



Systèmes d'Exploitations Centralisés

(mini)Projet (mini)Shell

Rapport

1ère Année, Département Sciences du Numérique

Hamid Oukhnini

2020-2021

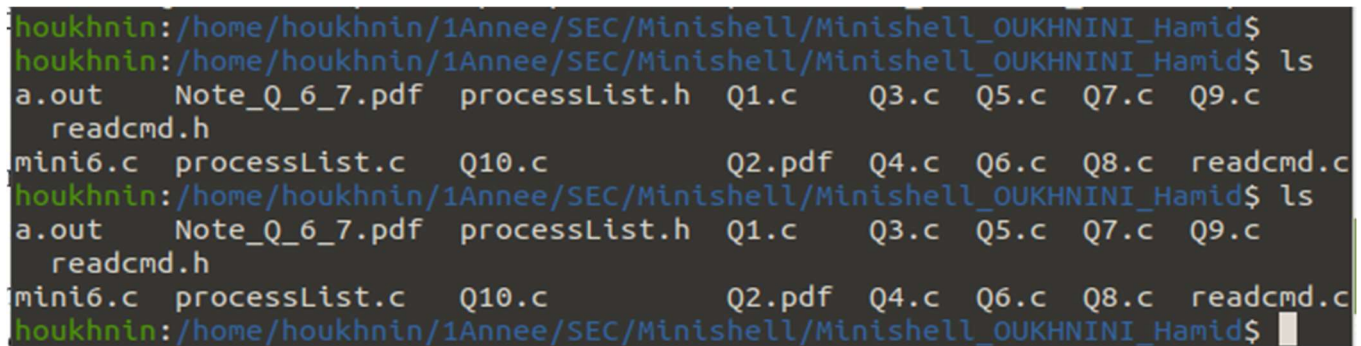
Question 01

Réalisation de la boucle de base de l'interpréteur, en se limitant à des commandes simples (pas d'opérateurs de composition), sans motifs pour les noms de fichiers.

L'implantation de cette question était assez simple : Il fallait d'abord une boucle infinie (while(1)) au sein de laquelle la création des processus était exécutée. En testant le retour des appels systèmes, et en ajoutant, dans la partie du fils, les commandes

```
execvp(cmd->seq[0][0], cmd->seq[0]);    //Executer la commande
exit(EXIT_SUCCESS);                    /* Terminaison normale (0 = sans erreur) */
```

Question 02



```
houkhnin:/home/houkhnin/1Annee/SEC/Minishell/Minishell_OUKHNINI_Hamid$
houkhnin:/home/houkhnin/1Annee/SEC/Minishell/Minishell_OUKHNINI_Hamid$ ls
a.out  Note_Q_6_7.pdf  processList.h  Q1.c  Q3.c  Q5.c  Q7.c  Q9.c
readcmd.h
mini6.c processList.c  Q10.c          Q2.pdf  Q4.c  Q6.c  Q8.c  readcmd.c
houkhnin:/home/houkhnin/1Annee/SEC/Minishell/Minishell_OUKHNINI_Hamid$ ls
a.out  Note_Q_6_7.pdf  processList.h  Q1.c  Q3.c  Q5.c  Q7.c  Q9.c
readcmd.h
mini6.c processList.c  Q10.c          Q2.pdf  Q4.c  Q6.c  Q8.c  readcmd.c
houkhnin:/home/houkhnin/1Annee/SEC/Minishell/Minishell_OUKHNINI_Hamid$
```

(a) Question 2

Figure 1.1: Mise en exergue du fait que l'affichage de l'invite se mêle à l'exécution du processus fils.

L'ajout des commandes suivantes `printf("houkhnin:");`, `printf("%s\n", getcwd(s,200));` avant le `fork()` engendre le problème remarqué dans la figure ci-dessus.

Question 03

Modification du code afin qu'il attende la fin de la dernière commande lancée avant de passer à la lecture de la ligne suivante.

Il suffisait d'ajouter `wait(NULL)` dans la partie exécutée par le père.

Question 04

Ajout de deux commandes internes, exécutées directement par l'interpréteur sans lancer de processus fils : `cd` et `exit`.

Pour ce faire, il fallait contrôler la saisie de ces commandes avant même la création du processus fils avec une simple structure `if.. else if...`

Question 05

Ajout de la possibilité du lancement de commandes en tâche de fond, spécifié par un & en fin de ligne. Pour achever cela, il fallait n'exécuter wait(NULL) que si l'attribut backgrounded de la structure cmdline était NULL, i.e., si la commande n'était pas en tâche de fond.

Question 06

Pour l'implantation de la commande jobs, j'ai dû créer un module processList d'une liste *dynamique* (et non une liste chaînée comme vu dans les cours de programmation). Ce module est en charge de l'ajout de processus, la mise à jour de leur état, leur suppression ainsi que leur affichage.

Tout d'abord, je définis 2 nouveaux types qui sont:

- enum pStatus
- struct pList

L'énumération pStatus contient les éléments Done, actif et suspendu, question d'imiter le vrai shell. La structure pList contient 4 attributs qui sont:

- La liste de PID int* pIDs
- La taille de la liste int size
- La liste des états de chaque processus pStatus* pStatuses
- La liste des commandes ayant engendré chaque processus char** commands

Ce module contient plusieurs procédures pour l'ajout, la suppression ; etc. Pour imiter le comportement de la primitive jobs du shell, la procédure d'affichage dans le module processList est void myJobs(pListprocessList).

Cette procédure affiche tous les processus. Si un processus a fini son exécution, il est signalé comme Done et son affichage avec la procédure Jobs le supprime de la liste de processus. C'est-à-dire, si je lance sleep 5&et sleep 30&, alors la saisi de list après 5 secondes donne:

```
[1]      5994      Done      sleep 5
[2]      5995      actif     sleep 30
```

Une deuxième exécution de list après que les 30 secondes soient écoulées affichera :

```
[2]      5995      Done      sleep 30
```

Les processus finis sont ainsi affichés une seule fois pour que l'utilisateur le sache puis supprimés de la liste de processus.

Pour l'exécution de jobs, fg, bg et stop, il fallait d'abord les ajouter dans la structure if.. elseif.. de la Question 4, s'assurer que chaque processus lancé en arrière-plan est ajouté à la liste de processus puis ajouter un handler du signal SIGCHLD, suivi_fils (Inspiré du tutoriel de Mr. Hamrouni sur les Signaux).

Ce dernier met à jour-là l'état des processus dans la liste de commandes (Done si WIFEXITED, suspendu si WIFSTOPPED ou WIFSIGNALED; etc.)

Pour ce qui est de la commande stop, on envoie le signal SIGSTOP au processus.

Pour reprendre un signal arrêté dans l'arrière-plan avec la commande bg, on envoie le signal SIGCONT sans attendre le fils.

Puisque l'attente du fils imite son exécution en avant-plan, c'est ce que j'ai fait pour implanter le processus

bg avec la commande wait(PID, 0, 0).

Puisque le signal SIGTSTP doit arrêter le processus en avant-plan sans quitter le minishell, j'ai fait ignorer le signal SIGTSTP aux fils avec signal(SIGTSTP, SIG_IGN et j'ai ajouté un autre handler de signal suivi_pere pour SIGTSTP qui, à la réception de ce signal,

envoie le signal SIGSTOP au processus. Sauf que puisque le père attend la fin d'un processus quelconque à cause de la commande wait(NULL) (Question 5) et que le processus est suspendu, nous ne revenons jamais au miniShell.

J'ai donc été obligé de créer un fils à la réception du signal SIGTSTP que je tue immédiatement. Ceci libère le père de la commande wait(NULL), et l'envoi du signal SIGSTOP au processus en avant-plan le suspend puis nous fait revenir au miniShell.

Question 07

Comme pour le signal SIGTSTP, j'ai fait ignorer aux fils le signal SIGINT avec la commande signal(SIGINT, SIG_IGN) et j'y ai assigné le handler suivi_pere (où je teste bien sûr si le signal vient d'un CTRL+Z ou CTRL+C). Le processus fils reçoit alors le signal SIGKILL et est supprimé de la liste de processus.

Question 08

Pour cette question, il fallait utiliser les attributs in et out de la structure du module readcmd. L'implantation, qui a été faite dans la partie du fils (après le fork), était assez simple :

```
if in != NULL && err == NULL
```

```
    Rediriger stdin vers le descripteur du fichier in if out !=  
    NULL && err == NULL
```

```
    Rediriger stdout vers le descripteur du fichier out
```

Il faut bien sûr aussi tester les retours des dup2 et des opens si jamais ces commandes engendrent une erreur. L'exécution de la commande est assurée par l'ancien execvp.

Question 09

. Puisque, pour cette question, on suppose qu'il n'y a qu'un seul pipe, il fallait alors, après avoir testé l'existence de redirections, vérifier si cmd->seq[1] était NULL; i.e., vérifier s'il y avait un pipe dans la commande.

Le cas échéant, il fallait créer un petit fils qui se chargera d'exécuter la première partie de la commande en redirigeant son résultat vers un pipe hérité de son père, puis rediriger stdin vers l'entrée du pipe dans la partie du père et exécuter la seconde partie de la commande

Question 10

Dans cette partie, je vérifie d'abord s'il y a un seul pipe et compte le nombre de pipes le cas échéant.

Je crée alors un tableau de pipes dont j'ouvre et ferme ceux dont j'ai besoin au sein d'une boucle while. Tant qu'on n'est toujours pas arrivé à la dernière commande, un nouveau petit fils est créé. Si on est à la première itération (première commande), on ne redirige pas stdin, sinon on le redirige vers le résultat de la commande précédente.

Si on est à la dernière commande, on ne redirige pas stdout, sinon on le redirige vers l'entrée de la commande suivante.

Après ces redirections en utilisant dup2, je ferme tous les pipes et exécute la commande dans la partie du petit fils. Toujours dans la boucle mais dans la partie du fils (père du petit fils), j'incrmente le compteur de commandes et celui des pipes. Une fois toutes les commandes exécutées, je ferme tous les pipes et attend que tous les petits fils finissent leur tâche avant de tuer leur père pour revenir au miniShell.