Rapport de projet: Chat – édition processus & pipe

Daniel Defoing (`ddef0003`) Ibrahim Dogan (`idog0003`)
Abdalrahman El Hussein (`aelh0003`)

28 novembre 2024

Table des matières

1 Introduction

Ce rapport décrit la conception du premier projet dans le cadre du cours de Systèmes d'Exploitation (INFO-F201). Il présente les choix de conception qui ont guidé notre développement, et les difficultés qui ont pu survenir durant celui-ci. Pour voir de façon précise les changements ayant eu lieu tout au long du projet et la contribution de chacun, veuillez consulter le repository GitHub de notre projet.

2 Choix du langage :pourquoi C++ plutôt que C ?

Une raison fondamentale qui a guidé cette décision est que C++ possède des fonctionnalités absentes en C tels que les références, les chaînes de caractères (ou *strings*) ou encore les classes. Ces éléments apportent des avantages majeurs :

- Ils facilitent le développement en enrichissant la panoplie mise à disposition des développeurs d'outils pratiques pour développer des applications en général, comme les classes.
- Ils évitent aux développeurs de devoir réinventer la roue en pouvant abstraire ou simplifier certaines opérations grâce aux outils susmentionnés du C++.
  Par exemple, l'utilisation des références ou classes a été importante pour résoudre certains problèmes rapidement, chose qui se serait avérée plus compliquée en C qui ne possède pas ces outils.

Ceci montre que des avantages ont été trouvés à utiliser C++ plutôt que C, alors qu'aucun avantage majeur n'a été identifié en faveur de l'utilisation de C par rapport à C++. Ce choix du langage s'est donc naturellement imposé comme le plus adapté.

3 Principes généraux de conception

Cette section discute des grands principes qui ont guidé le développement, des *idées* qui ont défini la façon dont le code a été écrit. Cette section n'est bien sûr pas exhaustive, les éléments mentionnés ci-dessous s'appuyant sur les principes généraux de codage qui ont été appris depuis le début du bachelier. La gestion des aspects techniques spécifiques à l'application (signaux, mémoire partagée...) sera discutée dans la section suivante.

3.1 Abstraction des opérations "de bas niveau" au maximum

On a cherché à masquer le fonctionnement interne de notre application le plus possible, afin de permettre à l'application d'éventuellement évoluer (par exemple en une application de chat en ligne). C'est évidemment l'un des fondements du *principe d'encapsulation*, qui favorise une séparation nette entre l'interface et l'implémentation interne d'un module [1]. Il suffit de voir des noms de méthodes comme `send_message` ou `open_sending_channel` pour s'en convaincre : ces noms disent simplement ce que la méthode *fait*, pas *comment elle le fait*.

3.2 Informer l'utilisateur tant que possible

En plus des erreurs critiques censées stopper l'exécution du programme, un système de warnings a été mis en place. Le but de ceci était d'être le plus explicite possible pour l'utilisateur sur l'état de l'application, et de proposer une séparation entre les erreurs dites *critiques* et les autres, moins graves.

4 Choix et difficultés d'implémentation

4.1 Makefile

Le makefile fourni au commencement du projet a été repensé sous plusieurs angles pour permettre une meilleure robustesse avec des features basiques mais pratiques, dont voici des exemples :

- Création automatique d'un dossier `obj` pour stocker les fichiers objet s'il n'existait pas auparavant
- Réorganisation des variables du makefile pour rendre les commandes de compilation plus flexibles, plus génériques. En clair, il suffit de modifier la valeur d'une variable pour adapter les options de compilation.

Cette partie n'a pas posé de difficultés particulières.

4.2 Gestion des exceptions et valeurs de retour

Une spécificité de ce projet est qu'il implique l'utilisation d'un grand nombre de fonctions système prédéfinies (`write` ou `read` notamment). Ces fonctions, construites à la base pour le langage C, ne lèvent pas d'exception (absentes en C), mais ont plutôt des valeurs de retour qui indiquent l'issue de l'opération exécutée. Par exemple, un appel à `write` retournera la valeur -1 si une erreur devait arriver d'une façon ou d'une autre.

Difficulté rencontrée : Une conséquence de cette façon de faire des fonctions système en C est que, contrairement à une exception, une valeur de retour indiquant une erreur n'interrompt pas le programme en cours. Cela signifie que, si l'on n'est pas assez rigoureux sur la vérification des valeurs de retour, on risque d'avoir des erreurs "silencieuses" en causant d'autres, et cela peut mener à des problèmes très difficiles à débugger. Une difficulté rencontrée lors du développement a donc été de gérer proprement toutes ces opérations prédéfinies dès le début, et d'éviter des conditions à tout va pour éviter de se retrouver avec un code désordonné, difficile à lire et à maintenir [2].

Résolution : afin de faciliter la maintenabilité du code et de minimiser les oublis, il a été décidé de centraliser autant que possible la gestion des erreurs dans une classe nommée `ExceptionHandler` qui a permis une uniformisation de la gestion des erreurs dans le programme, ainsi qu'à un raffinement des types d'erreurs que notre programme détecte (via un système de warnings, mentionné plus haut).

4.3 Gestion des signaux

Le programme gère les signaux `SIGINT` et `SIGTERM` uniquement. Lors de la réception d'un signal `SIGINT`, le programme affiche les messages en attente si l'option `manuel` est activée. Dans le cas contraire, le programme termine proprement les deux chats lors de la réception du signal `SIGINT`. Afin de traiter ce signal correctement dans les différents cas, il a fallu implémenter une solution pour que ce signal soit ignoré *dans le processus fils*. Si le signal n'était pas ignoré dans ce processus, plusieurs problèmes auraient fait surface, notamment la réception multiple de `SIGINT` (qui pouvait entraîner des interruptions d'autres processus, évidemment fortement indésirables).

Un choix d'implémentation discutable est l'utilisation d'un pointeur global pour accéder aux attributs et méthodes de la classe `ChatHandler`. Au début, cette fonction se trouvait dans le fichier `main.cpp`. Mais étant donné que certains attributs et méthodes de `ChatHandler` étaient nécessaires à cette fonction, il a été décidé de déplacer cette fonction dans la classe elle-même. Cependant, cela créait des problèmes (notamment appeler la fonction `signal()` avec `Signal_Handler` comme deuxième paramètre). Finalement, il a été décidé d'utiliser un pointeur `static` au sein de la classe `ChatHandler`

qui pointe vers l'instance actuelle de la classe après que le constructeur soit appelé. Cette approche a été choisie car elle minimisait les difficultés de gestion de la mémoire partagée.

4.4 Programme `chat`

4.4.1 Choix d'implémentation

La logique à suivre pour l'implémentation des canaux de communication entre deux utilisateurs semblait assez évidente, car mentionnée en travaux pratiques [2] : il fallait d'abord initialiser les canaux de communication (les *pipes nommés* [3], donc), et ensuite créer un processus fils (via un appel à `fork()`) pour avoir un processus *écrivain* de messages (via les pipes) et un processus *lecteur*.

4.4.2 Mode `bot`

L'utilisation du mode `bot` permet de ne plus afficher le message envoyé par l'utilisateur et les valeurs `ANSI` utilisées pour la mise en forme des messages. Dans le programme, un attribut booléen `bot` de la classe `ChatHandler` a été utilisé.
Par défaut, plusieurs variables définissent les caractères `ANSI` dans la classe `ChatHandler`. Elles sont mises à jour selon la valeur de l'attribut `bot`, naturellement : la décision d'afficher les messages envoyés par un utilisateur qu'il a lui-même écrits est également déterminée de cette façon, via une simple condition.

4.4.3 Mode `manuel`

On peut distinguer 3 situations où l'affichage doit être géré d'une façon particulière lorsque l'option `manuel` est activée.

Réception d'un signal SIGINT
Comme mentionnée dans la section 4.3 de l'énoncé, ce signal est ignoré *dans le processus secondaire* (le *processus fils*). Symétriquement, pour afficher les messages en attente, une fonction nommée `display_pending_messages` est appelée dans le processus d'origine (ou père).

Lorsqu'un message est envoyé par l'utilisateur (messages en attente)
De nouveau, cet affichage a lieu dans le processus d'origine, puisque ces messages sont affichés *après l'envoi du message de l'utilisateur* et qu'on sait que le processus d'origine à le rôle d'*écrivain*, comme vu plus haut. Cela se fait par un nouvel appel à la fonction `display_pending_messages` dans le "canal d'envoi".

***Buffer* supposé contenir les messages en attente rempli**
La dernière situation concerne l'affichage des messages lorsque plus de 4096 octets sont en attente d'être affichés. Dans ce cas, les messages sont affichés *dans le processus secondaire*. En effet, comme expliqué dans la section 4.4.4 de l'énoncé, le tableau de caractères faisant office de *buffer* est vidé lorsqu'il dépasse 4096 octets avec le message qu'il vient de recevoir.

Là encore, un appel à `display_pending_messages` était de rigueur, et ce n'est qu'après que le message reçu est ajouté dans le tableau. Il était donc crucial d'ajouter le message reçu et d'afficher les messages en attente dans le "canal de réception".

4.4.4 Mémoire partagée pour le mode `manuel`

Choix d'implémentation : Pour créer de la mémoire partagée entre les différents processus ayant un lien de parenté, l'utilisation de `mmap` a été privilégiée, un choix qui semble évident : pour une

mémoire partagée telle que demandée dans le projet, les fonctionnalités de `mmap` suffisaient amplement, là où utiliser une autre librairie comme `shm` se serait avéré plus complexe sans gain de clarté de code ou de performance notable.

Difficulté rencontrée : comme mentionné auparavant, `mmap` a été utilisé pour initialiser une mémoire partagée. Pour stocker les messages dans la mémoire partagée, l'utilisation de `std::queue` avait été décidée initialement. Cependant, cette option, bien que semblant *intuitive* de prime abord, n'était pas fonctionnelle. En effet, les éléments d'une structure de données STL peuvent en réalité être des pointeurs qui pointent vers une autre zone mémoire. Cela créait donc des crashs du programme.

Résolution : par après, ce problème a été résolu en stockant les messages dans un seul *buffer* (qu'on peut définir grossièrement comme une sorte de *conteneur temporaire de messages*).
La prise de décision concernant la taille et le type de cette mémoire partagée a été la suivante : pour stocker les messages, une structure simple, `SharedMemoryQueue`, a été implémentée. Cette structure contient un tableau de caractère (le *buffer* susmentionné) ainsi qu'un compteur `total_chars` afin de suivre le nombre de caractères stockés dans le tableau. Le choix de cette structure permet d'assurer une organisation claire des messages, d'optimiser la gestion de l'espace disponible, que l'espace de la mémoire partagée soit alloué dynamiquement, et d'éviter les problèmes rencontrés lors d'utilisation d'une structure de données `STL`.

En ce qui concerne la taille de la taille de la mémoire partagée, une taille de **4096** octets a été choisie. Ce choix est en accord avec les consignes du projet. En effet, comme mentionné dans les consignes concernant le mode manuel, les messages en attente doivent être affichés lorsque plus de 4096 octets sont en attente d'être affichés. Cette taille a donc été prise afin d'optimiser la mémoire et la gestion d'affichage tout en respectant les consignes.

5 Conclusion : la difficulté majeure de ce projet

Les difficultés spécifiques à chaque partie du programme ont été discutées en profondeur : il est temps de prendre une vue de plus haut niveau sur l'ensemble du projet, pour discuter de sa difficulté principale.

La plupart de ces difficultés rencontrées ont en fait une origine commune, dans le sens où la difficulté de la tâche ne réside pas dans les concepts derrière celle-ci, mais plutôt dans leur cohabitation au sein d'un même programme. Cela impliquait une gestion exhaustive des cas possibles tout en maintenant une clarté et une structure de code rigoureuses.

6 Bibliographie

Sources consultées pour la dernière fois le 28 Novembre 2024, 18h31.

1. *Encapsulation (programmation)* - Wikipédia en français, dernière modification le 9 Novembre 2021
2. J.GOOSSENS et O.MARKOWITCH, TPs 3 à 5 du cours de Systèmes d'Exploitation (INFO-F201), 2024
3. K. VERMA, *Named Pipe or FIFO with example C program*, 11 Octobre 2024