Projet Systémes d'Exploitation Centralisés
(Minishell)

MOUACHA BASSOU
Groupe C

Mai 2025

1 Introduction

Ce projet s'inscrit dans le cadre du module *Systèmes d'exploitation centralisés.* Il a pour but de concevoir et développer un interpréteur de commandes simplifié, appelé *minishell*, en langage C. L'objectif principal est de mettre en pratique les notions étudiées en cours et en travaux dirigés, telles que la gestion des processus, des signaux, des entrées/sorties, ainsi que la redirection et la communication entre processus.

À travers ce projet, nous avons progressivement construit les fonctionnalités de base d'un shell Unix, en suivant une démarche incrémentale durant les séances de travaux pratiques.

2 L'architecture de l'application :

Le minishell repose sur une boucle principale qui lit les commandes saisies par l'utilisateur via la fonction `readcmd()`. Pour chaque commande, le programme effectue les traitements suivants :

- **Analyse syntaxique** : détection des erreurs de saisie ou de la commande spéciale `exit` pour quitter le shell.
- **Gestion des processus** :
  - Création d'un processus fils pour chaque commande avec `fork()`.
  - Utilisation de `execvp()` dans le fils pour exécuter la commande.
  - Le père attend la fin du fils si la commande est en avant-plan.
- **Gestion des tubes** :
  - Si plusieurs commandes sont enchaînées (ex : `ls | grep .c`), des tubes sont créés via `pipe()`.
  - Les redirections des entrées et sorties standards sont réalisées à l'aide de `dup2()` selon la position de la commande.
  - Fermeture des descripteurs de fichiers inutiles pour éviter les fuites.
- **Redirections de fichiers** :
  - Utilisation de `open()` pour ouvrir les fichiers en lecture ou écriture.
  - Redirections avec `dup2()` de l'entrée (`<`) ou de la sortie (`>`).
  - Fermeture des fichiers ouverts après duplication des descripteurs.
- **Gestion des signaux** :
  - `SIGCHLD` est traité pour récupérer les statuts des fils et éviter les zombies.

— `SIGINT` et `SIGTSTP` sont bloqués dans le shell principal mais débloqués dans les fils.

```
// Etape 11 :

//Signal : SIGTSTP (Ctrl+Z)
action.sa_handler = traitement_SIGTSTP;
sigemptyset(&action.sa_mask);
action.sa_flags = SA_RESTART;
sigaction(SIGTSTP, &action, NULL);
signal(SIGTSTP, SIG_IGN);

//Signal : SIGINT (Ctrl+C)
action.sa_handler = traitement_SIGINT;
sigemptyset(&action.sa_mask);
action.sa_flags = SA_RESTART;
sigaction(SIGINT, &action, NULL);
signal(SIGINT, SIG_IGN);

//Ces deux signaux correspondent respectivement a Ctrl+C et Ctrl+Z

signal(SIGINT,SIG_IGN);  Ctrl+C
signal(SIGTSTP,SIG_IGN);  Ctrl+Z
```

Listing 1 – Gestion des signaux Ctrl+C et Ctrl+Z

— **Commandes en arrière-plan** :
  — Si la commande contient un `&`, le processus est lancé sans que le père attende sa fin.
  — Le shell reste réactif et affiche immédiatement un nouveau prompt.

Cette architecture permet de simuler un comportement de type shell Unix tout en respectant la structure d'un projet modulaire et évolutif.

3 Les choix & Spécificités de conception :

Lors de la conception de ce *minishell*, plusieurs choix techniques ont été faits afin d'assurer à la fois robustesse, lisibilité et modularité du code :

— **Gestion modulaire des signaux** : les signaux `SIGCHLD`, `SIGINT` et `SIGTSTP` sont traités explicitement à l'aide de la structure `sigaction`. Les signaux `SIGINT` et `SIGTSTP` sont bloqués dans le shell principal pour éviter l'arrêt brutal du processus parent, mais sont débloqués dans les processus enfants pour permettre à l'utilisateur d'interrompre les commandes.

— **Utilisation de `readcmd`** : la lecture et la découpe de la ligne de commande sont confiées à une bibliothèque externe (`readcmd.h`), ce qui permet de simplifier le traitement syntaxique des commandes (gestion des tubes, redirections, arrière-plan).

— **Support des tubes** : le shell gère les tubes (`|`) en créant un tableau de descripteurs de fichiers. Les redirections d'entrées et de sorties sont assurées avec `dup2()`, selon la position de la commande dans la chaîne.

— **Redirections de fichiers** : les redirections `<` et `>` sont implémentées via les appels systèmes `open()`, `dup2()` et `close()`, en respectant les permissions d'accès.

— **Commandes en arrière-plan** : les processus marqués comme étant en arrière-plan sont exécutés sans attendre leur terminaison, ce qui permet de continuer à saisir des commandes dans le shell principal.

— **Robustesse et prévention des zombies** : la gestion du signal `SIGCHLD` avec `waitpid()` dans une boucle permet de récupérer les statuts des processus terminés et d'éviter l'accumulation de processus zombies.

4 La méthodologie de tests :

Afin de valider le bon fonctionnement du *minishell*, une série de tests unitaires et fonctionnels ont été réalisés en ligne de commande :

— **Tests de base** :
  — Exécution de commandes simples `ls`, `pwd` ...
  — Vérification de la commande `exit` pour quitter proprement le shell.

— **Tests de redirection** :
  — Commandes avec redirection de sortie (la commande ls écrit le résultat dans `text.txt`) : `ls > text.txt`
  — Commandes avec redirection d'entrée (`text.txt` sert de source de données pour la commande `sort`) : `> sort < text.txt`

```
Commande: sort
minishell
minishell.c
minishell.o
readcmd.c
readcmd.h
readcmd.o
test_readcmd.c
text.txt
Le processus terminé (1479651).
```

— **Tests avec tubes** :
  — `> cat minishell.c | grep int | wc -l`

```
> Le processus terminé (933034).
25
Le processus terminé (933035).
Le processus terminé (933036).
```

— **Tests de signaux** :
  — Envoi de `Ctrl+C` et `Ctrl+Z` pour vérifier la gestion des signaux.
  — Observation sur les processus en avant-plan uniquement.

— **Tests en arrière-plan** :
  — `> sleep 50 &`

```
Commande sleep 50 &
> Le processus terminé (931383)
```

— **Test de cat** :
  — Avec la commande : `> cat < text.txt > text1.txt`
  Dans un premier temps, on crée un fichier texte nommé `text.txt` dont le contenu est `"bassou"` avec la commande exécutée. On génère ensuite un nouveau fichier nommé `texte1.txt` qui a le même contenu que le fichier `text.txt` (`"bassou"`).

L'ensemble de ces tests a permis de valider les fonctionnalités principales du *minishell* tout en assurant une exécution stable dans différents scénarios.