Manuel Technique

Lucas Coussement

30 décembre 2023

Résumé

Ce document est le manuel technique du projet de fin d'études pour la matière *Introduction aux systèmes d'exploitation* de l'année académique 2023-2024. Il contiendra toutes les informations relatives aux décisions techniques prises lors de la réalisation du projet.

Table des matières

1 Introduction

Le projet consiste à développer deux applications en respectant la norme *POSIX.1-2008* :

— Un lanceur de commandes

— Un client qui envoie des commandes au lanceur

En plus de ces applications, une bibliothèque statique sera créée. Elle contiendra des fonctions, des constantes et des structures de données dédiées à la gestion de files synchronisées.

1.1 Structure du projet

Le projet est divisé en trois parties :

— **file_synchronisee** : la bibliothèque statique

— **lanceur** : le lanceur de commandes

— **client** : le client qui envoie des commandes au lanceur

Chacune de ces parties sera programmée dans un fichier *.c* situé dans un dossier **src**. Les fichiers *.h* seront placés dans un dossier **include**. Enfin, les fichiers *.o* et les bibliothèques statiques générées par la compilation seront dans un dossier **lib**.

La compilation sera gérée par un *Makefile* situé à la racine du projet. Nous pouvons aussi trouver un dossier **man** qui contiendra les pages de manuel des applications *lanceur* et *client*.

1.2 Makefile

Le *Makefile* crée la bibliothèque statique *libfile_synchronisee.a* puis compile les applications *lanceur* et *client* en utilisant cette bibliothèque. Il permet aussi de générer une archive du projet et de tout nettoyer.

Voici la liste des commandes disponibles :

— **make** : compile le projet

— **make clean** : supprime les fichiers *.o*, l'archive, les exécutables et les bibliothèques statiques

— **make lanceur** : crée la bibliothèque statique et compile le lanceur

— **make client** : crée la bibliothèque statique et compile le client

— **make archive** : génère une archive du projet

2 File synchronisée

La file synchronisée, comme son nom l'indique, est une file[1] qui est synchronisée. Elle est implémentée en suivant le principe du problème des producteurs-consommateurs. La file est composée de 3 sémaphores pour gérer les accès aux sections critiques.

- **mutex** : un sémaphore binaire qui permet de gérer l'accès à la file
- **vide** : un sémaphore qui compte le nombre d'emplacements vides dans la file
- **plein** : un sémaphore qui compte le nombre d'emplacements occupés dans la file

La file synchronisée devra implémenter les fonctions d'initialisation, d'ajout, de suppression et de destruction. Elle devra aussi contenir une structure de données particulière qui permettra de stocker les commandes. Il faut donc définir ce qu'est une commande et comment elle est stockée dans la file.

2.1 Commande

Pour que le lanceur puisse exécuter une commande, il faut qu'il sache quelle est la commande à exécuter. Pour cela, il faut simplement faire passer une chaîne de caractères qui sera traitée par le lanceur. Cette chaîne de caractères sera appelée **commande**.

Une autre partie du travail du lanceur est de rediriger l'entrée standard, la sortie standard et la sortie d'erreur pour chaque commande vers des tubes afin que le client puisse envoyer ou récupérer les données de la commande. Il nous faut donc un moyen de communiquer les tubes au lanceur. Pour cela, nous avons choisi de faire passer le **pid** du client. Les clients construiront les tubes en suivant un schéma de nommage précis basé sur leur **pid**. Ainsi il sera possible de retrouver tous les tubes d'un client en connaissant son **pid**.

Maintenant que nous avons identifié les informations nécessaires pour exécuter une commande, il nous est possible de définir la structure de données qui sera stockée dans la file synchronisée. Cette structure sera nommée **commande** et sera composée de deux champs :

- **pid** : le **pid** du client qui a envoyé la commande
- **commande** : la commande à exécuter envoyée sous la forme d'une chaîne de caractères

1. Une file est une structure de données qui respecte le principe *FIFO* (First In First Out).

2.2 Fonctions

Maintenant que nous avons défini les informations à stocker dans la file, il nous faut déterminer comment les stocker, comment les récupérer et comment les détruire. Pour cela, nous allons définir les fonctions suivantes :

- **file_init** : initialise la file synchronisée
- **file_enfiler** : ajoute une commande à la file synchronisée
- **file_defiler** : récupère une commande de la file synchronisée
- **file_destroy** : détruit la file synchronisée
- **file_kill_all** : met fin à tous les processus en attente du traitement de leur commande

La fonction **file_init** initialise tous les sémaphores de la file. Cette fonction prend en entrée un pointeur vers une structure de données de type **file_synchronisee** et initialise les sémaphores **mutex**, **vide** et **plein** à 1, $MAX\_COMMANDES$[2] et 0 respectivement. Elle met également à 0 l'indice de la commande la plus ancienne ajoutée à la file.

La fonction **file_enfiler** ajoute une commande à la file synchronisée. La fonction attend que le sémaphore **vide** soit supérieur à 0, puis attend pour le mutex. Une fois le mutex obtenu, elle ajoute la commande à la file en dernière position, c'est-à-dire à la position $(last+1)$ mod $MAX\_COMMANDES$ où $last$ est l'indice de la dernière commande ajoutée à la file. Elle libère ensuite le mutex et incrémente le sémaphore **plein**.

La fonction **file_defiler** récupère la première commande de la file synchronisée. La fonction attend que le sémaphore **plein** soit supérieur à 0, puis attend pour le mutex. Une fois le mutex obtenu, elle récupère la commande la plus ancienne en utilisant $last$, puis libère le mutex et décrémente le sémaphore **plein**.

La fonction **file_destroy** détruit les sémaphores de la file synchronisée.

La fonction **file_kill_all** met fin à tous les processus en attente du traitement de leur commande. Elle attend que le mutex soit disponible, puis elle parcourt toutes les commandes de la file et envoie un signal **SIGKILL** à tous les processus en attente. Elle libère ensuite le mutex.

2. *MAX_COMMANDES* est une constante définie dans la file synchronisée qui représente le nombre maximum de commandes stockables dans la file.

3 Lanceur

Le lanceur a pour but d'exécuter toutes les commandes envoyées par les clients. Pour ce faire, il initialise une file synchronisée dans un segment de mémoire partagée. Il crée ensuite un thread qui va se charger d'attendre les commandes en défilant en boucle la file. Une fois une commande récupérée, le thread va créer un autre thread qui exécutera la commande puis attendre une autre commande et ainsi de suite. Le processus principal du lanceur va se charger d'attendre que l'utilisateur rentre **q** ou **EOF** pour mettre fin au programme de manière propre.

3.1 Attente d'une commande

Il est possible qu'aucun client n'envoie de commande au lanceur. Dans ce cas, le thread qui attend les commandes va rester bloqué indéfiniment. Pour éviter ce problème, nous avons décidé de créer un thread qui va utiliser un sémaphore qui sera incrémenté à chaque fois que le thread principal rentre dans une phase d'attente et à chaque fois qu'il ressort d'une phase d'attente. Au moment où le thread qui attend les commandes rentrent dans une phase d'attente, il incrémente le sémaphore ce qui a pour conséquence de débloquer le thread qui attend le signal. Celui ci va alors attendre un temps donné puis vérifier si le sémaphore a été incrémenté. Si le sémaphore a été incrémenté, cela signifie que le thread qui attend les commandes est sorti de sa phase d'attente et qu'il a traité une commande. Dans le cas contraire, cela signifie que le thread qui attend les commandes est toujours bloqué et qu'aucune commande n'a été ajoutée à la file. Alors le thread qui attend le signal va mettre fin au programme en libérant les ressources.

3.2 Exécution des commandes

Lorsqu'une commande est récupérée par le thread qui attend les commandes, un thread qui exécute les commandes est créé. Ce thread va ouvrir les tubes d'entrée, de sortie et d'erreur de la commande en fonction du **pid** du client qui a envoyé la commande. Il va ensuite exécuter la commande en redirigeant l'entrée standard, la sortie standard et la sortie d'erreur vers les tubes. Une fois la commande exécutée, le thread va fermer les tubes et libérer les ressources allouées par la commande.

3.3 Paramètres

Le seul paramètre que prend le lanceur est le temps de delai d'attente. Ce temps est utilisé par le thread qui attend le signal pour savoir combien de temps il doit attendre avant de vérifier si le thread qui attend les commandes est toujours bloqué. Par défaut, le programme attend indéfiniment. Mais si l'utilisateur rentre un temps de delai avec l'option **-d**, le programme attendra le temps fourni en seconde.

4 Client

Le client a pour but d'envoyer une commande au lanceur. Pour ce faire, il prend en argument lors de son appel la commande à exécuter. Il va ouvrir le segment de mémoire partagée contenant la file synchronisée. Dans le cas où cette file n'existe pas, le client mettra fin au programme puisque cela signifie qu'il n'existe pas de lanceur actif. Une fois la file synchronisée récupérée, le client va ajouter sa commande à la file puis créer un thread qui va lire en continu les tubes de sortie et d'erreur de la commande et les rediriger vers la sortie standard et la sortie d'erreur du client. Le client va également dans son thread principal écouter l'entrée standard pour écrire les données lues dans le tube d'entrée de la commande. Dans le cas où il écrirait sans lecteur, un appel sigpipe sera lancé et le client mettra fin à l'écoute de l'entrée standard. Une fois la commande terminée, le client mettra fin au programme.

4.1 Gestion des commandes

Lorsque le client envoie une commande au lanceur, il crée un thread. Ce thread a pour but d'écouter les tubes de sortie et d'erreur de la commande et de les rediriger vers la sortie standard et la sortie d'erreur du client.

5 Exemple d'utilisation

5.1 Exécution du lanceur

```
$ ./lanceur
Entrez q pour quitter
```

Cette commande crée le lanceur avec un temps de delai d'attente infini.

```
$ ./lanceur -d 5
Entrez q pour quitter
```

Cette commande crée le lanceur avec un temps de delai d'attente de 5 secondes. Ce qui signifie que si aucun client n'envoie de commande au lanceur pendant 5 secondes, le lanceur mettra fin au programme.

5.2 Exécution du client

```
$ ./client echo "message"
message
```

Cette commande renverra un message d'erreur dans le cas où le lanceur n'est pas actif.

```
$ ./client echo "message" | cat
message
```

Il est possible d'utiliser le client avec des tubes.

```
$ ./client cat
message
message
autre message
autre message
^D
```

Le client met à disposition l'entrée standard pour chaque commande envoyée. Il est donc possible d'utiliser des commandes comme **cat** qui attendent des données sur l'entrée standard.

5.3 Manuel d'utilisation

5.3.1 Lanceur

```
$ man man/lanceur.1
```

est la commande qui permet l'affichage du manuel d'utilisation du lanceur.

5.3.2 Client

```
$ man man/client.1
```

est la commande qui permet l'affichage du manuel d'utilisation du client.