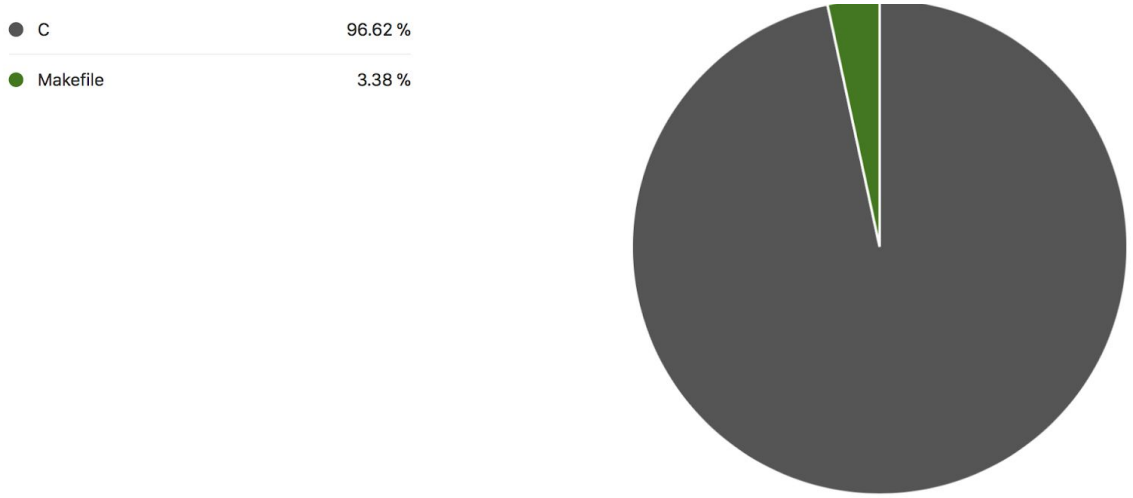
PROJET DE C

Polybash

Membre du groupe :

- **GUITTON Julien (Groupe 3)**
- **SOULEYMANE MOHAMED Ibrahim (Groupe 3)**
- **COUGNAUD Julien (Groupe 3)**
- **MOITEAUX Quentin (Groupe 3)**



Langages utilisés au cours de ce projet

Sommaire

Introduction

1. L'interpréteur

1.1. La compilation et l'exécution

La compilation du projet s'effectue grâce au Makefile présent dans le répertoire du projet. Celui-ci gère les 3 différentes compilations disponibles : avec les commandes incluses, avec les commandes en exécutables séparés et avec les commandes en librairie. Celui-ci va générer d'une part tous les exécutables des commandes ainsi que la librairie qui est placée dans le répertoire */lib* lui aussi créé, mais aussi 3 exécutables différents :

- **cmdr** : cet exécutable correspond à l'interpréteur dans le cas de la compilation des commandes en inclus.
- **cmdr2** : cet exécutable correspond à l'interpréteur dans le cas de la compilation des commandes en exécutables séparés.
- **cmdr3** : cet exécutable correspond à l'interpréteur dans le cas de la compilation des commandes en librairie indépendante.

Dans le cas de la compilation des commandes en tant que librairie indépendante, il est nécessaire d'exécuter la commande :

- **export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$(pwd)/lib**

Afin de permettre à la bibliothèque *dlfcn.h*, permettant le linkage dynamique de librairies, de fonctionner correctement.

Lors de la compilation via le Makefile, il vous sera demandé de fournir votre mot de passe pour permettre de compiler la commande **su** en tant que *root* via *sudo*. Celle-ci nécessite d'avoir comme propriétaire *root* et son setuid bit activé pour pouvoir fonctionner comme attendu.

1.2. Parsage des commandes

Lorsque l'interpréteur est lancé, l'utilisateur est invité à entrer des commandes afin que celles-ci soient exécutées. Pour cela, une boucle *tant que* tournant sur un *getchar()* récupère chacun des caractères fournis par l'utilisateur et les traite de la façon appropriée.

Dans une première version du parseur, on sauvegardait l'ensemble des caractères ce qui formait la commande puis utilisions *strtok()* de la bibliothèque *string.h* pour parser la chaîne de caractères en fonction des espaces et ainsi récupérer les différents arguments.

Mais il est apparu que dans le cas par exemple de la commande ***cd***, il existait des cas où il devait être possible de ne pas perdre la présence d'un espace dans le chemin d'un répertoire. Pour cela, nous avons dû prendre en compte le caractère d'échappement \. Alors il nous était impossible de continuer à utiliser la fonction *strtok()*, nous avons donc dû gérer nous-même le parsage de la commande.

Lorsqu'une commande est entrée, on sauvegarde l'ensemble des caractères fournies sauf exception. Lorsqu'un caractère d'échappement est suivi d'un espace, cet espace est sauvegardé sinon celui-ci ne l'est pas et nous indique que l'on a changé d'arguments. On détecte aussi les différents caractères de redirection (|, >, >>, <, <<) pour les sauvegarder et les traiter correctement.

Comme l'ensemble des arguments se trouve au final dans une seule variable, il était nécessaire de les récupérer. Pour cela à chaque fois qu'un espace était détecté, il était remplacé par le caractère de fin de chaîne *\0*. Puis on fait pointer les différents éléments d'un tableau de *char** vers les différents arguments et le dernier argument pointe toujours vers *NULL* pour qu'un programme exécuté via *execv()* par exemple puisse bien recevoir l'ensemble des arguments ainsi que leurs nombres, cette explication est illustrée par la figure suivante :

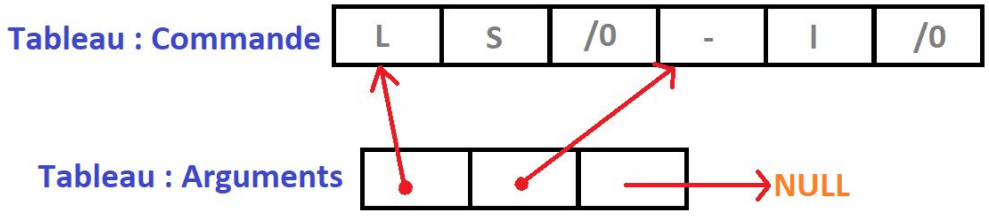


Figure 1 - Schéma du tableau d'arguments

1.3. Pipe et redirection

Lorsque la commande entrée par l'utilisateur a été parsée, on a sauvegardé la position des pipes ainsi que leur type, ainsi on sait quel type de redirection doit être mis en place entre 2 commandes et on peut aussi récupérer l'ensemble des arguments des commandes, vu qu'on sait que de chaque côté d'un pipe se trouve une commande.

Dans notre interpréteur seul le pipe classique représenté par | et la redirection de la sortie standard vers un fichier (> >>) sont implémentés. Pour créer les tubes, on utilise la fonction *pipe()* de la bibliothèque *unistd.h* qui avec un tableau d'entier initialise les 2 premières cases avec l'adresse d'un file descripteur, il est donc nécessaire de le faire plusieurs fois, toutes les 2 cases, s'il y a plusieurs pipes à la suite. Ensuite dans le cas d'un pipe classique avec la commande suivante par exemple : *ls -l* | *cat* | *cat*, l'interpréteur va d'abord via le premier tube rediriger la sortie standard de la première commande, avec *dup2(tubes[1], STDOUT_FILENO)*, vers l'entrée standard de la seconde commande, *avec dup2(tubes[0], STDIN_FILENO)*, puis va ensuite rediriger la sortie standard de la seconde commande, avec *dup2(tubes[3], STDOUT_FILENO)*, vers l'entrée standard de la première commande, avec *dup2(tubes[2], STDIN_FILENO)*.

Comme indiqué précédemment, un tube comporte 2 files descripteurs l'un correspond à l'entrée du tube (tubes[1]) et l'autre à la sortie (tubes[0]). La commande *dup2()* change simplement le file descripteur ainsi, on peut dire que l'entrée standard est en faite l'entrée du tube.

Pour la redirection vers un fichier, on redirige simplement la sortie vers la sortie standard vers le fichier en modifiant le file descripteur de la sortie standard par celui du fichier que l'on obtient grâce à la commande *open()* de la bibliothèque *fcntl.h*.

1.4. Mode connecté : côté client

Pour passer en mode client, c'est-à-dire ne plus envoyer des commandes au terminal mais à un serveur, on utilise la commande connect <adresseIP> <port>. Au début de l'interpréteur; le port d'écoute est indiqué, il suffit alors d'obtenir le port du terminal auquel on veut se connecter et son adresse ip pour se connecter. Une fois en mode connecté, (indiqué par le préfixe "server:" devant le working direct) l'ensemble des commandes entrées par l'utilisateur seront transmises au serveur qui s'occupera de transmettre les informations de retour. Le client s'occupe de transmettre les commandes grâce au socket généré et à la commande *send()*. Ensuite après que le serveur ait effectué les différentes actions nécessaire, le client reçoit les données via le socket et la fonction *recv()*, ces données sont alors affichées sur le terminal pour être rendues visibles à l'utilisateur.

Lorsque l'utilisateur entre les commandes exit ou disconnect la connexion est interrompue tout en informant le serveur de ce fait, pour éviter que le serveur ne reste bloqué dans l'attente de message du client dans le cas où l'utilisateur quitte le terminal en faisant CTRL + C, le signal correspondant est redirigé afin de notifier le serveur de la déconnexion du client avant son arrêt.

1.5. Mode connecté : côté serveur

Chaque terminal, à son lancement, lance dans un processus fils un terminal serveur qui gère toutes les connexions entrantes. Celui-ci, au lancement, entre en attente de la connexion d'un client grâce à la commande *accept()* lorsque cela arrive un file descripteur est associé à cette connection permettant ainsi la communication entre le client et le serveur.

Lors de l'acceptation de la connexion le serveur transmet son répertoire de travail courant afin que l'utilisateur sache où il travaille. Ensuite, le serveur rentre en attente de données via son socket grâce à la fonction *recv()*. Ensuite grâce à un pipe, on redirige la sortie standard dans l'entrée du tube puis on exécute les commandes, pendant ce temps on lit la sortie du tube, pour transmettre les données au client grâce au file descripteur précédent et à la fonction *send()*.

2. Les commandes implémentées

2.1. Commande ls

Syntaxe : ls -[**options**] [**fichiers ou répertoires**]

La commande ls affiche les fichiers donnés en argument et si un fichier est un répertoire alors son contenu est affiché. Quand cette commande est utilisée sans argument alors elle affiche le contenu du répertoire courant. Par défaut, la sortie de cette commande est triée par ordre alphabétique et donne tous les fichiers sauf ceux qui sont cachés c'est-à-dire commençant par un point (.). Les options que nous avons intégrées à cette commande permettant ainsi d'avoir plus ou moins des renseignements sur les fichiers sont :

- -a : le contenu complet du répertoire affiché y compris les fichiers cachés.
- -l : le contenu du répertoire affiché avec les renseignements complets sans les fichiers cachés.

2.2. Commande rm

Syntaxe : rm -[**options**] [**fichier/répertoire ou chemin**]

La commande rm permet la suppression des fichiers ou répertoires donnés en argument. Si le fichier ou répertoire est protégé en écriture et que l'utilisateur lui applique la commande rm alors un message d'accès refusé lui sera renvoyé. L'option **-r** permet de réaliser une suppression en mode récursif dans le cas où le dossier contient des fichiers ou répertoires.

2.3. Commande su

Syntaxe : su **[nom_utilisateur]**

La commande su permet de changer l'utilisateur courant de l'interpréteur en celui spécifié sachant que si aucun nom d'utilisateur n'est fourni, il sera changé en celui de *root*.

Pour que la commande puisse fonctionner, il était nécessaire que celle-ci ait comme propriétaire *root*, que son setuid bit soit activée et qu'elle soit toujours un exécutable (tout cela est géré par le Makefile : cf. *La compilation et l'exécution*). Ces nécessités sont présentes, car pour pouvoir modifier l'uid ou le gid d'un programme via *setreuid()*, il faut être *root,* car uniquement lui peut évoluer vers un rôle inférieur pour des questions de sécurité, ainsi que le setuid bit d'activité.

Grâce au fichier */etc/passwd*, il est possible de récupérer l'uid et le gid en fonction du nom d'un utilisateur ce qui nous permet de l'affecter. Mais le programme ayant comme propriétaire *root*, il était possible de changer d'uid sans aucune contrainte donc il était simple

de devenir *root*. Mais grâce au fichier */etc/shadow*, il a été possible de récupérer les mots de passe associés à chaque utilisateur car le fichier */etc/passwd* ne contient que x en tant que mot de passe. Mais pour des questions de sécurité, ces mots de passe sont cryptés, donc via la fonction *crypt()* de la bibliothèque *crypt.h,* il a été possible de crypter le mot de passe fourni par l'utilisateur via la fonction *getpass()* de la bibliothèque *unistd.h* qui permet de désactiver l'affichage. Pour des questions de sécurisation, le mot de passe non crypté était quant à lui supprimé en remplaçant tous ses caractères par des caractères de fin de chaîne.Il restait donc plus qu'à comparer le mot de passe crypté fourni par l'utilisateur à celui contenu dans */etc/shadow* et à valider le changement d'uid.

2.4. Commande cat

Syntaxe : cat -[**options**] [**fichiers | -**]

La commande cat permet de concaténer un ou plusieurs fichiers et d'afficher les différents contenus sur la sortie standard. Si aucun fichier n'est indiqué, ou si un tiret est indiqué après la commande sera lue l'entrée standard dont le contenu sera affiché à la validation par l'utilisateur (touche Entrée). Pour quitter la lecture de l'entrée standard, l'utilisateur peut utiliser le raccourci ^D qui classiquement va écrire le caractère EOF en entrée standard.

Pour réaliser cette commande, nous avons utilisé les appels systèmes read(), pour lire le contenu des différents fichiers (ou de l'entrée standard), et write() pour écrire ce contenu en sortie standard (provenant de unistd.h).

Nous avons réalisé deux options :

- -E qui affiche un '$' à la fin de chaque ligne.
- -n qui affiche le numéro de chaque ligne affichée.

2.5. Commande du

Syntaxe : du -[**options**] [**fichiers ou répertoires]**

La commande du permet, pour chacun des fichiers ou répertoires passés en paramètres, d'afficher la quantité d'espace disque que chacun d'eux utilise. Lorsqu'un répertoire est indiqué, on va également rechercher des informations sur les fichiers et répertoires qu'il contient récursivement.

La commande du implémentée sous Linux indique cette quantité d'espace disque en nombre de blocs de 510 octets ou 1024 octets sous POSIX ou 512 octets voire 1 Ko sous GNU. Pour notre part, nous avons décidé de choisir des blocs de 512 octets ce qui fait qu'il peut exister des différences avec la commande classique selon la configuration du système de l'ordinateur.

La réalisation de cette commande a nécessité l'utilisation des appels système stat() et lstat(). Ces deux fonctions permettent d'obtenir différentes informations sur un fichier dont le chemin est passé en paramètre, enregistrées dans une structure *stat* qui va contenir un champ nous intéressant : st_blocks (type blkcnt_t) et qui contient le nombre de blocs de 512 octets alloués pour le fichier, d'où notre choix d'utiliser des blocs de cette taille-là. L'utilisation de lstat() permet d'avoir des informations sur un lien symbolique lui-même, et non pas sur le fichier pointé, ce qui nous a permis d'implémenter le fonctionnement de base de la commande du qui, par défaut, affiche les informations sur le fichier lien lui-même.
Les différentes options que nous avons implémentées sont :

- -s : résume les informations d'un répertoire en 1 ligne en faisant la somme de la taille des fichiers et dossiers s'y trouvant.
- -S : pour les répertoires indiqués, ne comptabilise pas les tailles des sous-répertoires dans leur taille.
- -b : affiche la taille des fichiers et dossiers en bytes.
- -a : affiche la taille des fichiers présents dans les répertoires indiqués.
- -c : affiche la somme totale des tailles des différents fichiers / dossiers traités.
- -L : pour les liens symboliques, indique la taille des fichiers pointés (utilisation de stat()).

2.6. Commande chmod

Syntaxe : chmod -[**options**] [**0-7]** [**fichier/répertoire ou chemin]**
La commande chmod permet de changer les autorisations(lecture, écriture et exécution) sur des fichiers ou répertoires. La commande chmod intégrée à notre mini-shell est en mode numérique(valeur octale) et chaque chiffre se rapporte, dans l'ordre de lecture, à l'utilisateur propriétaire, aux utilisateurs du groupe et aux autres utilisateurs:
→ "4" pour le droit de lecture (**r**ead)
→ "2" pour le droit d'écriture (**w**rite)
→ "1" pour le droit d'exécution (e**x**ecute)
On peut effectuer les opérations d'additions sur ces chiffres pour attribuer des autorisations à l'utilisateur, au groupe ou aux autres utilisateurs.
Exemple: chmod 714 monfichier --->donne tous les droits à l'utilisateur, donne le droit d'exécution aux groupes de l'utilisateur et le droit de lecture pour les autres utilisateurs.
Les options que nous avons implémentées sont:

- -R : permet de propager la commande à l'ensemble de la branche de fichiers débutant au répertoire donné en argument.
- -v : permet d'activer le mode verbeux
- -f : permet d'activer le mode force

2.7. Commande echo

Syntaxe : echo -[**options**] [**message ...]**

La commande echo permet d'afficher un texte en sortie standard. Pour cela, nous avons utilisé l'appel système write() pour afficher en sortie standard les différents arguments passés à la commande. Nous avons implémenté l'option -n qui permet simplement de ne pas ajouter un saut à la ligne après l'affichage des différents messages, ce qui est fait par défaut.

2.8. Commande mkdir

Syntaxe : mkdir -[**options**] **répertoire(s)**

La commande mkdir permet de créer un ou plusieurs répertoires s'ils n'existent pas encore.

Pour réaliser cette commande, nous avons utilisé l'appel système mkdir() qui permet de créer un nouveau répertoire dont le chemin est passé en paramètre. Le mode par défaut utilisé lors de la création du répertoire créé est calculé comme ci : ~umask & 0777.

Pour permettre à l'utilisateur de changer le mode par défaut utilisé lors de la création d'un répertoire, nous avons implémenté l'option -m. Ainsi, en indiquant la valeur du mode en mode octal (comme pour la commande chmod), la valeur est calculée ainsi :
mode & ~umask & 0777.
Nous avons également implémenté l'option -v qui permet d'activer le mode verbeux c'est-à-dire d'afficher les dossiers créés au fur et à mesure des opérations réalisées. Mais aussi l'option -p qui nous paraissait très intéressante car cette option permet de créer directement une arborescence de dossiers en indiquant le futur chemin du répertoire le plus bas dans cette arborescence. Par exemple la commande *mkdir -p dossier/ss_rep/ss_rep2* permet, si le dossier "*dossier"* n'existe pas, de créer à la fois ce dossier puis les dossiers "*ss_rep"* et *"ss_rep2"*. Nous avons encore une fois utilisé l'appel système *mkdir*, à plusieurs reprises si nécessaire pour créer les dossiers non existants.

2.9. Commande cp

Syntaxe :

- cp -[**options**] **fichier chemin_fichier**
- cp -[**options**] **répertoire chemin_dossier**
- cp -[**options**] **fichiers… répertoire de destination**

La commande cp permet de réaliser différentes opérations de copie selon les arguments choisis :

- La copie d'un fichier vers un fichier existant (on remplace son contenu), ou non (on le crée dans ce cas-ci) en utilisant dans les deux cas les appels système open (pour ouvrir le fichier ou le créer) et write (pour écrire le contenu du fichier source). Cela correspond à la première syntaxe.
- La copie d'un dossier vers un dossier existant (on copie le contenu du dossier source dans celui de destination) ou non (on crée alors le dossier de destination en y ajoutant le contenu du dossier de destination). En plus des appels système open et write, on utilise également l'appel système mkdir pour créer le répertoire (voire les sous-dossiers) s'il y en a. Cela correspond à la deuxième syntaxe.
- La copie d'un ou plusieurs fichiers et / ou dossiers dans un répertoire existant (troisième syntaxe).

Les options qui ont été implémentées sont :

- -v : pour activer le mode verbeux et afficher les noms des fichiers / dossiers copiés.
- -r : pour permettre la copie des répertoires (récursivement).
- -i : pour demander à l'utilisateur s'il souhaite écraser le fichier / répertoire de destination si cela est nécessaire pour réaliser la copie.

2.10. Commande mv

Syntaxe :

- mv -[**options**] **source(s) cible**

La commande mv permet à la fois de déplacer et de renommer des fichiers. Les opérations sont différentes selon la nature de la cible :

- Si la cible est un répertoire existant, on copie les fichiers et dossiers sources dans la cible sans changer de nom.
- Si la cible n'est pas un répertoire existant, la source et la cible doivent correspondre soit à deux dossiers, doit à deux fichiers. Un renommage est alors réalisé de la source en cible. A noter qu'ici, l'utilisateur ne peut indiquer qu'une seule source, contrairement au cas précédent.

Pour réaliser cette commande, nous avons utilisé l'appel système rename() qui permet de renommer un fichier, voire de le déplacer dans un répertoire si nécessaire. Mais également l'appel stat() qui permet d'obtenir des informations sur les fichiers.

Nous avons implémenté les options suivantes :

- -i qui va demander à l'utilisateur une confirmation si l'opération nécessite l'écrasement d'un fichier existant.
- -u qui va être utile dans le cas d'une sauvegarde de fichiers car cette option va empêcher tout écrasement d'un fichier régulier modifié plus récemment que le fichier qui doit le remplacer.
- -v qui active le mode "verbeux" en indiquant les fichiers / dossiers traités.

2.11. Commandes annexes

2.11.1. whoami

Cette commande récupère l'uid de l'utilisateur via la fonction *getuid()* et récupère le nom de l'utilisateur dans la structure générée via la fonction *getpwuid()* que l'on transmet à l'utilisateur en l'affichant sur le terminal.

2.11.2. commands

Cette commande affiche tout simplement l'ensemble des commandes disponibles dans le terminal.

2.11.3. pwd

Cette commande permet d'afficher le nom du répertoire courant. On a utilisé pour cela l'appel système getcwd qui permet d'obtenir le chemin du répertoire de travail. Puis l'appel système write pour afficher ce chemin sur la sortie standard.

2.11.4. exit

Cette commande comme tout autre commande renvoie une valeur sauf qu'au lieu de renvoyer -1 ou 0 celle-ci renvoie uniquement 2. Cette valeur est interprétée par le terminal comme une indication qu'il doit se stopper. Non seulement, elle permet de stopper un terminal mais aussi la connexion entre un client et un serveur.