LinuxFocus article number 316
http://linuxfocus.org

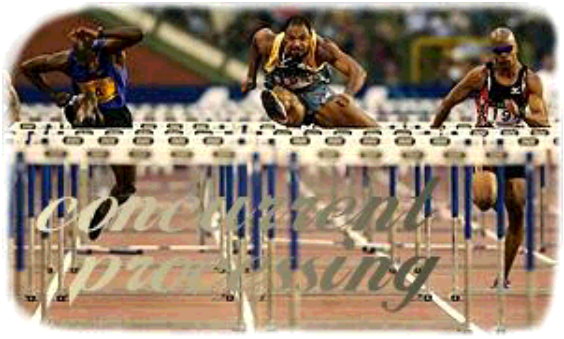
Programmation en temps partagé - Files d’attente de messages (3)

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

L´auteur:

Titulaire d’un diplôme d’ingénieur en télécommunications de l’école polytechnique de Milan, son centre d’intérêt majeur est la programmation (principalement l’assembleur et C/C++). Depuis 1999, il travaille exclusivement sous Linux/Unix.

Traduit en Français par:
Guillaume Baudot
<guillaume.baudot(at)caramail.com>



Résumé:

Voici venir le dernier article de cette série sur la programmation en temps partagé. Notre propos est d’implémenter la seconde et dernière couche de notre protocole : nous allons donc créer les fonctions qui régissent le comportement côté utilisateur et côté commutateur, en nous reposant, bien entendu, sur la première couche que nous avons vue précédemment.

Inutile donc de préciser qu’il est indispensable d’avoir préalablement pris connaissance des précédents articles avant d’aborder ce dernier. Pour mémoire :

- Programmation en temps partagé - Files d’attente de messages (2)
- Programmation en temps partagé - Files d’attente de messages (1)
- Programmation en temps partagé - Communication entre processus
- Programmation en temps partagé - Principes et notion de processsus

Mise en oeuvre du protocole - Couche 2 - Généralités

Le programme de démonstration ipcdemo simule un commutateur que les utilisateurs sollicitent pour s'envoyer des messages entre eux. Et pour agrémenter le tout, il met en pratique le concept de "service" : plus précisément, un message de service sert non pas à transférer des données, mais plutôt des informations de contrôle, il fait office de signal d'avertissement. Dans le cas qui nous préoccupe, les messages de service seront utilisés par les clients pour indiquer au commutateur qu'ils sont actifs, ou comment les contacter (grâce à l'identifiant de file d'attente IPC), ou encore le prévenir de leur arrêt. Deux autres services viennent compléter la panoplie, permettant au commutateur de spécifier à un utilisateur qu'il doit s'arrêter (terminaison) ou d'en mesurer le temps de réponse (chronométrage). Nous verrons tout cela plus en détail dans les sections qui suivent, commençons donc par quelques précisions concernant la couche 2 (layer2.*) du programme.

Il s'agit d'une couche d'abstraction dans laquelle sont utilisées les fonctions de la couche 1 (layer1.*) pour définir les fonctions, dites de haut niveau, de gestion des messages de service (initialisation, envoi, réception...). La complexité étant enfouie dans la couche infèrieure, vous pourrez constater que le code présent dans layer2.c est d'un abord particulièment aisé. Vous noterez en outre que le fichier layer2.h déclare un certain nombre d'alias définissant nos deux types de messages (message utilisateur et messages de service) ainsi que les différents services pris en compte (plus deux autres réservés à vos fins personnelles pour expérimentation - N.d.T. : au vu du code, je redoute cependant de grandes difficultés avec l'utilisation de "SERV_USERDEF1").

Ce programme a vocation d'exemple et n'est en aucun cas optimisé. Vous constaterez en particulier une profusion de variables globales : elles sont là pour vous permettre de passer outre les détails du code et ainsi mieux appréhender les fonctionnalités propres à IPC. Si toutefois certain point vous semble par trop abscons, n'hésitez pas à m'en faire part.

Implémentation du processus côté utilisateurs

Dans notre programme, les utilisateurs sont simulés par autant de processus fils dont le père, quant à lui, fait office de commutateur. De cette façon, toute variable initialisée avant de faire appel à "fork" est aussi bien connue des fils que du processsus père : c'est le cas en particulier pour l'identifiant de la file d'attente IPC du processus père, et s'agissant de la destination effective des messages de tous les utilisateurs, cela s'avère pour le moins pratique.

Pour pouvoir ensuite communiquer avec les autres, un utilisateur doit en tout premier lieu créer sa propre file d'attente de messages, puis indiquer au commutateur comment atteindre cette dernière. Deux messages de services sont successivement envoyés pour cela, SERV_BIRTH, puis SERV_QID, comme vous pouvez le voir ci-dessous.

```
/* Initialisation de la file d'attente de messages */
qid = init_queue(i);

/* message de type SERV_BIRTH : on spécifie qu'on est actif */
child_send_birth(i, sw);

/* message de type SERV_QID : on précise le moyen de nous joindre */
child_send_qid(i, qid, sw);
```

Vient ensuite la boucle principale qui se déroule en trois temps distincts :

- envoi éventuel d'un message vers un autre utilisateur,
- traitement des messages reçus (venant d'autres utilisateurs),
- traitement des requêtes de service (demandes spéciales du serveur)?

Comme la décision d'envoyer un message, de même que le choix du destinataire sont probabilistes, nous allons faire appel à un générateur de nombres aléatoires, en l'occurrence la fonction myrand() : pour un entier N (positif !) donné en paramètre, elle retourne un autre entier compris entre 0 et N (c.f. my_lib.*). Pour étayer notre propos, attardons nous un instant sur la première ligne du code ci-dessous : si l'on éxécutait ce test à cent reprises, le nombre de fois où la condition est vérifiée, devrait être en pratique proche de la valeur de send_prob...parfait.

```
if(myrand(100) < send_prob){
  dest = 0;

  /* Les messages à soi-même ou au commutateur sont prohibés */
  /* On évite aussi d'avoir 2 fois de suite le même destinataire */
  while((dest == 0) || (dest == i) || (dest == olddest)){
    dest = myrand(childs + 1);
  }
  olddest = dest;

  printf("%d -- U %d -- Message to user %d\n", (int) time(NULL), i, dest);
  child_send_msg(i, dest, 0, sw);
}
```

Les messages d'utilisateurs ne sont pas envoyés directement, mais transitent en fait par le commutateur. Ce qui les distingue des requètes de service, c'est qu'ils sont marqués avec le type TYPE_CONN (pour connexion).

```
/* A-t'on reçu un message d'un autre utilisateur ? */
/* Si oui, de qui et de quoi s'agit-il ? */
if(child_get_msg(TYPE_CONN, &in)){
  msg_sender = get_sender(&in);
  msg_data = get_data(&in);
  printf("%d -- U %d -- Message from user %d: %d\n",
         (int) time(NULL), i, msg_sender, msg_data);
}
```

Reste enfin à tester la possibilité d'une requète de service, autrement dit un message marqué du type TYPE_SERV (pour service), et au besoin y répondre. Deux cas sont à prendre en compte. Pour la terminaison, on commence par envoyer un accusé de reception, ainsi le serveur, nous sachant dorénavant inactif, n'aura plus à nous envoyer de messages. Ensuite de quoi, par politesse, nous lisons quand même les derniers messages s'il y'en a dans la file d'attente. Nous pouvons désormais sans remord supprimer la file avant de nous arrêter définitivement. Pour le chronomètrage, on se contente de renvoyer l'heure, laissant au commutateur le soin des calculs...

```
/* Le commutateur a-t'il requis un service particulier ? */
if(child_get_msg(TYPE_SERV, &in)){
  msg_service = get_service(&in);

  /* Si oui, lequel ? */
  switch(msg_service){
  case SERV_TERM:
    /* Dommage, c'est l'heure des adieux !... */
```

```
    /* Notre moyen de communication va être coupé, */
    /* il nous faut donc le signaler au commutateur. */
    child_send_death(i, getpid(), sw);

    /* S'il reste des messages dans la file, c'est le moment ou jamais */
    /* pour s'en aviser */
    while(child_get_msg(TYPE_CONN, &in)){
      msg_sender = get_sender(&in);
      msg_data = get_data(&in);
      printf("%d -- U %d -- Message from user %d: %d\n",
             (int) time(NULL), i, msg_sender, msg_data);
    }

    /* Ne reste plus qu'à supprimer notre file d'attente avant de disparaître */
    close_queue(qid);
    printf("%d -- U %d -- Termination\n", (int) time(NULL), i);
    exit(0);
    break;
  case SERV_TIME:
    /* Le serveur cherche a déterminer notre temps de réaction */
    /* On lui fait transmet donc l'heure à laquelle sa requête est traitée */
    /* pour qu'il puisse faire le calcul */
    child_send_time(i, sw);
    printf("%d -- U %d -- Timing\n", (int) time(NULL), i);
    break;
  }
}
```

Mise en oeuvre du commutateur

On peut distinguer deux phases dans le déroulement du processus père. La première, la phase d'initialisation couvre le début du programme, avant la création des processus fils. On génère l'identifiant de file d'attente IPC du commutateur à ce moment de sorte que l'information soit partagée entre tous les processus. Et l'on crée un tableau pour stocker les mêmes identifiants que les futurs utilisateurs ne tarderons pas à transmettre. L'identifiant du commutateur est une valeur clé pour notre tableau : elle indique que l'utilisateur est indisponible. Nous l'utilisons donc pour initialiser notre tableau, et l'utiliserons encore par la suite pour réagir à l'extinction d'un utilisateur. Une implémentation plus réaliste nécessiterait l'emploi d'une structure dynamique pour stocker ces identifiants sans être limité par le nombre de connexions, mais cela dépasse largement notre propos, et un simple tableau suffit amplement aux besoins de ce programme.

Nous abordons dans la seconde phase le code spécifique au commutateur : il consiste en une boucle dont le programme ne sortira qu'après extinction de tous les utilisateurs Reste à savoir ce qui se passe à l'intérieur de cette boucle. Et bien, pour commencer, il faut vérifier si un message a été envoyé, et le cas échéant, le transmettre à son destinataire.

```
/* Un utilisateur aurait-il envoyé un message ? */
if(switch_get_msg(TYPE_CONN, &in)){

  msg_receiver = get_receiver(&in);
  msg_sender = get_sender(&in);
  msg_data = get_data(&in);

  /* Le destinataire est-il actif ? */
```

```
  if(queues[msg_receiver] != sw){

    /* OK, on transmet */
    switch_send_msg(msg_sender, msg_data, queues[msg_receiver]);

    printf("%d -- S -- Sender: %d -- Destination: %d\n",
           (int) time(NULL), msg_sender, msg_receiver);
  }
  else{
    /* Destinataire injoignable */
    printf("%d -- S -- Unreachable destination (Sender: %d - Destination: %d)\n",
          (int) time(NULL), msg_sender, msg_receiver);
  }
```

Tout utilisateur ayant envoyé un message est ensuite susceptible d'être soumis à une requète de service. Il peut alors s'agir, soit d'une terminaison, auquel cas un simple message suffit à en informer l'intéressé pour que ce dernier agisse en conséquence, soit d'un chronométrage, cas à peine plus complexe : il faut s'assurer que cet utilisateur n'est pas déjà en cours de chronométrage et enregistrer l'heure avant d'envoyer le message idoine. Mais si par contre aucun message n'a été reçu, peut-être est-ce faute d'utilisateurs actifs : plutôt que de boucler indéfiniment, nous allons donc nous arrêter, non sans avoir auparavant attendu la fin effective des fils (qui peuvent encore lire leurs messages après déconnexion) et libéré la file d'attente de messages.

```
  /* L'émetteur du dernier message doit-il faire l'objet d'une requète de service ?
  if((myrand(100)  < death_prob) && (queues[msg_sender] != sw)){
    switch(myrand(2))
      {
      case 0:
    /* Terminaison */
    printf("%d -- S -- User %d chosen for termination\n",
          (int) time(NULL), msg_sender);
    switch_send_term(i, queues[msg_sender]);
    break;
      case 1:
    /* Chronométrage */
    if(!timing[msg_sender][0]){
      timing[msg_sender][0] = 1;
      timing[msg_sender][1] = (int) time(NULL);
      printf("%d -- S -- User %d chosen for timing...\n",
            timing[msg_sender][1], msg_sender);
      switch_send_time(queues[msg_sender]);
    }
    break;
      }
  }
}
else{
  if(deadproc == childs){
     /* Plus aucun processus n'est actif, on attend que tous les fils soient bel et
    waitpid(pid, &status, 0);

    /* On libère la mémoire de notre file d'attente de messages */
    remove_queue(sw);

    /* Fin du programme */
    exit(0);
  }
}
```

Et pour la bonne bouche, il nous reste à prendre en compte les messages de service que l'on pourrait recevoir, par un traitement approprié pour les quatre possibilités que nous avons prévues :

- SERV_BIRTH : l'émetteur du message signale qu'il est actif. Il n'y a rien à faire.
- SERV_DEATH : l'émetteur informe de son arrêt. On efface du tableau son identifiant de filede messages, et on incrémente le compte des utilisateurs déconnectés.
- SERV_QID : l'émetteur transmet son identifiant de file de messages. Enregistrons cette valeur dans notre tableau, et nous serons désormais en mesure d'envoyer des messages à ce dernier.
- SERV_TIME : l'émetteur transmet l'heure à laquelle il a traité la requète. En soustrayant l'heure du début de l'opération, nous obtenons son temps de réaction.

```
if(switch_get_msg(TYPE_SERV, &in)){
  msg_service = get_service(&in);
  msg_sender = get_sender(&in);

  switch(msg_service)
    {
    case SERV_BIRTH:
      /* Nouvel utilisateur */
      printf("%d -- S -- Activation of user %d\n", (int) time(NULL), msg_sender);
      break;

    case SERV_DEATH:
      /* Fin d'un utilisateur */
      printf("%d -- S -- User %d is terminating\n", (int) time(NULL), msg_sender);

      /* On oublie son identifiant de file de messages */
      queues[msg_sender] = sw;

      /* On garde une trace du nombre de clients déconnectés */
      deadproc++;
    break;

    case SERV_QID:
      /* l'utilisateur transmet son identifiant de file de messages */
      msg_data = get_data(&in);
      printf("%d -- S -- Got queue id of user %d: %d\n",
            (int) time(NULL), msg_sender, msg_data);
      queues[msg_sender] = msg_data;
      break;

    case SERV_TIME:
      msg_data = get_data(&in);

      /* Une simple soustraction entre la date reçue et celle */
      /* enregistrée en début d'opération */
      /* nous donne le temps de réponse */
      timing[msg_sender][1] = msg_data - timing[msg_sender][1];

      printf("%d -- S -- Timing of user %d: %d seconds\n",
            (int) time(NULL), msg_sender, timing[msg_sender][1]);
      /* Le chronométrage fini, on fait en sorte de */
      /* pouvoir en réaliser un autre : on "libère" le compteur */
      timing[msg_sender][0] = 0;
      break;
    }
}
```

Recommandations d’usage

Nous voici arrivés au terme de cette série d’articles sur la programmation en temps partagé : nous sommes certes bien loin d’en avoir exploré toutes les possibilités, mais au moins avons nous dorénavant une vision d’ensemble sur les communications inter-processus (IPC) et les solutions qu’elles apportent. Vous êtes bien sûr cordialement invités à modifier à loisir le programme d’exemple : si difficile que puisse se révéler le débogage d’un programme multi-processus, c’est en même temps une occasion de rève pour enrichir son expérience dans ce domaine, et le débogueur étant l’arme absolue du programmeur, ce serait malheureux de s’en passer... Pour ceux qui ne connaissent pas encore, voyez la liste de liens en fin d’article, vous y trouverez quelques références.

Une dernière remarque sur les IPC. Lors de vos expérimentations, il est fort probable que vos programmes n’aient pas le comportement attendu, vous obligeant à interrompre manuellement son exécution. Il est donc important de savoir faire un peu de ménage. Ainsi, si vous tapez la combinaison <Ctrl-C>, vous n’interrompez pas tous les processus du programme : je vous laisse donc le soin de lire la page de manuel de la commande "kill", pour vous affranchir de cet écueil. Mais qu’en est-il alors de nos structures IPC ? Si un processus est tué, il y a fort à parier qu’une file d’attente IPC allouée pas ses soins lui survive. Et c’est en effet ce qu’il advient, puisqu’une telle structure appartient au noyau (et pas au programme). On pourra constater les dégâts avec la commande "ipcs" qui liste l’ensemble des ressources IPC présentes (mémoire partagée, sémaphores, et files d’attente de messages), puis au besoin libérer la mémoire inutilement encombrée avec "ipcrm" : prenons bien soin toutefois de ne pas altérer les ressources d’un autre programme.

Et maintenant, forts de toute cette connaisssance, il nous reste à la mettre en application. Commençons par décompresser le programme ("tar -xvzf ipcdemo-0.1.tar.gz") et le compiler ("make", ou "make all"). Ensuite, pour un premier essai, je vous suggère de lancer le programme avec les paramètres suivants : 5 70 70. Libre à vous ensuite de tester d’autres valeurs, mais attention à ne pas choisir des probabilités trop faibles, ou vous risquez de vous ennuyer fermement, en attendant la fin du programme.

Conclusion

Je voudrais d’abord m’expliquer sur la publication tardive de cet article : j’avoue ne pas consacrer ma vie entière à la programmation. Mais, saurait-on me blâmer d’avoir d’autres centres d’intéret ?..

Sinon, comme de coutume, je suis impatient de connaître vos commentaires et autres suggestions sur cet article comme ceux à venir (pourquoi pas les "threads" ?).

N.d.T. : me voyant dans l’incapacité de donner une traduction valable du terme "thread" (dans le contexte bien particulier de la programmation système), je préfère une explication sommaire : les "threads" sont des processus un peu particuliers, conçus de façon à s’éxécuter séquentiellement, et non en totale concurrence. Si vous voulez en savoir plus, l’auteur propose justement d’en faire le sujet d’un prochain article : faites lui part de votre motivation (en italien, de préférence, voire en anglais). peut-être alors me faudra-t’il trouver mieux que "filament" ou encore "file d’exécution" pour évoquer les "threads". Et que dire encore de "fork" ?

Liens utiles

Pour les suggestions de lecture, je vous laisse revenir aux articles précédents. Et pour les liens Internet, je vous oriente aujourd'hui vers ce que l'on pourrait qualifier comme les indispensables du programmeur. Si vous avez le malheur de na pas comprendre l'anglais, utilisez votre moteur de recherche favori et vous trouverez pléthore d'information dans la langue de Molière...

Au risque de me répéter, le débogueur est le meilleur ami du programmeur. Apprenez donc à vous servir du premier (gdb), et quand vous maîtriserez les lignes de commande, le second (ddd) avec son interface graphique n'aura pas le moindre secret pour vous.

- GDB, le projet GNU de débogueur : www.gnu.org/directory/gdb.html
- DDD, l'afficheur de données de débogage : www.gnu.org/software/ddd

La dernière tentative d'exécution du programme s'est interrompue brutalement, vous gratifiant de ce message sibyllin : "Segmentation fault". Le programme a donc tenté d'accéder à une adresse mémoire interdite, mais cela ne nous dit pas où se trouve l'instruction fautive. Valgrind est là, avec ses possibilités de simulation de mémoire, pour nous sortir de ce mauvais pas.

- Valgrind, un débogueur de mémoire pour x86-linux : developer.kde.org/~sewardj

Enfin, si la complexité et la rigueur du C vous rebutent, vous pouvez aussi vous tourner vers un langage plus souple. Python constitue une excellente alternative : entre le support du "fork" et des extensions C, vous aurez en main tous les outils pour réussir.

- Python : www.python.org

Téléchargement

- Suivez ce lien pour obtenir le programme d'exemple

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

2005-01-14, generated by lfparser_pdf version 2.51