Minishell

Ce projet consiste à réaliser un shell simplifié en C, il n'est pas parfait mais il est fonctionnel. Certaine partie de l'implémentation n'est pas optimale mais elle fonctionne (cf pipes).

Lancer une commande élémentaire

Question 1

Pour cette première partie, j'ai rajouté la possibilité de lancer des commandes élémentaires. J'ai pu tester avec la commande `ls` et `pwd` .

```
> ls
> Makefile  minishell.c  minishell.pdf  readcmd.c  readcmd.h  test  test_readcmd.c
> pwd
> /home/evann/dev/minishell
```

Synchronisation entre le shell et ses fils

Question 2

Ici lors de tests élémentaires, nous observons que le prompt attend instantanément la prochaine commandes meme si la commande précédente n'est pas terminée. Ce qui pour un shell n'est pas le comportement standard.

```
> ls
> Makefile  minishell.c  minishell.pdf  readcmd.c  readcmd.h  test  test_readcmd.c
```

Question 3 : Enchaînement séquentiel des commandes

Ici nous avons testé avec la commande `ls` et `pwd` , on remarque que des a présent le prompt n'est affiché qu'après la fin de l'exécution de la commande, nous permettant d'obtenir un comportement plus standard.

```
> ls
Makefile  minishell.c  minishell.pdf  readcmd.c  readcmd.h  test  test_readcmd.c
```

>

Question 4 : Lancement de commandes en tâche de fond

Nous voulons maintenant pouvoir lancer des commandes en tâche de fond en utilisant le caractère `&`. Après avoir rajouté la gestion de ce caractère, nous obtenons le comportement attendu suivant :

```
> sleep 10 &
> sleep 5
...
> # au bout de 5 secondes (non bloquant)
```

Nous observons bien que la première commande s'effectue en arrière plan (non bloquante) ce qui permet l'execution d'une seconde commande en premier temps pour laquelle le shell attendra la terminaison de celle-ci avant d'afficher un nouveau prompt.

Gestion des processus lancés depuis le shell

Question 5 : Gérer les processus lancés depuis le shell

En utilisant un handler pour le signal SIGCHLD, nous pouvons determiner si un processus fils s'est terminé et afficher un message en conséquence (terminaison / erreur / suspension / reprise). De plus, chaque pid de processus fils est stocké dans une liste chainée pour pouvoir visualiser les processus en cours d'exécution. Nous avons également d'utiliser le pid comme identifiant côté minishell pour chaque processus.

Ainsi, nous obtenons le comportement suivant :

```
> sleep 50 &

> lj
pid: 95049, cmd: sleep, status: RUNNING
> sleep 20 &

> lj
pid: 95097, cmd: sleep, status: RUNNING
pid: 95049, cmd: sleep, status: RUNNING
> sj 95097
> Le processus 95097 a été stoppé par le signal 20

> lj
pid: 95097, cmd: sleep, status: SUSPENDED
```

```
pid: 95049, cmd: sleep, status: RUNNING
> bg 95097
> Le processus 95097 a été relancé
Le processus 95097 s'est terminé normalement avec le code 0

> lj
pid: 95049, cmd: sleep, status: RUNNING
> sj 95049
> Le processus 95049 a été stoppé par le signal 20
> fg 95049
Le processus 95049 s'est terminé normalement avec le code 0

>
```

Question 6-7: SIGINT et SIGTSTP

Ces deux signaux ne devant pas être traité par le minishell mais par les processus fils, nous ne devons plus les traiter dans le minishell. Pour ce faire nous avons décider de masquer ces signaux dans le minishell et de démasquer tout les signaux dans les processus fils tout en changeant sont pgid pour le pid du processus fils lorsque celui-ci est lancé en arrière plan.

```
> sleep 10
^CLe processus 74738 s'est terminé anormalement avec le code 2

> sleep 10 &

> ^C # aucun effet
> Le processus 74834 s'est terminé normalement avec le code 0
^C
> # le minishell fonctionne toujours
```

Même test avec le signal SIGTSTP :

```
> sleep 10
^ZLe processus 74738 a été stoppé par le signal 20

> sleep 10 &

> ^Z # aucun effet
> Le processus 74834 a été stoppé par le signal 20
^Z
> # le minishell fonctionne toujours
```

Gestion des redirections

Question 8

Ensuite en utilisant les données de la structure `cmd` nous pouvons gérer les redirections de fichiers. En utilisant les fonctions `dup2` et `open` nous pouvons rediriger les entrées et sorties des commandes.

```
> ls > test.txt
> cat test.txt
Makefile  minishell.c  minishell.pdf  readcmd.c  readcmd.h  test  test_readcmd.c
> cat < test.txt > test2.txt
> cat test2.txt
Makefile  minishell.c  minishell.pdf  readcmd.c  readcmd.h  test  test_readcmd.c
```

Tubes

Question 9-10

Ici, en implémentant la gestion des tubes, nous pouvons chaîner les commandes entre elles. Cependant l'implémentation actuelle chaine les commandes de manière séquentielle, ainsi si un tube est rempli cela risque de poser problème lors de l'exécution de la commande suivante. Cependant, nous pouvons observer que cela marche correctement pour des commandes simples.

```
> ls | wc
      7       7      84 # le chainage fonctionne
> ls | wc | wc
      1       3      24 # le chainage fonctionne
```

Evann DREUMONT - 2023/24