Rapport de projet

Introduction aux Systèmes d'Exploitation

Lanceur de commandes

LEROY Florent et SALLES Théo

Rapport rédigé pour
Université de Rouen Normandie

Informatique

Département d'informatique

Décembre 2023

Table des matières

1 Introduction

L'objectif de ce projet est de créer un lanceur de commandes dont les commandes à exécuter sont passés à l'aide d'un client. De plus, le client permet à l'utilisateur d'interagir directement avec les programmes exécutés. Le tout prenant en compte le fait que plusieurs clients peuvent envoyer ces commandes simultanément.

Ce projet se divise donc en 3 parties principales :

— La file synchronisée, qui doit servir de stockage et de traitement des demandes d'exécutions commun aux clients et au lanceur de commandes, tout en gérant de manière synchrone l'enfilement et le défilement de données, à l'aide de sémaphores ;

— Le client, qui doit traiter les demandes de l'utilisateur, créer les tubes servant d'entrées et de sorties standards à la commande et enfiler la demande d'exécution de commande dans la file ;

— Le lanceur de commandes en lui-même, qui, à partir d'une demande de commande issu d'une file, modifie ses entrées et sorties standards pour correspondre aux tubes cités ci-dessus et exécuter la commande dans un nouveau thread.

2 Manuel d'utilisation

Pour utiliser le code réalisé dans ce projet correctement, les étapes suivantes doivent être suivies rigoureusement et dans l'ordre.

Tout comportement non prévu issu du mauvais suivi de ces instructions ne sont pas de notre responsabilité. De plus, les conséquences issus de ces comportement sont entièrement de la faute de l'utilisateur.

1. Récupérer toutes les dépendances de ce projet (`gcc`, `make`)
2. Compiler les deux parties du code
   (a) Pour compiler le lanceur, exécuter la commande suivante :
   ```
   make lanceur
   ```
   (b) Pour compiler le client, exécuter la commande suivante :
   ```
   make client
   ```
3. Exécuter le lanceur avec la commande suivante :
   ```
   ./lanceur
   ```

4. Envoyer une commande au lanceur à partir du lanceur avec la commande suivante (remplacer les "[**CMDX**]" par les commandes à exécuter) :

   ```
   ./client [CMD1] | ... | [CMDN]
   ```

   Il est bon de remarqur que le client affichera la syntaxe attendue si exécuté avec aucun argument.

5. Pour fermer le lanceur, deux possibilités :
   — Récupérer le PID du lanceur avec la commande `ps` puis le tuer avec la commande `kill [PID]`
   — Tuer toutes les instances du lanceur avec la commande `killall lanceur`

3 File synchronisée

Notre implémentation des files synchronisées est utilisable à partir d'une bibliothèque `file_sync`.

Son fonctionnement se base sur l'utilisation d'un mutex pour éviter l'édition de la pile par un processus alors que celle-ci est activement utilisée par d'autres processus. Nous pouvons aussi remarquer que l'implémentation utilise l'algorithme solvant le problème du producteur/consommateur, c'est-à-dire que tout défilement sur une pile vide ou enfilement dans une file pleine provoquera un blocage, à l'aide d'une paire de sémaphores.

De plus, pour permettre à chaque acteur (clients, lanceur, ...) d'accéder à la file, et cela dans n'importe quelle situation (par exemple, en imaginant que chaque acteur possède un répertoire actif différent) nous devons la placer dans un segment de mémoire partagée, ce qui insinue que notre bibliothèque doit permettre à l'utilisateur de gérer (créer, détruire) ces segments de mémoire partagée.

Enfin, dans notre implémentation actuelle, nous avons codé la file de manière à ce qu'elle puisse stocker des identifiants de processus (codée en C sous forme de `pid_t`, type synonyme au type `signed int` sous Linux).

La bibliothèque `file_sync` offre l'accès aux fonctions suivantes :

3.1 `create_file_sync`

```
int create_file_sync(void)
```

Cette fonction crée une file synchronisée et de la stocker dans une zone mémoire partagée. Cette zone mémoire sera nommée selon la macroconstante `NOM_SHM`. Renvoie `-1` en cas d'erreur, `0` sinon. Les erreurs peuvent être causées par :

- L'existence d'un segment de mémoire partagée ayant pour nom `NOM_SHM` (il est à noter que ce cas de figure peut se produire si, lors d'une utilisation précédente de la bibliothèque, le segment n'a pas été supprimée correctement) ;
- Un manque d'espace suffisant pour créer le segment ;
- Une erreur lors de la création des sémaphores.

3.2 `destroy_file`

```
int destroy_file(void)
```

Cette fonction supprime la file précédemment crée portant le nom `NOM_SHM` et libère les ressources associées. Renvoie `-1` en cas d'erreur, `0` sinon. Les erreurs peuvent être causées par :

- L'absence d'un segment de mémoire partagée ayant pour nom `NOM_SHM` (il est à noter que ce cas de figure peut se produire si, lors d'une utilisation précédente de la bibliothèque, le segment n'a pas été créé correctement) ;
- Une erreur lors de la destruction des sémaphores.

3.3 `enfiler`

```
int enfiler(pid_t donnee)
```

Cette fonction enfile un identifiant de processus `donnee` dans la file synchronisée. Si la file est pleine et/ou en cours d'utilisation, l'appel sera bloquant jusqu'à ce que la file soit défilée au moins une fois et/ou que la file soit libre/inutilisée. Renvoie `-1` en cas d'erreur, `0` sinon. Les erreurs peuvent être causées par :

- L'absence d'un segment de mémoire partagée ayant pour nom `NOM_SHM` (il est à noter que ce cas de figure peut se produire si, lors d'une utilisation précédente de la bibliothèque, le segment n'a pas été créé correctement) ;
- Une erreur lors de l'édition d'un des sémaphores.

3.4 `defiler`

`pid_t defiler(void)`

Cette fonction défile depuis la file synchronisée un identifiant de processus et la renvoie. Si la file est vide et/ou en cours d'utilisation, l'appel sera bloquant jusqu'à ce que la file soit enfilée au moins une fois et/ou que la file soit libre/inutilisée. Renvoie `(pid_t) -1` en cas d'erreur, l'identifiant défilé sinon. Les erreurs peuvent être causées par :

- L'absence d'un segment de mémoire partagée ayant pour nom `NOM_SHM` (il est à noter que ce cas de figure peut se produire si, lors d'une utilisation précédente de la bibliothèque, le segment n'a pas été créé correctement) ;
- Une erreur lors de l'édition d'un des sémaphores.

4 Parseur

Le module parseur est une réutilisation du module `analyse` donnée lors du TP 2 qui contient ces deux fonctions.

- `char **parseur_arg(const char arg[])` : Prend une chaîne de caractère `arg` et la découpe en chaîne de caractère distincts selon les espaces.
- `void dispose_arg(char *argv[])` : Libère les ressources allouées pour le tableau retourné par `parseur_arg`.

Les fonctions n'ont uniquement subi comme modification des test rajoutés sur les `malloc`.

5 Lanceur de commande

Le nom des tubes est défini par une norme choisie commune aux clients et au lanceur. Cette norme veut que le nom d'un tube soit une chaîne de caractère qui définit l'utilité du tube suivit du PID du client afin que chaque client possède des tube qui lui soient propres. Tout les tubes sont créés par le client.

5.1 Macroconstantes

— `TUBE_CL` : préfixe du tube dans lequel le client écrit les informations pour le lanceur.
— `TUBE_RES` : préfixe du tube dans lequel le résultat de la commande sera écrit et envoyé au client.
— `TUBE_ERR` : préfixe du tube dans lequel une possible erreur lors de l'exécution de la commande sera écrit et envoyé au client.
— `BUF_SIZE` : Taille maximale en nombre de caractère des commandes avec leurs options.
— `CMD_SIZE` : Taille maximale en nombre de du nom de la commande (taille déterminer à partir du nombre de caractère de la plus grande commande linux avec une marge de sécurité).
— `PID_SIZE` : Taille maximale en nombre de caractère du pid du client.

5.2 Structure `my_thread_args`

La structure `my_thread_args` contient tous les arguments à transmettre au thread. Il est à noter qu'avec notre implémentation actuelle, seul le PID du cilent est transmis. Cette structure est codée de telle manière à faciliter l'amélioration du code, permettant d'ajouter aisément de nouveaux éléments utiles aux threads.

5.3 Bloc `main`

Le principe du `main` est de créer un thread pour chaque demande d'exécution de commande. Nous commençons par faire du lanceur un deamon, pour cela nous faisons un processus fils (avec un `fork`) dans lequel on le dissocie de la session avec la fonction `setsid`. Puis on s'occupe de la gestion du signal pour terminer le lanceur. En effet, comme le lanceur est un daemon, l'utilisateur va devoir lui envoyer un signal pour l'arrêter. On admet ici que l'utilisateur utilisera les commandes `kill [PID]` ou `killall [NOM]` pour arrêter le lanceur. Comme ces commandes envoient le signal `SIGTERM` par défaut, nous devons associer la réception de ce signal avec :

— La destruction de la file synchronisée avec la fonction `destroy_file`
— La fermeture du programme en lui-même.

Ensuite, tant que nous pouvons défiler un PID depuis la file synchronisée, on crée un thread dans lequel on passe comme argument le PID du client.

5.4 Fonction `run`

La fonction de lancement du thread crée un fils dans lequel on ouvre les 3 tubes lui permettant de communiquer avec le client. Puis il lit les informations donné par le client sur le tube prévu à cette effet. Ensuite on redirige les deux sorties standards (`stdout` et `stderr`) vers les deux tubes ouverts en écriture. On peut ensuite appeler la fonction `parseur_arg` du module `parseur` afin de récupérer la commande ainsi que chaque option de celle-ci. On finit par utiliser la fonction `execvp` afin d'exécuter la commande demandée. Le père lui ne fait que attendre son fils afin de libérer les ressources proprement.

6 Client

L'objectif du client est de faire parvenir au lanceur toutes les informations transmises par l'utilisateur à travers les arguments passés lors de son appel. Ces informations sont passées par deux canaux différents :

- L'envoi du PID du client au lanceur se fait à travers la file synchronisée ;
- L'envoi de la commande à exécuter par le lanceur se fait à travers le tube portant le nom `TUBE_CL`

On commence par générer le nom des tubes avant de créer ces tubes. Ensuite on envoie le PID du client dans la file afin que le lanceur puisse ouvrir les tubes. Maintenant on ouvre le tube de communication entre le client et le lanceur et on y écrit la commande à exécuter. Nous pouvons ensuite ouvrir les tubes de communication entre le lanceur et le client `TUBE_CL`. Enfin, le client essaye de lire sur chaque tube la sortie de la commande (à travers les tubes `TUBE_RES` et `TUBE_ERR`) et l'affiche sur la sortie correspondante.

7 Difficultés rencontrées

- Nous n'avons pas réussi à implémenter l'utilisation des tubes lors du passage des commandes.
- Nous avons rencontré des difficultés pour formater/parser la commande dans le lanceur afin de pouvoir utiliser `execvp`.
- Dans le client, nous avons eu des difficultés liés à la récupération des informations issus des tubes de réponses. En effet, nous pouvions soit rediriger les sorties du client ou plus simplement lire dans les tubes

puis écrire dans les sorties. Nous avons au final opté pour la seconde solution.

— La gestion du "timing" entre le client et le lanceur fut compliqué. En effet, nous avions dû constamment changer l'emplacement des blocs de code permettant les créations/lectures/écritures des différents tubes afin d'éviter une lecture/écriture sans écrivain/lecteur, bloquant indéfiniment l'exécution de notre code.

— L'aspect ouvert de ce projet a mené à de nombreux désaccords, chaque personne ayant participé au projet ayant une interprétation différente de l'énoncé.

8 Remerciements

Nous tenons à remercier tout particulièrement

— **HADDAG Édouard**

— **RENAUX VERDIÈRE Théo**

— **DUMONTIER Louis**

pour leur aide dans la compréhension et la réalisation de ce projet.