TELECOM Nancy

Rapport du Projet Système

October-Décember 2021

TELECOM Shell - Tesh 2021

Étudiants :
Omar CHIDA (G4)
Chaima TOUNSI OMEZZINE (G3)

Numéro Étudiant :
31730598
32025001

Encadrant du projet :
Lucas Nussbaum

Table des matières

1 Choix de conception

1.1 Conception

Après la lecture du sujet, nous avons commencé par mettre en place notre stratégie d'implémentation. Nous avons divisé le sujet en différentes petites tâches et nous avons fait le lien entre eux avec une priorisation des tâches les plus génériques. Globalement, nous avons suivi la ligne directrice proposée par l'énoncé. Nous avons également établi une stratégie des différentes méthodes (*fork, pipe, read, write,...*) à utiliser et dans quelle fonctionnalité.

De manière globale, nous avons adopté une méthode d'échange régulier. Avant chaque partie à implementer, nous discutions et nous nous mettions d'accord sur le choix de l'implementation (structures de données, la méthode à mettre en place...). Nous avons consacré également de temps d'implémentation commune (généralement en distanciel) que nous avons jugé nécessaire pour certaines parties délicates du sujet (traitement des lignes d'entrée *"Parsing"*, les pipes, les priorités entre les opérateurs comme &&, ||,...). Cela a permis également à chacun de mieux comprendre toutes les parties implémentées.

1.2 Architecture et Implémentation

Dès le début du projet, nous avons décidé d'adopter une architecture solide et claire afin de générer du code lisible et compréhensible. Ceçi facilite également les modifications que nous pourrons avoir au futur. Ainsi pour l'architecture globale de notre projet, nous avons adopté une variété du pattern/architecture `Pipeline`[1]. La saisie des entrés de la part de l'utilisateur *"Input"* ou le fichier du *script* sont lu ligne par ligne avant d'être traités et regroupés dans des différents *tokens* en fonction de la priorité de l'opérateur actuel. Chaque segment (déterminé par `&` ou `;`) est donc exécuté séquentiellement. Pour les commandes en arrière plan, un nouveau processus est créé pour exécuter la commande entière independamment du processus principal.

1.2.1 Représentations et termes utilisés

Pour avoir une bonne compréhension entre les membre de l'équipe, nous avons adopté des termes spécifiques non officiels pour identifier certaines parties des commandes entrées. Cela simplifie beaucoup le problème, facilite la communication et améliore la lisibilité et la compréhension du code.

- *Operators* : Les Operators sont les mot-clés du shell comme (`&&` : and, `||` : or, `|` : pipe, `;` : séparateur de commandes et les redirections)
- *High level control flow operators* : Ils sont appelés "haut niveau" car ils ont une priorité plus élevée que les autres opérateurs (`&` pour l'exécution en arrière-plan et `;` pour le séparateur de commandes)
- *Control flow operators* : Ce sont les opérateurs les plus courants comme les `pipes, and, or`
- *Redirections* : Ceux-ci ont la priorité la plus basse (>>, > et <)
- *Text* : C'est une séquence de caractères qui n'ont pas d'opérateur spécifique.
- *Builtin commands* : Ce sont les commandes spéciales exécutées par le shell sans la création d'un nouveau processus.

1. La sortie d'une phase est l'entrée de la phase suivante

— *Compound commands* : Ce sont des commandes qui peuvent ou non contenir des opérateurs de redirection.

Pour faciliter l'opération d'analyse, un tableau contenant des chaines de *tokens* a été déclaré dans `common.c`. Une énumération appelée `BUILTIN_TOKENS` a été déclarée également dans `common.h`. Les valeurs constantes de l'énumération sont de nombres binaires dont le i-ème bit est mis à 1 de sorte que i correspond à l'index du *token* dans le tableau des *Tokens*. Cette représentation binaire offre une grande flexibilité car les *Tokens* peuvent être regroupés à l'aide de l'opérateur `ou` binaire et testés à l'aide de l'opérateur `and` binaire.

Cette présentation a fait preuve de robustesse car, au cours de la toute dernière étape du développement, nous avons découvert un bug [2] dans la priorité de `;` (command separator). Mais grâce à l'architecture adoptée, la correction de ce bug consistait juste à faire une ou deux petites modifications (déplacer une constante d'enum qui représente le séparateur '`;`').

1.2.2 Parsing : Traitement des entrées

Initialement, l'entré est divisée en fonction des espaces. Simultanément [3], elle est analysée en différents *tokens* en fonction du symbole/opérateur. Une structure de donnée hybride spéciale que nous avons appelée `AbstractOp` est utilisée. Elle agit essentiellement comme un arbre ou une hiérarchie de tableaux qui nous aidera à représenter la priorité des différents opérateurs.

```
1 struct _AbstractOp
2 {
3     char** token; // Array containing the string associated with this node
4     int    count; // Count of strings
5     int    op;    // The parsed operator
6     struct _AbstractOp* opsArr; // Operators array that are one level below
    the current node in precedence
7     int opsCount; // Nodes count
8     int opsCap;   // Nodes capacity
9 };
```

Comme indiqué dans l'extrait du code ci-dessus, chaque commande analysée est représentée en utlisant cette structure de données qui contient toutes les informations nécessaires pour l'exécution.

— Le tableau des *tokens* représente la liste des chaines liées au noeud courant.

— L' `op` est un identificateur entier qui représente la nature actuelle du noeud. Il peut s'agir d'un opérateur, commande simple ou commande composée (Il prend une des valeurs de l'enum `BUILTIN_TOKENS` définies dans la sous section précédente).

— L' `opsArr` est un tableau de noeuds similaires qui inclue des éléments de la hiérarchie inférieure.

— `opsCount` et `opsCap` représentent respectivement le nombre d'élements du tableau et sa capacité maximale. Ceux-ci sont pratiques pour savoir quand réallouer le tableau [4].

Lors de l'analyse, le champ `op` est initialisé en fonction du *token* rencontré.

2. Au départ, nous avons pensé que la priorité de `&` est plus imporante que `;`. Cela signifie que `cmd1 ; cmd2 &` executerait `cmd1` et `cmd2` séquentiellement en arrière-plan. Cependant, *bash* exécute `cmd1` dans le plan principal et `cmd2` en arrière-plan

3. Dans la même boucle

4. La réallocation du tableau est faite en complexité amortie

1.2.3 L'exécution des commandes

La phase d'anlayse produit le format de données nécessaire qui simplifie la partie exécution de la commande. A chaque niveau de la struture de données, mentionnée précedement, nous parcourons son tableau d'opérateurs en commençant par les tableaux de haut niveau qui contient les opérateurs d'arrière-plan et les séparateurs de commande. A ce stade, il est décidé si la commande doit être exécutée en parallèle ou executée de manière séquentielle dans le shell.

Le tableau d'opérateurs (`opsArray`) de la commande précédente est ensuite passé à la phase suivante lorsqu'il est décomposé davantage en parcourant ses éléments recherchant d'autres opérateurs (comme `&&`, `||`, `|`, etc à l'exception des opérateurs de redirection). C'est à ce stade que sont manipulés les opérateurs `"and"`, `"or"`, et `"pipe"`. Les pipes sont créés uniquement lorsque l'opérateur suivant dans le tableau est un pipe. La lecture à partir des pipes se réalise lorsque l'opérateur précédent dans le tableau est un pipe. L'algorithme de lecture consiste à permuter l'ancien descripteur de fichier de lecture du pipe après l'avoir fermé avec l'ancien descripteur de fichier de lecture de pipe.

Enfin les données de commande atteindront l'étape de traitement finale où les opérateurs de redirection sont détectés. Selon, l'opérateur de redirection un fichier sera créé et `dup2` sera appelé pour rediriger le descripteur de fichier à l'entrée ou à la sortie standard avant que `execvp` ne soit appelé.

1.2.4 Structure du dépot/projet

Notre dépot git est organisé en différents dossiers et les fichiers ont chacun son propre spécificité comme expliqué ci-dessous :

- **tests** : Un dossier, contient les fichiers, dossiers et scripts nécessaires pour simuler les tests blancs fournis par le professeur.
- **rapport** : Un dossier contenant le fichiers LaTeX pour le rapport du projet.
- **run_tests** : Est un script bash qui lance une simulation de tests blancs et compare la sortie de notre programme directement à la sortie de bash.
- **Makefile** : utilisé pour compiler correctement et facilement `tesh`
- **common.h/c** : Contient des énumérations de *Tokens*, un tableau de *Tokens*, une énumération d'options et la structure `Shell` qui encapsule toutes les données liées à la session shell.
- **readline.h/c** : Inclut les signatures de fonctions et les initialisations liées à `libreadline.so`
- **parser.h/c** : Contient des fonctions liées à l'analyse des commandes et à la construction de la structure des données de commande.
- **utils.h/c** : Contient des fonctions utilitaires et d'aide qui sont principalement liées à la lecture de l'entrée de l'utilisateur. `read_input` est la fonction principale utilisée pour lire l'entrée. `my_readline` est juste un adaptateur de `read_input` pour unifier l'API appelante avec `libreadline` afin que le même code puisse être utilisé plusieurs fois sans avoir besoin de maintenir deux versions différentes du code en deux différents endroits et qui se chargent de la même fonction.
- **bg.c/h** : Contient tous les fonctions liées aux processus en arrière-plan
- **tesh.c/h** : Où se trouvent toutes les fonctionnalités de base.
- **main.c** : Entré du programme où le shell est initialisé et lancé.

2 Difficultés rencontrés

2.1 Parsing

La première étape fondamentale dans notre stratégie était l'étape de *Parsing* puisque tous les autres fonctionnalités en dépendent. En effet, nous avons rencontré quelques difficultés au cours de l'implémentation de cette partie. Nous avons dû changer le tout premier modèle implémenté pour faire le parsing et qui était basé sur des listes chainées. Se rendant compte que cette structure posera beaucoup de problèmes après pour implémenter les fonctionnalités demandées, nous avons choisi d'adopter à la place une structure de liste simple (comme détaillé dans la partie précédente) afin de faciliter la manipulation. La difficulté principale était alors de mettre en place un *Parser* qui prends en compte la spécificité de chaque fonctionnalité, par exemple le cas de `"; , <, >> "`, et la priorité entre les fonctions et les opérateurs. Principalement, le problème se posait avec `file < cmd` puisque il faut tenir en compte que la première partie de la commande doit être reconnue comme un fichier et ne doit pas être exécutée par le programme.

2.2 Les Pipes

Notre algorithme initial de gestion des pipes consiste en la mise en place de deux pipes existants en permanence et s'échangeant entre eux au fur et à mesure que nous exécutons les commandes. Cela a parfaitement fonctionné pour la plupart des cas. Mais quand nous l'avons testé avec des gros fichiers, cela a dépassé la limite par défaut du pipe mais nous n'avons pas voulu juste augmenter la limite de taille du pipe car nous avons estimé que ce n'est pas la bonne façon à faire.

De plus, cette version n'a pas réussi les tests blancs lancés par le professeur ; en particulier les tests de fuite de descripteur de fichier, car pour chaque pipe, un seul pipe devrait exister dans la commande alors que nous en avions deux, même lorsqu'un seul pipe est demandé. Nous avonsa alors modifié l'algorithme pour utiliser un seul pipe et éliminer l'attente immédiate après l'exécution du premier processus, au lieu de cela, nous attendrions la dernière commande (qui n'est pas suivi d'un pipe) pour terminer l'exécution.Cette approche a passé tous les tests.

Nous avons également pensé à une autre approche dans laquelle nous aurions un tableau de pipe initialisés au fur et à mesure que nous progressons dans la commande. Cette approche nous a paru plus correcte alors nous avons décidé de l'adopter, après l'avoir testée localement sur un ensemble de tests qui simulent les vrais tests blancs du professeur. Lorsque les résultats des tests blancs [5] sont sortis, nous n'avons pas compris pourquoi nous n'avions pas réussi le test pipe2, même si nous avions réussi un test similaire en locale.

Nous avons essayé de localiser le problème, mais nous n'avons toujours pas réussi le prochain test pipe2. A ce stade, sachant que nous n'avons plus qu'un seul test blanc à passer, nous avons décidé de revenir à l'ancienne approche fonctionnelle mais sans supprimer la nouvelle approche. Nous avons conservé les deux implémentations en profitant des directives du préprocesseur C [6]. La nouvelle approche peut-être activée ou désactivée en définissant (ou non) la macro `NEW_APPROACH` lors de la compilation ou en commentant/décommentant la ligne 12 dans `tesh.c`.

5. Les tests de 02/12/2021
6. Notamment les `#define`

2.3 Processus en arrière-plan

Au début du projet, Nous avons mal compris la partie des processus en arrière-plan. Nous avonc compris qu'il faut qu'on enlève les processus [7] dés qu'ils terminent leurs exécutions. Les tests sur l'exécution arrière-plan ont énormément clarifié la façon avec laquelle le système est sensé fonctionner pour être correcte. Le bug a donc été rapidement corrigé.

Bilan global du projet d'équipe

Le point sur l'équipe

Etapes	O. CHIDA	C. TOUNSI OMEZZINE
Documentation	1h	1.5h
Conception	3h	3h
Implémentation	20h	19h
Tests	2h	3h
Code Review	3h	2h
Rédaction du rapport	2h	3h
TOTAL	31h	31.5h

TABLE 1 – Le temps moyen consacré au projet par chaque membre

7. Dans le sens où ils sont plus accessibles en faisant `fg PID`