Programmation Système
TP no 1 : Processus, tubes et communications

Guillaume Mercier
email : mercier@enseirb.fr

11 Octobre 2019

1 Client-serveur local

Le but de cet exercice est d'écrire une sorte de programme client-serveur fonctionnant en mode local, c'est-à-dire que les processus clients et serveurs s'exécutent sur la même machine physique. Seuls les moyens de communications locaux sont autorisés pour cet exercice, à l'exception des sockets (domaine `AF_UNIX` ou autre ...). Bien évidemment, vous n'avez pas accès non plus aux primitives de types `connect` et `accept`.

1.1 Hypothèses et données de base

Affiliation des processus Les processus clients et le processus serveur ne sont *pas* affiliés. En revanche, le processus serveur va créer des processus fils afin de répondre aux requètes des processus clients. Pour le moment, le système que vous allez mettre en place sera limité car tous les processus (le serveur et les clients) appartiennent à un même utilisateur.

Identification du serveur On supposera que le processus serveur tourne continuellement sur la machine et que son PID est connu de tous les processus clients (dans le cas des communications réseau, on suppose que l'on connait l'adresse IP de la machine sur laquelle tourne le serveur, ainsi que le numéro de port). Ce PID va donc jouer dans notre cas le rôle du numéro de port dans le cas réseau habituel.

Demande de connexions de la part des clients Pour faire sa demande de connexion, un processus client va envoyer au serveur un signal de type `SIGUSR1`, avec deux numéros qui vont permettre d'identifier deux tubes *nommés* qui serviront aux communications exclusives entre le processus-client demandeur et le processus serveur (ou l'un de ses fils plus exactement). Les tubes nommés seront créés *par le processus client* dans le répertoire `/tmp` de la machine sur laquelle vous travaillez et leur nom devra suivre le format suivant :
`PIPE_<PID du processus client>_numéro1` et `PIPE_<PID du processus client>_numéro2`. Avec `numéro1` et `numéro2` des numéros générés (semi-)aléatoirement. Cependant, ces numéros seront stockés sur des *entiers non signés de 16 bits*.

Afin d'envoyer ces numéros au serveur en même temps que le signal, vous devrez utiliser le traitant de signal étendu (i.e le champ `sigaction` de la structure `struct sigaction`) ainsi que la fonction `sigqueue`.

`sigqueue` permet d'envoyer un entier 32 bits avec le signal, il va donc vous falloir utiliser ces 32 bits afin d'envoyer les deux numéros d'identification des tubes, codés chacun sur 16 bits (rappel : en C, il existe des opérations *bitwise*, et notamment de décalage » et «, ...).

Acceptation de la connexion par le serveur Le serveur devra mettre en place deux traitants de signaux :

— un premier traitant pour la gestion du signal `SIGCHLD` afin de bien éliminer au fur et à mesure les processus-fils zombies correspondants aux processus de service qu'il va créer pour répondre aux requêtes des clients ;

— un second traitant pour la gestion du signal `SIGUSR1` correspondant aux demandes effectives de connexion émanant des processus clients.

Dès qu'il reçoit une demande, le serveur devra créer un processus-fils qui sera chargé de répondre aux demandes du client qu'il sert. Les communications devront utiliser les tubes créés spécialement à cette occasion par le client.

Mise en place du service Maintenant que l'infrastructure d'échange est en place, vous pouvez implémenter un service, comme par exemple le cryptage d'un texte :

1. le client envoie le texte à crypter au serveur (ou plutôt au processus de service dédié) ;
2. le processus de service récupère ce texte et le crypte (vous pouvez utiliser le code de César fait l'an dernier en TP de C ...) ;
3. le processus de service renvoie au client le résultat du cryptage.

Bien entendu, il faudra mettre en place un protocle d'échange puisque les données transitent en tant que flux dans les tubes.

2 Processus, tubes, mémoire partagée

Le but de ce TP est de mettre en place un programme qui va permettre d'effectuer un calcul de façon distribuée, c'est à dire en répartissant la charge de ce calcul sur plusieurs processus. Afin de fixer les idées, nous allons supposer que le calcul en question sera la détermination du maximum d'un ensemble de valeurs entières (le maximum d'un tableau par exemple, les valeurs pouvant être déterminées aléatoirement si vous le désirez).

2.1 Communications par tubes

Dans un premier temps, nous allons supposer qu'un ensemble de N processus vont participer à ce calcul. Au début, un unique processus va posséder le tableau sur lequel tous les autres processus vont travailler. Ce processus-maître va donc être responsable de :

— créer les $N - 1$ autres processus participants ;

— initialiser le tableau d'entiers ;

— créer les tubes nécessaires pour communiquer ;

— distribuer aux processus les parties du tableau surs lesquelles ils doivent effectivement travailler (via les tubes).

Dans un schéma de type maître-esclave, les différents processus esclaves vont communiquer uniquement avec le processus-maître. Ils n'ont pas besoin de communiquer entre-eux (*a priori*. Chaque processus-esclave va calculer le maximum de sa partie de tableau et communiquer ce

maximum local au processus-maître qui va ensuite procéder au calcul du maximum global en travaillant sur tous les maxima locaux qui lui auront été communiqués.

2.2 Communication par mémoire partagée

Dans un deuxième temps, les processus utiliseront de la mémoire partagée pour effectuer ce calcul :

— le tableau d'entiers sera stocké dans une zone de mémoire accessible par tous les processus
— chaque processus va travailler sur un partie du tableau qui lui est propre
— un variable en mémoire partagée sera également utilisée pour stocker le résultat

Afin de synchroniser les accès à cette variable, les N processus utiliseront un tube (anonyme ou nommé), en mode bloquant. Au départ, ce tube contiendra *un* caractère. quand un processus voudra accéder à la variable (en lecture et/ou écriture), il devra effectuer une opération de lecture. Après avoir accédé (et éventuellement modifié) la variable, le processus écrira *un* caractère dans le tube.

2.3 Évaluation de performance

Si vous avez le temps, vous pouvez comparer les performances des différentes approches :

— l'approche maître-esclave (avec les tubes de communication) : vous pouvez faire varier le nombre de processus-esclaves (1, 2, 3, 4, etc.) et analysez ces performances au regard du nombre de processeurs disponibles sur votre machine. En particulier, que se passe-t-il lorsque le nombre de processus dépasse le nombre de processeurs ?
— l'approche mémoire partagée, là aussi en faisant varier le nombre de processus et en comparant avec les processeurs disponibles.

3 Signaux

3.1 Masquage de signaux

Un processus ne prend pas forcément en compte les signaux dans l'ordre où ils ont été reçus. À l'aide des masques, faites en sorte de forcer cet ordre. Effectuez des tests avec `SIGUSR1`, `SIGUSR2` et `SIGSEGV` et vérifiez que votre programme fonctionne correctement : par exemple vous allez envoyer `SIGSEGV` en premier, mais vous voulez qu'il soit pris en compte en dernier, après `SIGUSR1` et `SIGUSR2`.

3.2 Signaux : communications en Morse

Cet exercice a pour but de réaliser une communication utilisant le code *morse* (ou un quelconque code similaire) entre deux processus — un client et un serveur — au moyen de signaux. Un résumé du codage morse est disponible dans le fichier `morse.txt` du répertoire `/net/ens/mercier/IF210/TP1/`. Pour information, il faut y ajouter les particularités suivantes :

— les signes morses sont séparés par un blanc ;
— les lettres sont séparées par trois blancs ;
— les mots sont séparés par sept blancs.

Questions

— écrivez un programme `tstsig_1.c` qui boucle infiniment et affiche :
  — `'.'` lorsqu'il reçoit un signal `SIGUSR1` ;
  — `'-'` lorsqu'il reçoit un signal `SIGUSR2` ;
  — `' '` lorsqu'il reçoit un signal `SIGALRM`.

  Vous utiliserez la fonction `sigaction(3)` pour gérer la mise en place des gestionnaires (*handler*) de signaux. Vous pourrez utiliser la commande `kill(1)` pour envoyer des signaux au processus depuis un terminal.

— écrivez un programme `tstsig_2.c` à partir de `tstsig_1.c` qui — en plus des fonctionnalités de `tstsig_1.c` — se termine en affichant le message `"[over]"` lorsqu'il reçoit un signal `SIGTERM`.

— écrivez un programme `serveur.c` et un programme `client.c` avec les fonctionnalités suivantes :

  serveur Le serveur itère en demandant un message (directement en morse, ou en format texte si vous avez le temps de programmer la conversion en fin de TD) et un numéro de `pid`, puis envoie le message au processus client correspondant en utilisant les 4 signaux ci-dessus avec les mêmes conventions de signification.

  client Le client est une version étendue de `tstsig_2.c`. Il prend le `pid` du serveur en argument de ligne de commande puis reçoit et affiche un message envoyé par le serveur.

  Note : il est indispensable que le message soit transmis de manière fiable.

3.3 Signaux et zombies

Faites le TD du chapitre 7 (i.e la section 7.8).