La guerre des processus

Université Paris 7 — L3MI — Systèmes et Réseaux

Ce TP n'est pas noté

Signaux

`kill(pid,numsignal)` envoie un signal de numéro donné à un processus de numéro donné. La valeur de retour dit si l'envoi a réussi (0) ou non (-1). Voir `man 2 kill` pour plus de détails.

Il existe aussi une commande `kill` du shell avec la même syntaxe. Par exemple, taper `kill -9 -1` termine rapidement (mais pas proprement...) votre session.

Un *gestionnaire de signaux* (en anglais *signal handler*) est une procédure appellée quand un signal est reçu. Elle *doit* être de type `void` et prendre un seul paramètre, de type `int` : le numéro du signal reçu. Cette procédure n'est pas appellée dans votre programme[1] mais par le système d'exploitation, dès qu'un signal est reçu par votre processus. Elle constitue la réaction du processus face au signal.

Pour installer un gestionnaire de signal, on utilise la primitive `signal()`. Elle prend deux arguments

1. Le numero du signal a recevoir (le gestionnaire ne réagira qu'à ce signal-là !)
2. Un pointeur sur le gestionnaire de signal (pour ceux qui ont peur des pointeurs de fonction en C, c'est juste le nom de la méthode)

Vous pouvez utiliser le même gestionnaire pour plusieurs signaux. On peut aussi utiliser les macros prédéfinies dans `<signal.h>` à la place du pointeur sur le gestionnaire :

- `SIG_IGN` ignore le signal
- `SIG_DFL` est le comportement par défaut (dépend du numéro du signal).

La liste des signaux standards est définie dans `/usr/include/bits/signum.h` : regarder ce fichier vous donne une idée des différents évenements qui *peuvent* donner lieu à l'émission d'un signal. Voir aussi `kill -l` en ligne de commande, ou `man 7 signal` qui donne plein d'informations supplémentaires. Le manuel recommande d'utiliser `sigaction()` à la place de `signal()`, ce qui est plus portable et que vous êtes autorisés à faire. La syntaxe est toutefois un peu moins confortable...

Enfin, il est à noter que les signaux sont réçus même par les processus endormis.

1 Exercice 1 - signaux en rafale

Il faut programmer deux applications différentes : un émetteur et un récepteur.

- Le récepteur affiche son pid. Ensuite, il se contente d'afficher le numéro du signal reçu, dès qu'il en reçoit un (indéfiniment)
- L'émetteur prend en paramètre le pid du processus cible (normalement, le récepteur que vous avez écrit) puis lui envoie en boucle des signaux de numéro tiré aléatoirement

Questions à se poser :

1. Que se passe-t-il si on rajoute au gestionnaire de signaux du récepteur du code lent (un `sleep` ou une longue boucle inutile) ?
2. Pourquoi le récepteur finit-il par mourir, même si on a installé un gestionnaire pour tous les signaux ?
3. Comment l'émetteur réagit-il à la mort du récepteur ?
4. Comment faire pour que l'émetteur cesse sont activité une fois le récepteur mort ? (le code que vous rendez doit si possible inclure cette modification)

1. en fait, elle a le droit de l'être...

2 Exercice 2 - La guerre des processus

Dans cet exercice, N processus vont se battre jusqu'à ce qu'un seul survive ! Les participants ont tous un même père (qui, lui, ne participe pas). Problème, il faut connaître les PID de tous les participants. On opère donc en trois phases :

2.1 Phase préliminaire

Le programme est lancé avec comme paramètre le nombre N de processus. Le **père** commence par créer N tubes avec `pipe()`. Puis il crée N fils (avec `fork()`). Enfin, il envoie à *chacun* de ses fils le pid de *tous* les fils.

Un **fils**, quand il est créé, commence par lire les N pids dans son tube. Notez bien qu'il connaît N (car le père le connaît) et aussi le numéro du tube avec qui le père communiquera avec lui. En effet le père lance les fils dans une boucle "`for (i...`" donc le fils connaît la valeur de `i` à laquelle il a été créé, donc son numéro entre 0 et N. Il peut ainsi lire dans le ième tube le message de son père. Ce message consiste en un tableau de N ints. Il installe ensuite un gestionnaire de signal pour parer (presque) tous les signaux, comme dans l'exercice 1. Ensuite, il part en guerre.

2.2 Phase de combat

En phase de combat, chaque processus "tire dans tous les sens". C'est-à-dire qu'un processsus choisit un adversaire[2] au hasard parmi les autres[3] puis choisit un numéro de signal, tiré aléatoirement dans {1..64}. Le processus envoie le "coup" à sa victime, tout en affichant un message à la console comme "`le processus <PID> attaque <PID> avec <signal>`". Ensuite il recommence, inlassablement. Il est bon de tester la valeur de retour `kill()` à la recherche d'éventuels `EINVAL` ou `ESRCH` et réagir en conséquence.

Un processus recevant un "coup" (un signal) affiche un message du style "`le processus <PID> pare le coup numéro <signal>`".

Pour rendre les affichages plus lisibles vous pouvez ralentir un peu les combattants par `sleep()` ou `usleep()` (ce dernier prend un intervalle de temps en millisecondes) après un `kill()`.

2.3 Nettoyage final

Le père attend la mort des processus-fils par des `wait()`. Quand il n'en reste plus qu'un, il proclame par un message sur la console le nom de ce vainqueur puis... il le tue, avant de se terminer lui-même, ce qui met un terme à cette boucherie.

2. les processus ne sont pas freudiens et ne tuent donc pas le père !
3. un processus est darwinien : il évitera de s'attaquer lui-même...