LinuxFocus article number 281
http://linuxfocus.org

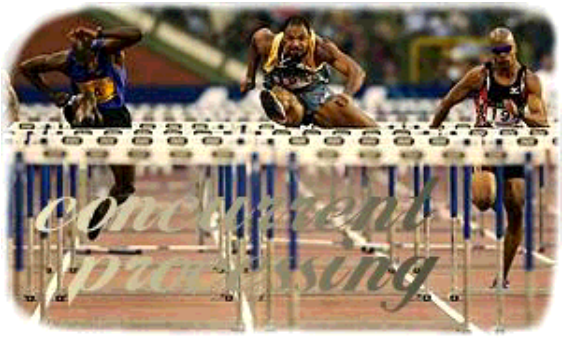
Programmation en temps partagé - communication entre processus

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

L´auteur:

Etudiant ingénieur en télécommunications à l'école Polytechnique de Milan, il travaille comme administrateur réseau et s'intéresse à la programmation (surtout en 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 oeuvre 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 comprendre l'article il faudrait avoir :

- Une connaissance minimale du shell
- Une connaissance basique du langage C (syntaxe, boucles, bibliothèques)

Vous devriez lire le premier article de la série, car il constitue une base indispensable à la compréhension de celui-ci : November 2002, article 272.

Introduction

Nous sommes à nouveau ici en face des fonctions multitâches du système Linux. Comme nous l'avons dit dans l'article précédent, dédoubler l'exécution d'un programme dans un processus différent ne nécessite que quelques lignes de code, puisque c'est le système d'exploitation qui prend en charge l'initialisation, la gestion et le partage du temps pour le processus créé.

Ce service offert par le système est fondamental, c'est la 'supervision de processus' ; chaque processus

est exécuté dans un environnement qui lui est dédié. Le développeur, qui perd ainsi le contrôle du déroulement de l'exécution, a donc un problème de synchronisation, qui se résume par la question : comment est-il possible de faire collaborer deux processus distincts ?

Ce problème est plus complexe qu'il ne semble : il ne suffit pas de synchroniser l'exécution des processus, mais aussi de partager des données, à la fois en mode lecture ou écriture.

Parlons un peu des problèmes classiques posés par l'accès concurrentiel à des données ; si deux processus lisent le même ensemble de données, il n'y a pas de problème majeur et l'exécution est dite CONSISTENTE. Mais si un des processus est autorisé à modifier ces données, le second pourra renvoyer des résultats différents selon l'instant où il lira les données, avant ou après que le premier ait écrit ces données modifiées. Par exemple : deux processus "A" et "B" partagent un entier "d". Le processus A incrémente d de 1, le processus B affiche sa valeur. Dans un meta-langage nous pourrions l'exprimer ainsi :

A { d->d+1 } & B { d->output }

en donnant à "&" le sens d'une exécution en 'concurrence'. Une première exécution pourra être :

(-) d = 5 (A) d = 6 (B) output = 6

mais si le processus B est exécuté le premier nous aurons :

(-) d = 5 (B) output = 5 (A) d = 6

Vous comprenez immédiatement l'importance cruciale de la gestion de telles situations: le risque d'INCONSISTENCE des données est grand et ne peut être accepté. Essayez d'imaginer que ces données concernent votre compte bancaire, et vous ne sous-estimerez plus jamais ce problème.

Nous avons abordé une première forme de synchronisation de processus dans l'article précédent, avec la fonction waitpid(2), qui force un processus à attendre l'achèvement d'un autre avant de poursuivre sa propre exécution. Ceci permet en fait de régler une partie des conflits provoqués par la lecture et l'écriture des données : dès que l'ensemble de données sur lequel un processus P1 doit travailler a été défini, un processus P2 qui doit travailler sur le même ensemble ou sur un sous-ensemble doit attendre l'achèvement de P1 avant de commencer lui-même à s'exécuter.

Si cette méthode présente une solution, il est clair qu'elle n'est pas la meilleure, car P2 va devoir attendre un temps indéterminé, qui peut être long, que P1 termine son exécution, et cela même s'il n'y a plus aucun travail sur des données partagées. Nous devons donc augmenter la granularité de notre contrôle, c'est-à-dire, régenter l'accès à une seule donnée ou un seul jeu de données. La solution à ce problème est un ensemble de primitives de la bibliothèque standard connue sous le nom de SysV IPC (System V InterProcess Communication).

Les clés de SysV

Avant d'aborder les arguments strictement liés à la théorie des processus concurrents et à leur mise en oeuvre, parlons un peu d'une structure type de SysV : les clés IPC. Une clé IPC est un nombre qui identifie de manière unique une structure de contrôle IPC (elles seront décrites plus loin), mais qui peut

aussi être utilisé pour générer des identifiants génériques, par exemple, pour organiser des structures non IPC. Une clé peut être créée par la fonction ftok(3) :

key_t ftok(const char *pathname, int proj_id);

qui a besoin d'un nom de fichier existant (pathname) et d'un entier. L'unicité de la clé n'est pas assurée, parce que les paramètres du fichier (les nombres désignant son i-node et son périphérique) peuvent mener à des combinaisons identiques. Une bonne solution consiste à créer une petite bibliothèque chargée de conserver une trace des clés affectées et d'éviter les doublons.

Sémaphores

Le concept de sémaphore utilisé dans la régulation du trafic automobile est applicable sans grandes modifications au contrôle d'accès aux données. Un sémaphore est une structure de données contenant une valeur plus grande ou égale à zéro et qui gère une file d'attente de processus attendant qu'advienne une condition particulière propre au sémaphore. Contrairement aux apparences, de simples sémaphores sont très puissants et par conséquent entraînent une complexité grandissante. Comme toujours, nous allons commencer en ignorant le contrôle d'erreurs : nous l'introduirons plus tard quand nos programmes deviendront plus complexes.

Les sémaphores sont utilisables dans le contrôle d'accès à une ressource : la valeur du sémaphore représente le nombre de processus qui peuvent accéder à la ressource; chaque accès d'un processus à la ressource décrémente la valeur, qui sera incrémentée à nouveau lorsque la ressource sera libérée. Dans le cas d'une ressource exclusive (i.e. un seul processus peut y accéder) la valeur initiale du sémaphore sera 1.

Une tâche différente peut être accomplie par un sémaphore, c'est le compteur de ressource : sa valeur est dans ce cas le nombre de ressources disponibles (par exemple le nombre de plages mémoire libres).

Examinons un cas réel dans lequel nous utiliserons des types de sémaphore : imaginons un tampon dans lequel plusieurs processus S1,..., Sn peuvent écrire mais dans lequel un seul processus L peut lire; de plus, les opérations ne peuvent être simultanées (i.e. un seul processus agit sur le tampon à un instant donné). Bien sûr les processus S peuvent toujours écrire, sauf quand le tampon est plein, alors que L ne peut lire que si le tampon n'est pas vide. Donc nous allons utiliser trois sémaphores : un premier pour l'accès à la ressource, un second et un troisième pour compter les éléments présents dans le tampon (nous verrons plus tard pourquoi deux sémaphores ne suffisent pas).

Comme le tampon est une ressource d'accès exclusif le premier sémaphore peut être binaire (sa valeur sera 0 ou 1), alors que les deux autres auront des valeurs liées à la taille du tampon.

Etudions comment mettre en oeuvre ces sémaphores en C grâce aux primitives de SysV. La fonction qui crée un sémaphore se nomme semget(2)

int semget(key_t key, int nsems, int semflg);

où *key* est une clé IPC, *nsems* le nombre de sémaphores que l'on désire créer et *semflg* est le contrôle d'accès représenté sur 12 bits, les 3 premiers relatifs à la création et les 9 autres aux accès en lecture et écriture de l'utilisateur ('user'), du groupe ('group') et des autres ('other') (notez la similitude avec le

système de fichiers Unix); pour plus de précisions lisez les pages de manuel d'ipc(5). Vous pouvez voir que SysV gère des ensembles de sémaphores plutôt que des sémaphores isolés, ce qui offre un code plus compact.

Créons notre premier sémaphore

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(void)
{
  key_t key;
  int semid;

  key = ftok("/etc/fstab", getpid());

  /* create a semaphore set with only 1 semaphore: */
  semid = semget(key, 1, 0666 | IPC_CREAT);

  return 0;
}
```

Avant d'aller plus loin apprenons à gérer et à supprimer des sémaphores; ceci se réalise avec la primitive semctl(2)

int semctl(int semid, int semnum, int cmd, ...)

qui effectue l'action désignée par *cmd* sur l'ensemble désigné par *semid* et (si l'action le requiert) sur le sémaphore particulier *semnum*. Nous introduirons des options si nécessaire, toutefois une liste complète peut être obtenue sur la page de manuel. Selon l'action *cmd* il peut être requis de spécifier un autre argument, dont le type est :

```
union semun {
 int val;                  /* value for SETVAL */
 struct semid_ds *buf;     /* buffer for IPC_STAT, IPC_SET */
 unsigned short *array;    /* array for GETALL, SETALL */
                           /* Linux specific part: */
 struct seminfo *__buf;    /* buffer for IPC_INFO */
};
```

Pour définir la valeur d'un sémaphore la directive SETVAL est obligatoire et la valeur doit être spécifiée dans la structure *union semun*; modifions le programme précédent en fixant la valeur du sémaphore à 1

```
[...]

  /* create a semaphore set with only 1 semaphore */
  semid = semget(key, 1, 0666 | IPC_CREAT);

  /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

[...]
```

Libérons maintenant le sémaphore en désaffectant les structures utilisées pour sa gestion; cette tâche est accomplie par la directive IPC_RMID de semctl. Cette directive efface le sémaphore et envoie un message à tous les processus en attente d'accès à la ressource. Une dernière modification du programme est :

```
[...]

   /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* deallocate semaphore */
  semctl(semid, 0, IPC_RMID);

[...]
```

Comme nous venons de le voir, créer et gérer des structures pour contrôler une exécution simultanée n'est pas trop difficile; mais si nous introduisons la gestion d'erreurs, les choses vont se compliquer un peu, mais seulement du point de vue de la complexité du code.

Le sémaphore peut maintenant être utilisé avec la fonction semop(2)

int semop(int semid, struct sembuf *sops, unsigned nsops);

où *semid* identifie l'ensemble, *sops* est un tableau contenant les opérations à effectuer et *nsops* le nombre de ces opérations. Chaque opération est représentée par une structure *sembuf* :

unsigned short sem_num; short sem_op; short sem_flg;

par exemple, le numéro du sémaphore dans l'ensemble (sem_num), l'opération (sem_op) et un drapeau définissant la politique d'attente; pour l'instant laissons ce *sem_flg* à 0. Les opérations que nous pouvons spécifier sont des entiers et suivent les règles suivantes :

1. sem_op < 0
   Si la valeur absolue du sémaphore est plus grande ou égale à celle de sem_op l'opération est effectuée et sem_op est ajoutée à la valeur du sémaphore (en fait elle est soustraite puisque sem_op est négatif). Si la valeur absolue de sem_op est inférieure à la valeur du sémaphore le processus se met en sommeil jusqu'à ce que la quantité de ressources requise soit disponible.
2. sem_op = 0
   Le processus est en sommeil jusqu'à ce que la valeur du sémaphore atteigne 0.
3. sem_op > 0
   La valeur de sem_op est ajoutée à celle du sémaphore, libérant les ressources utilisées jusqu'alors.

Le programme suivant va tenter de montrer comment utiliser les sémaphores pour mettre en oeuvre le précédent exemple de tampon : nous allons créer 5 processus nommés W (writers) et un processus nommé R (reader). Chaque processus W essaie de prendre le contrôle de la ressource (le tampon) en la verrouillant grâce à un sémaphore, et, si le tampon n'est pas plein, d'y ajouter un élément et de libérer la ressource. Le processus R essaie de verrouiller la ressource, d'y prendre un élément si le tampon n'est pas vide et de libérer la ressource.

Ces lectures et écritures sont virtuelles : en effet, comme nous l'avons vu dans l'article précédent, chaque processus dispose de son propre espace mémoire et ne peut accéder à celui d'un autre processus. Cela empêche la gestion correcte du tampon par 5 processus, puisque chacun voit sa propre copie du tampon. Pour l'instant nous devons renoncer au véritable échange de données jusqu'à ce que nous abordions la mémoire partagée.

Pourquoi faut-il 3 sémaphores ? Le premier (numéro 0) agit comme verrou d'accès au tampon et possède une valeur maximale de 1, alors que les deux autres gèrent les conditions de débordement par le haut (overflow) ou par le bas (underflow). Un seul sémaphore ne peut pas gérer les deux situations, puisque *semop* n'agit que dans un sens.

Soyons encore plus clairs : avec un seul sémaphore (nommé O), dont la valeur représente le nombre d'emplacements libres dans le tampon. Chaque fois qu'un processus S écrit quelque chose dans le tampon il décrémente de un la valeur du sémaphore, jusqu'à ce qu'elle atteigne zéro, i.e. que le tampon soit plein. Ce sémaphore ne peut gérer la condition d'underflow : le processus R peut augmenter sa valeur sans limite. Il nous faut donc un sémaphore spécial (nommé U), dont la valeur représente le nombre d'éléments dans le tampon. Chaque fois qu'un processus W écrit un élément dans le tampon il incrémente la valeur du sémaphore U et décrémente celle du sémaphore O. A l'inverse, le processus R décrémente la valeur du sémaphore U et incrémente celle du sémaphore O.

La condition d'overflow est ainsi identifiée par l'impossibilité de décrémenter le sémaphore O et la condition d'underflow par l'impossibilité de décrémenter le sémaphore U.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(int argc, char *argv[])
{
  /* IPC */
  pid_t pid;
  key_t key;
  int semid;
  union semun arg;
  struct sembuf lock_res = {0, -1, 0};
  struct sembuf rel_res = {0, 1, 0};
  struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
  struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};

  /* Other */
  int i;

  if(argc < 2){
    printf("Usage : bufdemo [dimension]\n");
    exit(0);
  }

  /* Semaphores */
  key = ftok("/etc/fstab", getpid());

  /* Create a semaphore set with 3 semaphore  */
  semid = semget(key, 3, 0666 | IPC_CREAT);
```

```
  /* Initialize semaphore #0 to 1 - Resource controller */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* Initialize semaphore #1 to buf_length - Overflow controller */
  /* Sem value is 'free space in buffer' */
  arg.val = atol(argv[1]);
  semctl(semid, 1, SETVAL, arg);

  /* Initialize semaphore #2 to buf_length - Underflow controller */
  /* Sem value is 'elements in buffer' */
  arg.val = 0;
  semctl(semid, 2, SETVAL, arg);

  /* Fork */
  for (i = 0; i < 5; i++){
    pid = fork();
    if (!pid){
      for (i = 0; i < 20; i++){
        sleep(rand()%6);
        /* Try to lock resource - sem #0 */
        if (semop(semid, &lock_res, 1) == -1){
          perror("semop:lock_res");
        }
        /* Lock a free space - sem #1 / Put an element - sem #2*/
        if (semop(semid, &push, 2) != -1){
          printf("---> Process:%d\n", getpid());
        }
        else{
          printf("---> Process:%d  BUFFER FULL\n", getpid());
        }
        /* Release resource */
        semop(semid, &rel_res, 1);
      }
      exit(0);
    }
  }

  for (i = 0;i < 100; i++){
    sleep(rand()%3);
    /* Try to lock resource - sem #0 */
    if (semop(semid, &lock_res, 1) == -1){
      perror("semop:lock_res");
    }
    /* Unlock a free space - sem #1 / Get an element - sem #2 */
    if (semop(semid, &pop, 2) != -1){
      printf("<--- Process:%d\n", getpid());
    }
    else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
    /* Release resource */
    semop(semid, &rel_res, 1);
  }

  /* Destroy semaphores */
  semctl(semid, 0, IPC_RMID);

  return 0;
}
```

Commentons les parties les plus intéressantes de ce code:

```
struct sembuf lock_res = {0, -1, 0};
struct sembuf rel_res = {0, 1, 0};
struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};
```

Ces 4 lignes désignent les actions possibles sur notre ensemble de sémaphores : les deux premières sont des actions simples, les autres sont doubles. La première action, *lock_res*, essaie de verrouiller la ressource : elle décrémente de 1 la valeur du premier sémaphore (numéro 0), si cette valeur n'est pas zéro, ou le processus reste en attente (none) si la ressource est déjà occupée. L'action *rel_res* est identique à l'action *lock_res* mais la ressource est libérée (valeur positive).

Les actions *push* et *pop* sont plus particulières. Ce sont des tableaux de 2 actions, la première sur le sémaphore numéro 1 et la seconde sur le sémaphore numéro 2; lorsque la valeur du premier est incrémentée la valeur du second est décrémentée et vice versa, mais le comportement n'est plus l'attente : IPC_NOWAIT force le processus à continuer son exécution si la ressource est occupée.

```
/* Initialize semaphore #0 to 1 - Resource controller */
arg.val = 1;
semctl(semid, 0, SETVAL, arg);

/* Initialize semaphore #1 to buf_length - Overflow controller */
/* Sem value is 'free space in buffer' */
arg.val = atol(argv[1]);
semctl(semid, 1, SETVAL, arg);

/* Initialize semaphore #2 to buf_length - Underflow controller */
/* Sem value is 'elements in buffer' */
arg.val = 0;
semctl(semid, 2, SETVAL, arg);
```

Nous initialisons ici la valeur des sémaphores : le premier à 1 puisqu'il contrôle l'accès à une ressource exclusive, le second à la longueur du tampon (indiquée sur la ligne de commande) et le troisième à 0, comme expliqué plus haut pour l'overflow et l'underflow.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
/* Lock a free space - sem #1 / Put an element - sem #2*/
if (semop(semid, &push, 2) != -1){
  printf("---> Process:%d\n", getpid());
}
else{
  printf("---> Process:%d  BUFFER FULL\n", getpid());
}
/* Release resource */
semop(semid, &rel_res, 1);
```

Le processus W essaie de verrouiller la ressource grâce à l'action *lock_res* ; ensuite il exécute un push et le signale sur la sortie standard : si l'opération ne peut s'effectuer il informe que le tampon est plein. Enfin il libère la ressource.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
```

```
/* Unlock a free space - sem #1 / Get an element - sem #2 */
if (semop(semid, &pop, 2) != -1){
  printf("<--- Process:%d\n", getpid());
}
else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
/* Release resource */
semop(semid, &rel_res, 1);
```

Le processus R agit plus ou moins a l'identique du processus W : il verrouille la ressource, effectue un pop et libére la ressource.

Nous parlerons dans le prochain article des files d'attente de messages, qui constituent une autre structure de l'IPC (InterProcess Communication) et de la synchronisation. Comme d'habitude si vous écrivez quelque chose de simple à partir de ce que vous avez appris ici, envoyez-le moi, avec votre nom et votre adresse mail, je serai heureux de le lire. 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
- The Linux Programmer's Guide: http://www.tldp.org/LDP/lpg/index.html
- Linux Kernel 2.4 Internals http://www.tldp.org/LDP/lki/lki-5.html

---

Site Web maintenu par l´équipe d´édition LinuxFocus	Translation information: it --> -- : Leonardo Giordani <leo.giordani(at)libero.it> it --> en: Leonardo Giordani <leo.giordani(at)libero.it> en --> fr: Paul Delannoy (homepage)

http://www.LinuxFocus.org

2005-01-14, generated by lfparser_pdf version 2.51