INP ENSEEIHT
TOULOUSE

Projet : Mini Shell

Première Année, Département SN

Hamza MOUDDENE

May 15, 2020

QUESTION 1-5

L'objectif de ce projet est de réaliser un minishell robuste, simple et efficace, afin de réaliser ceci, j'ai utilisé une boucle infini, à chaque itération de cette boucle lit une ligne sur l'entrée standard grace à la fonction **readcmd()**, puis le programme crée un process fils à l'aide de la fonction **fork()**, l'interprète avec **execvp()**. Le processus shell lance un fils, puis se met immédiatement en attente de lecture de la prochaine ligne de commande, il s'est avéré que l'affichage de l'invite précède ou se mele à l'exécution du processus fils, pour résoudre ce problème j'ai rajouté dans le tratement du processus père le fait que le père attend la fin de l'exécution du fils en utilisant **wait()**.
Après j'ai implementé deux commandes internes, **cd** en se servant d'une fonction native du langage C dite **chdir()** en gérant bien évidemment tous les erreurs probables qui peuvent accompagner l'exécution de cette commande, aisi que la commande **exit** ou **quit** qui se fait juste avec un **exit()** qui arrete le shell, après j'ai choisit de rajouter la commande UNIX **clear** en se servant de la fonction **printf()**, j'ai aussi implementé la possibilité de lancer des commandes en tache de fond, c'est à dire, les commandes qui se terminent par le caractère **&**, en utilisant **structure cmdline* cmd** qui contient toute la commande, la différence entre une commande en tache de fond et un commande en avant plan, c'est que la prèmière est lancée sans que le processus père attend la fin de son exécution, alors que la deuxième le processus shell attend bien la fin de l'exécution du processus fils.

QUESTION 6-7

Dans la suite de ce projet, j'ai implementé la commande **list** en utilisant une liste chainée puique nous savons pas le nombre de processus que nous allons ajouté dans la liste des processus, ce qui laisse l'ajout dans la liste de processus assez souple, et enfin la commande **list** consiste à afficher cette liste.
la commande **stop** est implémenté avec la commande **kill(pid, SIGSTOP)** en mettant le processus concerné en suspension, pour la commande **bg** consiste à utilisé kill pour mettre le processus en tache de fond et la commande **fg** est implementé avec la commande **wait**.
Pour le traitement du signal SIGINT, j'ai utilisé **handler_SIGINT(int sig)** puis dans la fonction **main()** j'ai utilisé **signal()** afin de gérer le signal SIGINT.

QUESTION 8

J'ai modifié le programme pour permettre d'associer l'entrée standard ou la sortie standard d'une commande à un fichier, la solution proposée consiste à

vérifier si la commande contient **¿**, si c'est le cas, je dirige la sortie standard de la commande vers le nom de fichier stocker dans **cmd-¿out**, j'ai procède de la meme façon pour les commandes contenant **¡** sauf que cette fois je dirige l'entrée standard vers le fichier dans le nom est donnée dans **cmd-¿in**, la question a été réaliser grace à la fonction **dup2**.

QUESTION 9-10

D'abord, j'ai commencé à implémenter la première question, qui consiste à mettre en place l'implantation de l'éxecution d'une commande qui contient un seul pipeline, ceci se fait simplement en ayant un process fils et process sous fils, par exemple si on a la commande : **ls -l** — **wc -l**, alors le sous fils va exécuter **ls -l** et le fils va terminer cette exécution en exécutant le reste de la commande, grace à cette question, j'ai remarqué que l'algorithme peut se factoriser d'une manière élégante en le rendant plus générique et plus souple. Alors j'ai choisit de faire une fonction **exec_pipelines** qui se base sur un algorithme récursif qui exécute n'implorte quelle commande contenant 1, 2 ou une infinité de pipe d'une façon simple et efficace en utilisant la question 9.