Rapport technique projet FaroShell

Résumé du projet

Nous sommes 5 étudiants en INFO3 : Arthur Millet / Bastien Jean / Guillaume Potier / Sébastien Berlioux / Simon Bessenay. Ce projet a pour but de créer un mini shell Linux en langage C. Pour cela, on a codé les principales commandes du Shell et utilisé un interpréteur de commande de type bash pour récupérer les saisies de l'utilisateur.
Un README.md est également disponible sur Git pour d'autres informations.

Répartition du travail

Le travail à été réparti entre deux équipes équipes, Simon et Bastien se sont occupés de l'interpréteur, Arthur, Sébastien et Guillaume se sont concentrés sur les différentes commandes à implémenter.

Notre bash le "faroShell"

Pour faire marcher notre bash, dans un premier temps il est nécessaire de télécharger la dernière version du programme, puis à l'aide d'un terminal, de se placer dans le dossier ainsi obtenu et enfin lancer la commande 'make && ./faroShell' : ceci va lancer la compilation de l'entièreté du projet ainsi que l'exécutable créé, il sera utilisable directement. Notre programme a la même façon de fonctionner qu'un terminal standard, une fois ce dernier lancé, il suffit d'exécuter les commandes.

Détails sur chaque commande usuelles

1. **mkdir** [option] chemin

L'option -m est disponible et permet de changer les droits sur le dossier directement à sa création. Nous avons utilisé la fonction C mkdir qui prend en argument le nom du dossier et le mode (les permissions). On récupère et traite les options avec la fonction C optget().

2. **cd** chemin

Ici la commande repose sur la fonction C chdir qui prend en argument le chemin. Avant d'appeler chdir, il faut vérifier que l'on ai le droit d'accéder au dossier. Pour cela, on utilise la structure stat qui nous permet d'obtenir entre autre les permissions du dossier. La commande vérifie si l'utilisateur a le droit de rentrer dans le dossier, si c'est le cas, va modifier le dossier courant. Pour revenir au dossier courant, on récupère l'UID et ensuite le 'mot de passe' qui contient le dossier personnel HOME.

3. **pwd**

La commande repose sur la fonction C getcwd qui prend en argument un tableau de caractères (tampon) et la taille du tampon. Elle renvoie le chemin absolu du répertoire courant dans la variable tampon que l'on affiche ensuite sur la sortie standard.

4. **ls** [option] fichier

Ls a pour effet d'afficher le contenu d'un dossier passé en argument ou alors si aucun argument n'est donné, du dossier courant. Les options disponibles sont : -a qui affiche les dossier "caché", c'est à dire ceux commençant par un point, -l permet d'afficher des détails sur les éléments et pas seulement leur nom et -d affiche les répertoires de la même façon que les fichier sans lister leur contenu. Dans l'option -l on va permettre à l'utilisateur de récupérer plus d'informations concernant les droits d'utilisateurs avec par exemple S_IRUSR, S_IWGRP,... On utilise la fonction isFolder pour savoir si il y a des dossier présent afin de les traités correctement (comme par exemple la couleur verte associé).

5. **cat** [option] fichier

La commande cat affiche le contenu d'un fichier sur la sortie standard. Les options disponibles sont -A qui permet d'afficher le contenu avec les options -E et -T, -E qui affiche un $ à la fin de chaque ligne, -n qui numérote toutes les lignes de sortie en commençant à 1 et enfin -T qui affiche le caractère TAB sous la forme '^I'. Pour ainsi faire, on regarde caractère par caractère le fichier passé en paramètre et on modifie la sortie en conséquent. Tout d'abord, on utilise fopen pour ouvrir le fichier et le stocker dans une variable FILE, ensuite on lit caractère par caractère le fichier avec la fonction C fgetc() qui prend en paramètre la variable FILE. Ainsi, on peut afficher le contenu du fichier en fonction des options. À la fin, on ferme le fichier avec fclose() qui prend en paramètre le variable FILE.

6. **mv** [option] fichier destination

Mv permet de déplacer un fichier, les options disponibles sont -i et -v, -i va demander confirmation pour tout écrasement de fichier et -v va permettre d'afficher les noms des fichiers qui sont déplacés.

7. **cp** [option] fichier destination

Cp est une commande qui permet de copier un fichier, elle prend en arguments le fichier à copier et la destination. Pour copier un dossier le fonction recherche les dépendance du dossier pour les copier aussi. La récursivité a était mise en place (cp_avanced()), c'est à dire que si l'on a un dossier test avec un fichier dedans et un dossier test2 sans rien dedans et que l'on fait un "cp test test2" le fichier dans test sera bien copié dans le dossier test2 comme voulu. On utilise aussi les fonctions estFichier() et estDossier() pour vérifier cela.

8. **rm** [option] fichier

Son fonctionnement est le suivant : dans un premier temps on analyse les arguments qui sont donnés avec la commande, on en extrait les options pour cette fonction, nous n'avons pas utilisé getopt() mais une fonction réalisé par nos soin. Ensuite on vérifie les caractéristiques du fichier cible et si les conditions sont remplies on le supprime. La commande repose sur la fonction C remove() qui prend en paramètre le chemin d'accès à un fichier et ne fait pas de distinction entre les propriétaire et les type de fichier, voilà pourquoi il vérifier cela nous même. Les options disponibles sont -r pour supprimer des dossier et -f pour supprimer des fichiers qui ne nous appartiennent pas.

9. **du** [option] fichier

Cette commande affiche la taille sur le disque qu'occupe le dossier passé en paramètre. Les options -a, -c, et -h sont disponibles. -a permet d'afficher tous les fichiers du dossier passé en paramètres et non pas uniquement les dossiers qu'il contient. -c affiche la taille totale des fichiers contenus dans le dossier et -h permet un affichage plus lisible par un humain en effectuant des conversions et en affichant l'unité. Pour se faire, on accède avec la structure stat, aux données des dossiers pour en récupérer le nom et la taille sur le disque. On utilise la fonction getcwd dans le cas ou on ne précise pas de dossier. On utilise la fonction C opendir pour accéder au dossier souhaité, ensuite, on utilise readdir qui prend en paramètre le dossier ouvert précédemment. Ainsi, on peut utiliser la structure stat afin d'obtenir les informations sur les sous-dossiers. On utilise closedir à la fin afin de refermer le dossier ouvert. On a créé une fonction conversionHumanReadable (qui prend en paramètre la taille du sous dossier, son nom et un indice pour faire les conversions) pour l'option -h qui rend la lecture plus aisée pour les humains.

10. **chmod** [option] mode fichier

La commande chmod permet de changer le mode d'un dossier. On va changer les droits de l'utilisateur ou du groupe ou pour les autres sur le fichier.

11. **chgrp** [option] groupe fichier

Cette commande permet de changer le groupe propriétaire d'un fichier, pour se faire on extrait les statistiques d'un fichier, on cherche l'id du groupe que l'on veut mettre en place et on le remplace dans les stats du fichier. Pour se faire, on commence par tester le fichier à

l'aide de la fonction isRegularFile puis on récupère l'ID du nouveau groupe avec fgets() puis si tout est ok, on change le groupe avec chown().

12. **echo** [option] message(s)

La commande echo permet d'afficher dans le bash les arguments qui lui sont donnés, on extrait les caractères du message et on les affiches dans le bash. Les options disponibles sont -n qui élimine le retour à la ligne en fin de message (\n), et -e qui a pour effet de rendre effectifs les caractère spéciaux * du message. Pour se faire, on regarde caractère par caractère le message et si l'on trouve un '\', on sait qu'il s'agit d'un caractère spécial ensuite donc on peut le traiter en conséquent.

13. **su**

La commande su permet de changer d'utilisateur, si aucun argument ne lui est donné elle considérera cela comme une connection de l'utilisateur "root". La command repose sur la librairie PAM qui permet de gérer l'authentification, la librairie est différente sur mac, linux d'où le test __APPLE__ à un moment du code. Pour valider une authentification on doit lancer une série de fonction pam…() qui, si elles se terminent toutes par un succès permettront de changer d'utilisateur. Cependant, dû à un problème de permission, la fonction su donné avec le code ne fonctionne pas car un des tests se soldera toujours par un échec et heureusement car sinon cela serait une faille de sécurité.

getopt()
Dans la majorité commandes codées, pour traiter les options, on a utilisé la fonction C getopt qui prend en paramètre argc, argv et la une chaîne de caractère contenant les différentes options possibles. Ainsi, on peut gérer avec un switch les options qui peuvent être activées ou non.

Organisation du code

Le répertoire principal comporte le Makefile et le README ainsi qu'un dossiers src qui contient le code source du projet. Celui-ci contient 3 dossiers :
- cmd : contient les dossiers avec les fichiers de code de chaque commandes exécutables dans le shell
- interpreter : contient les fichiers de code nécessaires à l'interpréteur
- utils : contient les fichiers de code des fonctions utiles à tout le projet

À la compilation du projet, 2 dossiers sont créés à la racine du projet :
- obj : contient les fichiers objets générés à la compilation
- bin : contient les fichiers binaires c'est-à-dire les exécutables et les libraires statiques et dynamiques

Le Makefile

Le projet comporte un Makefile principal qui est situé à la racine du projet et permet de compiler la totalité du projet grâce à la commande 'make'. Il est accompagné d'un fichier Makefile.variables qui contient toutes les variables nécessaires à son exécution.

Le Makefile commence d'abord par créer le dossier obj qui va contenir tous les fichiers objets qui vont être générés. Il va ensuite compiler le print personnalisé dans le dossier utils. Pour la prochaine étape, il s'agit de compiler toutes les commandes : pour cela, un autre Makefile à été créé dans chaque dossier de chaque commande afin de créer les librairies statiques et dynamiques de celles-ci ainsi que leur exécutables indépendants. Ces fichiers seront placés dans le dossier bin à la racine. Ces Makefile vont être exécutés grâce à un foreach qui va boucler sur tous les noms de commandes passés en variable dans le Makefile principal.

Enfin, le Makefile fini par créer l'exécutable de notre shell en compilant les fichiers de l'interpréteur pour en faire des fichiers objets qui seront ajoutés à la ligne de commande gcc finale de création de l'exécutable ./faroShell. Cette ligne de commande finale est également accompagné des librairies des commandes grâce à la spécification du dossier dans lequel elles sont placées (option -L) ainsi que leur nom précédé de l'option -l qui est ajouté grâce à au mot clé addprefix de makefile.

Il est également possible de supprimer les fichiers de compilations grâce à la commande 'make clean' qui va supprimer les dossiers bin/ et obj/ créés lors de la compilation.

Ainsi les 3 formes demandées sont possibles :

1. Exécutables indépendants
2. Intégrés à l'exécutable interpréteur de commande
3. Sous forme de librairie dynamique à chargement explicite (L'interpréteur de commande charge l'ensemble des commandes (librairies) depuis un répertoire donné)

Interpréteur de commande de type bash

Le programme affiche un prompt composé d'une ligne avec le nom de l'utilisateur récupérer grâce aux fonctions geteuid() et getpwuid() ainsi que le chemin du répertoire courant récupérer avec getcwd().

Pour chaque ligne de commande rentrée dans notre shell, elles sont lues caractère par caractère avec getchar() puis envoyer vers le parser qui va compter et séparer chaque arguments avec la fonction strtok() de la librairie "string.h". Nous utilisons aussi la commande strcmp() de cette même librairie afin de détecter les caractères de redirection.

Gestion des caractères de redirection :

Le problème principale dans la gestion des redirections est d'enregistrer ce que la première commande affiche dans la console à l'aide de printf. Il faut donc que ces printf écrivent dans un fichier plutôt que dans la console. Pour ce faire, nous avons créé une fonction printf "personnalisé" : faroprint. Celle-ci permet de faire un choix entre écrire dans la console ou alors écrire dans un fichier.

Gestion du PIPE (|)

L'opérateur PIPE sert à renvoyer la sortie d'une première commande vers l'entrée d'une seconde commande. Lorsqu'un caractère PIPE est rencontré dans la ligne de commande, le but du programme est d'exécuter la commande précédant le PIPE en écrivant dans un fichier temporaire (situé dans le fichier "farotmp" dans le dossier "tmp" à la racine de Linux : /tmp/farotmp). Ensuite, à l'aide du code de retour de la commande exécutée, on regarde si celle-ci à réussi ou non. En cas de réussite, on passe une variable à 1 permettant d'enregistrer le fait qu'un PIPE a été détecté. Si la commande a échoué, alors ne lis même pas la suite de la ligne de commande et on quitte la boucle d'interprétation de ligne de commande. On peut donc rentrer une nouvelle ligne de commande. Dans le cas d'une réussite, la boucle continu donc jusqu'à récupérer la commande suivant le PIPE. Celle-ci est donc exécuté en prenant comme argument le fichier créé par la première commande (/tmp/farotmp).

Gestion des chevrons (> , >>)

Le simple chevron droite > (tout comme le double chevron >>) sert à renvoyer le résultat d'une commande dans un fichier. Nous utilisons donc encore une fois la fonction faroprint pour écrire dans le fichier spécifié après le chevron dans la ligne de commande. La détection du chevron suit le même principe que pour le PIPE. Une fois un chevron rencontré, on enregistre une variable pour savoir qu'un chevron a été détecté. La commande n'est exécuté que lorsque le fichier suivant le chevron a été lu.
A la différence du simple chevron, le double chevron permet d'écrire dans un fichier mais sans supprimer le contenu déjà présent dans ce fichier. Pour des raisons techniques, notre bash interprète les simples chevrons comme des doubles chevrons.

Gestion du || et &&

Lorsque le programme détecte un ||, celui ci exécute de manière normale la commande précédant le ||. Si celle ci se déroule sans problème, alors on stop la boucle d'interprétation de la ligne de commande et on passe à une nouvelle ligne de commande. Sinon, on lit la suite de la ligne de commande pour exécuter la commande suivant le ||.
Lorsque le programme détecte un &&, celui-ci exécute de manière normale la commande précédant le &&. Si celle-ci se déroule sans problème alors on passe à la commande suivant le &&. La deuxième commande n'est exécuté seulement si la première commande a réussi. Donc si la première commande ne s'exécute pas, la suite de la ligne de commande n'est pas lue et on passe à une nouvelle ligne de commande.

Pour l'exécution des commandes, nous avons stockées les noms de fonctions à exécuter dans un tableau de pointeur de fonction. Ce tableau est associé avec un tableau de char* contenant le nom des fonctions disponibles. Ainsi lorsqu'une fonction est à exécuter, ce tableau est parcouru dans une boucle for avec la taille du tableau connue grâce à une division de la taille du tableau en octet par la taille d'un char* (fonction sizeof()), afin de savoir si la commande existe. Si ce n'est pas le cas, un print de message le stipulant est effectué.

Bilan final

Notre bash est au final capable de traiter correctement les commande de base qui on été demandé, certaines options sont disponibles pour la plupart des commandes que nous avons codé. Ce projet fut riche en enseignement tout, tout particulièrement sur la rigueur demandé par le langage C, les fonctions disponibles et les possibilités offertes par le langage. Nous n'avons malheureusement pas réussi à mettre en place la partie réseau, et les commandes ont le plus souvent deux ou trois options la ou la "vraie"commande peut en avoir plus d'une dizaine.

Idée d'amélioration :

Nous avions comme idée d'amélioration d'ajouter la fonctionnalité qui permet à l'intérieur du bash de faire la flèche du haut et donc de permettre à l'utilisateur de récupérer les lignes qu'il a tapé précédemment dans le bash (ce qui peut s'avérer très pratique au niveau de gain de temps). Il est aussi tout à fait possible d'ajouter la partie réseau à notre projet, il faudrait pour cela mettre en place une architecture client/serveur avec des messages standardisés pour transmettre les commandes.