Rapport : Projet Mini-Shell

Le but de ce projet est de simuler un shell simplifié, qui doit pouvoir gérer le lancement des commandes (au premier plan ou en arrière-plan), les redirections et les tubes.

PARTIE 1 : L'organisation du main(), le rôle de la fonction commande() et le comportement de la fonction execute()

Comme proposé dans le sujet, j'ai implémenté trois fonctions dans le fichier minishell.c :

int main(int argc, char* argv[])
TOKEN commande(int entree, int sortie, pid_t* pid, int* background)
pid_t execute(int entree, int sortie, char* argv[])

Dans le main, on utilise une boucle infinie qui à chaque itération appelle la fonction commande() pour lire et exécuter chaque commande tapée dans le terminal, tant que l'utilisateur ne sort pas du mini-shell. On passe les variables pid et background (initialisées dans le main) par pointeur à la fonction commande() pour que la valeur du PID du fils et celle indiquant un éventuel passage en arrière-plan (background = 0 si la commande doit être lancée au premier plan, background = 1 si la commande doit être lancée en arrière plan) soient mises à jour par commande().
Après le lancement et l'exécution de chaque commande, on teste la valeur de background. Si la commande a été lancée au premier plan, on fait une boucle avec un wait() pour récupérer tous les processus qui se sont terminés, jusqu'à atteindre la terminaison du processus fils d'exécution de la commande. A ce moment-là seulement on peut redonner la main à l'utilisateur. Si la commande est lancée en arrière-plan, on l'exécute et on redonne la main à l'utilisateur immédiatement après avoir lancé l'exécution. On affiche le prompt mini-shell> uniquement si le token renvoyé par la fonction commande() est T_NL, c'est-à dire un retour-chariot.

La fonction commande() appelée dans le main a pour but de lire chaque mot de la commande saisie par l'utilisateur, et s'il n'y a pas d'erreur, de l'exécuter et de renvoyer au main le token final. Chaque mot lu est stocké dans le tableau tabArgs. L'exécution de la commande se fait à travers la fonction execute().

Dans execute(), on duplique le processus via un fork() : le père est chargé de renvoyer le PID du fils tandis que le rôle du fils est d'exécuter la commande. On utilise pour cela la fonction execvp(), en utilisant le premier mot lu par commande() comme le nom de l'exécutable à chercher.

PARTIE 2 : Traitement d'une commande simple et du point-virgule dans la fonction commande()

Pour exécuter une commande simple, on utilise la fonction getToken() de l'analyseur lexical pour lire chaque mot de la commande et ainsi remplir tabArgs. Le tableau tabArgs a été déclaré statiquement avec une taille ARGS_MAX mais afin de ne pas gaspiller trop de mémoire, on alloue dynamiquement la place pour chaque mot lu (avec malloc) avant d'y recopier un mot (avec strcpy()).
Le premier mot qu'on extrait est le nom de l'exécutable. Puis, lorsque l'analyseur lexical détecte le retour chariot, le mini-shell doit exécuter la commande en appelant la fonction execute(). Les arguments donnés à cette fonction sont l'entrée et la sortie de la commande en cours, ainsi que la commande elle-même (dans tabArgs). Enfin, si l'exécution s'est déroulée sans problème, commande() retourne T_NL.
Si on rencontre un point-virgule,on le traite comme un retour-chariot car il sépare deux commandes distinctes. On renvoie T_SEMI et dans le main(), on peut continuer à lire le reste des commandes sans toutefois réafficher le prompt.

PARTIE 3 : Traitement de l'attente

Si la commande ne se termine pas par un &, on exécute le processus fils,et on attend que le fils se termine avec l'appel systeme wait() avant de redonner la main à l'utilisateur dans le main(). Si la commande se termine par & (token T_AMPER pour la mise en arrière-plan), on exécute la commande mais on ne fait pas le wait() immédiatement. On passe par pointeur la valeur 1 pour background au main(), et le shell continue son exécution.

Il faut également prendre en compte qu'une commande comportant un & ne se termine pas nécessairement après le &, elle peut être suivie d'une autre commande. Au début, je voulais lire le token suivant pour savoir s'il s'agissait d'un token T_NL (la commande serait alors terminée) ou un T_WORD (il y aurait une autre commande sur la même ligne). Mais en faisant cela, j'aurais déjà lu le prochain token et donc peut-être le prochain mot si la commande continuait. J'aurais donc potentiellement faussé la lecture de la commande suivante. C'est pourquoi je procède autrement : on se contente d'exécuter la commande qu'on a lue jusqu'au & et de retourner T_AMPER.
À la prochaine commande, si le mot qui suivait le & est un T_WORD, tout s'exécute normalement. Mais si c'est un T_NL, le tabArgs sera vide, on arrive directement au retour chariot. Donc dans le cas T_NL, on teste la taille du tableau. S'il est vide, on retourne directement T_NL, ce qui entraîne dans le main() l'affichage du prompt et une autre commande peut-être lue.

PARTIE 4 : Traitement des redirections

Afin de gérer les redirections, on ajoute deux arguments à la fonction commande(), de telle sorte qu'elle reçoive de main() les descripteurs de l'entrée et de la sortie standard (initialement 0 et 1). Dans commande(), on initialise deux variables fdin et fdout avec les valeurs d'entrée et de sortie reçues de main(). Si on rencontre les tokens T_LT, T_GT ou T_GTGT, on fait une nouvelle fois appel à getToken() pour lire le nom du fichier à ouvrir. On ouvre alors ce fichier avec open() et on met à jour fdin ou fdout selon le cas. Si on est dans le cas T_LT, on cherche à rediriger l'entrée donc on ouvre le fichier avec les modes O_RDONLY| O_CREAT afin de lire le contenu du fichier, et le créer s'il n'existe pas. Si on est dans le cas T_GT, on veut rediriger la sortie de la commande dans le fichier en écrasant son contenu donc on ouvre le fichier avec les modes O_WRONLY|O_TRUNC|O_CREAT. Dans le cas T_GTGT, on veut aussi rediriger la sortie dans le fichier mais en écrivant à la suite de son contenu : on ouvre donc le fichier avec le O_APPEND au lieu de O_TRUNC.

On peut alors appeler execute(), fonction à laquelle on passe fdin et fdout. Dans le processus fils de execute(), si le descripteur de fichier d'entrée et/ou de sortie est différent du descripteur standard, on utilise close() pour fermer le descripteur de fichier standard puis on emploie dup2() pour redéfinir l'entrée et/ou la sortie. On peut alors continuer l'exécution normale de la commande avec execvp().

PARTIE 5 : Traitement du tube

Dans la continuité des redirections, les tubes entraînent des modifications des descripteurs de fichier d'entrée et de sortie des commandes. On commence par lire la partie gauche de la commande jusqu'à rencontrer un token T_BAR puis on crée un tube et on appelle la fonction execute() pour exécuter la partie gauche avec redirection de la sortie dans le tube en écriture. Pour traiter la partie droite, on appelle récursivement la fonction commande() avec comme entrée le tube en lecture. Initialement je n'avais fait de deuxième fork() : je commençais par créer un pipe puis j'appelais execute() en remplaçant les descripteurs de fichier d'entrée/sortie par les extrémités écriture/lecture du pipe. Cependant cette méthode posait des problèmes au niveau de la fermeture des extrémités non utilisées du pipe.

J'ai donc changé de stratégie et je me suis inspirée d'un code trouvé sur StackOverflow (https://stackoverflow.com/questions/17166721/pipe-implementation-in-linux-using-c/17637791) que j'ai adapté car certaines redirections étaient déjà effectuées dans execute(). On refait donc un fork() dans commande() pour créer un nouveau processus. Dans le processus fils, on appelle execute() pour exécuter la partie gauche avec redirection de la sortie dans le tube en écriture. Dans le processus parent, on attend que le processus fils ait écrit dans le tube puis on traite la partie droite, en appelant récursivement la fonction commande() avec comme entrée le tube en lecture.
Pour finir, on renvoie récursivement le dernier token lu pour que toutes les fonctions commande () se terminent. Lors de l'appel récursif à commande(), on choisit la sortie standard comme sortie par défaut, qui sera modifiée si on rencontre un nouveau pipe par la suite. Cela permet que la sortie de la dernière commande soit bien la sortie standard, le terminal.

PARTIE 6 : Gestion des erreurs

6.1. Libération de la mémoire et codes d'erreur des appels système

En premier lieu, il faut penser à libérer la mémoire à chaque fois que commande() se termine ou à chaque erreur qui entraîne la terminaison de la fonction commande(). La fonction malloc() a uniquement été utilisée pour allouer dans tabArgs une place à chaque nouveau mot lu. Il s'agit donc seulement de libérer tabArgs avec la fonction free().

Ensuite, toutes les fonctions utilisées (wait, waitpid, pipe, open, fork, execvp, dup2) peuvent renvoyer un code d'erreur en cas de problème. Il est donc aussi nécessaire de vérifier chacun des appels à ces fonctions pour éventuellement écrire un message d'erreur.

6.2. Dépassement de mémoire

La troisième erreur à gérer est celle d'un dépassement de mémoire lorsqu'une commande comporte trop d'arguments. La longueur de tabArgs est fixée par la macro ARGS_MAX. Il faut donc compter les arguments lus et si on dépasse ARGS_MAX, on affiche un message d'erreur, on libère tabArgs et on continue à lire les arguments jusqu'à rencontrer un retour-chariot ou un point-virgule. En effet, si on arrête de lire les arguments dès que leur nombre dépasse la taille du tableau et que commande() retourne directement, le prochain appel à commande() posera problème. On continuerait à lire les arguments surnuméraires de la commande précédente au lieu de lire la vraie nouvelle commande().
Dans le cas où on rencontre une erreur avant l'exécution de la commande, il faut également penser à mettre la variable background à 1, sinon le main() attend indéfiniment la fin d'un processus fils qui n'a jamais été créé.

La dernière erreur à gérer est également un dépassement de mémoire, cette fois lors de la lecture d'un mot par getToken(). La variable word est déclarée statiquement avec une taille de TAILLE_MAX dans commande(). Cependant on ne peut pas vérifier que le mot lu est de la bonne longueur dans commande(), c'est pourquoi j'ai modifié la fonction getToken() et j'ai créé un token T_OVERFLOW qui signale à commande() que le mot est trop long . Il faut donc déclarer la macro TAILLE_MAX dans analex.h.

PARTIE 7 : Réflexions sur d'éventuelles fonctionnalités supplémentaires

7.1. Colorer le prompt pour qu'il soit plus visible

On modifie la fonction print_prompt() en remplaçant l'affiche initial par "\033[1;36mmini-shell> \ 033[0m". Le 1 permet de mettre le texte en gras, et 36 permet de mettre le texte en bleu. Ainsi chaque affichage du prompt est bleu et en gras, ce qui le rend plus visible.

7.2. Réflexion sur l'implémentation de l'auto complétion (non implémentée dans ce projet)

Il faudrait reconnaître qu'il y a tabulation : cela est simple, il suffit d'utiliser son code ASCII ou la chaîne '\t'. Toutefois, cela impliquerait de modifier getToken() car cette fonction utilise la fonction isspace() qui permet de ne pas considérer différents types d'espaces dont la tabulation. Cela pourrait fausser la lecture pour des commandes normales dont les mots sont espacés par des tabulations.

De plus, il faudrait utiliser grep pour chercher tout ce qui commence par ce qui a déjà été tapé : si c'est un nom de fichier, on cherche dans le dossier actuel, sinon il faut chercher dans les différents dossiers où se trouvent les noms des commandes.
D'ailleurs certaines commandes sont fournies directement par le shell donc certaines commandes classiques ne fonctionnent pas dans ce mini-shell.

7.3. Réflexion sur la récupération des commandes précédentes avec les touches directionnelles (non implémentée dans ce projet)

Plusieurs problèmes se posent : tout d'abord, il faudrait conserver un certain nombre de commandes déjà exécutées en mémoire. Cela supposerait donc de déclarer un tableau de commandes dans le main(), soit un tableau de type char***. Cela complique considérablement les choses car il faudrait le passer par pointeur à commande() et pouvoir le modifier malgré les appels récursifs à la fonction commande().

D'autre part, il faudrait aussi être en mesure de reconnaître lorsque l'utilisateur utilise les flèches du pavé directionnel. Or les quatre flèches ont le même code ASCII. J'ai donc cherché dans une autre direction et j'ai trouvé un code sur StackOverflow qui permet d'identifier chaque flèche (https://stackoverflow.com/questions/10463201/getch-and-arrow-codes). En effet, il se trouve que la flèche du haut est représentée par ^[[A et celle du bas par ^[[B. Il s'agirait donc d'utiliser getchar() pour lire le esc ^[, puis le [, puis le A ou le B. Cette méthode fonctionne mais l'utilisateur est obligé d'appuyer sur entrée après avoir utilisé une flèche directionnelle, ce qui est très peu naturel par rapport à un vrai shell.

Conclusion

Ce mini-shell est capable de traiter le lancement des commandes (au premier plan ou en arrière-plan), les redirections et les tubes. Étant une version simplifiée d'un shell, l'auto-complétion et la navigation dans l'historique de commandes n'ont pas été implémentées.