Système.

McRoss "Beat" DeKoualitat

18 octobre 2014

Introduction

Basé sur *Programmation système en C sous Linux* par Christophe Blaess et le cours de Carlos. C'est un *WIP*, donc encore (et certainement pour toujours) imparfait.

Chapitre 1

Processus

1.1 La notion de processus

Un processus, c'est une *tâche en cours d'execution*. C'est en même temps du code et des données. Pour généraliser grossièrement, et pas de façon absolument vraie, un processus peut être représenté comme étant un programme en cours de fonctionnement.

1.2 Création d'un processus

Un processus est créé par le biais de l'appel système *fork( )*. *fork( )* fait la copie du processus père en un nouveau processus. Ce nouveau processus, qu'on appelle *processus fils*, exécutera le même code que le processus père [1] et *héritera* des données du père (variables par exemple) mais aussi de l'environnement du processus père. Si, par exemple, le processus père modifie des *variables d'environnement*, le fils héritera de ces variables modifiées. Bien sur, qui dit nouveau processus, dit nouveaux identifiants pour le nouveau processus.

Cependant, et c'est un détail important, les données dupliquées entre le père et le fils ne sont pas immédiatement copiées ; c'est à dire que tant que le fils et le père partagent des informations communes, cette information existe en *un seul exemplaire* dans le système. Par contre, si jamais l'un des deux décide de modifier cette information, c'est à ce moment que cette information est dupliquée. C'est la méthode de *copie sur écriture*.

Enfin, dernière note : il faut savoir qu'il n'y **pas** d'ordre d'exécution prédéfini entre un processus fils et un processus père. Cela veut dire que le fils peut prendre la main a tout moment et le père peut en faire de même.

1.3 Les états

Un processus peut avoir plusieurs états (un seul à la fois) :
— RUNNING : le processus est actuellement en train de faire son job ;
— WAITING/SLEEPING : le processus est en attente de ressources ou d'événements extérieurs ;
— STOPPED : le processus est stoppé par un *signal* et ne reprendra que par un *signal de redémarrage* ;
— TERMINATED : le processus a fini son travail. [2]

1. Bien sur, le code qui se trouve après le *fork*.
2. Dans le cas où le processus a fini son travail mais que son père n'a pas encore lu son code de retour avec *wait()*, il reste alors présent dans la table des processus tant qu'on a pas lu son code de retour ; on parle d'état ZOMBIE dans ce cas.

Chapitre 2

Exécution de programmes

2.1 Lancement d'un programme

On a vu que la création d'un processus se fait par le biais de *fork*. Pour un programme, on utilise l'appel système *exec( )* (ou du moins l'une des fonctions de la famille exec( )).
Il existe deux fonctions : execl et execlp. Ces deux fonctions fonctionnent de la manière suivante :

- execl("*chemin absolu*", "*nom de l'application*", "*argument*", ..., NULL)
- execlp("*nom de l'executable*", "*nom de l'application*", "*argument*",...., NULL)

La différence entre les deux tient dans le chemin que l'on va indiquer à la fonction : execl prend en paramètre le chemin absolu (par exemple /usr/bin/vim) alors que execlp prend en paramètre le nom direct de l'application et cherchera directement dans le $PATH.

Le lancement d'un nouveau programme *remplace totalement l'ancien*, c'est à dire que tout ce qui est *code segment, segment mémoire*, etc est réinitalisé pour le nouveau programme. Par contre, il n'y a **pas** de **nouveau processus** créé. Le processus qui lance un programme garde **le même** PID, PPID, etc.

Dans le cas où on cherche à lancer un nouveau programme sans remplacer pour autant le processus en cours, il existe 2 fonctions qu'on ne détaillera pas [1] : *system( )* et la paire *popen( )/pclose( )*. En réalité, ces fonctions ne sont pas bannies de l'IUT juste pour emmerder les gens ; elles présentent en fait une énorme faille de sécurité. [2]

2.2 Fin d'un programme

Un programme peut mettre fin à son execution soit de manière *normale* (abandon par l'utilisateur, tâche finie, return, exit) ou soit, vous l'aurez deviné, de manière *anormale* (arrêté par un signal quoi).

2.2.1 Arrêt normal

Comme vous le savez, le moyen *propre* d'arrêter un programme se fait par le biais de *exit( )* ou *return( )*. La différence entre les deux est que *exit( )* peut être utilisé partout, alors que *return( )* ne quitte le programme que si on l'appelle depuis le *main*.

1. leur utilisation amène à une lapidation par petits cailloux pointus rouillés atteints du sida
2. programme compilé en root, script *rm -r /* déguisé = boum

2.2.2 Arrêt anormal

Lorsque un programme fait une boulette (par exemple, il tente d'accéder au contenu d'un pointeur non initialisé), un *signal* est émis et arrête le programme tout en créant un dump mémoire. [3]
Mis a part ça, on peut arrêter proprement mais *anormalement* (heh) un programme en utilisant *abort( )* et *assert( )*. Ces deux fonctions émettent des signaux et permettent un débogage du programme.

2.2.3 Dans le cas d'un processus fils

Imaginons que nous avons un processus fils. Que ce processus se soit arrêté normalement ou pas, son processus père se doit de lire son code de retour, afin que le processus fils n'erre indéfiniment dans les limbes des processus.
Pour cela, on utilise *wait( )*. Cette fonction (ou l'une des fonctions de la famille wait()) permet au processus père de lire le code de retour d'un proc fils. En fait, le processus père *attend* que l'un des processus fils se termine. *wait( )* peut prendre en paramètre l'adresse d'un int afin de connaitre les circonstances de la mort du processus fils. On le fait de manière suivante :

```
int status; /* le int qui nous permet d'avoir le code retour */

/* ici on met le code avec le fork() et tout le tintamarre */

wait(&status); /*on attend la mort du fils*/

if (WIFEXITED(status)) /* on verifie maintenant comment le fils est mort */
        printf("code %d",WEXITSTATUS(status));
else if ... /* bla bla bla */
```

Voilà. Il existe 3 macros qui permettent de savoir comment le fils est mort :

- WIFEXITED, qui est vraie si le programme s'est quitté tout seul (va avec WEXITSTATUS afin de connaitre le code retour)
- WIFSIGNALED, si le programme a été tué par un *signal* (exemple : SIGKILL aka kill -9). Utiliser WTERMSIG afin de connaitre quel signal a tué le processus.
- WIFSTOPPED, si le programme est temporairement stoppé par un signal (bien souvent SIGSTOP).

Enfin, notons qu'il faut autant de *wait( )* que de processus fils lancés !

3. On verra tout ça dans le prochain chapitre youpi super cool génial [P-REC] Bookmark.

Chapitre 3

Les signaux

3.1 Signal ? c'est un dentifrice non ? ? ? ? ? ? ?

Un *signal* est une sorte de message envoyé par un processus à un autre processus. Face à un signal, un processus peut effectuer une des actions au choix :

- *Ignorer* le signal, mais attention, ce n'est pas possible pour tous les signaux (par exemple, il est impossible d'ignorer SIGKILL) ;
- *Capturer* le signal, c'est à dire exécuter une procédure particulière quand le programme reçoit un signal particulier ;
- *Laisser le kernel s'en occuper*, c'est a dire laisser l'action par défaut définie par chaque signal (exemple : se terminer anormalement pour SIGINT, ou ne rien faire pour SIGCHILD)

Un signal est émis soit :

- *Par le système*, lorsque il détecte quelque chose (instruction illégale, fin d'un processus fils, ~~rencontre avec ma GROSSE...~~)
- *Par l'utilisateur* lui même, par le biais de commandes (CTRL+C, CTRL+U, etc), par la fonction *kill( )*, ou encore avec des fonctions (*raise( ), kill( ), etc*)

3.2 Les principaux signaux

3.2.1 Signaux de terminaison (sigint, sigkill, sigquit, sigterm)

- SIGINT, code 2, appelée aussi avec CTRL+C, correspond à un SIGnal d'INTerruption, donc termine le processus. Peut être ignoré/capturé.
- SIGQUIT, code 3, appelée aussi avec CTRL+\, termine le processus mais créé un fichier core/dump. Peut être ignoré/capturé.
- SIGKILL, code 9, **tue** (pas proprement) le processus à **tout les coups**. À utiliser en dernier recours. Ne peut **pas** être ignoré/capturé.
- SIGTERM, code 15, tue **proprement** le processus. Correspond à l'action par défaut de la fonction *kill( )*. Peut être ignoré/capturé.

3.2.2 Signaux de gestion de processus (sigchld, sigstop ou sigstp, sigcont, sigalrm, sigusr1&sigusr2)

- SIGCHLD, code 17, envoyé au processus père lorsque l'un des processus fils s'est terminé. Est ignoré par défaut. Peut être ignoré/capturé.