Programmation Unix 2 – cours n°8

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Janvier 2017

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°8

1. Réception des signaux
2. Envoi avec donnée
3. Compléments

1 - Réception des signaux

Vie d'un signal :

1. Un signal est généré par `kill` ;
2. il est en attente d'être délivré par le système ;
3. il est délivré au processus cible.

Un signal généré peut être bloqué par le système = il reste en attente.

Un processus peut placer un masque de signaux = une liste de signaux à bloquer.

Masque enlevé $\rightarrow$ les signaux en attente sont délivrés.

Atomicité

- Règle pour les signaux standards :

  une seule occurrence d'un signal X peut être en attente.

  $\rightarrow$ si un signal X est généré alors qu'un signal X est déjà en attente, il est perdu.

- Signaux temps réels (`SIGRTMIN .. SIGRTMAX`) :

  plusieurs occurrences d'un signal peuvent être en attente, voir `man getrlimit`

Handler de signal

Fonction appelée par le système lorsqu'il délivre un signal `sig` pour que le processus capte ce signal.

Prototype : `void handler (int sig);`

Handlers fournis :

- `SIG_DFL` : handler par défaut
- `SIG_IGN` : pour ignorer un signal

Comportement de `SIG_DFL` selon `sig` :

(a) Terminaison du processus
(b) Terminaison avec image mémoire
(c) Ignorer le signal
(d) Processus suspendu
(e) Processus repris

Installer un handler avec `signal()`

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal (int sig, sighandler_t func);
```

Installe le handler `func` pour le signal `sig`.
Renvoie l'ancien handler, ou `SIG_ERR`.

Exemple :

```
void capter (int sig)
{
  printf ("Reçu signal %d\n", sig);
}
int main ()
{
  signal (SIGTERM, capter);
  signal (SIGQUIT, capter);
  signal (SIGHUP, SIG_IGN);
  ...
}
```

Problèmes de portabilité

Interface historique : le comportement de `signal` varie selon

- le système ;
- les versions du système.

Exemple : lorsqu'un signal est capté,

- sur SysV : `SIG_DFL` est réinstallé
- sur BSD : le handler est maintenu

Pendant l'appel d'un handler (= la délivrance du signal),

- sur SysV : le handler peut être interrompu pour le même signal
- sur BSD : le signal est masqué

Linux $\sim$ SysV ; MacOS $\sim$ BSD

Réarmer `signal()` sur SysV et Linux

Exemple :

```
void capter (int sig)
{
  signal (SIGTERM, capter); // on réarme en premier
  printf ("Reçu signal %d\n", sig);
}
int main ()
{
  signal (SIGTERM, capter); // une fois au début de main
  while (1) {
      faire_quelque_chose();
  }
}
```

Fragile : dans capter, `SIG_DFL` avant réarmement
+ `SIGTERM` non masqué

Ne pas utiliser `signal()`

Seul usage portable de `signal()` : pour installer `SIG_DFL` ou `SIG_IGN`.

Solution : utiliser `sigaction()`

Installer un handler avec `sigaction()`

Complètement normalisé par POSIX, mais plus compliqué.

```
#include <signal.h>

int sigaction (int sig, const struct sigaction *act,
                                    struct sigaction *old);
struct sigaction {
  void     (*sa_handler)(int);
  sigset_t   sa_mask;
  int        sa_flags;
};
```

Installe le handler `act->sa_handler` pour le signal `sig`.

Si `old` $\neq$ `NULL`, récupère l'ancien handler dans `old->sa_handler`.

Renvoie 0 succès, -1 erreur.

Exemple

```
void capter (int sig)
{
  printf ("Reçu signal %d\n", sig);
}
int main ()
{
  struct sigaction act;
  act.sa_handler = capter;
  sigemptyset (&act.sa_mask);
  act.sa_flags = 0;
  sigaction (SIGTERM, &act, NULL);
  sigaction (SIGQUIT, &act, NULL);
  act.sa_handler = SIG_IGN;
  sigaction (SIGHUP, &act, NULL);
  ...
}
```

Masquer des signaux

Avec `sigaction`, un signal délivré `sig` est toujours masqué.
On peut rajouter des signaux dans le masque (en plus de `sig`) :

```
struct sigaction act;
sigemptyset (&act.sa_mask);
sigaddset (&act.sa_mask, SIGTERM);
...
sigaction (sig, &act, NULL);
```

Liste des opérations possibles :

```
#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);
```

Options

De nombreux paramétrages sont possibles.

```
act.sa_flags = flag1 | flag2 | ...;
```

- Par défaut, le handler est maintenu après délivrance.

  On peut demander la réinstallation de `SIG_DFL` :

  ```
  act.sa_flags |= SA_RESET_HAND;
  ```

- Par défaut, tout appel bloquant est interrompu par un signal, puis échoue avec `errno = EINTR`.

  On peut demander que les appels bloquants soient silencieusement repris : `act.sa_flags |= SA_RESTART;`

  → plus facile à gérer : aucun code supplémentaire

Une fonction utilitaire

```
int mysignal (int sig, void (*h)(int), int options)
{
  struct sigaction s;
  s.sa_handler = h;
  sigemptyset (&s.sa_mask);
  s.sa_flags = options;
  int r = sigaction (sig, &s, NULL);
  if (r < 0) perror (__func__);
  return r;
}
```

Exemple d'usage :

```
void capter (int sig)
{
  printf ("Reçu signal %d\n", sig);
}
int main ()
{
  mysignal (SIGTERM, capter, SA_RESTART);
  mysignal (SIGQUIT, capter, SA_RESTART);
  mysignal (SIGHUP, SIG_IGN, 0);
  ...
}
```

Compilation avec gcc

Pour compiler en C99 (ou C ANSI) avec les types et fonctions POSIX sur les signaux, il faut déclarer `_GNU_SOURCE`

- dans la ligne de commande :

```
gcc -Wall -W -std=c99 -D_GNU_SOURCE ex1.c -o ex1
```

- ou dans le source :

```
#define _GNU_SOURCE  // en premier
#include <stdio.h>
#include <signal.h>
```

puis :

```
gcc -Wall -W -std=c99 ex1.c -o ex1
```

2 - Envoi avec donnée

On peut envoyer un signal avec une donnée :

```
#include <signal.h>

int sigqueue(pid_t pid, int sig,
             const union sigval value);

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

`sigqueue()` fonctionne comme `kill()`.

Exemple

```
union sigval value;
value.sival_int = 123;
sigqueue (pid, sig, value);
```

ou encore

```
struct { int foo; char bar; } toto = { 123, 'a' };
union sigval value;
value.sival_ptr = &toto;
sigqueue (pid, sig, value);
```

Recevoir une donnée

Lorsqu'un signal est envoyé avec une donnée par `sigqueue`, on peut retrouver cette donnée en mettant `SA_SIGINFO` dans `sa_flags`, et en fixant `sa_sigaction` à la place de `sa_handler` :

```
struct sigaction {
    void  (*sa_handler)(int);
    void  (*sa_sigaction)(int, siginfo_t *, void *);
    ...
};
```

Le premier paramètre est le signal ; le second contient la donnée ; le troisième peut recevoir un contexte (inutile, mettre `NULL`).

Réception d'informations

Un signal capté par un handler `sa_sigaction` reçoit :

```
struct siginfo_t {
  ...
  pid_t    si_pid;
  uid_t    si_uid;
  sigval_t si_value;
  int      si_code;
  ...
}
```

La donnée envoyée par `sigqueue` est dans `si_value`.

De plus `si_code` décrit la cause de l'envoi :

`SI_USER`	par `kill` ou `raise`
`SI_QUEUE`	par `sigqueue`
`SI_KERNEL`	par le noyau
`...`	une 50$^{\text{aine}}$ de cas prévus !

Exemple de réception d'informations

```
void capter (int sig, siginfo_t *info, void *ctx)
{
  printf ("Reçu signal %d\n", sig);
  if (info->si_code == SI_QUEUE)
    printf ("PID émetteur %d, donnée %d\n",
      (int) info->si_pid, info->si_value.sival_int);
}

int main ()
{
  struct sigaction act;
  act.sa_sigaction = capter;  /* au lieu de sa_handler */
  sigemptyset (&act.sa_mask);
  act.sa_flags = SA_RESTART | SA_SIGINFO;
  sigaction (SIGTERM, &act, NULL);
  ...
}
```

Limitations

- Pas de moyen pour savoir s'il faut utiliser `si_value.sival_int` ou `si_value.sival_ptr`

- `sival_ptr` : uniquement pour des adresses partagées

3 - Compléments

- L'appel d'un handler peut avoir lieu lorsque le processus
  - revient d'une interruption matérielle ;
  - vient d'être élu par l'ordonanceur ;
  - revient d'un appel système.

- Un signal est sans effet sur un zombie, même `SIGKILL`.

Signaux captables

- Tous les signaux sont captables sauf :

  `SIGKILL`, `SIGSTOP`, `SIGCONT`

- La différence entre `SIGSTOP` et `SIGTSTP` :

  `SIGTSTP` peut être capté.

- Si le processus est stoppé (il a reçu `SIGSTOP` ou `SIGTSTP`)
  - un signal `SIGTERM` ou `SIGCONT` le réveille ;
  - les autres signaux seront délivrés au réveil.

Duplication et recouvrement

- Lors de la duplication avec `fork` :
  - les handlers sont conservés ;
  - les signaux en attente sont supprimés dans le fils.

- Lors d'un recouvrement avec `exec*` :
  - les handlers sont remis à `SIG_DFL` ;
  - sauf pour `SIG_IGN` : les signaux ignorés restent ignorés.
    But : protéger un programme de signaux.

Exemple (1/3)

Programme `protecsig.c` qui se recouvre avec une commande et la protège d'une liste de signaux.

Usage : `protecsig sig ... -r com [arg ...]`

Première étape : position de `"-r"`

```
int main (int argc, char *argv[])
{
    int ind_r;
    for (ind_r = 1; ind_r < argc; ind_r++)
        if (!strcmp (argv[ind_r], "-r")) break;
    if (ind_r <= 1 || ind_r >= argc-1) {
        fprintf (stderr, "Usage : %s sig ... -r com "
                         "[arg ...]\n", argv[0]);
        exit (1);
    }
```

Exemple (2/3)

Deuxième étape : on installe `SIG_IGN` pour les signaux.

```
for (int i = 1; i < ind_r; i++) {
    int sig = atoi (argv[i]);
    if (sig <= 0 || sig >= NSIG) {
        fprintf (stderr, "Erreur, signal '%s' "
                         "incorrect\n", argv[i]);
        continue;
    }
    if (mysignal (sig, SIG_IGN, SA_RESTART) < 0)
        fprintf (stderr, "Erreur, signal %d non "
                         "captable\n", sig);
    else printf ("Protégé du signal %d\n", sig);
}
```

Exemple (3/3)

Troisième étape : recouvrement.

```
        printf ("Exécution de la commande ...\n");
        execvp (argv[ind_r+1], argv+ind_r+1);
        perror ("exec");
        exit (1);
    }
```

Exemple : `sleep 60`
Dans un autre terminal : `killall -1 sleep`
Essayer avec -2, -3, -9

Recommencer avec : `./protecsig 1 2 9 -r sleep 60`

Suppression automatique des zombies

Dans le père :

```
mysignal (SIGCHLD, capter_SIGCHLD, SA_RESTART);

void capter_SIGCHLD ()
{
    int p;
    while ((p = waitpid (-1, NULL, WNOHANG)) > 0)
        printf ("Zombie %d supprimé\n", p);
}
```

Rappel : `waitpid` renvoie -1 : pas de fils ni zombie,
0 : pas de zombie,
> 0 : PID du zombie supprimé.