LinuxFocus article number 287
http://linuxfocus.org

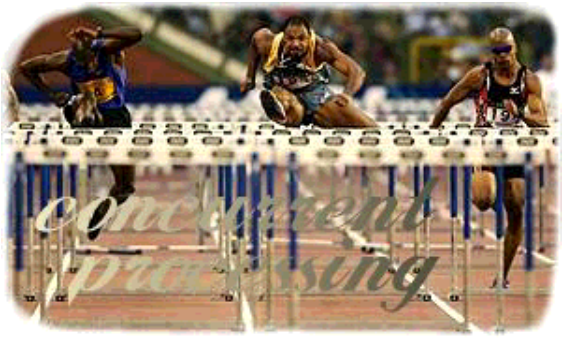
Programmation en temps partagé - Files d'attente de messages (1)

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

L´auteur:

Etudiant ingénieur en télécommunications à l'Ecole Polytechnique 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 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 standard du langage C (syntaxe, boucles, librairies)

Ce serait une bonne idée de lire les 2 précédents articles parus dans deux numéros de LinuxFocus (Novembre 2002 et Janvier 2003).

Introduction

Nous avons, dans les précédents articles, abordé le concept de programmation multitâche et étudié une première solution au problème de communication entre tâches : les sémaphores. Comme nous l'avons dit, les sémaphores nous permettent de gérer l'accès concurrentiel à des ressources partagées, afin de synchroniser deux ou plusieurs processus.

Synchroniser des processus nécessite de chronométrer leur travail, non pas dans un référentiel absolu

(qui donnerait un instant précis pour le démarrage des opérations d'un processus donné) mais dans un référentiel relatif, dans lequel nous pourrons indiquer quel processus agira le premier et lequel agira le second.

L'usage de sémaphores pour cet objectif se révèle complexe et limité : complexe parce que chaque processus doit gérer un sémaphore pour chaque processus distinct avec lequel il doit se synchroniser. Limité parce que cette solution ne permet pas d'échanger des paramètres entre les processus. Prenons l'exemple de la création d'un processus nouveau : cet événement doit être notifié à chacun des processus en cours d'exécution, mais la technique des sémaphores ne permet pas de le faire.

De plus, le contrôle par un sémaphore de l'accès concurrentiel à une ressource partagée peut conduire au blocage permanent d'un processus si l'un des autres processus concernés libère la ressource puis la verrouille à nouveau avant qu'un autre ait pu se l'approprier : comme nous l'avons vu, dans ce monde du multitâche, il n'est pas possible de prévoir quel processus sera actif, ni quand il le sera.

Ces quelques remarques rapides nous font comprendre que la technique des sémaphores n'apportera pas de solution à des problèmes de synchronisation de processus un peu complexes. Une solution élégante à ce problème est l'utilisation de files d'attente de messages : dans cet article nous allons étudier la théorie de ce système de communication entre processus et écrire un petit programme avec les primitives de SysV.

La Théorie des Files d'attente de Messages

Tout processus actif peut créer une ou plusieurs structures dénommées *files* : chacune de ces structures peut conserver un ou plusieurs messages de différents types, pouvant provenir de différentes sources et contenir des informations de nature quelconque; chaque processus peut envoyer un message à toute file disponible sous réserve de connaître son identificateur. Le processus accède à la file séquentiellement, lisant les messages en ordre chronologique (depuis le plus ancien, lu en premier, jusqu'au plus récent, le dernier arrivé), mais en opérant une sélection, en ne considérant que les messages d'un certain type : cela nous donne une sorte de contrôle de priorité sur les messages à lire.

L'utilisation des files est ainsi celle d'un système de courrier entre processus : chacun dispose d'une adresse et peut connaître celle des autres. Le processus peut lire ses messages dans un ordre préférentiel et il agira en fonction de ce qui lui sera ainsi notifié.

La synchronisation de deux processus peut ainsi être réalisée grâce à des échanges de messages entre eux : les ressources garderont leurs sémaphores pour que les processus connaissent leur état, mais le partage du temps entre processus peut être obtenu directement. Nous voyons immédiatement que cette technique va simplifier considérablement dès le départ ce qui était un problème complexe.

Avant de commencer l'écriture en langage C d'une mise en oeuvre de telles files d'attente, il faut parler d'un autre problème posé par la synchronisation : il nous faut un protocole de communication.

Créer un Protocole

Un protocole est un ensemble de règles définissant l'interaction entre des éléments d'un ensemble; dans l'article précédent nous avons mis en oeuvre l'un des plus simples possibles en créant un sémaphore et en programmant deux processus pour qu'ils agissent suivant l'état de ce sémaphore. L'utilisation de files d'attente de messages va nous permettre d'utiliser des protocoles plus complexes : il vous suffit de penser que chaque protocole réseau (TCP/IP, DNS, SMTP, ...) est construit sur une architecture d'échange de messages, même si la communication est établie entre machines et non interne à l'une d'elles. Cette comparaison est obligatoire : il n'y a pas de réelle différence entre la communication interne à une machine et celle entre deux machines. Comme nous le verrons dans un article à venir, l'extension des concepts dont nous discutons à un contexte distribué (sur plusieurs ordinateurs reliés entre eux) est très simple.

Voici un exemple simple de protocole basé sur l'échange de messages : deux processus A et B s'exécutent et traitent des données distinctes; lorsqu'ils ont terminé ces traitements, ils doivent regrouper les résultats. Un protocole pour régler une telle interaction pourrait être

PROCESSUS B:

- traitez vos données
- quand c'est fini envoyez un message à A
- lorsque A répond, commencez à lui envoyer vos résultats

PROCESSUS A:

- traitez vos données
- attendez un message de B
- répondez au message reçu
- recevez ses résultats et regroupez-les avec les vôtres

Le choix de celui des deux processus qui effectue le regroupement est dans ce cas arbitraire; ceci est souvent lié à la nature des processus concernés (client/serveur) mais discuter de cela ici desservirait l'article en cours.

Ce protocole peut simplement être étendu au cas de *n* processus : chacun des processus, sauf A, traite ses propres données puis adresse un message à A. Lorsque A répond chacun envoie ses résultats : la structure de chaque processus (hormis A) n'a pas à être modifiée.

Les Files d'attente de Messages dans System V

Il est temps d'aborder la mise en oeuvre de ces concepts dans le système d'exploitation Linux. Comme annoncé, nous disposons d'un ensemble de primitives permettant de gérer les structures relatives aux files d'attente de messages et qui fonctionnent comme celles permettant l'utilisation de sémaphores : je vais supposer maintenant que le lecteur connaît les concepts de base de la création d'un processus, l'usage d'appels au système et les 'clés IPC'.

La structure fondamentale du système de description d'un message se nomme `msgbuf;` elle est déclarée dans `linux/msg.h`

```
/* tampon de message pour les appels msgsnd et msgrcv */
struct msgbuf {
        long mtype;         /* type du message */
        char mtext[1];      /* texte du message */
};
```

Le champ mtype représente le type du message et c'est un entier strictement positif : la correspondance entre ces nombres et les types de message doit être établie par avance, comme une partie du protocole. Le second champ est le contenu du message mais n'a pas à être pris en compte lors de la déclaration. La structure `msgbuf` peut être redéfinie pour contenir des données plus complexes; il est par exemple possible d'écrire

```
struct message {
        long mtype;         /* type du message */
        long sender;        /* id émetteur */
        long receiver;      /* id destinataire */
        struct info data;   /* contenu du message */
        ...
};
```

Avant d'examiner les arguments plus strictement liés à la théorie de la concurrence, nous allons considérer la création d'un message de taille maximum, fixée à 4056 octets. Il est bien sûr possible, en recompilant le noyau, d'augmenter cette taille, mais cela rendra votre application non portable (et de plus cette limite a été choisie pour assurer de bonnes performances, aussi l'augmenter est certainement une mauvaise idée).

Pour créer une nouvelle file un processus doit appeler la fonction `msgget()`

```
int msgget(key_t key, int msgflg)
```

qui reçoit en argument une clé IPC et quelques drapeaux, qui pour l'instant seront fixés comme

```
IPC_CREAT | 0660
```

(crée la file si elle n'existe pas déjà et donne accès à cette file au propriétaire et au groupe), puis retourne comme résultat l'identificateur attribué à la file.

Comme dans les articles précédents nous allons supposer qu'aucune erreur ne survient, ce qui va simplifier notre code, même si dans un article à venir nous parlerons de programmes 'sécurisés IPC'.

Pour envoyer un message à une file dont on connaît l'identificateur nous utiliserons la primitive `msgsnd()`

```
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg)
```

où `msqid` est l'identificateur de la file, `msgp` est un pointeur vers le message à envoyer (dont le type est

ici identifié comme `struct msgbuf` mais qui est celui qui peut être redéfini), `msgsz` est la taille du message (sans celle du champ `mtype` qui est celle d'un *long*, usuellement fixée à 4 octets) et `msgflg` un drapeau relatif aux situations d'attente. La longueur du message est aisément accessible comme

```
length = sizeof(struct message) - sizeof(long);
```

tandis que la situation d'attente est celle d'une file pleine : si `msgflg` a la valeur IPC_NOWAIT l'émetteur n'attendra pas que de la place se libère et s'arrêtera en retournant un code d'erreur; nous reparlerons de ce cas lorsque nous aborderons la gestion d'erreurs.

Pour lire un message disponible dans une file nous utiliserons l'appel système `msgrcv()`

int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg)

où le pointeur `msgp` identifie un tampon dans lequel le message lu dans la file sera copié, et `mtype` identifie le sous-ensemble de messages qui nous intéresse.

La suppression d'une file peut être obtenue par la primitive `msgctl()` avec IPC_RMID comme valeur du drapeau.

```
msgctl(qid, IPC_RMID, 0)
```

Testons tout cela avec un programme simple qui crée une file, envoie un message et le lit; cela nous permet de tester le bon fonctionnement du système.

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

/* Redefinir la structure msgbuf */
typedef struct mymsgbuf
{
  long mtype;
  int int_num;
  float float_num;
  char ch;
} mess_t;

int main()
{
  int qid;
  key_t msgkey;

  mess_t sent;
  mess_t received;

  int length;

  /* Initialiser la base du générateur pseudo-aléatoire */
  srand (time (0));
```

```
  /* Longueur du message */
  length = sizeof(mess_t) - sizeof(long);

  msgkey = ftok(".",'m');

  /* Crée la file */
  qid = msgget(msgkey, IPC_CREAT | 0660);

  printf("QID = %d\n", qid);

  /* Construit un message */
  sent.mtype = 1;
  sent.int_num = rand();
  sent.float_num = (float)(rand())/3;
  sent.carattere = 'f';

  /* Envoie ce message */
  msgsnd(qid, &sent, length, 0);
  printf("MESSAGE ENVOYE...\n");

  /* Recoit le message */
  msgrcv(qid, &received, length, sent.mtype, 0);
  printf("MESSAGE RECU...\n");

  /* Controle l'égalité entre message envoyé et mesage reçu */
  printf("Interger number = %d (sent %d) -- ", received.int_num,
         sent.int_num);
  if(received.int_num == sent.int_num) printf(" OK\n");
  else printf("ERREUR\n");

  printf("Float numero = %f (sent %f) -- ", received.float_num,
         sent.float_num);
  if(received.float_num == sent.float_num) printf(" OK\n");
  else printf("ERREUR\n");

  printf("Char = %c (sent %c) -- ", received.ch, sent.ch);
  if(received.ch == sent.ch) printf(" OK\n");
  else printf("ERREUR\n");

  /* Detruit la file */
  msgctl(qid, IPC_RMID, 0);
}
```

Nous pouvons maintenant créer deux processus et les faire communiquer au travers d'une file d'attente de messages; rappelez-vous les concepts de dédoublement de processus : les valeurs de toutes les variables affectées par le processus père sont affectées à celles du processus fils (copie de mémoire). Cela signifie que la file doit être créée par le père avant qu'il se dédouble afin que le fils connaisse l'identificateur de la file pour pouvoir s'en servir.

Le code que j'ai écrit crée une file utilisée par le processus fils pour envoyer ses données au processus père : le fils génère des nombres aléatoires, les envoie au père, et les deux les affichent sur la sortie standard.

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

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

/* Redéfinit la structure message*/
typedef struct mymsgbuf
{
  long mtype;
  int num;
} mess_t;

int main()
{
  int qid;
  key_t msgkey;
  pid_t pid;

  mess_t buf;

  int length;
  int cont;

  length = sizeof(mess_t) - sizeof(long);

  msgkey = ftok(".",'m');

  qid = msgget(msgkey, IPC_CREAT | 0660);

  if(!(pid = fork())){
    printf("PERE - QID = %d\n", qid);

    srand (time (0));

    for(cont = 0; cont < 10; cont++){
      sleep (rand()%4);
      buf.mtype = 1;
      buf.num = rand()%100;
      msgsnd(qid, &buf, length, 0);
      printf("FILS - MESSAGE NUMERO %d: %d\n", cont+1, buf.num);
    }

    return 0;
  }

  printf("PERE - QID = %d\n", qid);

  for(cont = 0; cont < 10; cont++){
    sleep (rand()%4);
    msgrcv(qid, &buf, length, 1, 0);
    printf("PERE - MESSAGE NUMERO %d: %d\n", cont+1, buf.num);
  }

  msgctl(qid, IPC_RMID, 0);

  return 0;
}
```

Nous avons ainsi créé deux processus, qui peuvent collaborer de manière élémentaire par un échange de messages dans une file. Nous n'avons pas eu besoin d'un protocole (au sens formel du terme) car les opérations en cause sont très simples; nous parlerons plus avant des files d'attente de messages dans l'article suivant, ainsi que de la façon de gérer différents types de messages. Nous continuerons

à travailler sur le protocole de communication afin de commencer la construction de notre gros projet IPC (un téléphone avec simulateur).

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

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