Module SY5 – Systèmes d'Exploitation

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université de Paris (Diderot)
L3 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2021-2022

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)

Que mettre dans un *handler* ?

le processus peut avoir été interrompu absolument n'importe quand, en plein milieu d'une opération complexe agissant sur des variables globales par exemple – voire des variables « cachées », comme les structures utilisées pour la gestion des zones allouées ou les entrées/sorties de haut-niveau

on ne peut pas mettre n'importe quoi dans un *handler*, au risque d'interférer et de rendre ces structures incohérentes

il faut se limiter aux fonctions dites *async signal-safe*, cf
`man 7 signal-safety`

en particulier, ni `malloc`, ni `free`, ni `printf` ne sont sûres

Que mettre dans un *handler* ?

il faut se limiter aux fonctions dites *async signal-safe*

en particulier, ni `malloc`, ni `free`, ni `printf` ne sont sûres

une bonne pratique peut être de limiter le rôle du *handler* à la modification d'une variable globale dédiée de type `volatile sig_atomic_t`, pour déléguer le traitement réel au programme prinicipal

```
volatile sig_atomic_t signal_recu = 0;
void handler () { signal_recu = 1; }
int main () {
  ... /* mise en place du handler */
  while (1) {
    if (signal_recu == 1) { ... }
    ...
  }
}
```

Appels systèmes interrompus

sauf si `sa_flags` inclut `SA_RESTART`, un appel système (bloquant) interrompu par la réception d'un signal ne reprend pas : il retourne -1 et `errno=EINTR`

il faut donc en tenir compte tout au long d'un programme susceptible de recevoir des signaux

Exemple :

```
do {
  rc = write(...);
} while(rc < 0 && errno == EINTR);
```

Masquage de signaux

une solution pour améliorer la fiabilité des signaux en retardant le moment où un signal est délivré (pour éviter les sections critiques)

masque = ensemble de signaux (temporairement) masqués/bloqués

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- l'ancien masque est stocké dans `oldset` (si non `NULL`)
- si `set` est `NULL`, pas de changement, sinon le nouveau masque est déterminé à partir de `set` et `how`
- `how` vaut `SIG_BLOCK` (ajout au masque), `SIG_UNBLOCK` (suppression) ou `SIG_SETMASK` (remplacement du masque)

Manipulation des ensembles

```c
/* initialisation */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);

/* modification */
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

/* test d'appartenance */
int sigismember(const sigset_t *set, int signum);
```

Masquage durant l'exécution d'un *handler*

sauf si `sa_flags` inclut `SA_NODEFER`, le signal en cours de gestion est masqué

`sa_mask` permet de définir un masque additionnel (*i.e.* en plus du masque défini par `sigprocmask`)

Relâchement du masquage

attention à ce genre de situations :

```
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
... /* section critique protégée des interruptions par le masquage */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();      /* pb si le signal attendu arrive avant la pause */
```

il est ***indispensable*** de procéder au relâchement + attente de manière ***atomique*** $\Longrightarrow$ il faut un appel système spécifique

```
int sigsuspend(const sigset_t *mask);
```

- de manière atomique, remplace *temporairement* le masque par `mask` et met le processus en attente
- (si le processus ne termine pas) replace l'ancien masque avant de retourner -1, avec `errno=EINTR`