Système de communication client-serveur

Malek Slokom

1 Introduction

Les signaux constituent un moyen de communication entre processus, mais ils ne permettent pas d'échanger des données. Pour réaliser de telles échanges, il faut utiliser des fichiers. Deux processus UNIX peuvent communiquer par un fichier spécial appelé tube en terminologie UNIX (pipe). Un processus y met des données et un autre les prend. Il y a deux types de tubes: Les tubes ordinaires et les tubes nommés. Dans notre projet, nous allons nous intéresser aux tubes nommés. Ces tubes permettent de connecter des processus quelconques, sans lien de parenté particulier, ce qui est très utile quand on imagine un processus serveur permanent qui offre des services à tout processus demandeur.

2 Problématique

Dans notre projet, nous souhaitons réaliser un système de communication client/serveur qui utilise les pipes pour ordonnancer les requêtes. Plusieurs clients peuvent être en attente de réponse dans le même tube, donc il est nécessaire de définir un protocole assurant que chaque client reçoit les réponses qui lui sont destinées.

3 Solution

Pour résoudre ce problème, nous allons utiliser deux structures de données:

- Une structure "*Question*": Utilisé par le client dans laquelle il met son pid et un nombre aléatoire compris entre 1 et NMAX.
- Une structure "*Réponse*": Utilisé par le serveur dans laquelle il met son pid et un tableau rempli par n (le nombre envoyé par le client) nombres aléatoires.

```
struct question {
    int pid_client ;
    int question ;
};

struct response {
    int pid_server;
    int response[NMAX];
};
```

Figure 1: Les structures

De plus, pour synchroniser entre les processus, nous allons utilisé des signaux comme:

- "*SIGUSR1*": le serveur peut réveiller le client par l'intermédiaire du ce signal quand il a écrit la réponse. Et le client avertit par ce même signal le serveur quand il a lu les réponses.
- Tous les autres signaux auront pour résultat la terminaison du serveur.

pour assurer la communication entre les deux processus nous allons utiliser deux tubes nommé:

- "*Fifo 1*": dans lequel le client met ses questions.
- "*Fifo 2*": dans lequel le serveur met les réponses.

La figure ci-dessous est une schématisation de la solution proposée.

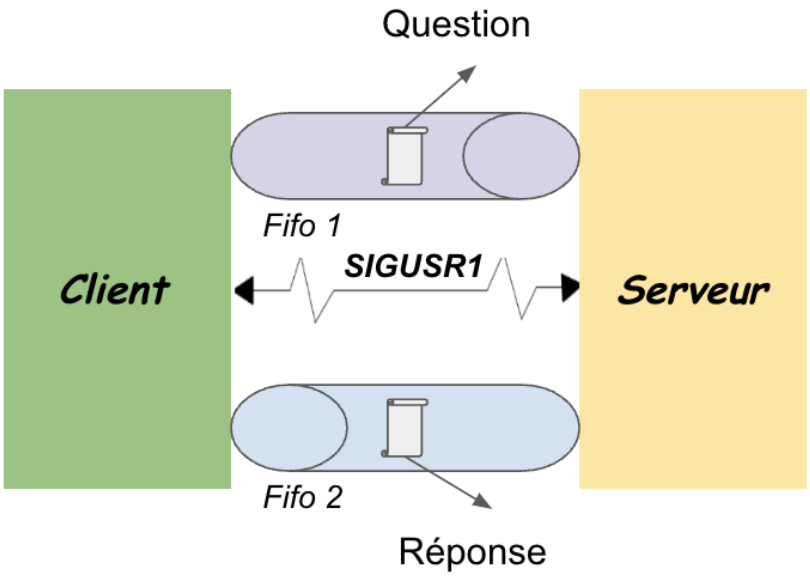

Figure 2: Solution proposée

4 Environnement logiciel

- **Visual Studio Code**:
  Visual Studio Code est un éditeur de code open-source créé par Microsoft, qui prend en charge un grand nombre de langues grâce à des extensions fonctionnant sous Windows, Linux et mac OS. Il prend en charge des fonctionnalités telles que la complétion automatique, la coloration syntaxique, le débogage et les commandes git.

Figure 3: Visual Studio Code logo

5 Technologies utilisées

- **MacOS**:
  macOS est un système d'exploitation partiellement propriétaire développé et commercialisé par Apple depuis 1998.MacOS est basé sur le système d'exploitation UNIX plus précisément BSD Unix. Il a été développé en utilisant C, C ++, Objective-C, le langage d'assemblage et Swift. Il s'agit du deuxième système d'exploitation le plus utilisé dans les ordinateurs personnels après Windows.

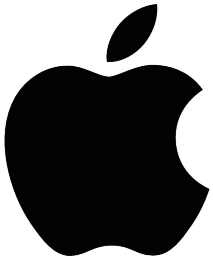

Figure 4: MacOS logo

- **C Langage**:
  Le C est un langage de programmation procédural. Il a été initialement développé par "Dennis Ritchie" comme un langage de programmation système pour écrire des systèmes d'exploitation. Les principales caractéristiques du langage C sont l'accès de bas niveau à la mémoire, un ensemble simple de mots-clés et un style épuré. Ces caractéristiques font du langage C un langage adapté à la programmation système comme le développement de systèmes d'exploitation ou de compilateurs.

Figure 5: C Langage logo

6 Réalisation

6.1 Structure du projet

Notre programme est développé en utilisant le langage de programmation C. La figure 6 représente la structure initiale de notre projet.

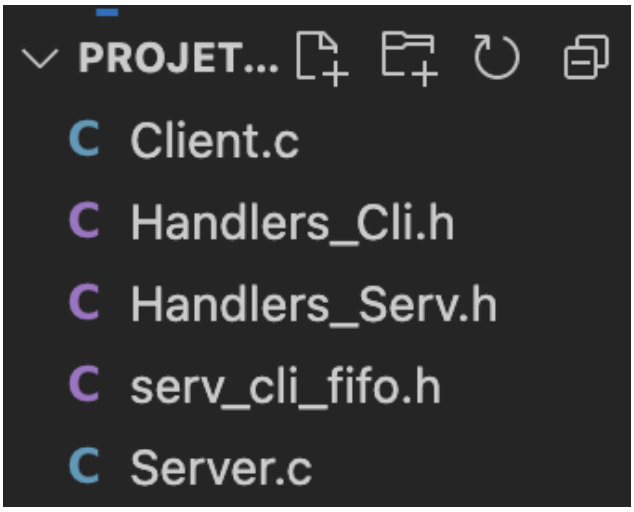

Figure 6: Structure du projet initiale

Notre répertoire contient 5 fichier:

- Trois fichier d'extension ".h" contenant les entêtes.
  - **Handlers_Cli.h**: Contient l'entête du handler "réveil client" qui réveille le client suite à la réception du signal SIGUSR1.
  - **Handlers_Serv.h**: Contient lest entêtes de deux handlers: "réveil serveur" qui réveille le serveur suite a la réception du signal SIGUSR1, et "fin serveur" qui termine le serveur suite a la réception d'un signal quelconque
  - **serv_cli_fifo.h**: Contient les entêtes des fichiers systèmes nécessaires, les constantes et les "macros" communes aux clients, aux serveurs et les structures des données pour représenter une question et une réponse.
- Deux fichier d'extension ".c" contenant le code.
  - **Client.c**: Contient l'implémentation des fonctionnalités décrites dans Handlers_Cli.h. Il contient aussi le code pour qu'il puisse communiquer avec le serveur à travers les pipes.
  - **Server.c**: Contient l'implémentation des fonctionnalités décrites dans Handlers_Serv.h. Il contient aussi le code pour la création des pipes et la communication avec le client.

Pour compiler notre projet, nous devons exécuter les deux commandes suivantes:

```
(base) malekslokom@Maleks-MacBook-Pro Projet_Unix % gcc -o ServerS  Server.c
(base) malekslokom@Maleks-MacBook-Pro Projet_Unix % gcc -o ClientS  Client.c
(base) malekslokom@Maleks-MacBook-Pro Projet_Unix %
```

Figure 7: Compilation des fichiers

Suite à l'exécution de ces deux commandes, nous allons avoir comme résultat deux fichier exécutables:

- **ClientS**
- **ServerS**

Enfin, pour lancer notre projet, nous devons utiliser au moins deux terminaux différents: un pour le serveur et d'autres pour les clients. Suite a l'exécution du serveur par l'utilisation du commande "**./ServerS**" nous allons avoir deux nouveaux fichiers, correspondants aux pipes, dans notre répertoire sous les noms:

- **fifo1**
- **fifo2**

Et pour le client nous devons utiliser la commande "**./ClientS**".

La figure 8 représente la structure finale de notre projet.

Figure 8: Structure du projet finale

6.2 Jeu d'essai

Tout d'abord, nous devons exécuter le serveur sinon nous allons avoir une erreur puisque les pipes n'ont pas encore étés crées comme l'illustre la figure 6.2.

```
(base) malekslokom@Maleks-MacBook-Pro Projet_Unix % ./ClientS
¤¤¤¤¤¤¤¤¤¤¤¤¤  Client-Serveur  ¤¤¤¤¤¤¤¤¤¤¤¤¤

¤ Coté Client (pid= 11552)
¤¤¤¤¤¤¤¤¤¤ Message ¤¤¤¤¤¤¤¤¤¤
          PID= 11552
          Nombre= 5
¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

-- Probléme d'écriture
: Bad file descriptor
(base) malekslokom@Maleks-MacBook-Pro Projet_Unix %
```

Après l'exécution du serveur, il crée les pipes "fifo1" et "fifo2" et reste en attente des messages des clients. dés la première requête reçue sur le tube "fifo1", le serveur sera réveillé et il attaque la phase d'exécution des requêtes en suivant la stratégie "First In First Out".

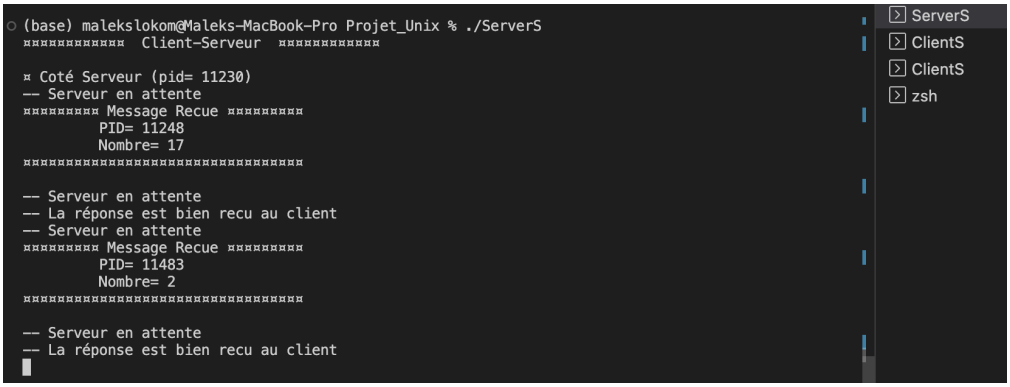

Figure 9: Serveur

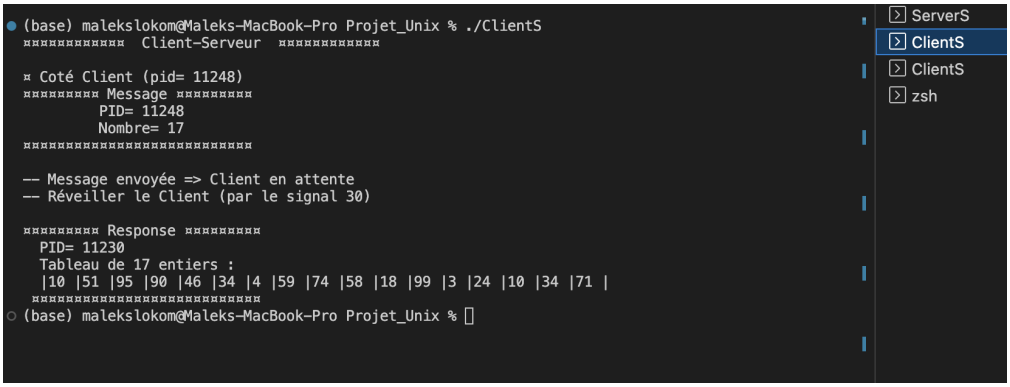

Figure 10: Client 1

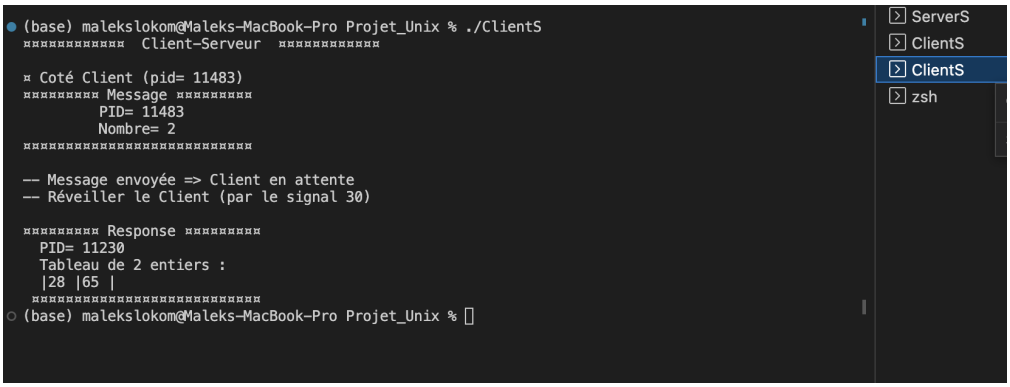

Figure 11: Client 2

Le processus serveur est terminé lorsqu'il reçoit tout signal qui n'est pas un signal "SIGUSR1" et les deux pipes sont automatiquement supprimés.