichier *core*

   `SIGQUIT`, `SIGSEGV`...

3. signal ignoré

   `SIGCHLD`, `SIGWINCH`

4. suspension du processus

   `SIGSTOP`, `SIGTSTP`

5. reprise du processus

   `SIGCONT`

Principaux signaux

problèmes matériels : `SIGBUS`, `SIGSEGV`, `SIGILL`, `SIGFPE` (2)

événements « externes » : `SIGCHLD` (3), `SIGPIPE` (1)

job-control :

- `SIGTERM`, `SIGKILL` pour terminer (1)
- `SIGSTOP`, `SIGTSTP` pour suspendre (4)
- `SIGCONT` pour reprendre (5)

`SIGSTOP` et `SIGKILL` ne peuvent être ni ignorés ni captés

Principaux signaux

problèmes matériels : `SIGBUS`, `SIGSEGV`, `SIGILL`, `SIGFPE` ❷

événements « externes » : `SIGCHLD` ❸, `SIGPIPE` ❶

job-control : `SIGTERM`, `SIGKILL`, `SIGSTOP`, `SIGTSTP`, `SIGCONT`
`SIGSTOP` et `SIGKILL` ne peuvent être ni ignorés ni captés

événements liés au terminal :

- `SIGHUP` : déconnexion ❶
- `SIGINT` ❶, `SIGTSTP` ❹, `SIGQUIT` ❷ : `ctrl-C`, `ctrl-Z`, `ctrl-\`
- `SIGTTIN`, `SIGTTOU` ❹ : tentative de lecture/écriture par un processus à l'arrière-plan
- `SIGWINCH` ❸ : redimensionnement

Principaux signaux

problèmes matériels : `SIGBUS`, `SIGSEGV`, `SIGILL`, `SIGFPE` (2)

événements « externes » : `SIGCHLD` (3), `SIGPIPE` (1)

job-control : `SIGTERM`, `SIGKILL`, `SIGSTOP`, `SIGTSTP`, `SIGCONT`

`SIGSTOP` et `SIGKILL` ne peuvent être ni ignorés ni captés

événements liés au terminal :

`SIGHUP`, `SIGINT`, `SIGTSTP`, `SIGQUIT`, `SIGTTIN`, `SIGTTOU`, `SIGWINCH`

auto-notification : `SIGABRT` (2), `SIGALRM` (1)

sans signification prédéfinie : `SIGUSR1`, `SIGUSR2` (1)

Envoyer un signal

```
int kill(pid_t pid, int sig);
int raise(int sig);  /* équivalent à kill(getpid(), sig) */
```

- `sig` est le numéro du signal à envoyer (ou 0 pour tester la faisabilité)
- `pid` indique le(s) processus cible(s) :
  `pid`>0 pour un processus unique,
  0 pour tous les processus appartenant au même groupe
  -1 pour tous les processus (sauf lui-même sous Linux),
  `pid`<-1 pour tous les processus appartenant au groupe -`pid`
- retourne 0, ou -1 en cas d'erreur

un processus `p1` est autorisé à envoyer un signal à un processus `p2` si le propriétaire (réel ou effectif) de `p1` est privilégié, ou s'il est le propriétaire (réel ou effectif) de `p2`

Attendre l'arrivée d'un signal

il est possible de bloquer un processus jusqu'à la réception d'un signal, par exemple avec :

```
int pause(void);
```

le plus simple, ne fait rien d'autre qu'attendre (indéfiniment) ; en cas de retour, retourne -1, avec `errno=EINTR`

```
unsigned int sleep(unsigned int seconds);
```

pour éviter de bloquer indéfiniement... retourne le nombre de secondes restant en cas d'interruption (et de retour), 0 sinon

Redéfinir la réaction

chaque processus peut choisir son comportement à la réception d'un signal particulier (sauf `SIGKILL` et `SIGSTOP`) :

- ignorer le signal : rien ne se passe
- capter le signal et exécuter une fonction particulière (*handler*)
- revenir au comportement par défaut

cela doit être mis en place ***avant*** la réception du signal concerné avec la primitive suivante :

```
int sigaction(int signum, const struct sigaction *act,
                                  struct sigaction *oldact);
```

il existe également une primitive ***obsolète*** et ***absolument pas fiable***

tl;dr : ***ne jamais utiliser*** `signal()`

Redéfinir la réaction

```
int sigaction(int signum, const struct sigaction *act,
                                  struct sigaction *oldact);
```

- `signum` est le signal concerné par la modification
- `act` décrit le comportement souhaité (si non `NULL`)
- `oldact` permet de récupérer le comportement actuel (si non `NULL`)
- retourne 0, ou -1 en cas d'échec

```
struct sigaction {  /* contient entre autres : */
  void     (*sa_handler)(int);
  sigset_t  sa_mask;
  int       sa_flags;
}
```

- `sa_handler` est soit un pointeur vers une fonction explicite, soit `SIG_IGN`, soit `SIG_DFL`
- les deux autres champs permettent de préciser le comportement pendant et après l'exécution du *handler* ; ils peuvent être mis à 0

Comment ça marche ?

chaque processus a sa *table des signaux*, qui inclut (entre autres) :

- une table des signaux pendants, *i.e.* émis mais non encore délivrés ; c'est une *bitmap* – il n'y a ***pas de possibilité de décompte*** des occurrences d'un signal donné reçues
- une table des signaux masqués – cf plus loin
- une table des *handlers* : `SIG_IGN`, `SIG_DFL` ou un pointeur de fonction (donc vers la zone de texte du processus)

la table des signaux pendants est a priori incluse dans la table des processus, et mise à jour lors de l'envoi d'un signal (donc alors que le processus cible n'est pas actif), puis lors du traitement

à certaines occasions (non spécifiées, mais souvent lors de la bascule entre mode noyau et mode utilisateur), le processus actif prend connaissance des signaux pendants et en traite un (pas de critère de choix spécifié)