SYSTÈME D'EXPLOITATION CENTRALISES : RAPPORT PROJET MINISHELL

XAMBILI Robin

Table des matières

1 INTRODUCTION

Le but de ce projet est de développer un interpréteur de commande simplifié offrant les fonctionnalités de base des shells UNIX, comme le bash.

2 ARCHITECTURE ET CHOIX DE CONCEPTION

2.1 RENDU

L'archive rendue contient :
Le code final minishell.c
Les fichiers jobs.h et jobs.c gérant les listes
Les fichiers readcmd.c et readcmd.h fournis de base
Le rapport du projet.

2.2 QUESTIONS

2.2.1 Question 1 (Lancement d'une commande)

On réalise une boucle infinie (do ... while(1) ;) dans laquelle on utilise la fonction readcmd() fournie pour lire la commande. Ensuite, on crée un processus fils (à l'aide d'un fork()) dans lequel on exécute la commande à l'aide de la primitive exec().

2.2.2 Question 2 (Exemple)

```
$ ls
$ minishell  readcmd.c  minishell.c  readcmd.h  LisezMoi.html  jobs.c  LisezMoi.md  jobs.h
```

On observe que le "$" est placé au mauvais endroit c'est-à-dire avant le résultat du "ls".

2.2.3 Question 3 (Enchaînement séquentiel des commandes)

Afin que le minishell attende la fin de la dernière commande avant de passer à la lecture de la ligne suivante il faut attendre que le processus fils se termine avec la primitive wait().

2.2.4 Question 4 (Commande Internes)

On ajoute une fonction commandeInterne(struct cmdline *cmd) qui retourne 1 si une commande interne a été exécuté 0 sinon.

Implémentation des commandes internes :
— cd : on utilise la fonction chdir(), on met le dossier personnel si aucun chemin n'est passé en argument,
— exit : on quitte le minishell avec exit(EXIT_SUCCESS).

2.2.5 Question 5 (Lancement de commande en tâche de fond)

On teste si la commande doit être exécuté en tâche de fond si c'est le cas alors le processus père n'attend pas la fin du processus fils.

2.2.6 Question 6 (Gérer la suspension des processus)

On ajoute la gestion d'une liste permettant de stocker les processus non terminés. On ajoute les options WUNTRACED et WCONTINUED à la primitive waitpid() permettant de détecter les changements d'états des processus fils. On ajoute le processus à la liste dans les cas suivant :
— Processus suspendu
— Processus en tâche de fond

De plus, on change le traitant du signal SIGCHLD à l'aide de la primitive signal(). Le traitant de SIGCHLD épuise tout les changements d'états grâce à l'option WNOHANG de la primitive waitpid() et traite les changements d'états.

Les commandes internes cont, stop et jobs sont ajouté à la fonction commandeInterne() :

— cont : envoie le signal SIGCONT au processus concerné, change son état dans la liste et attend qu'il se finisse à l'aide de la primitive waitpid()
— stop : envoie le signal SIGSTOP au processus concerné et change son état dans la liste
— jobs : affiche tous les processus non terminés.

2.2.7 Question 7 (SIGINT et SIGTSTP)

On ajoute un traitant pour SIGINT et SIGTSTP qui envoie respectivement le signal SIGQUIT et SIGSTOP au processus en avant plan et on masque les signaux SIGINT et SIGTSTP pour les processus fils.

2.2.8 Question 8 (Redirections)

On ajoute la fonction redirection() permettant de rendre le code plus lisible.

Si l'attribut in fourni par readcmd() est non nul alors on ouvre le fichier passé en argument et on redirige l'entrée standard du processus vers le fichier à l'aide de la primitive dup2(). Enfin, on ferme le fichier avec close().

Si l'attribut out fourni par readcmd() est non nul alors on ouvre le fichier passé en argument et on redirige la sortie standard du processus vers le fichier à l'aide de la primitive dup2(). Enfin, on ferme le fichier avec close().

2.2.9 Question 9 (Tubes simples)

Dans le cas d'une commande en tube simple, on crée un tube à l'aide de la primitive pipe() puis on crée un processus fils dont l'entrée standard est redirigé vers la sortie du tube p[0] et qui exécute la deuxième commande. Le processus père redirige sa sortie standard vers l'entrée du tube p[1] et exécute la première commande. On pense bien à fermer les fichiers ouvert pour éviter les blocages.

2.2.10 Question 10 (Pipelines)

On ajoute la fonction exec_pipelines() améliorant la lisibilité du code.

On commence par compter le nombre de tubes nécessaires, puis on crée les tubes à l'aide de la primitive pipe(). Puis on fait une boucle for dans laquelle on crée un processus fils dont la sortie standard et l'entrée standard sont redirigées sur les tubes puis on exécute les commandes. On pense à fermer les fichiers ouverts pour éviter les blocages.

3 MÉTHODOLOGIE DE TEST

3.1 Lancement de commandes

On lance différentes commandes et on compare le résultat obtenu avec le résultat attendu. Exemple :

```
$ ls
minishell  readcmd.c  minishell.c  readcmd.h  LisezMoi.html  jobs.c  LisezMoi.md  jobs.h
```

3.2 Commandes internes

Exemple :

```
$ cd ..
$ ls
minishell/ minichat/
$ exit
```

3.3 Tâche de fond

On lance des commandes en arrière plan et on regarde si c'est effectivement le cas.
Exemple :

```
$ ls&
$ minishell  readcmd.c  minishell.c  readcmd.h  LisezMoi.html  jobs.c  LisezMoi.md  jobs.h
```

3.4 Gestion de suspension des processus

On ajoute différent processus à la liste des jobs et on l'affiche.
Exemple :

```
$ sleep 10&
$ jobs
[1] ACTIF sleep 10
$ cont 1
$ jobs
$ sleep 10
^Z
$ sleep 5
^Z
$ jobs
[2] SUSPENDU sleep 5
[1] SUSPENDU sleep 10
$ cont 2
^C
$ jobs
[1] SUSPENDU sleep 10
$ sleep 10&
$ jobs
[2] ACTIF sleep 10
[1] SUSPENDU sleep 10
$ cont 1
[2] FINI
$ jobs
$ exit
```

3.5 Redirections

On fait des redirections et on affiche le contenu des fichiers.
Exemple :

```
$ ls > test
$ cat test
minishell
readcmd.c
minishell.c
readcmd.h
LisezMoi.html
jobs.c
LisezMoi.md
jobs.h
$ cat < test > test2
$ cat test2
minishell
```

```
readcmd.c
minishell.c
readcmd.h
LisezMoi.html
jobs.c
LisezMoi.md
jobs.h
$ exit
```

3.6 Tubes

On lance des commandes en tubes et on compare le résultat obtenu avec le résultat attendu. Exemple :

```
$ ls -l | wc -l
19
$ cat minishell.c | grep int | wc -l
42
$ cat minishell.c | grep int | grep e | wc -l
24
$ exit
```