Jean-Paul Rigault, Erick Gallesio, Stéphane Lavirotte, Hélène Collavizza
Polytech'Nice – Sophia
SI3
2019-2020

TD n°7
Signaux

1 Introduction

La gestion des signaux est une partie très délicate de la programmation-système avec Posix. Nous touchons ici à la programmation concurrente et aux problèmes de synchronisation entre activités parallèles.

> **Très important** : Il y a 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`. Nous allons utiliser ici le mécanisme et la sémantique de Posix. Dans tous ces exercices, sauf mention explicite du contraire, vous devez donc utiliser `sigaction()` et non pas `signal()`, et vous devez compiler avec `gcc` sans l'option `-ansi`.

Vous trouverez à l'adresse suivante des ressources pour débuter rapidement le TD :

`https://lms.univ-cotedazur.fr/mod/resource/view.php?id=92455`

2 Manipulations élémentaires de signaux

Avant de programmer la gestion des signaux, vous allez pratiquer les signaux depuis un terminal pour bien voir et comprendre leur utilité et leur fonctionnement.

Exercice n°1:

Voici quelques manipulations à effectuer dans un terminal. Vous répondrez aux questions dans votre fichier de compte-rendu.

1. Lancez la commande xeyes. Que constatez-vous sur l'utilisation du terminal ?
2. Faites `Ctrl-Z` dans le terminal. Que constatez-vous au niveau du terminal et de l'application `xeyes` ?
3. Lancez la commande `bg`. Que se passe-t-il au niveau du terminal et de l'application `xeyes` ?
4. Lancez la commande `ps` et notez le numéro du processus correspondant à la commande xeyes.
5. Dans un autre terminal (ceci nous permet de vérifier que l'on peut envoyer un signal sans qu'il y ait de lien de filiation), lancer la commande `kill -SIGSTOP` *`pid`* (où *`pid`* est le numéro du processus correspondant à `xeyes` que vous avez noté). Que constatez-vous ? A quelle autre action faite précédemment pouvez-vous relier ce signal `SIGSTOP`.
6. A l'aide de la commande `kill -l` (« l » pour liste) trouver le numéro de signal correspondant à CONTinuer le processus `xeyes`. Envoyez ce signal au processus avec son numéro au lieu du nom du signal.
7. Enfin pour terminer la commande `xeyes`, au lieu de spécifier son numéro de *pid*, on préférrait utiliser son nom. Quelle est la commande à lancer pour utiliser le nom plutôt que le numéro de *pid* ?

3 Mise en œuvre élémentaire de signaux

Maintenant que vous avez pratiqué les signaux, nous allons voir comment faire un programme qui réagisse à l'envoi d'un signal.

3.1 Capture des signaux

Exercice n°2:

Écrire un programme, `tst_signal.exe`, qui capture, 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
- par ailleurs, 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 (true) {}` pendant laquelle vous aurez tout le loisir d'envoyer des `SIGINT` et des `SIGSEGV` (voir la note ci-après).

> **Remarque** :
> Si vous processus a été lancé en avant-plan (*foreground*, donc commande lancée sans le &) vous pouvez 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 votre processus a été lancé en arrière-plan (*background*, donc commande lancée avec le &), il ignore les signaux produits au terminal (vérifiez-le). Mais on peut toujours utiliser la seconde méthode depuis une autre fenêtre Shell (`kill -SEGV` *`pid`* ou `kill -INT` *`pid`*).

Exercice n°3:

Réalisez le même programme, mais cette fois-ci en utilisant la primitive Posix `sigaction()` au lieu de `signal()`. Nous nommerons `tst_sigaction.exe` ce second programme.

3.2 Signaux reçus pendant `sleep()`

Exercice n°4:

Dans les programmes précédents, remplacez 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 avec soin (elle est très courte !) la page de manuel de `sleep()` (`man 3 sleep`) pour comprendre le phénomène observé. Vous noterez que la fonction `sleep` renvoie `0` si elle est arrivée à son terme, sinon, le nombre de secondes restantes à dormir quand elle a été interrompue par un signal.

3.3 Compilation avec l'option `-ansi` de gcc

Comme nous l'avons rappelé en préambule, il est nécessaire de spécifier au compilateur si l'on souhaite générer son code en respectant le norme Ansi, à l'aide de l'option `-ansi`. Le comportement des signaux est très différent suivant la norme Ansi ou Posix. C'est ce que nous allons constater avec l'exercice suivant.

Exercice n°5:

Juste pour avoir une concrétisation de l'effet de cette option, compilez `tst_signal.exe` avec l'option `-ansi`. Exécutez cette nouvelle version compilée avec l'option `-ansi`. Vous constaterez, peut-être[2], 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 alors pas connu (il s'agit d'une primitive Posix, pas Ansi C).

3.4 Ignorer ? Bloquer ? Capturer sans rien faire ?

Posix propose plusieurs manières de ne pas réagir sur un signal :

[1] En effet, `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 « déréférençage » du pointeur nul, par exemple).
[2] Cela dépend en fait de la version de votre noyau Linux et de celle de `gcc`.

- ignorer le signal (`SIG_IGN`),
- bloquer le signal (grâce à `sigprocmask()`),
- capturer le signal sans *handler*.

Cependant, ces manières de faire sont loin d'être équivalentes.

Exercice n°6:

Pour le montrer, nous vous proposons le programme `tst_ignore.exe` qui réalise successivement les actions suivantes :

- ignorer `SIGINT`, puis attendre 5 secondes (grâce à `sleep()`)
- capturer `SIGINT` avec un *handler* qui se contente d'afficher qu'il a reçu le signal, bloquer le signal `SIGINT`, attendre 5 secondes, puis débloquer `SIGINT`
- capturer `SIGINT` avec un *handler* vide, puis attendre 5 secondes

Pendant chacune des périodes d'attente de 5 secondes, vous enverrez des `SIGINT` à votre processus (`^C` au terminal). Concluez.

4 Communication interprocessus

Exercice n°7:

Écrire 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 ainé et se termine, en émettant un message. Sur réception de `SIGUSR2`, le fils ainé se termine également, avec un message. Pour envoyer le signal `kill`, vous utiliserez la fonction Posix `kill`, bien entendu (vous ne ferez pas l'`exec` de la commande Unix `kill` qui elle-même a été implémentée grâce à la commande Posix `kill`).

Question 1: Pourquoi les rôles des fils ainé et cadet sont difficilement interchangeables ?

5 Signaux Win32 = Evénements

Faute de temps, nous n'allons pas pouvoir explorer l'équivalent Win32 pour les signaux. Sachez toutefois que si la philosophie est différente, un tel mécanisme de communication entre les processus existe aussi. Vous pouvez consulter dans le cours la section sur les Event pour en savoir plus sur les communications interprocessus sous Win32.

https://lms.univ-cotedazur.fr/mod/resource/view.php?id=83526

TD n°7
Signaux

Pour aller plus loin

6 Exercices complémentaires en Posix

Voici quelques exercices supplémentaires pour pratiquer les signaux et beaucoup d'autres choses que nous avons vues jusqu'à présent sous Posix

Exercice n°8:

Écrire un programme qui montre que la récupération des signaux est (partiellement) transmise lors de l'utilisation d'une primitive de la famille `execXX()`. (Vous aurez besoin ici de deux programmes en fait, dont un sera exécuté grâce à `execXX()`).

Question 2:

Pourquoi cette transmission ne peut être que partielle ?

Exercice n°9:

Ecrire un programme qui simule (de façon TRES simplifiée) la boucle centrale d'un Shell. La boucle de votre programme doit afficher un prompt, lire une commande sur l'entrée standard et exécuter la commande lue. Vous pouvez utiliser votre fonction system. Récupérez le signal `SIGINT` afin que votre programme se comporte convenablement sur la réception d'un `^C`. Vous utiliserez les primitives `sigsetjmp` et `siglongjmp` vues en cours.

Exercice n°10:

Réécrire la commande Unix `nohup`. Vous consulterez la page de manuel pour savoir ce que vous devez programmer.