
RAPPORT PROJET SYSTEME

Mini-Shell en C

Hugo Bommart & Cédric Ribet

M. Lavergne, M. Ouffoue & M. Motii

Introduction	3
I - Cahier des charges.....	3
II - Méthodes de travail	3
Programmation Modulaire.....	3
Répartition des tâches.....	4
Outils	4
III - Réalisation du projet	4
Code	4
Main.....	4
Cat	5
Touch.....	5
Find.....	5
Copy.....	6
Parsing	6
Pipe.....	6
Recherche	7
Amélioration supplémentaire.....	7
IV - Problèmes rencontrés	8
Individuellement.....	8
Cédric Ribet.....	8
Hugo Bommart	8

INTRODUCTION

Dans le cadre de notre formation en Système, nous avons réalisé projet en binôme. Le sujet était la création d'un mini-shell en C reprenant les codes des shells classiques. Ainsi, il fallait implémenter des commandes (internes) courantes telles que cat, history, touch, cd, cp, find et gérer le path pour pouvoir exécuter des commandes externes.

Ce projet a été réalisé par le binôme composé de Hugo Bommart et Cédric Ribet, tous deux étudiants en 1ère année du cycle Ingénieur Informatique de Polytech Paris Sud, encadré par M Ouffoue.

I - CAHIER DES CHARGES

Dans un premier temps, nous avons rendu une version préliminaire complète et opérationnelle. Ensuite, nous avons réalisé toutes les commandes internes demandées. Puis, nous avons implémenté la gestion des commandes externes avec une gestion du path. Nous avons réalisé les pipes (la gestion des deux éléments dans la même commande également). Cependant, nous n'avons pas pu réaliser les jobs controls car le temps nous faisait défaut. Néanmoins, nous avons réalisé la commande bonus find.

De plus, nous avons autant que possible essayé de "sécuriser" notre shell. En effet, à chaque appel d'une fonction native du C, nous vérifions la réussite de cette dernière et si elle échoue, nous arrêtons l'exécution de la commande et nous affichons l'erreur grâce à *perror* ou manuellement.

II - METHODES DE TRAVAIL

PROGRAMMATION MODULAIRE

Pour ce projet, nous avons fonctionné avec une programmation modulaire. Notre projet est très décomposé. En effet, nous avons deux fichiers par commande (le code source et le header), cela permet une lecture rapide et agréable de notre projet. Cela nous fait un total 13 fichiers (incluant le shell_final). Nous avons décidé d'avoir des noms de fichiers explicites pour améliorer la clarté du projet.

La répartition est la suivante :

- *shell_final* : contenant la boucle principale,
- *parsing* : contient toutes les fonctions relatives aux chaînes de caractères, (traitement, découpage et analyse),
- *cat* : contient le code de la fonction cat et les différentes options gérées pour cat (plusieurs fichiers, stdin, -n),
- *touch* : contient le code de la fonction touch et les différentes options gérées (plusieurs fichiers, création de fichier, -m),
- *cp* : contient le code de la fonction copy, cette fonction est découpé en plusieurs sous-fonctions permettant facilement la compréhension du code,
- *find* : contient le code de la fonction bonus find,
- et *pipe* qui contient la gestion des pipes, de la redirection et l'exécution des commandes entrées par l'utilisateur.

REPARTITION DES TACHES

Nous avons tous les deux travaillé entièrement sur ce projet. Nous nous sommes répartis les différentes fonctions mais nous avons également codé ensemble certaines parties. Notamment pour la vérification et le débogage des fonctions créées. De ce fait, nous maîtrisons et pouvons expliquer tous les deux l'intégralité de notre projet. Cela est possible grâce à un intéressement mutuel sur les fonctions implémentées.

Nous avons pu facilement intégrer nos différentes fonctions grâce à notre programmation modulaire, il nous suffisait de fixer préalablement les prototypes des fonctions pour qu'elles puissent s'intégrer sans soucis à notre programme.

OUTILS

Nous avons utilisé comme demandé un gestionnaire de version, pour se faire nous avons choisi la plateforme GitHub.

III - REALISATION DU PROJET

CODE

Le principal type de structure est un tableau 2D (*args*) contenant la commande entrée, avec *args[0]* contenant le programme et *args[i]* contenant les arguments de la commande.

L'architecture de notre shell a beaucoup évolué depuis le début du projet. Nous avons commencé en séparant le cas avec et sans pipe : une fonction *executeCommand* qui traitait les commandes reçues (sans pipes) et une autre fonction *execPipe* qui les traitait si elles contenaient des pipes. Nous nous sommes vite rendus compte que cela n'était pas pertinent car *execPipe* pouvait également gérer une commande qui ne contenait pas de pipes. De ce fait, nous avons supprimé la fonction *executeCommand* au profit de la fonction *execPipe* plus complète.

A présent, nous allons détailler l'architecture de quelques fonctions que nous trouvons intéressantes afin d'offrir au lecteur une vision globale de notre programme.

MAIN

Voici par conséquent l'architecture finale de notre shell :

- 1 - Saisie utilisateur : on rentre une commande.
- 2 - Étape de parsing : on la découpe en arguments selon son contenu (le traitement sera différent si elle contient un pipe ou un chevron afin que ces caractères ne soient pas considérés comme des arguments).

3 - Étape de traitement préliminaire : stockage de la commande rentrée dans history et on l'effectue si il s'agit de cd ou exit.

4 - Traitement : on effectue la commande. Il y a une différence de traitement entre une commande avec et sans pipe. Avec pipe, nous devons allouer des chaînes de caractères supplémentaires.

5 - Fin : fin du traitement de la commande.

6 - Attente : on attend la suivante puis on repart à l'étape 1 si une commande est entrée.

Si la commande entrée est "exit", on n'atteint jamais l'étape 6, on quitte le programme à l'étape 3. Des sous-fonctions interviennent à chaque étape afin d'obtenir un fichier principal lisible.

CAT

Il y a 3 cas possibles de cat : stdin (cat sans arguments), -n (affiche les lignes) et sans options. Nous avons implémenter cat de telle manière qu'elle puisse gérer plusieurs fichiers en même temps. Nous procédons de la manière suivante :

1 - S'il s'agit de cat stdin, on boucle sur l'entrée terminal. Tout ce que l'utilisateur rentre est affiché une deuxième fois.

2- Sinon si l'option -n est détectée : on récupère les lignes du fichier ligne par ligne avec *fgets* et on les affiche avec un certain format précisé avec *printf*. L'indice des lignes est remis à zéro entre chaque fichier comme nous pouvons le constater sur un shell classique.

3- Sinon l'option -n n'est pas détectée : on lit et on affiche simplement ce que l'on voit dans le fichier.

4- Fin

TOUCH

Il y a 2 cas possibles pour touch : sans options ou avec l'option -m qui ne modifie que la dernière date de modification du fichier. Touch crée le fichier s'il n'existe pas. On accepte plusieurs fichiers, qu'ils soient existants ou non : on peut utiliser notre touch avec à la fois des fichiers existants et des fichiers non-existants.

1 - Si il n'y a pas d'arguments, on quitte et on affiche une erreur.

2 - Si -m est détectée : on ne modifie que la date de modification et on crée les fichiers s'il le faut.

3 - Sinon, on modifie la date d'accès et de modification et on crée également des fichiers s'il le faut.

4 - Fin.

FIND

Une fonction récursive qui parcourt les répertoires et les sous-répertoires et ainsi de suite jusqu'à ce qu'il ne reste plus rien à explorer. Elle ne gère pas d'options. Cependant, on peut préciser à partir de quel répertoire nous voulons lancer la commande sans pour autant y être. Ainsi les arguments possibles sont : "un chemin /path", "." ou aucun argument (affichera alors depuis le répertoire courant).

1 - On parcourt le répertoire jusqu'à trouver un élément sinon on s'arrête.

- 2 – S’il s’agit d’un fichier, on va l’afficher.
- 3 - Sinon il s’agit d’un dossier, on affiche son nom puis on lance la fonction find sur ce même dossier (récursivité).
- 4 - Fin

COPY

Nous ne décrivons pas la fonction cp car elle est algorithmiquement très proche de find. En effet, il s’agit d’une fonction récursive qui parcourt des dossiers sauf que nous les copions également, nous ne faisons pas que les afficher.

Attention, il faut bien respecter la syntaxe de cette fonction qui est cp path1 path2.

PARSING

Ce fichier contient toutes les fonctions relatives au traitement des chaînes de caractères : découpage, dénombrement de caractères et récupération du path. Nous avons utilisé à la fois des fonction incluses dans “string.h” du C et des algorithmes que nous avons faits nous-mêmes.

PIPE

La fonction majeure de notre programme. Elle gère les pipes et exécute les commandes. Son fonctionnement est complexe, nous devons respecter une méthode bien précise qui est la suivante. En effet, si une des étapes n’est pas respectée, la fonction n’aboutit pas.

- 1 - On ouvre un tableau pipes contenant les pipes.
- 2 - On déclare les pipes grâce à la fonction pipe. On en déclare 2 fois plus de fois qu’il y a de pipes.
- 3 - On fork pour lancer les commandes dans un processus pour être sûr qu’en cas d’erreur, seul la commande est arrêtée et non tout le shell.
- 4 - Nous faisons les liens entre les entrées et sorties de chaque commande.
- 5 - On les ferme juste après leur ouverture pour éviter tout conflit.
- 6 - On traite la commande.
- 7 - Une fois la commande exécutée, on referme une nouvelle fois tous les pipes pour être sûrs que les sorties soient correctes.
- 8 - On attend que tous les forks se finissent avant de quitter la fonction.

RECHERCHE

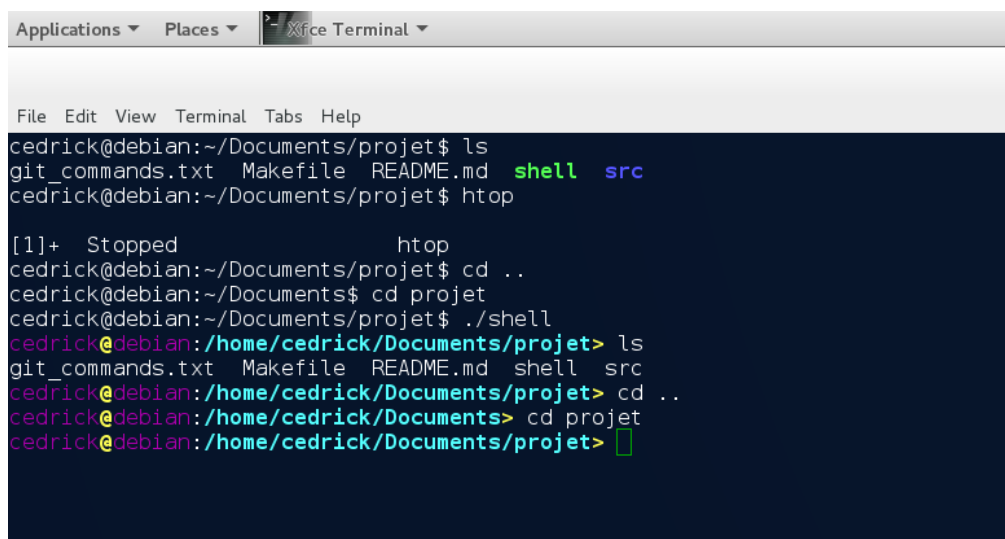
En parallèle de la programmation, ce projet a également nécessité de longues heures de recherche afin de trouver les fonctions du C les plus judicieuses pour notre projet et de bien comprendre le comportement des fonctions que nous avons à implémenter. Dans le cadre de nos recherches, plusieurs pages du manuel ont dû être lues.

Grâce à ce projet, nous avons pu apprendre et appliquer une démarche de recherche constructive afin d'obtenir des réponses pertinentes à nos interrogations.

Par exemple, *dup2* utilisé pour les pipes et la redirection. La compréhension de la fonction *dup2* fût longue mais il s'agissait d'une étape nécessaire par laquelle nous devons passer si nous voulions obtenir une fonction gérant les pipes.

AMELIORATION SUPPLEMENTAIRE

Comme implémentation supplémentaire, nous avons pensé à mettre en couleur le répertoire courant de l'utilisateur. Comme on peut le voir sur l'image ci-dessous, cela permet de le distinguer aisément le répertoire des commandes. Cela donne un meilleur aspect à notre interface.

A screenshot of a terminal window titled "Xfce Terminal". The terminal shows a user named cedrick on a debian system. The user is in the directory ~/Documents/projet. They run 'ls' and see a list of files: git_commands.txt, Makefile, README.md, shell, and src. The 'shell' file is highlighted in green. Then they run 'httpd' and see a message: [1]+ Stopped httpd. They then run 'cd ..' and 'cd projet' to navigate between directories. Finally, they run './shell' which shows a prompt where the current directory is highlighted in green: cedrick@debian:/home/cedrick/Documents/projet>. The prompt then changes to cedrick@debian:/home/cedrick/Documents/projet> after running 'cd ..' and 'cd projet' again.

```
Applications ▾ Places ▾ Xfce Terminal ▾  
File Edit View Terminal Tabs Help  
cedrick@debian:~/Documents/projet$ ls  
git_commands.txt Makefile README.md shell src  
cedrick@debian:~/Documents/projet$ httpd  
[1]+ Stopped httpd  
cedrick@debian:~/Documents/projet$ cd ..  
cedrick@debian:~/Documents$ cd projet  
cedrick@debian:~/Documents/projet$ ./shell  
cedrick@debian:/home/cedrick/Documents/projet> ls  
git_commands.txt Makefile README.md shell src  
cedrick@debian:/home/cedrick/Documents/projet> cd ..  
cedrick@debian:/home/cedrick/Documents> cd projet  
cedrick@debian:/home/cedrick/Documents/projet> █
```

Capture d'écran de notre shell

IV - PROBLEMES RENCONTRES

Lors de la réalisation du projet, nous avons été confrontés à un certain nombre de problèmes liés à la gestion des pipes. Cela a bloqué notre avancée pendant 5 jours, durant lesquels nous avons tenté d'implémenter une fonction pipe qui stockait les commandes rentrées dans un tableau 3D (char ***). Lors de l'implémentation, cette structure alambiquée s'est révélée être une contrainte. Nous n'avons jamais réussi à mettre en place la gestion des pipes avec cette structure. Nous avons maintes fois amélioré, changé notre code mais cela n'a jamais fonctionné.

Constatant bien malgré nous notre échec à gérer les pipes, nous avons décidé de tout reprendre à zéro. Ainsi, nous sommes repartis sur la gestion des pipes en utilisant cette fois ci un tableau 2D qui ne stocke lui que 2 parties de commandes à la fois. Nous avançons le long de la commande rentrée et les exécutons une par une. Cette restructuration fut salvatrice, elle nous a permis d'avancer dans notre projet et de continuer sereinement sur le reste du projet.

Nous avons tous les deux acquis grâce à ce problème une expérience significative, nous savons désormais qu'il ne faut plus persister à corriger une fonction qui n'est pas pertinente.

En outre, le travail en binôme fut agréable, travailler sur ce projet était une expérience enrichissante.

V - INDIVIDUELLEMENT

CEDRICK RIBET

Grâce à ce projet, j'ai pu maîtriser l'éditeur de texte Vim, éditeur que je n'avais jamais utilisé auparavant. Je suis désormais en mesure d'utiliser cet éditeur dans un cadre professionnel car j'ai pu appliquer tout au long du projet les commandes de cet éditeur de texte.

De plus, ce projet était l'occasion pour moi d'installer un environnement Linux (Debian).

HUGO BOMMART

Au cours de ce projet j'ai pu apprendre à utiliser les pipes entre les processus afin de gérer au mieux les entrées sorties.