TELECOM Nancy

Rapport du Projet de Réseaux & Systèmes

Octobre - Décembre 2021

Projet de RS 2021

Étudiants :
Théo BOUGUET
Thomas BALDUZ

Numéro Étudiant :
32021152
32026314

Encadrant du projet :
Lucas NUSSBAUM

Table des matières

1 Introduction

Ce projet tesh avait pour objectif de nous faire faire un interpréteur de commande exécutable ressemblant aux shells Unix comme sh ou encore bash. Nous allons décrire dans ce rapport le déroulement de la réalisation de ce projet.

Dans un premier temps nous allons rapidement aborder la gestion de ce projet. Dans un second temps, nous allons parler de la conception du tesh. Ensuite nous parlerons des difficultés que nous avons tous les deux rencontrées et comment nous avons essayé de les résoudre. Enfin nous détaillerons le temps que nous avons passé dans les différentes étapes du projet.

Ce projet a été réalisé dans le cadre du premier semestre de deuxième année à Télécom Nancy s'inscrivant dans le module STIC 3.

2 Gestion de projet

En ce qui concerne l'organisation de ce projet, nous avons essayé de régulièrement faire des réunions afin de se tenir informés de l'avancement de chacun et de décider des tâches à accomplir pour la prochaine réunion.

Au tout début du projet, nous avons essayé de réfléchir à la base ensemble, puisqu'il était impossible de se donner des fonctionnalités à implémenter, puis nous nous sommes réparti les tâches de cette manière :

Table 1 – Tableau de répartition des tâches

Tâches	Responsables
Fonctionnement de base	Théo, Thomas
Commande interne (cd)	Théo
Affichage d'un prompt	Thomas
Enchaînement conditionnel de commandes	Thomas
Redirections d'entrée et de sortie	Thomas
Mode interactif / Non interactif	Théo
Sortie sur erreur	Théo
Lancement de commandes en arrière plan	Théo
Édition de la ligne de commande avec readline	Thomas
Chargement dynamique de readline	Thomas

Ce tableau n'a pas forcément été respecté à la lettre. Évidemment si l'un d'entre nous avait besoin d'aide, l'autre membre du binôme pouvait venir l'aider. Par exemple, pour les enchaînements de commandes et les redirections, nous avons tous les deux travaillé dessus puisque c'était les tâches les plus compliquées.

3 Conception

Pour la conception, comme écrit plus haut, nous nous sommes d'abord concentré sur le fonctionnement de base pour avoir un programme qui lit une commande rentrée au clavier, l'exécute,

attend sa fin, puis lit une autre commande. Ceci afin d'avoir une base de programme solide nous permettant ensuite d'ajouter des fonctionnalités. Nous allons détailler dans les sous-parties suivantes la conception de certaines fonctionnalités qu'il semble pertinent de détailler.

3.1 Fonctionnement de base

D'abord, il a fallu réfléchir à la manière de récupérer les commandes afin de pouvoir implémenter toutes les fonctionnalités par la suite sans avoir à retoucher au fonctionnement global du programme. Dans un premier temps, le programme attend que l'utilisateur entre une commande tant que l'utilisateur n'a pas indiqué la fin des entrées (avec CTRL+D). Pour chaque entrée de l'utilisateur, on retire tous les espaces entre les mots. Ces mêmes mots sont stockés dans un array. Le programme va alors ensuite parcourir le tableau, et décider de quelle action entreprendre en fonction de chaque mot rencontré. Le mot NULL indique bien entendu la fin de la ligne de commande.

Cette méthode paraissait la plus intuitive au début du projet, En effet, lorsque l'on parcours le tableau, tant que l'on ne rencontre pas de séparateur de commande ou de symbole de redirections/-pipes, on ajoute le mot à un autre array qui sera l'array passé en paramètre du execvp. Et dès que l'on rencontre un de ces symboles, on fait un traitement spécifique lié à ce symbole, et on exécute la commande.

Cette méthode fonctionne bien pour des enchaînements de commandes simples, cependant, comme nous le détaillerons dans la partie décrivant nos difficultés rencontrées, celle-ci pose certains problèmes.

3.2 Enchaînement conditionnel de commandes

Avec notre méthode pour parcourir les commandes, on peut assez facilement réaliser des enchaînements simples ou conditionnels de commandes. En fait, nous avons mis en place une variable qui est modifiée à chaque fin de commande afin de savoir si l'on doit exécuter la commande suivante. Donc si on rencontre un " ;", la commande suivante sera toujours exécutée (sauf si l'on avait indiqué l'argument "-e" lors de l'exécution du tesh). Si l'on rencontre un "", alors si la commande a terminé avec le code 0, on indique que l'on peut exécuter la suivante, sinon, on n'exécute pas les commandes suivantes (sauf si l'on rencontre un " ;"). Si c'est un "||", c'est l'inverse, tant que le code de retour n'est pas 0, on exécute les commandes suivantes.

3.3 Mode interactif / non interactif

Pour réaliser le mode non interactif, il faut remplacer le fichier indiqué dans la fonction fgets() que nous utilisons pour récupérer les commandes par le fichier indiqué par l'utilisateur quand celui-ci démarre le tesh. Pour faire cela, on a introduit une variable qui contient le descripteur d'entrée que le programme va utiliser, Par défaut, on indique que c'est stdin. Et si lors du démarrage du tesh, on voit qu'il y a des arguments précisés, on vérifie d'abord s'il s'agit du '-e' pour les sorties sur erreur ou du '-r' du readline. Si ce n'est pas le cas, on considère que c'est un fichier qui est passé, et on essaye de l'ouvrir. La fonction fgets() va alors lire les commandes contenues dans le fichier et les exécuter ligne par ligne.

3.4 Lancement d'une commande en arrière plan

Comme pour les fonctionnalités d'avant, on détecte si on veut mettre une commande en arrière-plan si l'on rencontre un symbole spécifique, ici, c'est "". On l'indique alors à la fonction qui s'occupe d'exécuter les commandes à l'aide d'un paramètre. Ensuite, contrairement aux autres cas, on exécute la commande sans appel de wait. L'utilisateur peut donc continuer à lancer des commandes en parallèle.

Afin de pouvoir faire revenir la commande au premier plan, nous avons mis en place vérification comme avec la commande "cd". Si la commande entrée est fg, on n'exécute pas de commande avec execvp, mais on utilise juste la fonction wait() s'il n'y a pas de pid de précisé, et waitpid si un pid est précisé.

4 Difficultés rencontrées

4.1 Théo

Sur la majorité des fonctionnalités à implémenter dans ce projet, je n'ai pas rencontré de difficultés particulières. A l'exception des redirections et des pipes sur lesquelles je me suis penché à la fin du projet.

4.1.1 Les redirections

En effet, au départ, Thomas devait se charger des redirections et des pipes. Cependant, comme il a rencontré des difficultés sur cette partie, j'ai essayé de compléter ce qu'il avait déjà fait. J'ai donc réussi à faire fonctionner les redirections "$>$", "$\gg$" et "$<$". Mais seulement en mode terminal lorsque je testais en local. Je me suis rendu compte que les tests blancs sur git ne passaient pas pour ces fonctionnalités. En fait, sur ces tests, je ne pouvais même pas voir quelles étaient les commandes testées. J'ai donc réalisé durant tout le dernier week-end avant la deadline du projet une série de modifications et de tests afin que, dans un premier temps j'identifie la cause du problème, puis que je puisse ensuite le régler.

Pour l'identification, je pense avoir trouvé d'où venait le problème. D'abord, je pensais que le souci venait de mon code, par exemple d'un fichier que je ne fermais pas. Mais je ne comprenais pas pourquoi le code fonctionnait lorsque je le testais de mon coté, mais qu'il ne fournissait même pas d'output sur les tests blancs. C'est alors que je me suis rendu compte que le problème venait des redirections lorsqu'elles étaient utilisées en mode non interactif. En effet, la redirection "$<$" ne fonctionnait pas dans ce mode.

Cependant, je n'ai au final pas réussi à trouver comment corriger ce problème même en modifiant de plein de façon mon code.

Un autre problème des redirections est lié à notre manière de récupérer les commandes. Comme expliqué plus tôt, le programme lit chaîne de caractère par chaîne de caractère. Et lorsque l'on rencontre un symbole spécial, on exécute la commande et on réinitialise l'array permettant de lancer la commande. Mais les symboles de redirections ne sont pas des séparateurs de commandes. En effet, si l'on veut par exemple lancer une première commande dont on redirige la sortie vers un fichier, puis lancer une deuxième commande, l'utilisateur doit écrire "cmd1 $>$ fichier ; cmd2". On voit dans notre cas que cela va poser un problème. Au symbole "$>$" on arrête d'ajouter des commandes, on

s'avance d'une chaîne de caractère pour récupérer le nom du fichier, on fait la redirection, on lance la commandes, puis on rencontre un " ;". Donc le programme va vouloir lancer une commande, sauf que l'array sera vide, cela va alors provoquer une erreur. Le problème est le même si l'on veut faire des enchaînements conditionnels, sachant donc qu'il faut différencier chaque séparateurs. De plus, il est impossible ici de rediriger la sortie et l'entrée d'une commande. Dans ce cas, j'ai pu régler le problème avec les séparateurs en faisant regarder le programme d'une chaîne de caractères en avant, on peut alors décider quoi faire. Cependant, je n'ai pas réussi à faire des redirections multiples.

4.1.2 Les pipes

J'ai aussi essayé de me pencher sur la réalisation des pipes. Cependant, ici encore, notre méthode pour récupérer les commandes pose problème. En effet, le principe d'un pipe "cmd1 | cmd2" est de rediriger la sortie de cmd1 vers l'entrée de cmd2. Pour pouvoir implémenter les pipes, il aurait donc sûrement fallu modifier grandement le fonctionnement de notre programme. Je n'ai cependant pas eu le temps nécessaire pour pouvoir vraiment effectuer ces modifications, voulant en priorité corriger les problèmes liés aux redirections.

4.2 Thomas

La principale difficulté rencontrée lors de ce projet a été pour moi la maîtrise du langage C. N'étant pas très à l'aise dès le départ, le début a été assez laborieux. Au fil de lectures de documentation et de tests, mon niveau s'est amélioré. Cependant, je n'ai pas pu finir l'implantation du chargement dynamique de readline car j'ai passé trop de temps sur les redirections.

4.2.1 Fonctionnement de base

Pour la première étape du projet, j'ai rencontré mes premiers problèmes. L'utilisation de tableaux de pointeurs était pour moi nécessaire, car je voulais que chaque commande soit incluse dans un tableau où chaque élément de cette commande constituerait une case. La complexité de ce choix d'implantation ne m'a pas permise d'aller au bout de mon idée. Après une réunion avec Théo, l'idée de "tokeniser" chaque élément de la commande et ensuite de l'analyser fut la solution prise.

4.2.2 Les redirections & pipe

Concernant les redirections ">", "»" et "<", je n'ai pas réussi à gérer les descripteurs de fichiers. Après réflexion, mes erreurs devaient provenir du fait que lors du pipe, je confondais les stdin et stdout des parents et des fils. Je pense que je pourrais aujourd'hui identifier mes erreurs et adapter mon code en fonction.
La mise en place des pipes a été très laborieuse car notre code ne garde pas en mémoire une commande qui vient de s'exécuter. Il aurait donc fallu stocker la sortie d'une commande précédant un pipe, et la donner en entrée à la commande suivante. Cependant, la complexité du code ne m'a pas permis de faire des tests fonctionnels.

4.2.3 Chargement dynamique de readline

Après avoir passé beaucoup de temps sur la partie des redirections, je n'ai eu que peu de temps pour étudier l'utilisation du readline. L'implantation en édition de ligne de commande de readline fut rapide, mais je n'ai pas eu le temps nécessaire pour le chargement dynamique, à savoir permettre à la fonction "dlopen" de chercher automatiquement le path de la librairie *readline*.

5 Gestion du temps

La vraie difficulté pour ce projet était de bien gérer notre temps. En effet, certaines fonctionnalités à mettre en place étaient assez complexes et demandaient une bonne maîtrise du langage C, et nous avions un certains nombre d'autre projets à réaliser en parallèle. Et si nous l'avions mieux gérer, surtout sur la fin, nous aurions pu corriger les problèmes de redirection et mettre en place les pipes.

TABLE 2 – Tableau de répartition du temps

Etape	Temps	
	Thomas	Théo
Conception	5h	
Implémentation	Entre 7h et 10h	Entre 10h et 15h
Tests	Entre 3h et 7h	Entre 5h et 10h
Rapport	3/4h	

6 Conclusion

En conclusion, ce projet a été un vrai défi pour nous. En effet, il nous a permis d'une part de mieux comprendre certaines parties du cours, et d'autre part, il nous a aussi aidé à nous améliorer en C. En effet, l'implémentation de certaines fonctionnalités était assez compliqué à développer, mais cela nous a permis une meilleure compréhension du langage C. Cependant, nous aurions pu être plus efficaces et peut-être terminer toutes les fonctionnalités si nous avions eu une meilleure gestion de notre temps.