
Rapport Projet Système d'exploitation centralisé

Auteur :

VIGNAUX Adrien

Professeurs :

CAZANOVE Cédric

ERMONT Jérôme

Projet de première année de SN

année : 2024-2025

Table des matières

1. Introduction	2
2. TP 1 : Exécution de commandes	2
3. TP 2 et 3 : Gestion des signaux	2
4. TP 4 : Gestion des fichiers et redirections	3
5. TP 5 : Gestion des tubes	4
6. Conclusion	5

1. Introduction

Le projet consiste à développer un shell à partir d'un squelette de code. Le shell doit être capable d'exécuter des commandes, de gérer les processus et de gérer l'enchaînement de commandes grâce à des tubes. Le projet est divisé en plusieurs étapes, chacune visant à ajouter des fonctionnalités spécifiques au shell.

2. TP 1 : Exécution de commandes

Dans cette première étape, nous nous sommes intéressé à la capacité d'exécuter des commandes simples. Nous avons utilisé la fonction *fork()* pour créer un processus fils dans lequel nous avons exécuter les commandes avec *execvp()*. Nous avons également géré l'attente du processus fils avec *waitpid()* pour éviter les zombies.

De plus nous avons géré l'utilisation de commandes en background (ex: *&sleep(10)*). Pour cela, nous avons utilisé le processus fils créer précédemment et un booléen pour savoir si la commande que l'on exécutait était en premier plan ou non. Ainsi le processus père attendait la fin du processus fils si la commande était en premier plan, sinon il ne l'attendait pas.

```
avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$ ./minishell
> sleep 10
commande : sleep Le processus fils de pid : 1259491 vient de se terminer avec le code 17
> &sleep 10
commande : sleep
> exit
Au revoir ...
avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$
```

Fig. 1. – Exécution de commandes simples

3. TP 2 et 3 : Gestion des signaux

Dans cette étape, nous avons ajouté la gestion des signaux pour permettre au shell de réagir aux interruptions et aux arrêts sans que le programme minishell n'en soit impacté. Nous avons utilisé la fonction *sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction)* pour gérer les signaux SIGINT (Ctrl+C) et SIGTSTP (Ctrl+Z). Lorsque l'utilisateur envoie un signal d'interruption, le shell doit afficher un message et continuer à fonctionner. Pour le signal d'arrêt, nous avons mis en place un traitement qui permet de suspendre le processus en cours.

Cependant cette gestion ne doit pas s'appliquer aux processus en arrière plan. Pour cela nous avons fait des groupes pour que les signaux envoyés au shell ne soient pas envoyés aux processus en arrière plan.

```
avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$ ./minishell
> sleep 10
^Ccommande : sleep
Signal d'interruption envoyé au processus fils de pid : 1260765 avec le signal 2
> &sleep 10
commande : sleep
> ^C
> exit
Au revoir ...
avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$
```

Fig. 2. – Exécution de commandes avec gestion des signaux

Nous avons également mis en place un traitement pour le signal SIGCHLD afin de gérer la terminaison des processus fils. Car nous n'utilisons plus *waitpid()* puisque sinon on attends aussi les processus en arrière plan. Nous utilisons donc *pause()* dans le processus père que si notre booléen foreground est vrai. Sinon on ne fait rien et on laisse le processus fils s'exécuter en arrière plan.

4. TP 4 : Gestion des fichiers et redirections

Dans cette étape, nous avons ajouté la gestion des fichiers et des redirections pour permettre au shell de rediriger l'entrée et la sortie des commandes. Nous avons utilisé les fonctions *open(const char *path, int oflag)* et *dup2(int oldfd, int newfd)* pour gérer les redirections des commandes et l'ouverture ou la création de fichier si nécessaire.

```
avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$ ./minishell
> ls
Makefile minishell minishell.c minishell.o non.txt oui oui.txt readcmd.c readcmd.h readcmd.o test_readcmd.c
commande : ls Le processus fils de pid : 1262853 vient de se terminer avec le code 17
> cat oui.txt
absolument d'accord
commande : cat Le processus fils de pid : 1262917 vient de se terminer avec le code 17
> cat non.txt
non
commande : cat Le processus fils de pid : 1262960 vient de se terminer avec le code 17
> cat < oui.txt > non.txt
commande : cat Le processus fils de pid : 1263262 vient de se terminer avec le code 17
> cat non.txt
absolument d'accord
commande : cat Le processus fils de pid : 1263323 vient de se terminer avec le code 17
> cat < oui.txt > peut.txt
commande : cat Le processus fils de pid : 1263508 vient de se terminer avec le code 17
> cat peut.txt
absolument d'accord
commande : cat Le processus fils de pid : 1263562 vient de se terminer avec le code 17
> exit
Au revoir ...
avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$
```

Fig. 3. – Exécution de commandes avec gestion des redirections

De plus nous avons implémenter les commandes *cd* et *dir*. Ces commandes permettent de changer le répertoire courant et d'afficher le contenu d'un répertoire. Nous avons utilisé la fonction *chdir(const char *path)* pour changer le répertoire courant et *opendir(const char *name)* pour ouvrir un répertoire. Nous avons également utilisé la fonction *readdir(DIR *dirp)* pour lire le contenu du répertoire et afficher les fichiers et sous-répertoires.

```

avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$ ./minishell
> ls
Makefile minishell minishell.c minishell.o non.txt oui oui.txt peut.txt readcmd.c readcmd.h readcmd.o test_readcmd.c
commande : ls Le processus fils de pid : 1263853 vient de se terminer avec le code 17
> cd oui
> ls
non
commande : ls Le processus fils de pid : 1263959 vient de se terminer avec le code 17
> cd
> ls
Arduino Bureau Documents eclipse-workspace Images MesDocs Modèles Musique Public snap Téléchargements Vidéos
commande : ls Le processus fils de pid : 1264051 vient de se terminer avec le code 17
> dir Bureau
.
..
TOB_MN01
1er_année
N7
> dir
.
..
.mozilla
.arduino15
Téléchargements
.xsession-errors.old
.gnatstudio
.gps
.dotnet
.wget-hsts
.vscode
.Xauthority
.matlab
Images
Vidéos
.nv
.tooling
.gnupg
.config
MesDocs
.local
.gitconfig
Public
.my-credentials
.ipython
.MathWorks
Documents
.swt
.viminfo
.als
.nfs0000000001217f61000005f3
Musique
.xsession-errors
.vscode-server

```

Fig. 4. – Exécution de commandes avec gestion des redirections et des commandes `cd` et `dir`

5. TP 5 : Gestion des tubes

Dans cette étape, nous avons ajouté la gestion des tubes pour permettre au shell de chaîner plusieurs commandes ensemble. Nous avons utilisé la fonction `pipe(int pipefd[2])` pour créer un tube et rediriger l'entrée et la sortie des commandes à l'aide de `dup2(int oldfd, int newfd)`. Notamment lorsque la chaîne est plus longue que 2 commandes, nous avons utilisé un entier qui prend la valeur de sortie du tube de la commande précédente et qui est redirigé vers l'entrée de la commande suivante. Cependant, le résultat de l'enchaînement de commande se fait avant l'affichage de la dernière commande.

```

avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$ ./minishell
> ls | wc -l
commande : ls Le processus fils de pid : 1266729 vient de se terminer avec le code 17
12
commande : wc Le processus fils de pid : 1266730 vient de se terminer avec le code 17
> ls | grep oui | wc -l
commande : ls Le processus fils de pid : 1266978 vient de se terminer avec le code 17
commande : grep Le processus fils de pid : 1266979 vient de se terminer avec le code 17
2
commande : wc Le processus fils de pid : 1266980 vient de se terminer avec le code 17
> exit
Au revoir ...
avx5062@kernighan:~/Bureau/N7/SEC/TP1/minishell$ █

```

Fig. 5. – Exécution de commandes avec gestion des tubes

6. Conclusion

Dans l'ensemble, le projet a été une expérience enrichissante qui nous a permis de mieux comprendre le cours et de l'appliquer dans un cas plus concret. Nous avons notamment appris à gérer les processus, les signaux et les redirections dans un shell. Nous avons également appris à utiliser des tubes pour chaîner plusieurs commandes ensemble. Le projet nous a permis de mieux comprendre le fonctionnement d'un shell et de développer nos compétences en programmation système.