Info F-201 : Projet d'OS 2
Rapport

Auteurs: Liefferinckx Romain, Rocca Manuel, Radu-Loghin Rares
Matricules: 000591790, 000596086, 000590079
Section: INFO

2024, 17 Décembre

Contents

1 Introduction

1.1 Présentation du projet et contexte

Dans le cadre de notre cours, nous avons réalisé un second projet qui, en s'appuyant sur les bases établies par le premier, vise à implémenter une messagerie en ligne utilisant un serveur implémenté en C. Cette messagerie permet donc la communication entre deux clients utilisant des ordinateurs différents, grâce à des sockets qui gèrent la connexion en ligne. Elle se compose de deux parties. La première est un programme 'chat' s'occupant de, comme son nom l'indique, d'établir un système de chat entre le client et le serveur. La seconde est un script bash faisant office de 'chat-auto' facilitant l'utilisation du programme chat.
Ce rapport décrit donc les choix d'implémentation, les difficultés rencontrées et les solutions mises en œuvre pour arriver à un résultat final satisfaisant nos attentes.

1.2 Objectifs du projet

A l'image du précédent, l'objectif de ce second projet est de mettre en pratique les concepts vus en cours d'OS. Comme évoqué dans l'introduction, nous nous basons donc sur le premier projet. De ce fait, certains concepts comme les threads, la gestion des signaux et le bash sont une fois de plus utilisés. De nouveaux concepts s'ajoutent naturellement au premiers, à savoir les sockets et les mutexs.
Ce projet permet donc d'encore élargir notre éventail de compétences concernant les outils liés aux OS.

2 Choix d'implémentation

2.1 Choix du langage

Le choix du langage de programmation entre le C et le C++ s'est une fois de plus offert à nous. Le C tombait sous le sens pour deux raisons bien distinctes. La première étant simplement le fait que le C est le langage utilisé durant les séances d'exercices et les cours théoriques. Ensuite, nous avons réalisé le premier projet en C.
Notre choix est donc une continuation logique de ces deux raisons.

2.2 Choix de la méthode de gestion des clients

Concernant la gestion des clients nous avons fait le choix de les gérer à l'aide de threads. Autrement dit, à chaque client, un thread différent est assigné qui exécute la fonction 'handle_client'. Cette approche possède divers avantages décrits ci-dessous:

- **Concurrence et Réactivité:** L'utilisation des threads permet de traiter les requêtes des clients de manière concurrente. Un message envoyé par un client n'affecte pas les autres clients.
- **Fluidité du serveur:** Le serveur étant indépendant des clients, ceux-cis ne peuvent pas le bloquer à cause d'une erreur qui leur est inhérente. Ceci permet donc d'avoir un serveur fluide et réactif.
- **Simplicité de l'implémentation:** Les threads sont relativement simples à utiliser (pthread).
- **Utilisation efficace des ressources:** Les threads partagent le même espace mémoire, ce qui permet une communication rapide et efficace entre eux.
- **Gestion à grande échelle:** Avec les threads, le serveur peut facilement gérer un grand nombre de clients simultanément. Chaque nouveau client est simplement un nouveau thread, ce qui permet une gestion claire et dynamique des connexions.

Alternatives considérées:

- **Multiprocessing:** Remplacer les threads par des processus assignés à chaque client est certes une alternative, cependant, cette méthode consomme nettement plus de ressources. En effet, chaque processus a son propre espace mémoire. Ceci nécessite donc une gestion de communication entre processus qui peut s'avérer plus complexe et moins efficace que celle entre threads.
- **Multiplexing/Polling (select, poll, epoll):** Une autre alternative est l'utilisation des techniques de polling comme `select`, `poll` ou `epoll`. Bien que ces techniques sont certainement tout aussi efficaces que les threads pour gérer de nombreuses connexions simultanées sans causer de bloquage, elles sont plus complexes à implémenter. N'ayant pas beaucoup de temps pour la réalisation du projet et nos connaissances étant restreintes sur le sujet, nous ne nous sommes pas risqués a les utiliser.

En définitive, l'utilisation des threads nous a permis de créer un serveur fluide, efficace et facile à maintenir, capable de gérer plusieurs clients simultanément.

2.3 Gestion des accès concurrents

Afin d'assurer le fonctionnement correct de notre messagerie, la gestion des accès concurrents s'est révélée être un défi crucial à surmonter pour assurer l'intégrité des données partagées entre les différents threads et éviter leur corruption potentielle. Nous avons donc utilisé des mutexs pour synchroniser ces accès concurrents aux ressources partagées, comme la liste des clients connectés. Soulignons ci-dessous certains exemples de leur utilisation dans notre projet:

- **Protection de la liste des clients:** La liste des clients connectés est une ressource partagée entre les différents threads. Pour éviter les conditions de course et garantir la sécurité des données, nous avons utilisé un mutex nommé `clients_mutex`. Avant d'ajouter ou de supprimer un client de la liste, le thread doit verrouiller ce mutex. Une fois l'opération terminée, le mutex est déverrouillé. Par exemple, ces fonctions modifient la liste des clients avec les mutex: `addClient` et `remove_client`. Cela garantit que ces fonctions ne peuvent pas être exécutées simultanément par plusieurs threads, évitant ainsi les conflits et les incohérences.
- **Comptage des clients actifs:** Pour compter le nombre de clients actifs, nous avons également utilisé un mutex pour protéger l'accès à la liste des clients. Cela garantit que le comptage est précis et que la liste n'est pas modifiée pendant l'opération.

2.4 Script bash

L'aspect technique du script bash se trouve dans la redirection des entrées et sorties du programme chat. Nous avons d'abord opté un usage de la commande 'exec'. Or celle-ci remplace le processus courant par le celui exécuté par la commande. Ceci ne correspondant donc pas à nos besoins, nous avons simplement opté pour l'opérateur '|', appelé 'pipe operator', qui permet une redirection de la sortie standard d'une commande vers l'entrée standard d'une autre commande. En utilisant la syntaxe suivante, ' { code } | ./chat', cela nous permet de rediriger l'IO du programme chat vers le code entre crochets qui s'occupe d'ajouter le pseudo de l'utilisateur devant le message envoyé et le retourner complété.
En ce qui concerne la gestion des 'ctrl-D', nous avons simplement opté pour deux boucles 'while read -r'. En effet, si l'entrée standard est fermée, la boucle interne s'arrête et passe à la boucle externe qui redemande le pseudo de l'utilisateur.

3 Difficultés Rencontrées et Solutions

3.1 Gestion des signaux dans client

Notre plus grosse difficultées, fut que le contrôle D marchait sur Mac OS mais pas sur Linux. En effet, sur Mac OS, le contrôle D est interprété et permet d'interrompre l'exection de la boucle de lecture et d'ecriture. Or, sur Linux, apres que la boucle s'interrompe, la lecture se poursuivait indéfiniment. Pour résoudre ce problème, nous avons utilisé la fonction shutdown() qui permet de bloquer toutes les tentatives de lecture du socket spécifié.

3.2 Confusion dans les consignes

Les consignes du point 3.4 spécifiaient que nous devions effectuer la tache suivant: "Afin de connaître le destinataire du message écrit par l'utilisatrice ou utilisateur, chaque message sur stdin doit être précédé du pseudonyme de son destinataire. Si ce pseudonyme contient des espaces, ces espaces sont remplacées par des tirets (p.ex., le pseudo <chat chat>s'écrira <chat-chat>)". Après une longue réflexion, cela nous a paru impossible. En effet, pour lancer le programme chat, il a été spécifié que le 1er argument devrait être le pseudo et que les suivant sevaient obligatoirement être bot et manuel. Dès lors, un utilisateur ne pourra jamais avoir un nom d'utilisateur contenant un espace. Aussi, lorsqu'on écrit un message (après avoir lancé le programme) il nous a été demandé d'igniorer les messages ne contenant pas au minimum 2 mots. Mais alors, il n'y a plus aucun moyen de vérifier que les 2 entrées sont un seul et unique nom d'utilisateur ou bien un nom et un message ce qui ne permet plus de savoir si c'est une erreur ou un message valide. Ou encore, si nous oublions ce dernier point, lorsque le programme cherchera a envoyer un message au premier mot entré, et qu'il ne le trouvera pas, le serveur le signalera, alors que nous cherchions peut être a contacter la combinaison des premiers mots. du message. Dès lors nous avons conclu que ces lignes portaient a confusion et nous avons décidé d'en revenir a l'enoncé de base en laissant cette partie non-faite.

4 Solutions Originales et Améliorations

4.1 Gestion des constante globales

Du fait de l'utilisation de C comme langage de programmation, l'orienté objet nous est inaccessible et nous n'avons donc pas accès aux constantes d'instances d'objets. Le premier réflexe est donc de mettre 'const <nom de la constante>= 'valeur" lorsque celle-ci ne doit pas être modifiée et '<nom de la constante>= 'valeur" lorsqu'elle peut l'être. Cette méthode est tout à fait viable mais, malgré ça, nous avons utilisé une autre approche, à savoir mettre tout les constante globales non modifiable sous la forme '#define <nom de la constante><valeur>'. L'avantage de cette approche est l'économie d'espace mémoire. En effet, les variables globales déclarées de cette façon sont gérées par le préprocesseur qui remplace les occurences de cette variable dans le code avant la compilation et ne consomme pas d'espace en mémoire. Des problèmes de compatibilité entre types pourraient apparaître, certes, mais nous n'utilisons les constante globales que comme types simples comme 'int' ou 'str', tout en faisant attention à leur contexte d'utilisation, nous permettant ainsi d'éviter ces erreurs.

4.2 File d'attente

Comme expliqué dans les consignes, le serveur doit être muni d'une file d'attente pour les clients qui se connectent lorsque le serveur est plein. Pour cela nous avons utilisé un atomic pour calculer le nombre de personnes dans la file d'attente. Ceux-ci, nous permettent de compter les clients dans la boucle d'attente sans les compter plusieurs fois. De plus, cela nous

permet de facilement reconnecter un cleint en attente a la place d'un autre client qui vient de se déconnecter.

5 Conclusion

Ce second et dernier projet de ce cours d'OS nous a permis d'approfondir nos connaissances concernant les threads, les signaux et le bash déjà utilisés dans le cadre du premier. A ces concepts s'ajoutent des nouveaux comme les mutexs et les sockets et leurs fonctions relatives comme 'listen', 'bind' et autres, également étudiés en cours. De plus, notre groupe étant resté le même pour la réalisation de chacun de ces deux projets, nous avons pu réitérer l'expérience de travail ensemble, de répartition des tâches et tout autre chose liée de près ou de loin aux travaux à plusieurs en nous basant sur notre modeste expérience lors du premier projet. Ceci nous a permis d'améliorer certains aspects, comme, notablement, la répartition de travail qui s'est faite de manière plus claire et plus efficace qu'à la première itération.
En combinant les travaux pratiques, les cours théoriques et les deux projets de ce cours de systèmes d'exploitation, chaque membre de notre groupe a pu progresser tant sur le plan personnel que professionnel. D'une part en apprenant de nombreux concepts liés aux OS, les appliquant et en les maitrisant et, d'autre part, en découvrant les divers aspects inhérents aux projets en groupes et en les intégrant pour mieux s'adapter dans le monde professionnel à l'avenir!