Programmation Système VI
Signaux

Juliusz Chroboczek

19 mars 2016

1 Notifications asynchrones

Dans ce cours, ainsi que dans les cours de L3, nous avons souvent demandé au système de nous notifier d'un événement. Par exemple, lorsqu'un processus fait un `read` ou `select` sur une *socket* ou un tube, il demande a être réveillé lorsque les données seront disponibles.

Toutes les notifications que nous avons vues jusque là sont *synchrones*, ou sens qu'elles sont communiquées au processus lorsqu'il les demande[1]. Il est parfois utile de recevoir une notification *asynchrone*, dès qu'un événement a lieu même si on ne l'attend pas explicitement.

Sous Unix, les notifications asynchrones sont représentées par un *signal*. Un processus ou le noyau peut à tout moment envoyer un signal a un processus. Un *gestionnaire de signal* est exécuté par le processus qui a reçu le signal.

1.1 Quelques signaux utiles

La liste complète des signaux peut être obtenue à l'aide de la commande du *shell* `kill -l`. On peut citer en particulier :

- `SIGSEGV`, `SIGBUS` et `SIGILL` qui sont envoyés à un processus qui a effectué un accès illégal à la mémoire ou une opération illégale ;
- `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGKILL` qui sont envoyés pour demander la terminaison d'un processus — en particulier, le pilote de clavier envoie `SIGINT` pour `^C` et `SIGQUIT` pour `^\` ;
- `SIGPIPE` qui est généré lors d'une écriture sur un tube fermé ;
- `SIGUSR1` et `SIGUSR2` qui sont réservés à l'utilisateur.

Chacun de ces signaux a une *action par défaut*, une action qui est effectuée par un processus qui le reçoit en l'absence de dispositions explicites. `SIGINT` et `SIGPIPE` terminent le programme par défaut, les autres signaux génèrent un *core dump*[2] puis terminent le programme. Voyez `man 7 signal` pour plus de détails.

1. Attention, le mot *synchrone* est ici utilisé différement du sens qu'il a lorsqu'on parle de communication synchrone ou asynchrone.
2. Une image de la mémoire stockée dans un fichier nommé `core` et exploitable par un déboggueur.

2 Envoi de signaux

On peut envoyer un signal a un processus à l'aide de l'appel système `kill` :

```
int kill(pid_t pid, int sig);
```

La fonction `raise` envoie un signal au processus courant : un appel à `raise(signo)` est équivalent à `kill(getpid(), signo)`. Enfin, un appel à `abort` débloque le signal `SIGABRT` puis invoque `raise(SIGABRT)`, ce qui cause un *core dump* puis termine le programme.

3 Gestion des signaux

Traditionnellement, les signaux étaient capturés à l'aide de l'appel système `signal`. Cependant, `signal` n'est pas portable, et c'est `sigaction` qu'il faut utiliser.

L'appel système `sigaction` a le prototype suivant :

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

Le paramètre `signum` est le numéro du signal à capturer. Le paramètre `act` décrit la nouvelle disposition du signal ; le paramètre `oldact` vaudra `NULL` pour nous.

La structure `sigaction` est définie par :

```
struct sigaction {
     void     (*sa_handler)(int);
     void     (*sa_sigaction)(int, siginfo_t *, void *);
     sigset_t   sa_mask;
     int        sa_flags;
 };
```

Le champ `sa_handler` définit l'action à effectuer ; c'est soit un pointeur sur un gestionnaire (une fonction prenant un paramètre entier), soit une des constantes `SIG_DFL` (action par défaut) ou `SIG_IGN` (ignorer le signal). Le champ `sa_flags` vaudra 0 dans ce cours, `sa_mask` vaudra 0 pour le moment, et `sa_sigaction` vaudra `NULL`. Un exemple est donné dans la figure 1.

Vous remarquerez que le gestionnaire utilise un appel à `write` plutôt que `printf` — les signaux sont asynchrones, et il n'est pas sûr que les structures de données de `stdio` soient dans un état cohérent lorsqu'il est invoqué.

4 Les signaux sont asynchrones

Un gestionnaire de signal peut s'exécuter à tout moment. De ce fait, un gestionnaire de signal ne doit appeler aucune fonction qui peut dépendre de la consistance de structures de données globales. Un gestionnaire de signal ne peut effectuer de façon fiable que les actions suivantes :

– lire et affecter une variable globale de type `volatile sig_atomic_t`, et

```
void
bennon_handler(int signo)
{
    write(1, "Ben non.\n", 8);
}

...

struct sigaction sa;
int rc;

memset(&sa, 0, sizeof(sa));
sa.sa_handler = bennon_handler;
sa.sa_flags = 0;
rc = sigaction(SIGINT, &sa, NULL);
if(rc < 0) {
    perror("sigaction");
    exit(1);
}
```

Figure 1 — Utilisation triviale de `sigaction`

– appeler un petit nombre d'appels système et de fonctions documentées comme *async signal-safe* dans la norme POSIX.

En pratique, je connais trois techniques pour écrire des gestionnaires de signal fiables : écrire des gestionnaires triviaux qui n'appelent que des fonctions *async signal-safe*, convertir les notifications asynchrones en notifications synchrones à l'aide d'une variable globale, et convertir les notifications en notifications synchrones à l'aide de `sigblock`.

4.1 Gestionnaires triviaux

Dans certains cas, le gestionnaire de signal n'a pas besoin d'accéder aux structures de données du programme, et peut être entièrement écrit à l'aide de fonctions *async signal-safe*. C'est le cas dans le fragment de code de la figure 1.

4.2 Utilisation d'une variable globale

Si le programme est écrit à l'aide d'une boucle à événements, il passe régulièrement au même endroit de la boucle. Il est alors possible d'écrire un gestionnaire d'événement qui signalise la capture d'un signal à l'aide d'une variable globale, et c'est le programme principal qui s'occuppera d'effectuer l'action désirée, en contexte synchrone.

Un exemple de cette technique est donné dans la figure 2.

```
volatile sig_atomic_t quitter = 0;

void
quit_handler(int signo)
{
    quitter = 1;
}

int
main()
{
    struct sigaction sa;
    int rc;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = quit_handler;
    sa.sa_flags = 0;
    rc = sigaction(SIGINT, &sa, NULL);
    if(rc < 0) abort();

    while(!quitter) {
        printf("J'ai pas encore fini.\n");
        sleep(1);
    }
    printf("J'ai fini.\n");
    return 0;
}
```

Figure 2 — Conversion d'un signal asynchrone en événement synchrone

4.3 Signaux bloqués

Il est possible de demander au système de retarder la livraison *(delivery)* d'un signal au processus. Un signal dont la livraison est retardée est dit *bloqué*, et il est livré lorsqu'il est *débloqué*.

Il existe toute une zoologie d'appels système permettant de bloquer et débloquer les signaux. L'appel système portable est `sigprocmask`, qui manipule un ensemble de signaux représentés par un `sigset_t`.

Manipulation des ensembles de signaux Deux fonctions sont utiles pour manipuler les ensembles de signaux représentés par le type opaque `sigset_t` :

```
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
```

La fonction `sigemptyset` initialise son paramètre à l'ensemble vide. La fonction `sigaddset` ajoute le signal `signum` à son paramètre `set`.

Blocage des signaux Un ensemble de signaux est bloqué ou débloqué à l'aide de `sigprocmask` :

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

Le paramètre `how` indique l'action à effectuer : s'il vaut `SIG_BLOCK`, les éléments de `set` sont ajoutés à l'ensemble de signaux bloqués ; s'il vaut `SIG_UNBLOCK`, ils sont supprimés, et s'il vaut `SIG_SETMASK`, l'ensemble des signaux bloqués est remplacé par `set`. Dans tous les cas, l'ancien ensemble de signaux bloqués est stocké dans `oldset` (qui peut valoir `NULL` si cette information ne nous intéresse pas.

La figure 3 donne un exemple artificiel d'utilisation de cet appel système ; nous verrons un exemple plus convaincant dans le cours suivant.

5 Signaux et appels système bloquants

Lorsqu'un signal est livré pendant qu'un appel système est bloqué, l'appel système retourne -1 avec `errno` valant `EINTR`[3]. Il est *essentiel* de gérer ce cas dans tout programme qui peut recevoir des signaux durant son fonctionnement normal. Nous verrons des exemples dans le cours suivant.

3. Il est possible de désactiver ce mécanisme en spécifiant `SA_RESTART` dans le champ `sa_flags` lors de l'appel à `sigaction`. En pratique, ce mécanisme est rarement utile, car on voudrait spécifier le comportement selon l'appel système, pas selon le signal.

```
volatile sig_atomic_t value = 42;

void
handler(int signo)
{
    char buf[40];
    int rc;
    rc = snprintf(buf, 40, "%d\n", (int)value);
    if(rc >= 0 && rc < 40)
        write(1, buf, rc);
}

int
main()
{
    struct sigaction sa;
    int rc;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handler;
    rc = sigaction(SIGUSR1, &sa, NULL);
    if(rc < 0) abort();

    while(1) {
        sigset_t old, new;
        sigemptyset(&new);
        sigaddset(&new, SIGUSR1);
        rc = sigprocmask(SIG_BLOCK, &new, &old);
        if(rc < 0) abort();
        value = 57;
        sleep(1);
        value = 42;
        rc = sigprocmask(SIG_SETMASK, &old, NULL);
        if(rc < 0) abort();
    }

    return 0;
}
```

Figure 3 — Un exemple d'utilisation de signaux bloqués. Ce programme affiche toujours 42 lorsqu'il reçoit un signal `SIGUSR1`, avec au plus une seconde de délai.