Systèmes d'exploitation — Exercices
Évènements

Seweryn Dynerowicz (Seweryn.DYNEROWICZ@umons.ac.be)

1 Signaux

Il est possible d'adopter une manière différente d'organiser la logique d'un programme. Par contraste avec la programmation *impérative* dans laquelle l'ordre des instructions détermine l'ordre des traitements, il est possible d'associer un traitement (décrit par une fonction) qui doit être effectué lors de la réception d'un signal correspondant par le processus en cours d'exécution [1].

Afin d'implémenter un programme dans ce paradigme, le système GNU/Linux met à disposition des mécanismes d'envoi et de réception de signaux par les processus, dont l'interface est déclarée dans `signal.h`. La plupart de ces signaux sont typiquement émis par le noyau et peuvent faire le lien avec une exception du processeur (*e.g.* `SIGFPE`, `SIGILL`). La table suivante reprend les signaux les plus courant, leur description ainsi que l'action par défaut. Ces signaux sont, par défaut, soit ignoré, soit donne lieu à une *terminaison* qui peut produire un fichier `core` contenant une sauvegarde du contexte d'exécution au moment de la réception du signal [2]. Lorsqu'un processus a associé une fonction de gestion à un certain signal, on dit que ce processus *capture* ce signal. **Attention :** il existe deux signaux qui ne sont pas capturables ; `SIGKILL` (terminaison inconditionnelle) et `SIGSTOP` (suspension inconditionnelle).

Nom	Description	Note	Action
`SIGABRT`	Arrêt prématuré du processus	*cfr.* `abort(..)`	`Core`
`SIGALRM`	Alarme temporisée	*cfr.* `alarm(..)`	Terminaison
`SIGCHLD`	Terminaison/arrêt d'un processus enfant	*cfr.* `wait(..)`	Ignoré
`SIGFPE`	Erreur arithmétique (*e.g.* division par 0)	*n/a*	`Core`
`SIGILL`	Instruction illégale (*e.g.* invalide, privilégiée)	*n/a*	`Core`
`SIGINT`	Interruption du processus	Ctrl + c	Terminaison
`SIGKILL`	Arrêt immédiat du processus	*cfr.* `kill`	Terminaison
`SIGUSR1`	Signal utilisateur 1	*n/a*	Terminaison
`SIGUSR2`	Signal utilisateur 2	*n/a*	Terminaison

FIGURE 1 – Table des signaux

Il existe deux fonctions pour envoyer un signal ; la fonction `kill(..)` qui permet d'émettre un signal à destination d'un processus sur base de son *Process IDentifier* (*i.e.* `pid`) et la fonction `raise(..)` qui permet d'émettre un signal à destination de soi-même.

Alternativement, il est possible d'utiliser la commande `kill` pour envoyer un signal à un processus donné. Dans le cas où il n'existe pas de processus ayant le `pid` stipulé, le terminal vous le signalera. Dans le cas contraire, la commande n'affiche aucun *feedback* par défaut.

```
sdy@mentat $ kill -USR1 19768
bash: kill: (19768) - No such process
sdy@mentat $ kill -USR1 19772
sdy@mentat $
```

FIGURE 2 – Envoi d'un signal `SIGUSR1` vers un processus

Pour trouver le `pid` d'un processus, vous pouvez utiliser les commandes `top` ou `ps` et chercher le processus qui vous intéresse. Il est également possible d'utiliser la fonction `getpid()` qui renvoye son `pid` au processus appelant.

1. On entre dès lors dans le paradigme de la programmation dite *évènementielle*
2. Selon le système, ce fichier est placé dans le répertoire courant ou peut être localisé via la commande `coredumpctl`

2 Gestion classique — `signal(..)`

Le canevas typique d'un programme utilisant la gestion classique des signaux est donné dans le listing ci-dessous. La structure du code de la fonction `main(..)` comprend toujours deux phases.

```
1  #include <stdio.h>  // printf(..)
2  #include <stdlib.h> // EXIT_SUCCESS
3  #include <signal.h> // signal(..)
4
5  void handler(int signum) {
6    if(signum == SIGINT)
7      printf("Caught a SIGINT signal.\n");
8  }
9
10 int main(int argc, char* argv[]) {
11   signal(SIGINT, handler);
12
13   while(1) {
14     pause();
15     printf("Waiting loop resumed.\n");
16   }
17
18   return EXIT_SUCCESS;
19 }
```

FIGURE 3 – Programmation *évènementielle* avec `signal(..)`

Dans un premier temps, le programme doit installer des fonctions de gestion pour chaque type de signal qui doit être traité (*cfr.* ligne 11). La valeur de retour de l'appel à `signal(..)` décrit l'état de cette installation ; une valeur de -1 dénote un problème. Dans le cas où aucun problème ne s'est produit, à chaque fois que le signal en question sera reçu par le processus, son exécution sera déroutée de l'endroit où il se trouve vers la fonction de gestion associée.

Dans un second temps, le programme entre dans une boucle infinie qui empêche que la fonction `main(..)` ne se termine et permet de garder le processus en vie pour la durée des traitements. L'utilisation de la fonction `pause()` (ligne 14) a pour effet de suspendre l'exécution du processus jusqu'à ce qu'un signal soit reçu et traité.

Important. De la même façon qu'un programme purement impératif peut se reposer sur des variables *globales* pour faire évoluer son état à travers son fil d'exécution, les fonctions de gestion de signaux que vous utilisez peuvent faire évoluer l'état du processus à travers ces variables *globales*.

Par exemple, suite à l'exécution de la ligne 11, à chaque fois que le processus recevra un signal `SIGINT`, l'exécution sera déroutée vers la fonction `handler`. Au terme de l'exécution de cette fonction, pour autant que le signal reçu ne nécessite pas la terminaison, l'exécution se poursuivra à l'endroit dans la fonction `main(..)` où se trouvait le processus lorsque le signal a été reçu. La boucle infinie de la ligne 13 étant cadencée à l'aide de la fonction `pause(..)`, l'envoi par l'utisateur de trois signaux `SIGINT` successifs produira le listing suivant dans le terminal ;

```
sdy@mentat $ ./program
^CCaught a SIGINT signal.
Waiting loop resumed.
^CCaught a SIGINT signal.
Waiting loop resumed.
^CCaught a SIGINT signal.
Waiting loop resumed.
```

FIGURE 4 – Illustration de réception de signaux

L'implémentation de `signal(..)` n'est pas parfaitement standardisée ; son comportement exact pouvant varier d'un système à un autre. Il est dès lors préférable d'utiliser exclusivement la gestion dite *portable* des signaux que nous décrivons dans la section suivante.

3 Gestion portable — `sigaction(..)`

Le canevas typique d'un programme utilisant la gestion portable des signaux est donné dans le listing ci-dessous. La structure du code de la fonction `main(..)` présente la même structure que pour la gestion classique mais présente deux différences principales.

```
1  #include <stdio.h>  // printf(..)
2  #include <string.h> // memset(..)
3  #include <unistd.h> // pause(..), sleep(..)
4  #include <stdlib.h> // EXIT_SUCCESS
5
6  #define __USE_GNU
7  #include <signal.h>   // struct sigaction, sigaction()
8  #include <ucontext.h> // ucontext_t, REG_RIP
9
10 #define IP(context) ((ucontext_t*) context)->uc_mcontext.gregs[REG_RIP]
11
12 void handler(int signum, siginfo_t* info, void* context) {
13   if(signum == SIGINT) {
14     printf("Caught a SIGINT signal. [rip=%p]\n", IP(context));
15     // Traitement ...
16   }
17 }
18
19 int main(int argc, char* argv[]) {
20   struct sigaction descriptor;
21   memset(&descriptor, 0, sizeof(descriptor));
22   descriptor.sa_sigaction = handler;
23   descriptor.sa_flags = SA_SIGINFO;
24   sigaction(SIGINT, &descriptor, NULL);
25
26   while(1)
27     pause();
28
29   printf("Finished!\n");
30
31   return EXIT_SUCCESS;
32 }
```

FIGURE 5 – Programmation *évènementielle* avec `sigaction(..)`

Premièrement, la signature d'une fonction de gestion (ligne 12) ne se limite plus à simplement recevoir le numéro du signal reçu (*i.e.* `signum`). Le second paramètre, `info`, permet de recevoir des informations additionnelles sur les circonstances entourant la réception du signal. Le troisième paramètre, `context`, permet d'accéder au contexte d'exécution du processus au moment où il a reçu le signal. Les directives des lignes 6 à 10 rendent possible l'accès à la valeur du pointeur d'instruction courante stocké dans le contexte passé en paramètre. La ligne 14 illustre comment il est possible d'afficher le pointeur d'instruction courante [3] lorsque le signal a été reçu.

Deuxièmement, l'installation d'une fonction de gestion repose sur la construction d'un descripteur (ligne 19) au lieu de passer directement la fonction. Cette structure de type `struct sigaction` (ligne 14) sera remplie de 0 à l'aide de la fonction `memset(..)` (ligne 20) pour l'initialiser. La fonction à utiliser sera introduite dans le champ `sa_sigaction` (ligne 21). Le *flag* `SA_SIGINFO` stipule que le paramètre `info` passé à la fonction de gestion devra contenir, au besoin, les informations relatives aux circonstances du déroutement.

L'implémentation de `sigaction(..)` est parfaitement standardisée ; son comportement est identique quelque soit le système. Par contraste avec le comportement ré-entrant lié à `signal(..)`, la gestion des signaux à l'aide de `sigaction(..)` n'est pas ré-entrante ; si un second signal est reçu pendant l'exécution de la fonction de gestion d'un premier signal, le second signal sera mis en attente jusqu'à ce que la fonction de gestion du premier signal se termine.

3. Registres d'instruction courante en `x86` : `eip` en 32 bits, `rip` en 64 bits.

4 Énoncés

Exercice 1

Implémentez un programme qui capture les signaux `SIGUSR1`, `SIGUSR2` et réagit de la façon suivante à la réception d'un de ces signaux ;

— `SIGUSR1` : affiche la chaîne de caractères `"hello !"`.

— `SIGUSR2` : affiche le nombre de signaux `SIGUSR1` reçus pendant l'exécution avant de se terminer.

Exercice 2

Implémentez un programme qui capture le signal `SIGINT` et qui parcourt trois fois de suite toutes les valeurs d'une variable de type `unsigned int`. Votre programme réagit au signal `SIGINT` en affichant le numéro du parcours courant (1, 2 ou 3) et adopte un comportement additionnel selon le nombre d'occurrences de ce signal déjà reçues ;

— 1× : affiche la chaîne de caractères `"Just give me a moment."`

— 2× : affiche la chaîne de caractères `"I said I need a moment!"`

— 3× : affiche la chaîne de caractères `"Fine. I'm out of here."` et se termine

Exercice 3

La conjecture de Collatz [4] stipule que, pour toute valeur initiale $n \geqslant 1$, l'application repétée de la fonction suivante [5] converge vers 1 en un nombre fini d'applications :

$$\text{collatz}(n) = \begin{cases} \frac{n}{2} & n \mod 2 = 0 \\ 3n+1 & n \mod 2 = 1 \end{cases}$$

e.g. Implémentez un programme qui, au départ d'une valeur initiale de `n`, applique de façon répétée une fonction `collatz` jusqu'à ce que la valeur courante de `n` soit égale à 1. Au terme de l'exécution, votre programme affichera le nombre d'applications de `collatz` qui auront été nécessaires. Utilisez la fonction `sleep(..)` pour que votre programme attende 1 seconde après chaque application de la fonction. Ajoutez la possibilité d'interrompre son exécution via un signal `SIGINT`. Dans ce cas de figure, votre programme affichera la valeur atteinte ainsi que le nombre d'applications effectuées.

Exercice 4

Implémentez un programme qui réagit à l'occurrence d'une division par zéro (signal `SIGFPE`) dans son code en passant à l'instruction suivante. **Indications :** Vous devez utiliser `sigaction` pour pouvoir accéder au contexte d'exécution. Pour un processeur `x86`, l'instruction `idiv` est codée sur 2 octets.

Exercice 5

Implémentez un programme qui capture les signaux `SIGINT`, `SIGTERM` et `SIGQUIT`. Votre programme réagit au signal `SIGINT` en entrant dans une boucle infinie dans laquelle il affiche `"I cannot diiiiiiiiie!"` continuellement. Comment pouvez-vous terminer un tel processus ?

Exercice 6

Implémentez un programme qui capture le signal `SIGALRM`. Votre programme réagit à ce signal, mis en place à l'aide de la fonction `alarm()`, en se terminant après un temps prédéfini.

Travail de groupe

Implémentez un programme qui accumule les frappes au clavier à l'aide de la fonction `getchar()` dans un buffer. Toutes les 5 secondes, le programme affiche les caractères accumulés en remplaçant les minuscules par des majuscules et vide le buffer. Utilisez un signal `SIGALRM` comme déclencheur. Si le buffer est vide lors de la réception du signal `SIGALRM`, le programme se termine.

4. `https://en.wikipedia.org/wiki/Collatz_conjecture`
5. En C, `%` représente l'opérateur modulo