C : projet mini-shell

Nicolas Cuadros
Alexandre Nonnon
Antoine Marjault
Romain Andrieux

Introduction

Objectif

L'objectif de ce projet était de recréer le shell Linux avec les commandes suivantes :

- mkdir / cd / pwd
- ls
- cat
- rm / cp / mv
- du
- chmod / chown / chgrp
- echo
- su

Mais aussi avec les fonctionnalités de chaînage de commande suivante :

- Redirection de flux : | , < , << , > , >>
- Opérateurs logiques : | | , &&
- Détachement du terminal : &

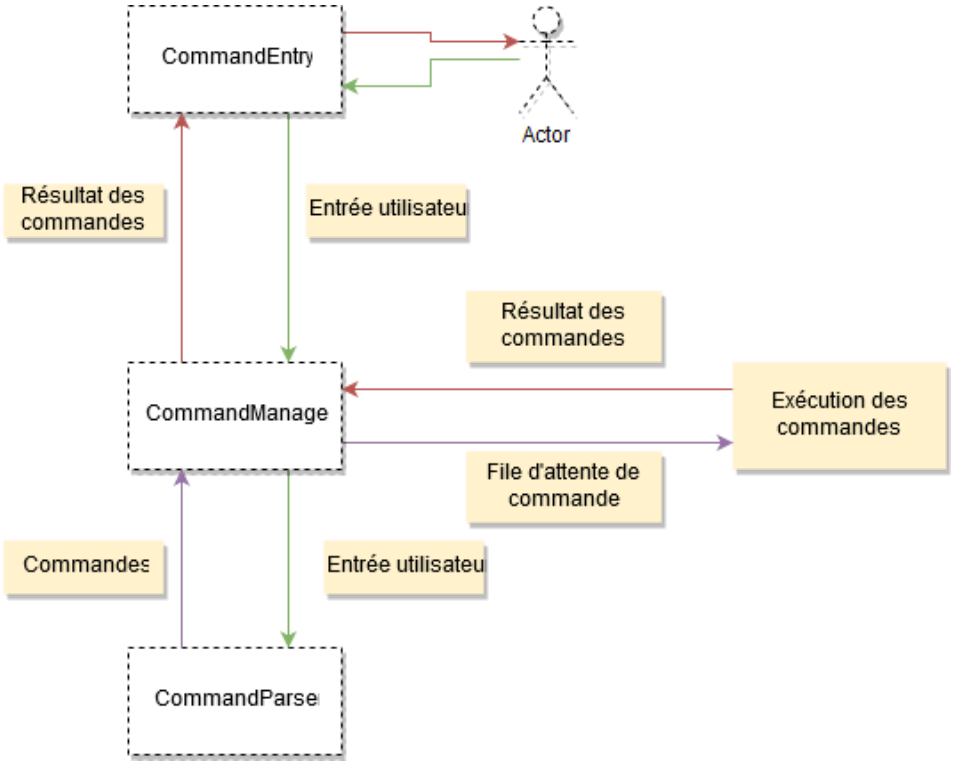
Le shell résultant de ce projet devait pouvoir exister sous 3 formes :

- Version intégrée : toutes les commandes doivent être directement inclues dans le code avec l'interpréteur
- Version libraire : les commandes doivent être stockées dans une librairie chargée par l'interpréteur.
- Version processus : Chaque commande doit être disponible sous forme de processus indépendant de l'interpréteur.

Ces 3 versions devaient être totalement transparentes pour l'utilisateur, chaque version devait fonctionner de la même façon pour lui.
Le shell devait aussi pouvoir exécuter des programmes autres que des commandes.
Une communication de la même manière que le ssh devait être possible via une commande spéciale 'connect'.

Organisation

Afin de mener le projet à bien nous avons scindé les tâches en 2 parties : Interpréteur pour Nicolas et Antoine; Commandes pour Romain et Alexandre.
Au tout début du projet un schéma de fonctionnement à été créé afin de voir comment allaient transiter les informations entrées par l'utilisateur jusqu'au résultat d'une commande :



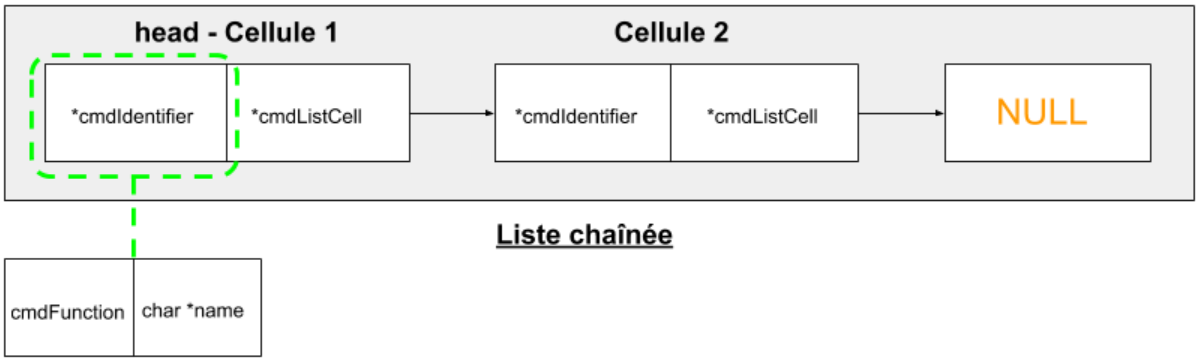
Du côté de l'interpréteur, le but était de diviser le travail en modules, avec chacun leurs fonctionnalités.
Ainsi :

- CommandEntry s'occupe de la saisie de l'utilisateur ainsi que de l'affichage.
- CommandManager s'occupe des différentes étapes de traitement.
- CommandeParser extrait de l'entrée utilisateur les commandes et modes d'exécution.

Structures utilisées

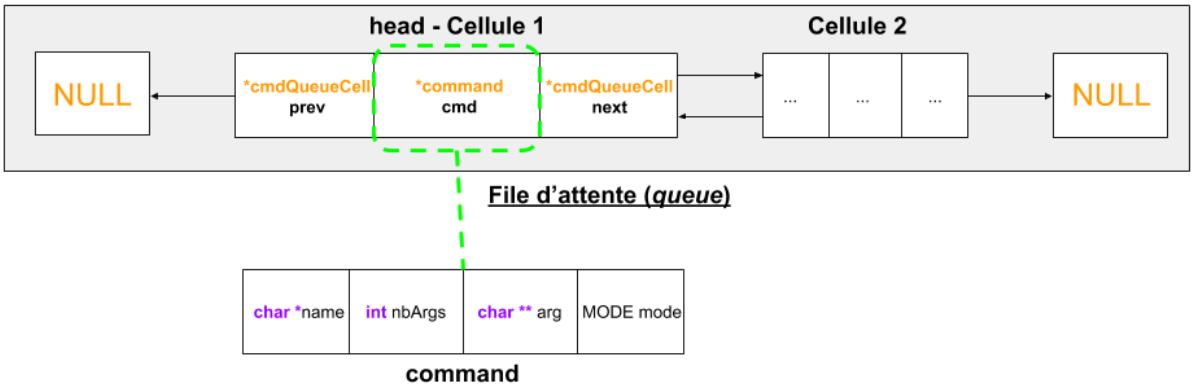
1. *Liste chaînée*

La liste chaînée se présente sous la forme d'une structure *commandList* composée de cellules (*commandListCell*) permettant de stocker des références vers les fonctions du shell ainsi qu'un pointeur sur la cellule suivante (la liste est simplement chaînée).
Les fonctions intervenant sur cette structure sont assez classiques : ajout de commande dans la liste, suppression de la liste, fonction de recherche de commande…
Cette structure nous permet de charger en mémoire des références vers les fonctions du shell (*ls*, *rm*, *chmod…)* à son lancement et d'y accéder facilement grâce à la fonction *find()*.



2. *File d'attente*

La file d'attente (*queue*) nous sert à stocker les commandes parsées. A chaque bout de commande parsée on la met dans la file d'attente grâce à la fonction *pushQ()*. Une fois que la commande a fini d'être parsée, le *command manager* exécute les commandes contenues dans la queue grâce à la fonction *executeQueue()*. A chaque commande exécutée on la dépile de la file d'attente grâce à la fonction *popQ()*.
La file d'attente se présente sous la forme d'une liste *FIFO* (First In First Out), pour simplifier les fonctions de dépilement nous avons décidé de chaîner doublement la liste. Une cellule (*commandQueueCell*) contient donc un pointeur vers la cellule précédente, un pointeur vers la cellule suivante et une commande de type *command* que nous détaillerons après.



3. *Command*

La structure *command* (qui est représentée à la figure précédente) sert à stocker une commande. Elle contient donc le nom de la commande en question, le nombre et la liste de ses arguments et son mode d'exécution. Le mode correspond aux fonctionnalités de chaînage que le shell intègre tel que les redirections de flux, les opérateurs logiques et le détachement du terminal.

4. *CommandParser*

Cette structure sert à stocker un parseur. Elle est composée de la chaîne de caractère qu'a entré l'utilisateur, de la position actuelle dans la chaîne et de *hasEnded* qui indique si la chaîne a fini d'être parsée.

char *chaîne	int actualPosition	int hasEnded

Fonctionnement du shell

Parcours de l'entrée utilisateur jusqu'à l'extraction des commandes.

Une fois le shell lancé, l'utilisateur a la possibilité d'effectuer une saisie de texte pour réaliser différentes opérations représentées par des commandes.
Une fois le texte saisie celui-ci va devoir suivre un parcours avant de pouvoir être interprété.
Tout d'abord la saisie est envoyée au CommandManager qui lui-même va passer par un commandParser chargé d'extraire les commandes du texte saisi.
Chaque commande est composée de cette façon :

nom_de_la_commande argument1...argumentX mode_exécution

Ainsi le commandParser va parcourir le texte jusqu'à l'apparition d'un mode d'éxécution('&','&&,'||','<','<<','>>','>) ou de la fin du texte. Une fois un mode trouvé la commande est enregistrée avec comme premier argument le nom, puis enregistrement des arguments. Cela est répété jusqu'à la fin du texte.

Les commandes extraites par le parser sont renvoyées au commandManager qui les enregistre dans une file d'attente. A l'ajout dans la file d'attente, chaque nom de commande est vérifié afin d'éviter les erreurs lors de l'exécution.
Une fois l'analyse et le parsing terminé la file d'attente est exécutée afin d'obtenir le résultat de la saisie.

Exécution des commandes

Pour exécuter une commande, on extrait et exécute une à une les commandes présentes dans la liste d'attente (FIFO) préalablement remplie par le par commandManager et le Parser. L'exécution se fait donc dans l'ordre de saisie.
Avant d'exécuter une commande, le mode d'exécution est testé afin de déterminer comment celle-ci doit être exécutée, de plus notre shell intègre une fonction qui permet de lancer des exécutables dans le cas où l'entrée de l'utilisateur ne correspond pas à une commande standard (ls, cat…) mais à un fichier exécutable.
Au fur et à mesure de l'exécution l'utilisateur peut voir apparaître le résultat de ses commandes.

Gestion des redirections et de la communication interpréteur/commandes

Un des problèmes majeurs lors de la conception de l'interpréteur était la gestion des résultats des commandes qui devaient pouvoir être redirigés ou affichés. L'utilisation de variables pour stocker les résultats n'était pas recommandée puisque le mode processus empêche la communication directe entre l'interpréteur et les commandes.
Pour palier à ce problème, l'idée a été de stocker les résultats sous forme de fichier de communication (de la même manière que le véritable bash avec stdout).
Ainsi chaque commande n'utilise pas la fonction 'printf' du C mais une fonction print, envoyant les résultats vers un fichier de communication.

Le fichier de communication est généré à partir du PID de l'interpréteur parent, passé en paramètre une variable d'environnement. L'utilisation direct du PPID n'était pas possible puisqu'il fallait détecter le cas où le processus parent n'était pas notre interpréteur.
Le PID du parent permet de créer un dossier portant le nom du PID de l'interpréteur, stocké dans le dossier /tmp du système.
Dans ce dossier il existe 3 fichiers : std, err et log respectivement pour les sorties standard, d'erreur et de processus détaché du terminal.

Ainsi par le biais de cette fonction, les commandes envoient leurs résultats vers ces fichiers qui seront lus par l'interpréteur.
En cas de redirection, std sera envoyé dans la commande suivante en tant qu'argument.
En cas d'opérateur logique, la taille de err sera lue afin de déterminer si une erreur est survenue.
Chacun des fichiers est vidé après chaque exécution de commande, le fichier log est obligatoirement lu à la fin de chaque commande alors que std n'est pas forcément dépendant des redirections de la commande en cours.

Finalité du projet

Fonctionnalités et problèmes recensés

Au niveau des fonctionnalités, le shell remplit presque entièrement le cahier des charges. Premièrement nous avons produit 3 exécutables qui permettent de lancer le shell sous 3 formes différentes : un mode où l'interpréteur et les commandes sont dans des exécutables séparés, un mode avec des commandes intégrées dans un seul exécutable et enfin un mode extensible par librairie qui sont chargées au lancement du shell.
Un prompt est proposé à l'utilisateur pour qu'il puisse entrer ses commandes, puis ces dernières sont analysées et exécutées. Elles peuvent soit être sous forme de commandes intégrées (*ls, cat, rm…)* soit sous forme de programmes exécutables (c'est à dire que l'utilisateur peut lancer ses propres programmes exécutables). Les opérateurs de redirection sont tous pris en compte, que ce soit la redirection de flux, les opérateurs logiques ou le détachement du terminal.
La fonctionnalité *connect()* n'a pas été développée par manque de temps vers la fin du projet.
Au niveau des commandes, elles ont toutes été réalisées, en voici la liste (avec leurs options) :

- *mkdir* / *cd* / *pwd*
- *ls* (avec les options *-l -a*)
- *cat*
- *mv* / *cp*
- *rm* (avec les options *-i -d -f*)
- *du*
- *chmod* / *chown* / *chgrp*
- *echo* (avec l'option *-n*)
- *su*

Parmis les problèmes que nous avons pu rencontrer, un seul persiste et c'est celui qui empêche les programmes exécutés depuis le shell n'utilisant pas notre fonction print, de pouvoir bénéficier des redirections, voire pire cela peut même avoir des comportements assez imprévisibles.

Idées d'amélioration

Afin d'améliorer notre shell nous pourrions ajouter la fonction *connect()* que nous n'avons pas intégrée par manque de temps, elle nous permettrait de se connecter à distance à un serveur par exemple.
L'ajout d'un manuel (*man*) permettrait de mieux appréhender le shell puisqu'il proposerait une documentation pour chaque fonction. Dans la même optique nous avons pensé à ajouter une fonction (de type *--help*) qui permettrait d'afficher toutes les fonctions qu'intègre notre shell.
L'expérience utilisateur (*UX*) pourrait davantage être travaillée en ajoutant quelques fonctionnalités telles que l'autocomplétion (via la touche *TAB*), la sauvegarde des commandes précédentes dans une pile (touche ↑), l'amélioration des couleurs (différentes couleurs en fonction du type de fichier par exemple), ...
Enfin la possibilité d'installer de nouvelles commandes et packages (une sorte de *apt-install*) pour que n'importe qui puisse améliorer le projet (open-source).

Conclusion

Notre shell étant totalement fonctionnel, nous sommes très satisfaits du résultat obtenu d'autant plus que son architecture permet une évolution durable dans le temps, les fonctionnalités étant faciles à ajouter sans retoucher pour autant à la structure du programme.
La réalisation de ce shell nous a permis d'améliorer nos compétences en langage C bien sûr mais aussi de fixer certaines choses apprises en cours de *Système* notamment les appels systèmes, les forks, la gestion des droits, etc.
Enfin dans la continuité du projet de Python de début d'année, la réalisation de ce shell en équipe a permis d'expérimenter le travail en groupe nous rappelant combien la répartition des tâches et la gestion d'un projet est importante.