# TP 3 – IPC : Les files de messages

On souhaite mettre en place un Serveur de calcul auquel des clients accèdent pour demander la réalisation d'une opération arithmétique.

Le comportement est le suivant :

- ◇ un programme dénommé *serveur* attend les requêtes de clients sur un medium[1] de communication ; à l'arrivée d'une requête d'un client, le serveur traite la requête et communique au client le résultat du traitement ;
- ◇ le programme *client* envoie une requête au serveur, attend l'arrivée du résultat puis l'affiche.

Voici l'affichage que l'on doit obtenir au final :

```
TP4 158 % serveur &
Serveur (21091) : file de messages créée (589824)
TP4 159 % client 2 + 3
client 21097 : 2 + 3 = 5.000000
TP4 160 % client 20 / 3
client 21098 : 20 / 3 = 6.000000
```

Le résultat final sera obtenu étape par étape (exercice par exercice).

### Lien avec le cours : Producteur-Consommateur

Ce problème est de type "producteur-consommateur" : N producteurs (les clients) et 1 consommateur (le serveur). Nous n'allons pas utiliser de sémaphore ni de moniteur (comme en cours, chapitre 6), mais un mécanisme système qui est dédié à ce type de problème : les files à messages des IPC System V.

Ce mécanisme permet aux producteurs de mettre des messages avec une étiquette (un *type*) et aux consommateurs de récupérer les messages dans l'ordre d'entrée (façon tampon circulaire) et que l'on peut filtrer par étiquette (*type*). Il n'y a pas besoin de gérer les pointeurs sur le tampon.

On profitera de ce mécanisme pour renvoyer la réponse du serveur aux clients.

1. c'est-à-dire quelque chose qui sert à communiquer (un tube, une file, un truc, ...)

## Exercice n°0

Lisez les paragraphes sur les généralités des IPCs (voir chapitre 4 suite) ainsi que sur les files à messages.

**Fichiers**

☞ Pour chaque exercice, vous ferez des fichiers séparés.

## Exercice n°1 : 1 serveur, 1 client

Dans un premier temps, pour vous familiariser avec les files de messages, vous allez réaliser un client qui envoie un entier (donné en argument) et un serveur qui renvoie cet entier après l'avoir multiplié par 2. Au retour du message, le client affichera le résultat. Le serveur se finira après avoir envoyé le message. Suivez les indications suivantes (ainsi que les exemples donnés dans le cours sur les IPCs) pour réaliser cet exercice :

Rappelons que la file de messages est à partager entre deux processus donc cela implique l'utilisation d'une même clé pour le serveur et les clients. Et pour ce TP, **les clients et le serveur n'ont pas de lien de filiation**. C'est le serveur qui créera la file de messages (**et la détruira, quelques secondes après avoir renvoyé le résultat**) :

- Pour **générer la clé** ou la retrouver, on utilisera la fonction *key_t ftok(const char *path, int id);* où *path* est un fichier existant non-vide et *id* un (petit) entier : ils seront définis (avec #define) dans un fichier .h à inclure dans les programmes serveur et client. La fonction *ftok* renvoie -1 si la clé n'a pas pu être créée. Pensez à tester cette valeur . . .surtout lors du débuggage !
- Pour récupérer le numéro de la file dans le client, on utilise comme pour sa création (dans le serveur), la fonction *msgget* mais avec l'option 0 : elle renvoie -1 si la file n'existe pas. Pensez à tester cette valeur . . .surtout lors du débuggage !
- L'envoi et la réception se font avec *msgsnd* et *msgrcv* (voir le cours : Chapitre 4 suite). Ces fonctions renvoient -1 si l'envoi ou la réception n'ont pas réussi. Pensez à tester cette valeur . . .surtout lors du débuggage . . .euh . . .j'ai l'impression de l'avoir déjà dit, non ? ;-)
- Attention au *type* des messages : si vous n'utilisez que des messages de type 1 (ou de même valeur pour le client et pour le serveur) que risque-t-il de se passer ? Indice : Si le client envoie un message de type 1, et qu'il arrive à sa ligne de code pour la réception avant le serveur, que se passe-t-il ?

  Réponse : Le client risque de récupérer son propre message avant que le serveur n'ait eu le temps de le récupérer. Il faut donc différencier le type d'un message client-vers-serveur d'un message serveur-vers-client.
- C'est le serveur qui doit détruire la file avec la fonction *int msgctl(msgid, IPC_RMID, 0)* avec *msgid* le numéro de la file donnée lors de la création (il attendra quelques secondes pour laisser le temps au client de récupérer le message. Cette fois-ci nous n'utiliserons pas de signal).

**Vérifiez vos IPCs**

☞ Lors de réalisation du programme, vérifiez que la file a bien été détruite : taper *ipcs* (dans le shell) et effacer, si besoin est, la file de messages existante *ipcrm -q numéro_msqid*

## Exercice n°2 : 1 serveur, N clients

Maintenant, on suppose que plusieurs clients peuvent envoyer des messages au serveur. Lors de l'exercice précédent, vous avez utilisé un *type* pour les envois d'un client et un autre pour la réponse du serveur. À partir de maintenant, plusieurs clients peuvent envoyer un message.

- Tous les clients peuvent envoyer un message de même *type*, le serveur traitera les demandes dans l'ordre de dépôt dans la file. Mais que risque-t-il de se passer si le serveur utilise un même *type* pour toutes les réponses aux clients ?
- Il faut trouver un moyen pour que chaque client retrouve sa réponse. Indice : comment identifie-t-on de manière unique un processus ? Utilisez cela comme *type* pour la réponse du serveur (il faut que le client le communique au serveur par l'intermédiaire de la structure).
- Le serveur bouclera tant qu'il n'aura pas reçu le signal INT (CTRL + C) provenant de l'utilisateur (pas d'un client). N'oubliez de détruire la file de messages.

- Testez avec plusieurs clients simultanément : réalisez un script Bash pour lancer plusieurs clients en même temps (en tâche de fond).

## Exercice n°3 : Mise en place du véritable serveur de calcul

- Un client envoie dans un message (= une structure) l'opération et le(s) opérande(s) (le *type* du message sera par exemple 1 pour l'ensemble des clients). Il attend le retour, affiche le résultat puis se finit. Les opérations possibles sont :
  - ◇ - : pour la soustraction ; exemple : 2 - 4
  - ◇ + : pour l'addition ; exemple : 2 + 4
  - ◇ * : pour la multiplication ; exemple : 2 * 4
  - ◇ / : calcule la division **entière** ; s'utilise avec 2 opérandes ; exemple : 20 / 3

  C'est au client de "traduire" l'opération, et au besoin de vérifier les valeurs des opérandes. L'opération doit être traduite avec un entier (par le client avant l'envoi du message) pour faciliter le communication (utilisez des #DEFINE pour les valeurs choisies). Pour la comparaison de chaines de caractères, voir l'encart "Aide en C".
- le serveur reçoit le message du client puis renvoie un message (= une structure) contenant le résultat. Il est possible d'utiliser la même que celle du client (en réutilisant les champs par exemple).

L'opération à effectuer sera donnée par les arguments du client : `client 2 + 4`. Le serveur bouclera tant qu'il n'aura pas reçu le signal INT (CTRL + C) provenant de l'utilisateur (pas d'un client).

**Attention** : pour l'opérateur de multiplication, il faut mettre des guillemets : *client 2 "*" 2* (ou alors prenez d'autres symboles).

### Aide en C

- L'argument *argc* de la fonction *main* donne le nombre d'arguments.
- Pour tester 2 chaînes de caractères, on peut utiliser la fonction *int strcmp( const char *str1, const char *str2 )* (de <string.h>). Cette fonction renvoie 0 si les deux chaines sont identiques.

## Exercice n°4 : Opérations prioritaires

On souhaite maintenant que certains clients soient servis prioritairement par rapport aux autres clients. Les opérations '*' et '/' sont prioritaires par rapport à '+' et '-'. On peut jouer sur le *type* des messages et sur l'argument de la fonction *msgrcv*. Quelles sont les modifications à apporter pour prendre en compte ce mécanisme de priorité ? (regardez le poly)

Réponse : On utilise les types négatifs comme argument dans la fonction de réception (pas à l'envoi) : -4 signifie que l'on va attendre tous les types < 4 en prenant prioritairement les plus petits types en premier. Il faut donc, à l'envoi, donner des types (positifs) différents suivant la priorité de l'opération.

**Remarque** : comme l'on cherche à tester la priorité, mettez une attente de 1 seconde pour chaque boucle du serveur (les messages auront le temps de s'accumuler). Réalisez un script shell pour lancer plusieurs clients en même temps.

### FIN de séance

- Vérifiez qu'il ne reste pas de processus ou d'IPC en mémoire. Pour vérifier les IPC : taper *ipcs* (dans un shell) et pour effacer une file de messages existante *ipcrm -q numéro_ msqid*.