# Rapport Projet Minishell

Arthur Jaumet

## Table des matières

# Introduction

Le minishell est constitué d'un fichier minishell.c et d'un module readcmd.h fourni. Le but du minishell est qu'une fois exécuté il émule le comportement d'un shell tel bash. A la fin du projet, le minishell doit être capable de gérer les processus lancés depuis celui-ci, de gérer les signaux reçus, de gérer la redirection des entrées/sorties avec des fichiers ou avec des tubes.

# Conception

## Méthode

La méthode utilisé a été de suivre les questions et pour chaque question se rapporter au TP/Tuto fourni sur moodle. Cependant la gestion du ctrl-Z et ctrl-C ont été remise à plus tard. De plus chaque appel système est testé mais j'ai également essayé de prévenir toute erreur qui se produirait dans des fonctions internes à mon minishell.

## Questions 1-4

### Tâches réalisées

Réalisation de la boucle de base de l'interpréteur, on affiche le chemin courant puis on lit ce que rentre l'utilisateur grâce à readcmd(). Ensuite on vérifie que l'utilisateur n'ait pas rentré de command interne au minishell (ex: exit et cd) pour enfin lancer un processus fils qui lance la commande à l'aide de execvp().

### Tests

- ls -> affiche correctement les fichiers du répertoire courant
- sleep 10 -> attend 10 secondes avant de lire la commande suivante
- exit -> quitte le minishell
- cd x -> permet de changer de répertoire, si x est nul le répertoire devient /home

## Question 5-6

### Tâches réalisées

Gestion des processus lancés depuis le shell, permet une gestion plus simple des commandes lancés en tâche de fond avec également une gestion des signaux envoyés aux différents processus. Des commandes internes au minishell pour la gestion de ces processus appelés jobs. Pour la gestion de ctrl-C et ctrl-Z voir la fin du rapport.

### Fonctionnement

Les jobs sont gérés de la manière suivante:

```
typedef struct processus{
     int id_shell;
     pid_t pid;
     char status; //'A' active et 'S' stopped
     char command[1024]; //fixe mais de grande taille
     int ended; //initialisé à 0, égal 1 si terminé/interrompu
  }p;
  p jobs[1000];
  int nb_jobs = 0;
```

jobs est une variable globale (un tableau) dans laquelle nous allons ajouter les informations des processus à chaque nouveau lancement. Pour des raisons de facilité de développement les capacités pour une commande et le nombre de job sont fixes. De plus, lorsqu'un processus se termine il n'est pas supprimé de la table jobs mais ended passe à 1.

list est la commande qui permet d'afficher le tableau jobs, on y affiche l'ID interne au minishell, le PID, l'état (actif/suspendu) et la commande. Si le processus a terminé son exécution, c'est à dire que ended = 1 alors on ignore l'affichage de ce processus.

Pour les commandes stop, bg et fg, elles prennent l'ID du processus en paramètre. Bien évidemment une série de test est là pour vérifier que l'ID passé en paramètre soit conforme.

### Tests

- stop -> vérifier que le bon processus passe de l'état 'A' à 'S' à la fois avec list mais aussi réellement (utilisation de la commande "ps fj" dans un terminal)
- Envoie du signal SIGSTOP -> mêmes vérifications que pour stop
- bg -> vérifier que le bon processus passe de l'état 'S' à 'A'
- fg -> même fonctionnement que bg mais bloquant

## Question 8

### Fonctionnement

Que ce soit pour la redirection de l'entrée ou de la sortie standard, le fonctionnement est le même. Si une redirection a été indiqué dans la commande rentré par l'utilisateur alors on ouvre le fichier correspondant soit en lecture (dans le cas de l'entrée) soit en écriture (dans le cas de la sortie) avec l'appel système open(). Puis on associe l'entrée standard avec le fichier en lecture et la sortie standard avec le fichier en écriture avec l'appel système dup2().

### Test

- "cat < f1 > f2" -> simple vérification que l'on retrouve bien le contenu de f1 dans f2

## Questions 9-10

### Fonctionnement

Lorsqu'une commande est rentrée par l'utilisateur, on compte le nombre de "sous-commandes" (séparées par un '|'). On peut en déduire le nombre de tubes et les initialiser dans un tableau. Ensuite pour chaque "sous-commande", on va créer un nouveau processus pour laquelle on va rediriger sa sortie standard vers la suivante et l'entrée vers la précédente. Il faut également fermer tous les tubes non utilisés. Tout cela est géré grâce à la fonction close_pipes().

### Tests

- “cat f1 f2 | grep int | wc -l” fonctionne
- “cat < f1 | grep int | wc -l > f2” fonctionne, donc les redirections marchent mais seulement pour la première sous-commande (avec l’entrée) et la dernière sous-commande (avec la sortie).

## Question 6-7 (ctrl-C et ctrl-Z)

### Fonctionnement

On recouvre les deux signaux associés qui seront gérer par un handler (handler_chld()). De plus, on va enregistrer le pid du processus en avant plan dans une variable fg_pid. Lorsque l’on envoie un signal SIGINT ou SIGTSTP, ils sont interceptés et on envoie un signal semblable (SIGUSR1 et SIGSTOP respectivement) au processus en avant-plan à l’aide de fg_pid. On ignore les signaux (SIGINT et SIGTSTP) pour les processus en fond de tâche.

### Tests

Pour les tests, l’idée est de vérifier le comportement des processus à l’envoie de SIGINT ou SIGTSTP. Par exemple si un processus est en background, qu’il ne perçoive pas le signal. Ensuite que les signaux ont les bons comportement que ce soit une commande classique (sleep 100) ou bien après un fg. La plupart des tests ont été effectué avec un terminal à côté avec la commande sleep.

# Conclusion

Au cours de ce projet, j’ai pu manipuler différents appels systèmes pour émuler un shell (type bash). De la gestion des processus aux redirections et tubes en passant par l’utilisation des signaux, j’ai pu mieux comprendre comment fonctionne le système d’exploitation UNIX d’un point de vu logiciel. Pour aller plus loin, on pourrait imaginer ajouter tout l’ergonomie (accès aux commandes précédentes, gestion de script) déjà accessible à travers un shell tel bash.