Rendu Projet Open Source

Lanceur de commande

Killian MOUTINARD

Alexandre MOALA

Mathursan MAHENDRARASAN

Samir RAHMOUNI

Lorie NOWICKI

Table des matières

Fonctionnalité du projet

Ce projet vise à créer un lanceur de commandes en langage C, sous la forme d'un démon qui répond aux requêtes émises par des clients. Voici comment le projet sera organisé :

Démon

- Un processus démon sera en écoute des demandes de connexion des clients via un tube nommée. Cela signifie qu'il restera actif en permanence pour accepter les connexions des clients.
- Lorsqu'un client souhaite se connecter, il envoie une demande de connexion via ce tube nommé vers le démon.

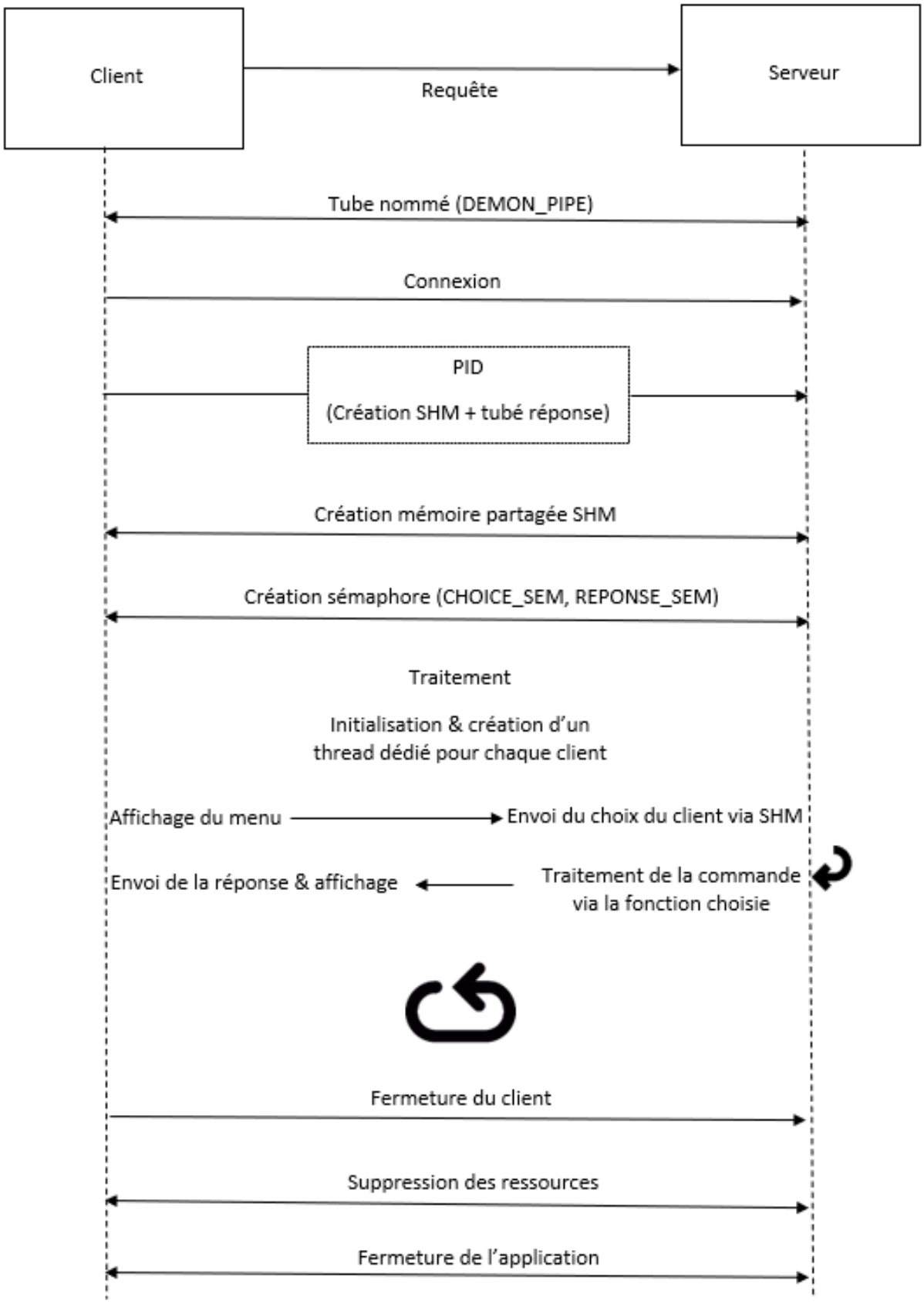
Client

- Un programme client sera développé. Ce client utilisera le tube nommé pour demander une connexion au démon.
- Une fois la connexion établie, le client enverra ses requêtes au démon pour exécution.

Threads

- Le démon utilisera des threads pour gérer les requêtes des clients de manière concurrente. Chaque fois qu'un client envoie une requête, le démon lancera un nouveau thread pour prendre en charge cette requête spécifique.
- Ces threads auront accès à une zone de mémoire partagée (SHM). La SHM est une région de mémoire qui peut être partagée entre plusieurs processus. Elle sera utilisée pour transmettre les données entre les clients et le démon de manière efficace.
- Une fois qu'un thread a traité une requête, il retournera les résultats dans la zone de mémoire partagée, où le client pourra les récupérer.

En résumé, le démon sera responsable de l'acceptation des connexions des clients, de la création de threads pour gérer les requêtes, et de la communication des résultats via une zone de mémoire partagée. Les clients, quant à eux, enverront leurs requêtes au démon via un tube nommée, puis récupéreront les résultats via la SHM.


Client
Requête
Serveur
Tube nommé (DEMON_PIPE)
Connexion
PID
(Création SHM + tubé réponse)
Création mémoire partagée SHM
Création sémaphore (CHOICE_SEM, REPONSE_SEM)
Traitement
Initialisation & création d'un
thread dédié pour chaque client
Affichage du menu
Envoi du choix du client via SHM
Envoi de la réponse & affichage
Traitement de la commande
via la fonction choisie
Fermeture du client
Suppression des ressources
Fermeture de l'application

Manuel technique

Explication détaillée du code client

Le code ‘client.c’ est un programme écrit en langage C. Son but est d’agir comme un client pour interagir avec un serveur distant via des mémoires partagées et des tubes nommées.

Fonctionnalités principales

- Lecture de la configuration à partir du fichier ‘demon.conf’.
- Envoie d’une demande de connexion au serveur via un tube nommée.
- Communication avec le serveur via une mémoire partagée pour échanger des données.
- Gestion d’un menu interactif pour l’utilisateur

Structure du code

Inclusion de bibliothèques : Le code inclut les bibliothèques standard nécessaires pour les opérations d’entrée / sortie, la gestion des threads, la gestion des mémoires partagées, etc.

Définition des constantes : Des constantes telles que le nom du fichier de configuration (‘**CONFIG_FILE’**) et le nom tube nommée (‘**PIPE_NAME’**) sont définies.

Définition de structures : La structure ‘**demon_config’** est utilisée pour stocker la configuration du démon.

Fonctions :

- ‘**read_config()**’ : Cette fonction lit les paramètres à partir d’un fichier (‘**demon.conf’)** et les stocke dans une structure ‘**demon_config’**, qui contient des informations telles que le nombre minimum de threads, le nombre maximum de threads, le nombre maximum de connexions par thread et la taille de la mémoire partagée.
- ‘**request_connection()**’ : Cette fonction gère l'envoi d'une demande de connexion au serveur via un tube FIFO et récupère le nom de la mémoire partagée associée. Elle ouvre un tube FIFO pour écrire la demande de connexion, puis attend de lire le nom de la mémoire partagée à partir d'un tube FIFO distinct après avoir envoyé la demande. En cas d'erreur lors de l'ouverture des tubes FIFO ou de la lecture du nom de la mémoire partagée, la fonction affiche un message d'erreur et quitte le programme.
- ‘**printMenu()**’ : Cette fonction affiche un menu permettant à l’utilisateur de choisir parmi plusieurs options, telles qu’afficher la date, afficher l’heure, afficher l’identifiant de l’utilisateur ainsi que ses groupes, afficher le répertoire de travail actuel, fermer le thread actuel côté serveur ou fermer le client.
- ‘**write_choice_to_shm (int choice)’** : Cette fonction prend en paramètre le choix de l’utilisateur, ouvre la mémoire partagée en utilisant le nom stocké dans la variable globale, écrit le choix dans la mémoire partagée, puis signale (poste) le sémaphore pour indiquer au serveur que le choix est prêt à être lu.

- '**communicate_with_server()**' : Cette fonction communique avec le serveur en lisant le contenu de la mémoire partagée après que le serveur y a écrit une réponse. Elle ouvre la mémoire partagée en utilisant le nom stocké dans la variable globale, lit le contenu de la mémoire partagée, puis affiche la réponse du serveur.

Fonction 'main()' :

- Lit la configuration
- Envoie une demande de connexion au serveur
- Affiche un menu utilisateur et envoie le choix au serveur via la mémoire partagée
- Attend la réponse du serveur et l'affiche
- Répète le processus jusqu'à ce que l'utilisateur choisisse de quitter

Utilisation des mécanismes IPC

- **Tubes nommés** : Utilisés par la communication entre le client et le serveur
- **Mémoires partagées** : Utilisées pour échanger des données entre le client et le serveur de manière efficace

Gestion des erreurs

Le code effectue une gestion appropriée des erreurs en utilisant les fonctions '**perror()'** pour afficher les messages d'erreur et '**exit(EXIT_FAILURE)'** pour quitter le programme en cas d'erreur critique.

Conclusion

Le code **'client.c'** est un programme client qui communique avec un serveur distant via des mémoires partagées et des tubes nommés. Il est conçu pour être robuste et offre une gestion adéquate des erreurs.

Explication détaillée du code serveur

Le fichier 'serveur.c' est un programme écrit en langage C. Son objectif est de fonctionner comme un serveur qui reçoit les demandes de connexion des clients, traite leurs requêtes, puis retourne les résultats. Le serveur utilise des mémoires partagées pour communiquer avec les clients.

Fonctionnalités principales

- Lecture de la configuration à partir du fichier **'demon.conf'**
- Attente de demandes de connexion des clients via un tube nommé
- Traitement des requêtes des clients en utilisant des threads
- Communication avec les clients via des mémoires partagées
- Implémentation de différentes fonctionnalités comme l'affichage de la date, de l'heure, de l'ID de l'utilisateur, et du répertoire de travail actuel

Structure du code

Inclusions de bibliothèques : Le code inclut diverses bibliothèques standard nécessaires pour les opérations d'entrée / sortie, la gestion des threads, la gestion des mémoires partagées, etc.

Définition des constantes : Des constantes telles que le nom du fichier de configuration ('**CONFIG_FILE'**) et le nom du tube nommé ('**PIPE_NAME'**) sont définies.

Définition de structures : La structure '**demon_config'** est utilisée pour stockée la configuration du démon.

Fonctions :

- '**read_config()**' : Cette fonction lit les paramètres à partir d'un fichier ('**demon.conf')** et les stocke dans une structure '**demon_config'**, qui contient des informations telles que le nombre minimum de threads, le nombre maximum de threads, le nombre maximum de connexions par thread et la taille de la mémoire partagée. '**getCurrentDate()**' : Renvoie la date actuelle au format texte.
- '**sigint_handler(int sig)'** : Cette fonction gère le signal SIGINT, modifiant la variable '**keep_running'** pour arrêter proprement le programme lorsqu'une interruption de type CTRL+C est reçue. Elle est utilisée pour définir le comportement en cas d'interruption système, assurant une terminaison correcte du programme.
- '**getCurrentDate'** : Cette fonction obtient la date actuelle au format « jour-mois-année » et renvoie une chaîne de caractères représentant la date.
- '**getCurrentTime()**' : Cette fonction obtient l'heure actuelle au format « heure:minute:seconde » et renvoie une chaîne de caractères représentant l'heure.
- '**printUserIdInfo()'** : Cette fonction obtient les informations sur l'utilisateur actuel, y compris l'UID, le nom de l'utilisateur, le GID, le nom du groupe de l'utilisateur et la liste des groupes auxquels l'utilisateur appartient. Ces informations sont renvoyées sous forme de chaîne de caractères.
- '**getCurrentWorkingDirectory()**' : Cette fonction obtient le répertoire de travail actuel (directory) et renvoie une chaîne de caractères représentant le chemin absolu du répertoire (résultat de pwd).

- '**handle_client(void* arg)'** : Cette fonction est exécutée par chaque thread du serveur pour traiter les demandes des clients. Elle prend en paramètre l'identifiant du client (PID) et effectue les opérations suivantes :
  - Crée ou ouvre une mémoire partagée pour communiquer avec le client
  - Envoie le nom de la mémoire partagée au client via un tube nommé
  - Attends que le client écrive son choix dans la mémoire partagée et le lit
  - Traite la demande du client en fonction de son choix et stocke le résultat dans la mémoire partagée
  - Libère les ressources et termine le traitement du client.
- '**startdemon()**' : Cette fonction est responsable de démarrer le serveur. Elle lit d'abord la configuration à partir du fichier de configuration, puis crée un tube nommé (demon_pipe) pour accepter les demandes de connexion des clients. Ensuite, elle entre dans une boucle qui s'exécute tant qu'un signal CTRL+C (SIGINT) n'est pas reçu, cela est géré par la variable '**keep_running**' où elle accepte les demandes de connexion des clients, crée un thread pour chaque client connecté et démarre le traitement.

Fonction 'main()' :

- Lit la configuration
- Affiche la configuration du serveur
- Lance la fonction '**startdemon()**' pour commencer à écouter les demandes de connexion

Utilisation des mécanismes IPC

Tubes nommés : Utilisées pour la communication entre les clients et le serveur.

Mémoires partagées : Utilisées pour échanger des données entre les clients et le serveur de manière efficace.

Gestion des erreurs

Le code effectue une gestion adéquate des erreurs en utilisant les fonctions '**perror()'** pour afficher les messages d'erreur et en sortant du programme en cas d'erreur critique.

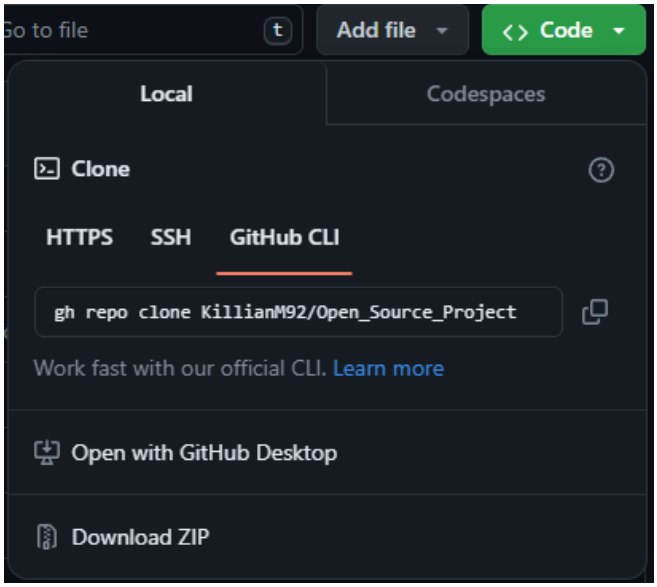
Conclusion

Le code '**serveur.c'** est un serveur qui traite les requêtes des clients de manière concurrente en utilisant des threads et des mémoires partagées pour la communication. Il est conçu pour être robuste et offre une gestion appropriée des erreurs.

Tutoriel utilisateur

Afin de télécharger notre code, il suffit de vous rendre sur notre repository sur GitHub à l'adresse suivante : https://github.com/KillianM92/Open_Source_Project

Vous avez ensuite le choix de cloner notre projet sur votre poste dans le répertoire que vous souhaiterez ou alors de télécharger le fichier compressé de notre avec « Download ZIP ».



Notre repository comprend un dossier « TEST » qui est le dossier de développement ou les versions ont été échangés entre les membres du groupe afin de gérer le développement et le dossier racine qui contient tous les fichiers utiles au lancement de notre application.

Une fois que le code a été téléchargé et placé dans votre machine Linux (exemple : Kali), il faut vous rendre dans ce dossier et décompresser l'archive (si besoin) mais vous devriez avoir ces fichiers (au minimum) :

```
┌──(kali㉿kali)-[~/…/Open_Source_Project-main/TEST/Killian/V5]
└─$ ls
client.c  demon.conf  makefile  serveur.c
```

Une fois que vous avez ces 4 fichiers, il faut effectuer la compilation à l'aide du makefile et taper « make » :

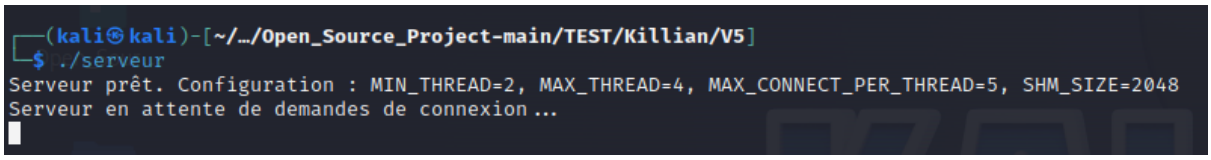
```
┌──(kali㉿kali)-[~/…/Open_Source_Project-main/TEST/Killian/V5]
└─$ make
gcc serveur.c -o serveur -pthread
gcc client.c -o client

┌──(kali㉿kali)-[~/…/Open_Source_Project-main/TEST/Killian/V5]
└─$ ls
client  client.c  demon.conf  makefile  serveur  serveur.c
```

Les fichiers vont se compiler et 2 exécutables « client » et « serveur » vont être ajoutés au dossier.

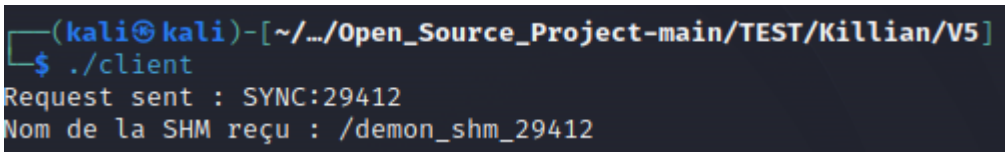
Pour la suite de ce tutoriel, je vous invite à ouvrir un deuxième terminal et de se placer dans le même dossier que précédemment.

La prochaine étape va constituer à lancer le serveur avec la commande « ./serveur » :

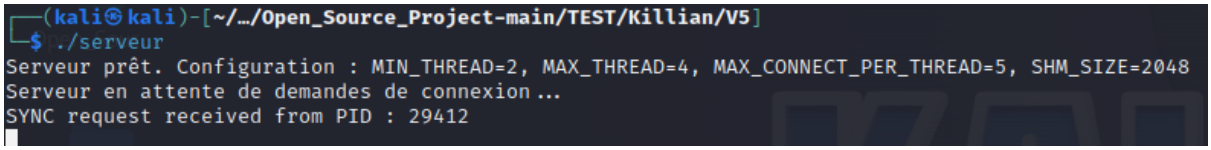


On voit bien ici que le serveur est prêt, il a chargé sa configuration et est en attente de connexions via le tube « demon_pipe ».

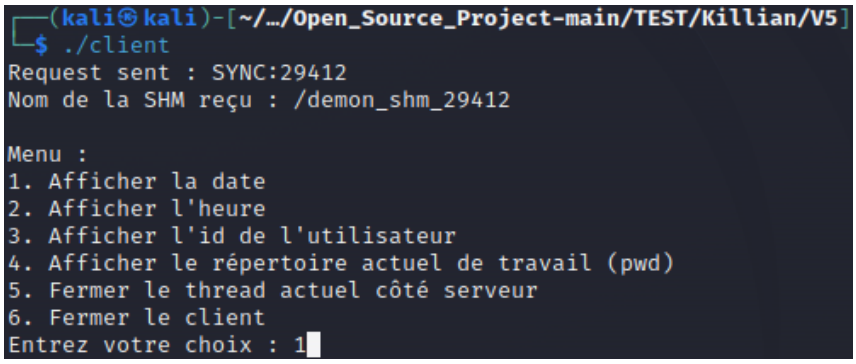
Nous allons maintenant lancer le client à l'aide de la commande « ./client » dans le second terminal :



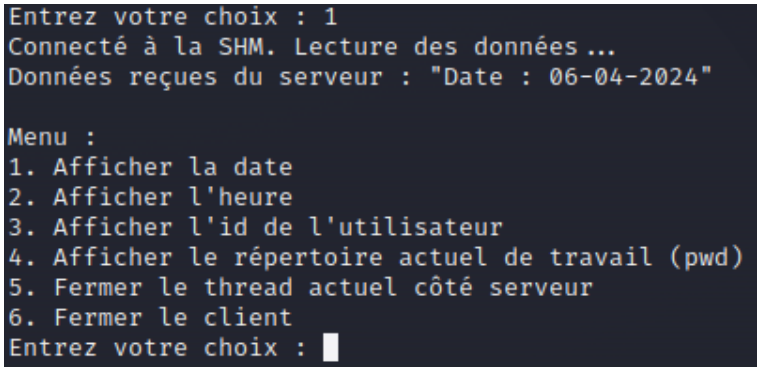
Lors du lancement du client, ce dernier envoie une requête de type SYNC avec son PID via le tube demon_pipe puis reçoit en retour le nom de la SHM qui a été créé par le serveur. De son côté le serveur a démarré son thread dédié à la suite de la réception de la requête du serveur :



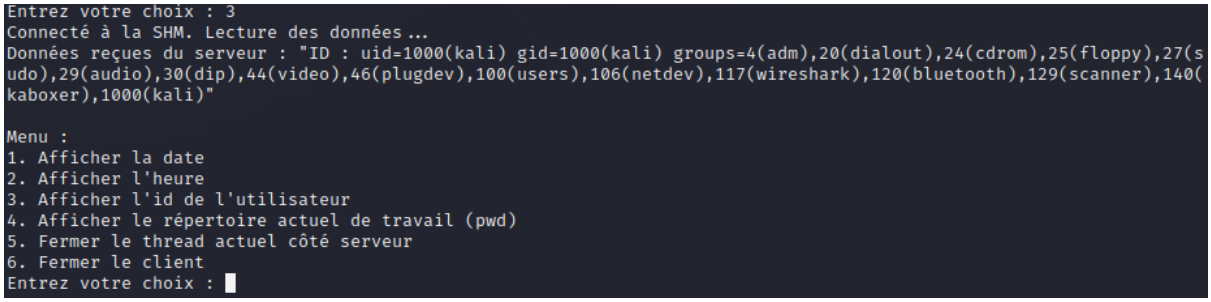
Côté client, instantanément après la réception de toutes les ressources nécessaires (tubes, SHM, sémaphores), le menu va s'afficher :



Si vous voulez utiliser une de nos 4 commandes il faudra choisir un chiffre entre 1 et 4, si vous voulez fermer le thread, il faudra taper « 5 » et si vous voulez fermer le client et nettoyer les ressources utilisées il faudra taper « 6 ». Ici, pour le test, on décide de taper « 1 » :



Le résultat est reçu par le client à la suite de l'appel de la fonction dédié côté serveur et le menu s'affiche encore une fois (tant que le choix est compris entre 1 et 5). On décide donc d'afficher l'id de l'utilisateur et ses groupes associés avec « 3 » :



Nous pouvons effectuer des demandes de traitements des commandes via le menu tant que nous choisissons des chiffres entre « 1 » et « 4 », le chiffre « 5 » supprimera lui la SHM et le thread créé donc va couper toute communication établie auparavant entre le client et le serveur. « 6 » reste le dernier choix afin de fermer le client et de nettoyer les ressources associées.

Exemple de fin de session :

Lors de l'échange entre client et serveur, 2 sémaphores, 1 SHM et 2 tubes sont utilisés. Le tube de réponse est supprimé dès lorsqu'il a fourni l'information de la SHM au client. On voit ici dans le dossier « /dev/shm » les 2 sémaphores et la SHM :

```
┌──(kali㉿kali)-[/dev/shm]
└─$ ll
total 12
-rw-r--r-- 1 kali kali 2048  6 avril 00:58 demon_shm_29412
-rw-r--r-- 1 kali kali   32  6 avril 00:56 sem.choice_sem_demon
-rw-r--r-- 1 kali kali   32  6 avril 00:56 sem.reponse_sem_demon
```

Nous allons tenter de fermer le thread actuel via l'envoi de la commande « 5 » :

```
Entrez votre choix : 5
Connecté à la SHM. Lecture des données...
Données reçues du serveur : "Session terminée."
```

Côté dossier « /dev/shm », voici le résultat :

```
┌──(kali㉿kali)-[/dev/shm]
└─$ ll
total 8
-rw-r--r-- 1 kali kali 32  6 avril 00:56 sem.choice_sem_demon
-rw-r--r-- 1 kali kali 32  6 avril 00:56 sem.reponse_sem_demon
```

La SHM a bien été supprimée.

Maintenant, nous allons fermer le client via l'envoi de la commande « 6 » :

```
Menu :
1. Afficher la date
2. Afficher l'heure
3. Afficher l'id de l'utilisateur
4. Afficher le répertoire actuel de travail (pwd)
5. Fermer le thread actuel côté serveur
6. Fermer le client
Entrez votre choix : 6

┌──(kali㉿kali)-[~/…/Open_Source_Project-main/TEST/Killian/V5]
└─$
```

On voit bien ici que le client s'est terminé correctement sans erreur, allons vérifier du côté du dossier « /dev/shm » pour vérifier que les sémaphores de synchronisation utilisés ont bien été supprimés :

```
┌──(kali㉿kali)-[/dev/shm]
└─$ ll
total 0
```

Il nous reste encore le serveur actif avec son « demon_pipe » qui est en attente constante de connexions de la part de clients tant qu'un signal CTRL+C n'est pas envoyé. Ce tube est situé dans le dossier « /tmp » :

```
┌──(kali㉿kali)-[/tmp]
└─$ ls
demon_pipe
ssh-XXXXXXDsZw9m
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-colord.service-rMCmZ2
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-haveged.service-KT6D5c
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-ModemManager.service-ogji0P
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-systemd-logind.service-tgcbuf
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-systemd-timesyncd.service-mdLT9t
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-upower.service-xLWydf
VMwareDnD
vmware-root_501-2117483949
```

Nous allons donc lui envoyer ce signal :

```
┌──(kali㉿kali)-[~/…/Open_Source_Project-main/TEST/Killian/V5]
└─$ ./serveur
Serveur prêt. Configuration : MIN_THREAD=2, MAX_THREAD=4, MAX_CONNECT_PER_THREAD=5, SHM_SIZE=2048
Serveur en attente de demandes de connexion ...
SYNC request received from PID : 29412
Traitement du client (PID: 29412) terminé.
^C
┌──(kali㉿kali)-[~/…/Open_Source_Project-main/TEST/Killian/V5]
└─$
```

On voit ici par ailleurs que lors de l'envoi de la commande « 5 » tout à l'heure le message « Traitement du client (PID : 29412) terminé. » a été envoyé par le serveur. CTRL+C envoyé = coupure du serveur et de l'exécutable.

Après fermeture du serveur, le tube « demon_pipe » est supprimé comme on peut le voir ici :

```
┌──(kali㉿kali)-[/tmp]
└─$ ls
ssh-XXXXXXDsZw9m
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-colord.service-rMCmZ2
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-haveged.service-KT6D5c
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-ModemManager.service-ogji0P
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-systemd-logind.service-tgcbuf
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-systemd-timesyncd.service-mdLT9t
systemd-private-fb48c0572f084cc5b4d3a6f5845b0de5-upower.service-xLWydf
VMwareDnD
vmware-root_501-2117483949
┌──(kali㉿kali)-[/tmp]
└─$
```

En résumé, si vous suivez ces étapes dans le bon ordre, vous ne devriez pas garder de fichiers parasites sur votre poste qui ont été utilisés pendant l'utilisation de notre application.

Lien GitHub

https://github.com/KillianM92/Open_Source_Project