PR6 – Programmation réseaux
TP n° 7 : Processus légers et verrous en C

Exercice : Un client et un serveur pour un service de tuteurs

On souhaite programmer un serveur et un client IPv6 afin de gérer un ensemble de tuteurs disponibles pour aider des étudiants ; ceux-ci peuvent se connecter via une application auprès d'un serveur qui leur permet de demander l'allocation d'un tuteur et de converser avec lui.
Dans la suite, un «message» est une «ligne» de caractères (donc terminée par un `\n`!!!) ne dépassant pas 100 caractères, `\n` compris.

Serveur Le serveur attend les messages des clients sur son port. Il maintient à jour une liste chaînée de tuteurs disponibles, où un tuteur est représenté par son nom (appelé *id*) et sa discipline. Les messages peuvent être :

- `LIST` signifie que le client demande la liste des disciplines pour lesquels un tuteur est disponible (il peut y avoir plusieurs tuteurs de même champ de compétence). Une discipline peut donc apparaître plusieurs fois. Le serveur renvoie donc au client cette liste. Pour ce faire, il commence par envoyer au client un premier message contenant juste le nombre n (en ASCII) d'éléments de la liste. Il envoie ensuite n messages du type : `<discipline>` (les caractères `<` et `>` ne font pas partie du message.
- `ACQUIRE <subj>` signifie que le client demande un tuteur de la discipline `subj`. S'il y a un tuteur de cette discipline disponible et aucun tuteur n'est déjà affecté au client, le serveur renvoie au client l'identifiant de ce tuteur et le retire de la liste des tuteurs disponibles. Si un tuteur est déjà affecté au client, le serveur renvoie le message `TUTOR_ALREADY_ASSIGNED`. S'il n'y a pas de tuteur de cette discipline disponible, le serveur renvoie le message d'erreur `TUTOR_NOT_AVAILABLE`.
- `QUESTION <text>` signifie que le client pose une question au tuteur qui lui a été attribué, auquel cas, le serveur répond toujours par le message `ANSWER lis ton cours` ou n'importe quel message de type `ANSWER <message>`. Ce message ne peut être envoyé que si l'acquisition d'un tuteur a été correctement réalisée auparavant, sinon le serveur renvoie `TUTOR_NOT_ASSIGNED`.
- `RELEASE` signifie que le client rend disponible son tuteur. Le serveur rend ce tuteur à nouveau disponible. Ce message ne peut être envoyé que si l'acquisition d'un tuteur a été correctement réalisée auparavant, sinon le serveur renvoie `TUTOR_NOT_ASSIGNED`.
- Pour toutes les autres messages, le serveur renvoie `ERROR`.

Client Le client se connecte au serveur et lui envoie des requêtes. Il interagit avec l'utilisateur via un terminal. Sur ce terminal, un prompt (`TuteurOnLine> `) invite l'utilisateur à entrer sa nouvelle requête tant que celui-ci ne demande pas à se déconnecter et le client y affiche les réponses. Depuis le prompt, l'utilisateur peut taper :

- `LIST`, alors le client affiche les disciplines actuellement disponibles ;
- `ACQUIRE <subj>`, alors s'il y a un tuteur affecté le client affiche l'identifiant ; sinon il affiche un message d'erreur ;

- `QUESTION <text>` pour poser une question,
- `RELEASE`, alors le client affiche le message `Merci!` et termine proprement la connexion avec le serveur ;

Une conversation pourrait être :

```
$ ./client adresse_serveur port_serveur
tuteurOnLine> LIST
prog_java
proba
prog_c
prog_reseaux
tuteurOnLine> ACQUIRE proba
Ok, connecté avec FiFi
tuteurOnLine> QUESTION quel est la probabilité de gagner au loto
FiFi: lis ton cours
tuteurOnLine> QUESTION combien font 1 + 1
FiFi: lis ton cours
tuteurOnLine> RELEASE
Ok, fin de session. Merci!
$
```

Commencez par télécharger les fichiers `tutor.h`, `tutor.c` et `tutors_list.txt`.
Dans le fichier `tutor.h`, vous trouverez les définitions des structures `struct tutor` et `struct assignment`, aisnsi que le prototype de fonctions qui pourront vous être utiles et dont la définition se trouve dans le fichier `tutor.c`.
La fonction `init_tutors` (voir dans `tutor.h`) initialise la liste chaînée de tuteurs à partir des informations contenues dans un fichier. Le format du fichier doit être celui du fichier `tutor_list.txt`.

1. Programmez les fonctionnalités `LIST` et `ACQUIRE` du client et d'un premier prototype mono-filaire du serveur : présumez qu'au plus un client à la fois est connecté.

   Pour le serveur, vous pourrez déclarer deux listes chaînées comme variables globales, une liste de tuteurs de type `struct tutor *` et une liste d'associations (tuteur, étudiant) de type `struct assignment *`

2. Modifiez le serveur pour gérer plusieurs clients en parallèle en utilisant des processus légers. Garantissez la cohérence en vous servant du mécanisme de verrouillage (*mutex*). Par exemple, deux clients peuvent demander simultanément de modifier la liste des tuteurs disponibles. Par ailleurs, la liste peut être mise à jour entre le moment où sa taille est envoyée au client et le moment où elle est transmise. Il faudra donc commencer par effectuer une copie de la liste (la section critique du code), puis vous enverrez au client le nombre d'éléments de la liste copiée, ainsi que la liste copiée.

3. Ajoutez les fonctionnalités `QUESTION` et `RELEASE`, ainsi que la gestion des erreurs, au client et au serveur.