LinuxFocus article number 272
http://linuxfocus.org

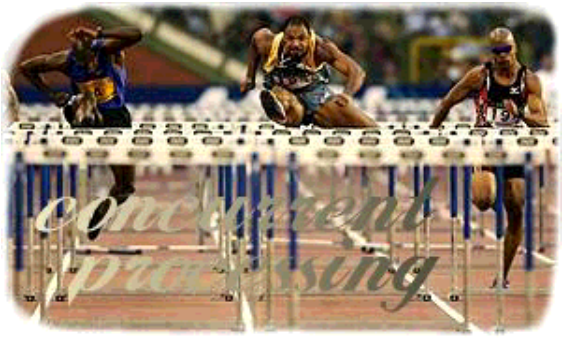
Programmation en temps partagé - Principes et notion de processus

par Leonardo Giordani
<leo.giordani(at)libero.it>

L´auteur:

Etudiant à la Faculté d'Ingénieurs Polytechniciens en Télécommunications de Milan, il travaille comme administrateur réseau et s'intéresse à la programmation (surtout en langage assembleur et en C/C++). Depuis 1999 il ne travaille que sous Linux/Unix.

Traduit en Français par:
Paul Delannoy (homepage)



Résumé:

Cette série d'articles se propose d'initier le lecteur au concept de multitâche et à sa mise en pratique dans le système d'exploitation Linux. Nous partirons des concepts théoriques de base concernant le multitâche pour aboutir à l'écriture complète d'une application illustrant la communication entre processus, avec un protocole simple mais efficace. Pour la compréhension de l'article il faut :

- Une connaissance minimale du shell
- Une connaissance de base du langage C (syntaxe, boucles, librairies)

Toute référence à une page de manuel sera placée entre parenthèses après le nom de la commande. Toutes les fonctions de glibc sont expliquées dans les pages info gnu (info Libc, ou tapez info:/libc/Top dans konqueror).

Introduction

Un des moments les plus importants dans l'histoire des systèmes d'exploitation est l'apparition du concept de programmation multiple, technique d'entrelacement de l'exécution de plusieurs programmes afin d'utiliser au mieux et de manière constante les ressources du système. Pensez à une station de travail, sur laquelle il est possible d'exécuter simultanément un traitement de texte, un lecteur audio, une file d'attente d'impression, un navigateur, et plus encore. C'est un concept important dans les OS modernes. Comme nous allons le découvrir cette petite liste n'est qu'une infime partie de l'ensemble des

programmes qui fonctionnent couramment sur votre machine, même si c’est la plus évidemment visible.

Le concept de processus

Pour réaliser l’entrelacement des programmes, une complication significative du système d’exploitation est nécessaire; pour éviter les conflits potentiels entre programmes, un choix incontournable consiste à encapsuler chacun avec l’ensemble des informations nécessaires à son exécution.

Avant d’explorer ce qui se passe dans votre machine Linux, un peu de vocabulaire technique : Soit un **PROGRAMME** donné, à un instant donné, le **CODE** est l’ensemble des instructions qui le constituent, l’**ESPACE MEMOIRE** est la quantité de mémoire de la machine occupée par ses données, et l’**ETAT du PROCESSEUR** est la valeur des paramètres du microprocesseur, comme les "flags" (drapeaux) ou le "Program Counter" (l’adresse mémoire de la prochaine instruction à exécuter).

Nous définissons le terme **PROGRAMME S’EXECUTANT** comme un ensemble d’objets constitué de CODE, d’ESPACE MEMOIRE et d’ETAT du PROCESSEUR. Si, à un moment donné du fonctionnement de la machine, nous pouvons sauver cet ensemble et le remplacer par l’ensemble des informations d’un autre programme en cours d’exécution, ce dernier va continuer au point où il avait été arrêté : faire cela une fois avec l’un, puis une fois avec l’autre procure l’entrelacement que nous décrivions ci-dessus. Le terme **PROCESSUS** (ou **TACHE**) désigne un tel programme en cours d’exécution.

Regardons à nouveau la station de travail dont nous parlions dans l’introduction : à chaque instant il n’y a qu’une TACHE en exécution (elle n’a qu’un seul microprocesseur qui NE peut PAS faire deux choses différentes en même temps), et la machine exécute une part de son code; après une certaine durée nommée QUANTUM, le processus en cours est suspendu, ses informations sont sauvées et remplacées par celles d’un autre processus en attente, dont le code sera exécuté pendant un quantum de temps, et ainsi de suite. C’est ce que nous appelons le "fonctionnement multitâche".

Comme nous l’indiquions précédemment, l’introduction du multitâche pose une série de problèmes, dont beaucoup ne sont pas triviaux, comme celui de la gestion de la file d’attente des processus suspendus (SCHEDULING); pourtant l’architecture de chaque système d’exploitation doit être prise en compte : ce sera peut-être le sujet principal d’un futur article de la série, qui présentera probablement quelques "morceaux" de code du noyau Linux.

Processus sous Linux et Unix

Découvrons un peu les processus actifs sur notre machine. La commande qui nous donne ces informations est **ps(1)**, acronyme de "process status". Dans une fenêtre shell, tapez la commande ps et vous obtenez l’équivalent de

```
PID TTY          TIME CMD
 2241 ttyp4    00:00:00 bash
 2346 ttyp4    00:00:00 ps
```

Comme je l’ai déjà noté, cette liste n’est pas complète, mais examinons-la un instant : ps nous a donné la liste de tous les processus actifs sur notre terminal. Nous reconnaissons dans la dernière colonne le nom

sous lequel le processus a été lancé (comme "mozilla" pour le navigateur Mozilla et "gcc" pour le Compilateur C GNU). Evidemment "ps" apparaît dans cette liste puisqu'il était en cours d'exécution lors de l'affichage du résultat de la commande. L'autre processus présent est celui du Bourne Again Shell, le shell "bash" qui fonctionne sur mes terminaux.

Ignorons (pour le moment) les informations TIME et TTY et attardons-nous sur PID, le "Process IDentifier". Ce *pid* est un nombre positif entier (non nul) qui est associé à chaque processus lors de sa création; lorsqu'il sera terminé, son nombre pid pourra resservir, mais nous sommes assurés que pendant toute la durée de son exécution ce nombre restera le même. Tout cela implique que chacun de vous aura obtenu une réponse de la commande ps différente de l'exemple ci-dessus. Pour vérifier la véracité de mes propos, ouvrez un autre shell sans fermer le premier et tapez-y la commande ps : la liste affiche les mêmes processus mais avec des nombres pid différents, prouvant ainsi qu'il s'agit bien de deux processus distincts même si le programme est le même.

Nous pouvons aussi afficher la liste de tous les processus actifs sur notre ordinateur Linux : la page de manuel de la commande ps indique que l'option -e signifie "sélection de tous les processus". Donc tapons "ps -e" sur un terminal et ps va afficher une longue liste au format illustré ci-dessus. Pour pouvoir l'analyser tranquillement, redirigeons la sortie de la commande ps vers le fichier ps.log :

```
ps -e > ps.log
```

Nous pouvons alors lire ce fichier dans notre éditeur favori (ou simplemant avec la commande less); comme indiqué au début de l'article le nombre de processus est supérieur à ce que nous imaginions. Nous voyons tout de suite que cette liste comporte non seulement l'ensemble des processus que nous avons créé (par la ligne de commande ou depuis un environnement graphique) mais aussi un autre ensemble de processus, certains avec des noms étranges : le nombre et l'identité des processus de la liste dépendent de la configuration de votre système, avec certains points communs. Tout d'abord, quelle que soit votre configuration système, le processus dont le pid vaut 1 est toujours "init", le père de tous les processus; il a ce pid numéro 1 parce qu'il est toujours le premier processus lancé par le système d'exploitation. Une autre remarque évidente concerne la présence de plusieurs processus dont le nom se termine par la lettre "d" : on les appelle "daemons" et ils font partie des processus les plus importants du système. Nous étudierons en détail init et les daemons dans un prochain article.

Le multitâche dans la libc

Nous comprenons maintenant le concept de processus et sa grande importance pour notre système d'exploitation : nous allons poursuivre et commencer à écrire du code multitâche; après les questions soulevées par la simultanéité, nous allons rencontrer un nouveau problème : celui de la communication entre des processus concurrents et leur synchronisation; nous découvrirons deux solutions élégantes à ce problème, les messages et les sémaphores. Ces derniers seront abordés plus en profondeur dans un article à venir concernant les "threads". Après avoir étudié les messages il sera temps de commencer l'écriture de notre application basée sur tous ces concepts.

La bibliothèque C standard (libc, mise en oeuvre sous Linux par glibc) utilise les caractéristiques multitâches d'Unix System V; l'Unix System V (maintenant appelé SysV) est une variante commerciale d'Unix, constituant l'une des deux grandes familles d'Unix, l'autre étant l'Unix BSD.

Dans la libc, le type pid_t est défini comme un entier susceptible de contenir un pid. Nous l'utiliserons

pour caractériser un pid, uniquement pour la clarté : utiliser un entier revient au même.

Découvrons la fonction qui va nous donner la valeur de pid du processus contenant notre programme.

```
pid_t getpid (void)
```

(qui est définie par pid_t dans unistd.h et dans sys/types.h) puis écrivons un programme pour afficher le pid obtenu sur la sortie standard. Tapez le code suivant dans l'éditeur de votre choix :

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main()
{
  pid_t pid;

  pid = getpid();
  printf("The pid assigned to the process is %d\n", pid);

  return 0;
}
```

Enregistrez ce programme sous print_pid.c et compilez-le :

```
gcc -Wall -o print_pid print_pid.c
```

Ceci va construire un exécutable nommé print_pid. Rappelez-vous que si le répertoire courant n'est pas dans le chemin, vous devrez l'exécuter en tapant "./print_pid". L'exécution ne réserve pas de grosses surprises : il affiche un nombre positif et, s'il est exécuté plusieurs fois, ce nombre augmente de un à chaque fois; ce n'est pas systématique, puisqu'un autre processus peut être créé par un autre programme entre une exécution de print_pid et la suivante. Essayez, par exemple, de lancer ps entre deux exécutions de print_pid...

Il est temps d'apprendre comment créer un processus, mais je dois encore dire quelques mots sur ce qui se passe lors d'une telle action. Quand un programme (contenu dans le processus A) crée un autre processus (B), les deux sont identiques, c'est-à-dire ont le même code, occupent la mémoire avec les mêmes données (mais pas la même plage de mémoire) et ont le même état de processeur. Toutefois les deux peuvent s'exécuter de manière différente, par exemple selon les saisies de l'utilisateur ou des données aléatoires. A est le "processus père" alors que B est le "processus fils"; cela donne tout son sens à l'expression "père de tous les processus" désignant init. La fonction qui crée un nouveau processus est :

```
pid_t fork(void)
```

et son nom lui vient de sa capacité à séparer en deux l'exécution du processus. Le nombre renvoyé par cette fonction est un pid, qui mérite un peu d'attention. Nous avons dit que le processus en cours se partage en un père et un fils, qui vont s'exécuter en s'entrelaçant avec tous les processus en cours, pour effectuer différentes choses; mais juste après la duplication lequel va être exécuté, le père ou le fils ? Eh bien, la réponse est simplement : un des deux. Cette décision est prise par une partie du système appelée scheduler, selon un algorithme basé sur d'autres paramètres et qui ne tient pas compte du fait qu'un processus soit père ou fils.

Cela dit, il est important de savoir quel processus est en cours, puisque le code est le même. Les deux processus contiendront chacun le code du père et celui du fils, mais chacun n'exécutera que l'un des deux. Un regard sur l'algorithme suivant pourra éclairer cette situation :

```
- FORK
- IF YOU ARE THE SON EXECUTE (...)
- IF YOU ARE THE FATHER EXECUTE (...)
```

ce qui représente dans une espèce de meta-langage le code de notre programme. Voici la clé du mystère : la fonction fork renvoie '0' au processus fils et renvoie au père le nombre pid du fils. Il suffit donc de vérifier si le pid renvoyé est zéro pour savoir quel est le processus qui exécute ce code. Traduit en C nous obtenons :

```
int main()
{
  pid_t pid;

  pid = fork();
  if (pid == 0)
  {
    CODE OF THE SON PROCESS
  }
  CODE OF THE FATHER PROCESS
}
```

Ecrivons maintenant notre premier code multitâche : vous pouvez le sauver comme fichier fork_demo.c puis le compiler comme précédemment. Je mets des numéros de ligne pour la clarté. Le programme va se partager et le pére et le fils vont tous les deux écrire dans l'écran; le résultat final sera l'entrelacement de ces sorties (si tout se passe bien).

```
(01) #include <unistd.h>
(02) #include <sys/types.h>
(03) #include <stdio.h>

(04) int main()
(05) {
(05)   pid_t pid;
(06)   int i;

(07)   pid = fork();

(08)   if (pid == 0){
(09)     for (i = 0; i  < 8; i++){
(10)       printf("-SON-\n");
(11)     }
(12)     return(0);
(13)   }

(14)   for (i = 0; i < 8; i++){
(15)     printf("+FATHER+\n");
(16)   }

(17)   return(0);
(18) }
```

Les lignes de (01) à (03) contiennent les fichiers à inclure pour les bibliothèques requises (Entrées/Sorties standard, multitâche).

La procédure main (comme toujours en GNU), renvoie un entier, qui vaudra zéro si le programme se termine sans erreur, ou un code d'erreur si quelque chose ne va pas; supposons pour l'instant que tout ira bien (nous ajouterons la gestion d'erreurs lorsque les concepts seront bien clairs). Nous définissons le type de données contenant un pid (05) et un entier compteur de boucle (06). Ces deux types, nous l'avons dit, sont identiques, mais ne sont utilisés que pour apporter une clarté supplémentaire.
A la ligne (07) nous appelons la fonction fork qui renverra 0 au programme exécuté dans le processus fils et le pid du processus fils au père; le test est à la ligne (08). Ainsi le code des lignes (09) à (13) sera exécuté dans le processus fils, alors que le reste dans les lignes (14) à (16) sera exécuté dans le processus père.
Chacune de ces parties écrit simplement 8 fois sur la sortie standard le mot "-SON-" ou le mot "+FATHER+", suivant le processus qui l'exécute, puis s'arrête en retournant zéro. Ceci est très important, car sans ce dernier "return" le processus fils, après la fin de la boucle, exécuterait le code écrit pour le père (essayez, ça ne fera aucun mal à votre machine, mais ça ne donnera pas le résultat espéré). Une telle erreur est très difficile à détecter, puique l'exécution d'un programme multitâche (d'autant plus qu'il est complexe) donne des résultats différents à chaque exécution, ce qui rend le déboguage à partir des résultats tout simplement impossible.

En exécutant ce programme vous ne serez peut-être pas satisfait : je ne peux pas assurer que vous obtiendrez un mélange des deux chaînes, ceci en raison de la vitesse d'exécution d'une aussi courte boucle. Votre sortie sera probablement une succession de "+FATHER+" suivie par une suite de "-SON-" ou le contraire. Essayez d'exécuter le programme plusieurs fois : le résultat peut varier.

En insérant un délai aléatoire avant chaque instruction printf, nous pouvons obtenir une image plus visuelle du multitâche : nous obtenons cela avec les deux fonctions sleep et rand :

```
sleep(rand()%4)
```

avec ceci le programme dort (sleep) pour un nombre de secondes fixé au hasard de 0 à 3 (% renvoie le reste de la division entière). Maintenant le code ressemble à

```
(09)  for (i = 0; i < 8; i++){
(->)    sleep (rand()%4);
(10)    printf("-SON-\n");
(11)  }
```

et de même pour le code du père. Sauvez comme fork_demo2.c, compilez et exécutez. C'est un peu plus lent, et nous notons une différence dans l'ordre de sortie :

```
[leo@mobile ipc2]$ ./fork_demo2
-SON-
+FATHER+
+FATHER+
-SON-
-SON-
+FATHER+
+FATHER+
-SON-
-FIGLIO-
+FATHER+
+FATHER+
-SON-
-SON-
-SON-
```

```
+FATHER+
+FATHER+
[leo@mobile ipc2]$
```

Regardons en face le problème qui émerge maintenant : nous sommes capables de créer un certain nombre de processus fils selon un certain processus père, qui exécuteront des opérations différentes de celles effectuées par le processus père lui-même, dans un environnement multitâche; le père doit souvent communiquer avec ses fils, ou au moins se synchroniser avec eux, afin d'exécuter les opérations au bon moment. Le premier moyen d'obtenir une synchronisation entre processus est fourni par la fonction wait :

```
pid_t waitpid (pid_t PID, int *STATUS_PTR, int OPTIONS)
```

où PID désigne le process_id du processus dont on attend la fin, STATUS_PTR un pointeur vers un entier qui contient l'état du processus fils (NULL si on ne désire pas l'information) et OPTIONS un ensemble d'options que nous n'aborderons pas pour l'instant. Voici un exemple de programme dans lequel le père crée un processus fils et attend jusqu'à ce qu'il se termine :

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main()
{
  pid_t pid;
  int i;

  pid = fork();

  if (pid == 0){
    for (i = 0; i < 14; i++){
      sleep (rand()%4);
      printf("-SON-\n");
    }
    return 0;
  }

  sleep (rand()%4);
  printf("+FATHER+ Waiting for son's termination...\n");
  waitpid (pid, NULL, 0);
  printf("+FATHER+ ...ended\n");

  return 0;
}                                    }
```

La fonction sleep a été insérée dans le code du père pour différencier les exécutions. Sauvons ce code comme fork_demo3.c, compilons-le puis exécutons-le. Nous venons d'écrire notre première application multitâche synchronisée !

Dans le prochain article nous approfondirons nos connaissances sur la synchronisation et la communication entre processus; maintenant écrivez vos propres programmes et envoyez-les moi afin que je puisse montrer de bonnes solutions ou des erreurs graves. Envoyez-moi le fichier .c avec votre code commenté et un petit fichier texte décrivant le programme, avec votre nom et votre adresse de courrier électronique. Bon travail !

Lectures recommandées

- Silberschatz, Galvin, Gagne, **Operating System Concepts - Sixth Edition**, Wiley&Sons, 2001
- Tanenbaum, WoodHull, **Operating Systems: Design and Implementation - Second Edition**, Prentice Hall, 2000
- Stallings, **Operating Systems - Fourth Edition**, Prentice Hall, 2002
- Bovet, Cesati, **Understanding the Linux Kernel**, O'Reilly, 2000

Site Web maintenu par l´équipe d´édition LinuxFocus © Leonardo Giordani "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Translation information: it --> -- : Leonardo Giordani <leo.giordani(at)libero.it> it --> en: Leonardo Giordani <leo.giordani(at)libero.it> en --> fr: Paul Delannoy (homepage)

2005-01-14, generated by lfparser_pdf version 2.51