Travail pratique #2

IFT-2245

April 24, 2019

¡¡ Dû le 5 avril à 23h55 !!

1 Survol

Ce TP vise à vous familiariser avec la programmation avec des threads et des sockets dans un système de style POSIX. Les étapes de ce travail sont les suivantes:

1. Lire et comprendre cette donnée.
2. Lire, trouver, et comprendre les parties importantes du code fourni.
3. Compléter le code fourni.
4. Écrire un rapport. Il doit décrire **votre** expérience pendant les points précédents: problèmes rencontrés, surprises, choix que vous avez dû faire, options que vous avez sciemment rejetées, etc... Le rapport ne doit pas excéder 5 pages.

Ce travail est à faire en groupes de 2 étudiants. Le rapport, au format LaTeX exclusivement et le code sont à remettre par remise électronique avant la date indiquée. Chaque jour de retard est -25%. Indiquez clairement votre nom au début de chaque fichier.

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Si un étudiant ne trouve pas de partenaire, il doit me contacter au plus vite. Des groupes de 3 ou plus sont **exclus**.

2 Introduction

Pour ce TP, vous devez implémenter en C l'algorithme du banquier, un algorithme qui permet de gérer l'allocation des différents types des ressources, tout en évitant les interblocages (deadlocks). Le code fourni implémente un modèle de client-serveur qui utilise les prises (sockets) comme moyen de communication.

D'un côté l'application serveur est analogue à un système d'exploitation qui gère l'allocation des ressources, comme par exemple la mémoire, le disque dur ou les imprimantes. Ce serveur reçoit simultanément plusieurs demandes de ressources des différents clients à travers des connexions. Pour chaque connexion/requête, le serveur doit décider quand les ressources peuvent être allouées au client, de façon à éviter les interblocages en suivant l'algorithme du banquier.

De l'autre côte, l'application client simule l'activité de plusieurs clients dans différents fils d'exécution (threads). Ces clients peuvent demander des ressources si elles sont disponibles ou libérer des ressource qu'ils détiennent à ce moment.

Ce TP vous permettra de mettre en pratique quatre sujets du cours différents:

- Fils d'exécution multiples (multithreading).
- Prévention des séquencements critiques (race conditions).
- Évitement d'interblocages (deadlock avoidance).
- Communication entre processus via des prises (sockets).

3 Mise en place

Des fichiers vous sont fournis pour vous aider à commencer ce TP. On vous demande de travailler à l'intérieur de la structure déjà fournie.

Les deux applications, client et serveur, utilisent les librairies du standard POSIX pour l'implémentation des structures dont vous allez avoir besoin, notamment les sockets, les threads, les semaphores et les mutex. Ces librairies sont par defaut installées sur une installation du système GNU/Linux.

3.1 Makefile

Pour vous faciliter le travail, le fichier `GNUmakefile` de l'archive vous permet d'utiliser les commandes suivantes:

- `make` ou `make all`: Compile les deux applications.
- `make release`: Archive le code dans un `tar` pour la remise.
- `make run`: Lance le client et le serveur ensembles.
- `make run-client`: Lance le client.
- `make run-server`: Lance le serveur.

- `make run-valgrind-client`: Lance le client dans valgrind.
- `make run-valgrind-server`: Lance le serveur dans valgrind.
- `make clean`: Nettoie le dossier build.

4 Protocole client serveur

Le protocole de communication entre le client et le serveur est **binaire** et constitué de commandes et de réponses qui sont specifié par:

```
CMD_TPYE TAILLE INT1 INT2 INT3 ...
```

Les type de commande sont donner dans le fichiers "common.h":

```
enum cmd_type {
  BEGIN,
  CONF,
  INIT,
  REQ,
  ACK,
  WAIT,
  END,
  CLO,
  ERR,
  NB_COMMANDS
};
```

BEGIN(0) 1 *RNG*	Requête pour commencer le serveur
CONF(1) *nb_resources* rsc_0 rsc_1 ...	Provisionne les ressources
END(6) 0	Termine l'exécution du serveur
INIT(2) *nb_resources*+1 *tid* max_0 max_1 ...	Annonce client avec son usage maximum
REQ(3) *nb_resources*+1 *tid* rsc_0 rsc_1 ...	Requête de ressources
CLO(7) 1 *tid*	Annonce la fin du client

Figure 1: Requêtes du client

ACK(4) 1 *RNG*	Requête de commencement de serveur exécutée avec succès
ACK(4) 0	Requête de ressources exécutée avec succès
ERR(8) *NB_DE_CHARS msg*	Commande invalide, *msg* explique pourquoi
WAIT(5) 1 *sec*	Demande au client d'attendre *sec* secondes

Figure 2: Réponses du serveur

Note: Vous pouvez utiliser "RNG" pour spécifier le nombre de clients qui se connecteront au serveur (points bonus pour ne pas spécifier)

Le protocole est dirigé par le client qui fait des requêtes au serveur. Les réponses du serveur sont toujours ACK, ERR, ou WAIT, comme montré à la Figure 2.

Les requêtes, montrées à la Figure 1, se divisent en deux parties:

- les commandes globales utilisées au tout début pour configurer le serveur puis à la fin pour le terminer.
- Les commandes par client (où chaque client est en fait un thread du programme `tp2_client`).

Donc après avoir lancé le serveur, le programme client le configure par exemple avec:

```
0 1 7
1 5 10 5 3 23 1
```

Suite à cela, les différents threads du client peuvent chacun ouvrir une connection et y envoyer leurs requêtes, qui pourraient alors avoir la forme suivante pour le thread client avec le id 332:

```
2 6 332 1  2  1 10 0
3 6 322 1  0  0  1 0
3 6 332 0  2  0  0 0
3 6 332 0  0 -1  0 0
3 6 332 0 -2  0  0 0
7 1 332
```

À remarquer que pour libérer des ressources, il suffit d'utiliser des quantités négatives dans une *requête*.

À remarquer que si un client est mis en attente par un WAIT, il ne fait que répéter la même requête après l'attente jusqu'à recevoir un ACK.

C'est une erreure s'il reste des ressources occupées lors du CLO ou des clients encore connectés lors du END. À la fin, le serveur effectue alors l'impression du journal à l'intérieur d'un fichier et le client fait de même à la réception de ACK. Le serveur peut alors se fermer ainsi que le client.

4.1 Boni

1. (1 point) Autoriser un nombre dynamique de clients (ne pas spécifier le nombre de clients avec RNG).
2. (1 point) Mettre en œuvre un schéma de cryptage pour permettre une communication sécurisée entre le client et le serveur. Vous devez modifier le Makefile afin que votre schéma de cryptage puisse être utilisé facilement ou non. En outre, après la compilation, le cryptage doit être activé via un indicateur de ligne de commande `--secure`.

4.2 Remise

Pour la remise, vous devez remettre tous les fichiers C ainsi que le `rapport.tex` dans une archive `tar` par la page StudiUM du cours. Vous pouvez utiliser `make release` pour créer l'archive en question.

Le programme doit être exécutable sur `arcade.iro` ou sur une nouvelle image de docker d'ubuntu.

Cela ne vous empêche pas bien sûr de le développer sur un système différent, e.g. sous Windows avec Cygwin, mais assurez-vous que le résultat fonctionne *aussi* sur `arcade.iro`.

5 Détails

- La note sera divisée comme suit: 20% pour chacune des 4 sujets (multi-threading, race conditions, deadlock avoidance, et sockets), et 20% pour le rapport.
- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, ...) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat.
- Le code ne doit en aucun cas dépasser 80 colonnes.
- Vérifiez la page web du cours, pour d'éventuels errata, et d'autres indications supplémentaires.
- La note sera basée d'une part sur des tests automatiques, d'autre part sur la lecture du code, ainsi que sur le rapport. Le critère le plus important, et que votre code doit se comporter de manière correcte. Ensuite, vient la qualité du code: plus c'est simple, mieux c'est. S'il y a beaucoup de commentaires, c'est généralement un symptôme que le code n'est pas clair; mais bien sûr, sans commentaires le code (même simple) est souvent incompréhensible. L'efficacité de votre code est sans importance, sauf s'il utilise un algorithme vraiment particulièrement ridiculement inefficace.
- Les fuites de mémoire sont pénalisées.