TD7 – Signaux

1 Objectif

Ces exercices introduisent à la manipulation élémentaire des signaux.

2 Capture de signaux

Écrire un programme `tst_signal.exe` qui capte, grâce à la primitive d'ANSI C `signal()`, les signaux `SIGSEGV` et `SIGINT` :

- lors de la réception d'un des signaux, le programme doit afficher un message indiquant quel est le signal qui a été capturé
- la réception de 5 signaux `SIGINT` consécutifs doit provoquer la terminaison du programme.

Après avoir mis en place la capture des signaux, le programme principal entrera dans une boucle infinie, du type `while (1) {}` pendant laquelle vous aurez tout le loisir d'envoyer des `SIGINT` et des `SIGSEGV` depuis le shell.

Ecrire un programme `tst_sigaction.exe` analogue utilisant la primitive POSIX `sigaction()` au lieu de `signal()`.

Ecrire un programme `tst_sleep` en remplaçant dans l'un des programmes précédents la boucle d'attente infinie par un appel de la primitive `sleep(5)`.

Pendant les 5 secondes d'attente ainsi obtenues, envoyez `SIGINT`. Que se passe-t-il ? Lisez la page de manuel de `sleep()`.

Indication sur l'envoi de signaux depuis le shell Si un processus a été lancé en avant-plan (*foreground*) on peut lui envoyer le signal `SIGINT` en tapant `^C` au terminal. En revanche, il n'y a pas de raccourci au clavier pour `SIGSEGV`¹, on doit donc taper, depuis une autre fenêtre shell, la commande `kill -SEGV pid`, où `pid` est l'identification du processus visé.

Si un processus a été lancé en arrière-plan (*background*), il ignore les signaux produits au terminal. Mais on peut toujours utiliser la seconde méthode depuis une autre fenêtre shell (`kill -SEGV` ou `kill -INT`).

Indication sur la compilation avec l'option -ansi de gcc Il coexiste en fait deux mécanismes de gestion de signaux : celui d'ANSI C, disponible sur toute plate-forme supportant ce langage (et qui correspond, entre autres, à la primitive `signal()`), et celui de POSIX.1 (`sigaction()`, `sigprocmask()`...), plus complet et plus sûr, mais limité aux systèmes d'exploitation qui implémentent cette norme. Ces deux mécanismes ont des sémantiques qui sont parfois très différentes. D'autre part, le choix de la sémantique utilisée est sensible à la présence (ou l'absence) de l'option `-ansi`, lors de la compilation avec `gcc`.

¹ `SIGSEGV` est un *signal synchrone*, une *exception*, normalement produite par le processus lui-même quand il tente d'accéder à une mauvaise adresse (tentative de déréréférencage du pointeur nul, par exemple).

Pour avoir une concrétisation de l'effet de cette option, compilez (et exécutez) `tst_signal.exe` avec cette option. Vous constaterez, peut-être², que votre handler de signal n'est valide qu'une seule fois, et qu'il faut le réactiver, dans le handler lui-même, en appelant `signal()` à nouveau. Vérifiez aussi qu'avec cette option on ne peut compiler `tst_sigaction.exe`, car `sigaction()` n'est pas alors connu (il s'agit d'une primitive POSIX non ANSI C).

3 Transmission entre fils

Ecrire un programme `tst_children.exe` qui lance deux processus fils, envoie le signal `SIGUSR1` à son fils cadet, et attend la terminaison de ses deux fils. A la réception de ce signal, le fils cadet envoie le signal `SIGUSR2` au fils aîné et se termine, en émettant un message. Sur réception de `SIGUSR2`, le fils aîné se termine également, avec un message. Pourquoi les rôles des fils aîné et cadet sont difficilement interchangeables ?

4 Transmission de l'état des signaux par `execXX()`

Ecrire un programme qui montre que la récupération des signaux est (partiellement) transmise lors de l'utilisation d'une primitive de la famille `execXX()`. Pourquoi cette transmission ne peut être que partielle ?

Indication Vous aurez besoin ici de deux programmes, dont l'un sera exécuté grâce à `execXX()`.

² Cela dépend de la version de votre noyau LINUX et de celle de gcc