Deleuze-Dordron Axel
Grousson Dylan

Projet mini-shell
Compte rendu

Organisation générale:

La première chose que nous avons faite a été de mettre en place un git pour ce petit projet, afin de pouvoir y accéder facilement depuis n'importe quelle machine et de pouvoir travailler sur des branches séparées.
Cependant, ayant principalement travaillé ensemble en présentiel, nous avons surtout utilisé la fonction LiveShare de Visual Studio Code et nous travaillons à deux sur l'implémentation de telle ou telle partie.
Nous nous sommes réparties certaines tâches, comme par exemple l'implémentation de la gestion spéciale du `^C` spécifique pour notre shell pour Dylan tandis qu'Axel a créé des tests.

Modularisation:

Nous avons découpé notre code en six grande parties:

- `interpreter.c :` contient la fonction permettant l'exécution des commandes reçu par la fonction de lecture donnée, ainsi qu'une fonction auxiliaire pour gérer correctement les pipes.
- `internal_cmd.c:` contient les fonctions d'exécutions des commandes internes prisent en compte par notre mini shell
- `handler.c :` contient nos différents handlers de signaux.
- `error.c :` contient de courte fonctions appelées en cas d'échec d'une primitive système ou d'une fonction.
- `list.c :` contient tout ce qui est relatif à l'implémentation et la gestion des listes, utilisées pour la gestion des processus en foreground et background.
- `shell.c :` contient le main et donc la boucle principale de notre mini-shell

Choix d'implémentation:

Gestion des commandes:

Au niveau de l'exécution des commandes dans notre Shell, nous avons fait le choix pour modulariser au mieux notre code de passer par une fonction qui gère le ^D et qui permet de rediriger la commande vers les fonctions d'exécution de commandes internes et enfin vers la fonction exécutant les commandes externes.

Deleuze-Dordron Axel
Grousson Dylan

Gestion des pipes multiples:

Sachant qu'on a le nombre de commandes sur une ligne, on ouvre ce nombre moins un de pipe que l'on stock dans un tableau pseudo dynamique, le nombre de pipe ne variant que d'une ligne de commande à l'autre. Une fois cela fait, chaque fils va appeler peu après sa création la fonction `pipes_handling`, qui s'occupe de fermer, en fonction de la place de la commande correspondante dans la ligne de commande, les bonnes entrées ou sorties des tubes, ainsi que de faire les dup2 nécessaires. Enfin, le père ferme toutes les entrées/sorties des tubes.
L'implémentation de la fermeture des tubes multiples a été la partie la plus compliquée que nous ayons géré. Nous avons hésité à l'origine entre cette version et une autre, où chaque tube ne serait connu que des deux processus qui les utiliseraient. Cependant, cette deuxième possibilité s'est révélée assez compliquée, pour des gains en termes d'efficacité qui sont dérisoires.

Processus en background:

Nous avons modifié la fonction de lecture des lignes de commandes ainsi que la structure contenant lesdites lignes afin de prendre en compte la possibilité de lancer des commandes en arrière-plan. Cela pose désormais le problème de la captation de la terminaison du signal de mort de ses processus, qui passe donc par un handler. Seulement, il faut que notre handler puisse aussi gérer le cas où il capte un signal provenant d'un processus mort en foreground et qu'il nous rende éventuellement le prompt si il n'y a plus de processus en foreground vivant.
Pour cela, nous avons créé deux listes chaînées, pour avoir une structure de données dynamique, l'une stockant les pids des processus en foreground (`fg_list`) et l'autre les pids des processus en background (`bg_list`). Chaque signal `SIGCHLD` est capté par le handler `sigchild_handler` qui va alors faire un `wait()` et ensuite essayer de retirer le pid du processus terminé de la liste des processus en foreground. Si il n'appartient pas à cette liste, la fonction de retrait va renvoyer 0 et il cherchera donc à le retirer de la liste des processus en background.
Dès que la liste des processus en foreground est vide, le prompt nous est rendu.

Tests:

A l'ajout d'une nouvelle fonctionnalité, nous créons un nouveau jeu de test et lui donnons un nom explicite, puis nous le faisons passer à notre mini-shell, en plus des tests précédents. Cela nous permet de nous assurer que notre programme reste compatible avec ses fonctionnalités antérieures, tout en traitant correctement ses nouvelles capacités.
Pour exécuter les tests, nous passons par `driver.pl` afin de faciliter la tâche.
Une remarque est que nous n'avons pas réussi à créer de fichier de test concluant pour vérifier la gestion différenciée du `^C`. Les tests à la main montre que cette dernière

fonctionne - ie les fils meurent tous et notre shell reste en vie - mais lorsque nous utilisons le fichier de test ctrl_c.txt, nous avons deux comportements, soit rien n'est affiché et le shell meurt, soit il affiche les deux ps montrant qu'aucun processus fils n'est mort et boucle à l'infini dans le vide.

Fonctionnement global:

Au début de notre fonction main, nous initialisons nos listes globales, créons une variable pour contenir les lignes de commande et assignons des handlers aux signaux SIGCHLD et SIGINT.
Nous entrons ensuite dans la boucle principale, qui affiche le prompt, lit une ligne de commande et appelle la fonction `cmd_intern_extern`. Si la première commande de notre ligne est une commande interne, alors la fonction correspondante est appelée et nous rendons le prompt. Sinon, on appelle la fonction exec_cmd_line qui est chargée de l'exécution des lignes de commandes externes.

`Exec_cmd_line` compte tout d'abord le nombre `n` de commandes contenues dans la ligne puis alloue un tableau `n-1` tableaux de deux entiers. A l'aide de ce dernier, nous créons les `n-1` pipes séparant nos commandes.
Ensuite pour chaque commandes, nous faisons un fork:

- Dans le cas du fils, nous vérifions pour la première ou la dernière commande s' il y a respectivement un fichier d'entrée ou de sortie et appelons la fonction `pipes_handling` pour configurer correctement nos pipes. Une fois cela fait, on fait un execvp de la commande avec ses arguments.
- Dans le cas du père, nous ajoutons à la liste des processus en foreground ou en background les commandes, en fonction de leur mode d'exécution.

Lorsque le père quitte la boucle, il ferme tous les pipes et attend que la liste des processus en foreground soit vide avant de retourner et nous rendre ainsi le prompt.