Signaux

François Trahay

Contents

Signaux

Rappel (CSC3102)

- Signal: mécanisme de communication inter-processus
- Message: un entier entre 1 et 31
- Ordre de réception aléatoire (différent de l'ordre d'émission)
- Une routine de réception est automatiquement invoquée chez le récepteur dès que le signal arrive

---

Envoyer un signal

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

  - Envoie le signal `sig` au processus `pid`
  - Quelle valeur pour `sig`?
    * valeur entière (par ex: `9`): pas portable (dépend de l'OS)
    * constante (par ex: `SIGKILL`) définie dans `signal.h`

Voici un exemple de programme utilisant la fonction `kill`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>

int main(int argc, char**argv) {
  if(argc != 2) {
    fprintf(stderr, "usage: %s PID\n", argv[0]);
    return EXIT_FAILURE;
  }

  pid_t pid = atoi(argv[1]);
  int signo = SIGKILL;

  printf("Sending signal %d to %d\n", signo, pid);
  kill(pid, signo);

  return EXIT_SUCCESS;
}
```

Recevoir un signal

- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - Spécifie le comportant lors de la réception du signal `signum`
  - `struct sigaction` est une structure de la forme:

```c
struct sigaction {
        void     (*sa_handler)(int); // pointeur sur la fonction à appeler
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
};
```

Signaux interceptables

Il est possible d'utiliser `sigaction` pour "intercepter" tout signal sauf les signaux `SIGKILL` et `SIGSTOP`

`struct sigaction`

La valeur prise par `sa_handler` est: * l'adresse d'une fonction (par ex: `void signal_handler(int signo)` * Le paramètre `signo` est le numéro du signal reçu * la valeur `SIG_DFL` pour restaurer l'action par défaut (tuer le processus) * la valeur `SIG_IGN` pour ignorer le signal: à la réception de ce signal, aucune action ne sera effectuée

Sauf cas d'usages particuliers, les autres champs de la structure `sigaction` sont à mettre à `0`.

`oldact`

La fonction `sigaction` modifie le comportant du processus lorsqu'il reçoit le signal `signum`. Si `oldact` n'est pas `NULL`, l'ancien comportement y est stocké.

Variables globales

Si la fonction traitant le signal manipule des variables globales, il est conseillé de les déclarer `volatile`. Par exemple:

```
volatile int var;
```

Lorsqu'une variable est déclarée `volatile`, le compilateur limite les optimisations faites sur cette variable. Par exemple, le compilateur ne met pas en cache (dans un registre) la variable.

Si une fonction (par exemple `foo`) qui manipule la variable `var` non `volatile` est interrompue par un traitant de signal (`sig_handler`) modifiant `var`, la modification de la variable risque de ne pas être "vue" par `foo` qui travaille sur une copie en cache de la variable. La fonction `foo` risque donc de travailler sur une version obsolète de la variable.

Exemple

Voici un exemple de programme utilisant `sigaction` pour intercepter un signal:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>

/* Function to call upon signal reception */
void signal_handler(int signo) {
  printf("Received: signal %d\n", signo);
}

int main(int argc, char**argv) {
```

```c
  if(argc != 2) {
    fprintf(stderr, "usage: %s signo\n", argv[0]);
    return EXIT_FAILURE;
  }

  /* Initialize the sigaction structure */
  int signo = atoi(argv[1]);
  struct sigaction s;
  s.sa_handler = signal_handler;

  /* Install the signal handler */
  printf("Installing signal handler for signal %d\n", signo);
  int retval = sigaction(signo, &s, NULL);
  if(retval<0) {
    perror("sigaction failed");
    abort();
  }

  /* Wait to receive signals */
  while(1) {
    printf("[%d] Sleeping...\n", getpid());
    sleep(1);
  }

  return EXIT_SUCCESS;
}
```

Attendre un signal

- `int pause( );`
  - Attend qu'un signal (non ignoré) soit reçu

Programmer une alarme

- `int alarm(unsigned int s);`
  - Programme l'envoi de `SIGALRM` après `s` secondes

Pour programmer une alarme avec une granularité plus fine, utilisez la fonction `setitimer`. Cette fonction permet de programmer des alarmes périodiques (qui se répètent) avec une granularité de l'ordre de la microseconde.

Pour aller plus loin

Les 3 sous-sections suivantes présentent des notions pour les étudiant · e · s Ninja qui ont vraiment envie d'aller encore plus loin.

`sigsetjmp` et `siglongjmp`

Permet de faire un "saut (`goto`) non local"

- `int sigsetjmp(sigjmp_buf env, int savesigs);`
  - sauvegarde l'environnement d'appel courant (pile d'appel, pointeur d'instruction, etc.)
- `void siglongjmp(sigjmp_buf env, int val);`
  - restaure l'environemment sauvegardé `env`
  - le programme continue son exécution comme s'il retournait de la fonction `sigsetjmp`

Ces fonctions peuvent être utile pour la gestion des signaux :

- Avant de mettre en place le gestionnaire de signaux, l'appel à `sigsetjmp` permet de définir l'endroit du code où revenir après l'appel à `siglongjmp`.
- Dans le gestionnaire de signaux (i.e. la fonction `signal_handler`); appeler `siglngjmp`.

Elles sont également utilisées pour mettre en place des mécanismes d'exception.

Exemple : suivez le lien pour visualiser cet exemple tiré du livre "Développement système sous Linux" de Christophe BLAESS. FYI, ce livre est disponible à la médiathèque.

Programmation événementielle

libuv permet de faire de la programmation événementielle

- "Asynchronous I/O made simple" dit le site de `libuv`.
- "Networking in `libuv` is not much different from directly using the BSD socket interface, some things are easier, all are non-blocking, but the concepts stay the same. In addition libuv offers utility functions to abstract the annoying, repetitive and low-level tasks like setting up sockets using the BSD socket structures, DNS lookup, and tweaking various socket parameters." (extrait du chapitre Networking de la documentation libuv).

Pour récupérer des exemples, c'est ici.

Il existe aussi d'autres bibliothèques, mais qui nous semblent moins pertinentes (vu leurs dernières dates de mise à jour): libevent et libev.

Coroutines

- inconvénient de la programmation événementielle: définition de nombreuses functions de *callback*
  - callback appelé quand un appel asynchrone est terminé
  - réduit la lisibilité du code
- solution: **coroutines**
  - possibilité de suspendre l'exécution d'une fonction pour la reprendre plus tard

Cet extrait de l'exemple de `tcp-echo-server` de `libuv` (cf. `libuv-1.48.0/docs/code/tcp-echo-server/main.`
illustre le problème de lisibilité:

```c
void echo_write(uv_write_t *req, int status) {
    if (status) {
        fprintf(stderr, "Write error %s\n", uv_strerror(status));
    }
    free_write_req(req);
}

void echo_read(uv_stream_t *client, ssize_t nread, const uv_buf_t *buf) {
    if (nread > 0) {
        write_req_t *req = (write_req_t*) malloc(sizeof(write_req_t));
        req->buf = uv_buf_init(buf->base, nread);
        uv_write((uv_write_t*) req, client, &req->buf, 1, echo_write);
        return;
    }
    // ...
}

void on_new_connection(uv_stream_t *server, int status) {
    // ...
    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
    // ...
}
```

- Quand une nouvelle connexion est établie avec ce serveur, la fonction `on_new_connection` est appelée. Cette fonction appelle la fonction `uv_read_start` qui contient un paramètre `echo_read`, i.e. la fontion à appeler quand la lecture sur la socket par `uv_read_start` sera terminée.
- Et `echo_read` appelle la fonction `uv_write` avec le paramètre `echo_write`, i.e. la fonction à appeler quand l'écriture sur la socket sera terminée.

Ce serait bien d'avoir une seule fonction qui enchaîne ces 3 codes. Cela supposerait une fonction capable de relâcher la main en plein milieu de son exécution et reprendre à cet endroit quand on lui redonnerait la main. C'est la notion de coroutine.

Hélas, cette notion n'est pas implémentée en langage C. Mais, il existe des contournements, cf. articles que nous vous invitons à lire :

- Coroutine in C Language
- Coroutines in C