LinuxFocus article number 296
http://linuxfocus.org

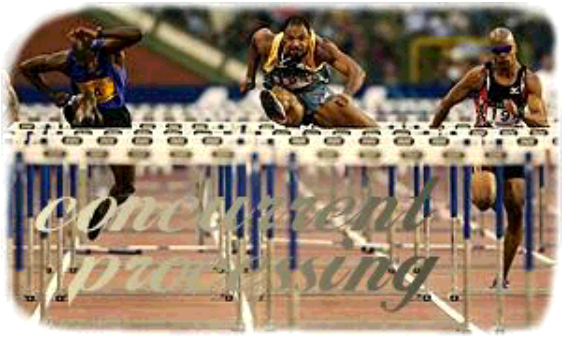
Programmation en temps partagé - Files d’attente de messages (2)

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 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)

Toute référence à des pages de manuel est placée entre parenthèses après le nom de la commande concernée. Toute fonction de la *glibc* est documentée par la commande "info Libc".

Introduction

Nous avons appris dans l’article précédent à synchroniser et à faire travailler ensemble deux processus (ou plus) en utilisant des files d’attente de messages. Nous allons maintenant un peu plus loin, en commençant à créer notre propre protocole d’échange de messages.

Nous avons précédemment défini un protocole comme un ensemble de règles qui permettent le dialogue entre deux personnes ou machines, même si elles sont différentes. L’usage de la langue anglaise par

exemple est un protocole, puisqu'il me permet de parler à mes lecteurs indiens (qui ont toujours été très intéressés par ce que j'écris). Pour parler de choses plus proches de Linux, si vous recompilez votre noyau (pas de panique, ce n'est pas si compliqué), vous remarquerez sans doute la section Networking (Réseau), qui permet de faire comprendre à votre noyau différents protocoles réseau, comme TCP/IP.

Afin de créer un protocole, nous allons commencer par définir le type d'application que nous envisageons. Notre exemple sera un simulateur de commutateur téléphonique. Un processus principal simulera le commutateur lui-même, et ses processus fils simuleront les actions des utilisateurs : chaque utilisateur devra pouvoir échanger des messages avec un autre au travers du commutateur.

Le protocole doit gérer trois situations différentes : la naissance d'un utilisateur (i.e. l'utilisateur existe et se connecte), le travail courant d'un utilisateur, et la disparition d'un utilisateur (il n'est plus connecté). Abordons les trois cas :

Lorsqu'un utilisateur se connecte au système, il crée sa propre file d'attente de messages (n'oublions pas : c'est un processus), dont les identifiants doivent être envoyés au commutateur afin que celui-ci sache comment atteindre cet utilisateur. Il a le temps d'initialiser des structures de données si nécessaire. Il reçoit du commutateur l'identifiant d'une file d'attente de messages, dans laquelle il pourra écrire les messages délivrés à d'autres utilisateurs par le commutateur lui-même.

L'utilisateur peut donc envoyer et recevoir des messages. Lorsqu'il émet un message vers un autre utilisateur, deux cas sont possibles : le destinataire est connecté ou non. Nous décidons que dans les deux cas, un accusé de réception sera envoyé à l'expéditeur, afin qu'il sache ce qu'il advient de son message. Ceci peut être accompli par le commutateur lui-même, sans que le destinataire n'ait à faire quoi que ce soit.

Lorsqu'un utilisateur se déconnecte, il doit en informer le commutateur, mais aucune autre action n'est nécessaire. Le métacode décrivant cette façon de faire est le suivant

```
/* Naissance */
create_queue
init
send_alive
send_queue_id
get_switch_queue_id

/* Travail courant */
while(!leaving){
 receive_all
 if(<send condition>){
  send_message
 }
 if(<leave condition>){
  leaving = 1
 }
}

/* Disparition */
send_dead
```

Maintenant il faut définir le comportement de notre commutateur téléphonique : lorsqu'un utilisateur se connecte il nous envoie l'identifiant de sa file d'attente de messages; nous devons conserver cet identifiant, afin de faire parvenir à cet utilisateur les messages qui lui sont destinés, et nous devons

répondre en lui fournissant l'identifiant d'une file d'attente dans laquelle il peut écrire les messages destinés à d'autres utilisateurs. Nous devons ensuite analyser les messages en attente et vérifier que les destinataires soient vivants : si le destinataire est connecté nous délivrons le message, sinon nous l'ignorons; dans les deux cas nous informons l'expéditeur. Lors de la disparition d'un utilisateur, nous supprimons simplement l'identifiant de sa file d'attente, il devient ainsi injoignable.

A nouveau, le métacode est le suivant :

```
while(1){
 /* Nouvel utilisateur */
 if (<birth of a user>){
  get_queue_id
  send switch_queue_id
 }

 /* Disparition utilisateur */
 if (<death of a user>){
  remove_user
 }

 /* Distribution des messages */
 check_message
 if (<user alive>){
  send_message
  ack_sender_ok
 }
 else{
  ack_sender_error
 }
}
```

Gestion des erreurs

Gérer les erreurs est une des choses les plus importantes et difficiles, dans la conduite d'un projet. De plus, un bon ensemble de routines de gestion d'erreurs, peut représenter la moitié des lignes de code nécessaires au projet. Nous n'allons pas expliquer ici comment développer un bon système d'analyse des erreurs, c'est un sujet trop conséquent, mais à partir de maintenant je vais toujours tester les conditions d'erreur. Une bonne introduction à ces questions d'analyse d'erreurs est la lecture du manuel de la bibliothèque glibc (www.gnu.org) mais, si vous êtes intéressés, j'écrirai un article sur le sujet.

Mise en oeuvre du protocole - Couche 1

Notre petit protocole aura deux couches : la première (la plus basse) est constituée des fonctions de gestion des files d'attente, de préparation et d'envoi des messages, alors que la plus haute met en oeuvre le protocole grâce à des fonctions ressemblant au métacode utilisé pour décrire le comportement du commutateur et des utilisateurs.

La toute première chose à faire est de définir la structure des messages, en utilisant le prototype msgbuf fourni par le noyau

```
typedef struct
```

```
{
  int service;
  int sender;
  int receiver;
  int data;
} messg_t;

typedef struct
{
  long mtype; /* Type du message */
  messg_t message;
} mymsgbuf_t;
```

Il y a ici quelque chose de général que nous pourrons étendre plus tard : les champs expéditeur et destinataire contiennent un identifiant d'utilisateur et le champ données contient des données quelconques, tandis que le champ service permet d'adresser une demande particulière au commutateur. Par exemple il pourrait y avoir deux services prévus : l'un pour la distribution immédiate d'un message, l'autre pour la distribution différée, le champ data devant alors contenir le délai en secondes. Ce n'est qu'un exemple, mais il permet de comprendre que le champ service offre de nombreuses possibilités.

Nous pouvons maintenant définir quelques fonctions pour traiter nos structures de données, particulièrement pour lire ou écrire les valeurs des champs des messages. Ce sont plus ou moins toujours les mêmes, aussi je ne vous en propose que deux, vous trouverez les autres dans les fichiers *.h

```
void set_sender(mymsgbuf_t * buf, int sender)
{
  buf->message.sender = sender;
}

int get_sender(mymsgbuf_t * buf)
{
  return(buf->message.sender);
}
```

Le but de telles fonctions n'est certes pas de compresser le code (elles ne contiennent qu'une seule ligne !) : elles ont pour but de conserver la définition du protocole proche d'un langage humain, et donc d'être plus simples à utiliser.

Ecrivons maintenant les fonctions destinées à générer des clés IPC, créer et supprimer des files d'attente de messages, envoyer et recevoir des messages : construire une clé IPC est simplement ceci :

```
key_t build_key(char c)
{
  key_t key;
  key = ftok(".", c);
  return(key);
}
```

La fonction qui crée une file d'attente

```
int create_queue(key_t key)
{
  int qid;

  if((qid = msgget(key, IPC_CREAT | 0660)) == -1){
    perror("msgget");
```

```
    exit(1);
  }

  return(qid);
}
```

comme vous le voyez la gestion d'erreurs est ici extrêmement simple. La fonction suivante supprime une file d'attente

```
int remove_queue(int qid)
{
  if(msgctl(qid, IPC_RMID, 0) == -1)
  {
    perror("msgctl");
    exit(1);
  }
  return(0);
}
```

Et enfin les fonctions de réception et d'envoi de messages : pour nous, émettre un message c'est l'écrire dans une file d'attente spécifique, celle qui nous a été indiquée par le commutateur.

```
int send_message(int qid, mymsgbuf_t *qbuf)
{
  int result, lenght;
  lenght = sizeof(mymsgbuf_t) - sizeof(long);
  if ((result = msgsnd(qid, qbuf, lenght, 0)) == -1)
   {
     perror("msgsnd");
    exit(1);
    }

  return(result);
}

int receive_message(int qid, long type, mymsgbuf_t *qbuf)
{
  int result, length;
  length = sizeof(mymsgbuf_t) - sizeof(long);

  if((result = msgrcv(qid, (struct msgbuf *)qbuf, length, type, IPC_NOWAIT)) == -1)
   {
    if(errno == ENOMSG){
      return(0);
    }
    else
    {
      perror("msgrcv");
      exit(1);
    }
  }

  return(result);
}
```

C'est tout. Vous trouverez des instructions complémentaires dans le fichier layer1.h : essayez de créer un programme (par ex. celui du dernier article) avec elles. Dans le prochain article nous aborderons la couche 2 du protocole et sa mise en oeuvre.

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
- Le guide du programmeur Linux : **http://www.tldp.org/LDP/lpg/index.html**
- Linux Kernel 2.4 Internals **http://www.tldp.org/LDP/lki/lki-5.html**
- Site Web du canal IRC #kernelnewbies **http://www.kernelnewbies.org/**
- La FAQ de la liste de diffusion "linux-kernel" **http://www.tux.org/lkml/**

Comme toujours vous pouvez m'adresser commentaires, corrections, questions à mon adresse électronique (leo.giordani(at)libero.it) ou par la page de discussion. S.V.P écrivez en anglais, allemand ou italien.

Site Web maintenu par l´équipe d´édition LinuxFocus	Translation information: 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