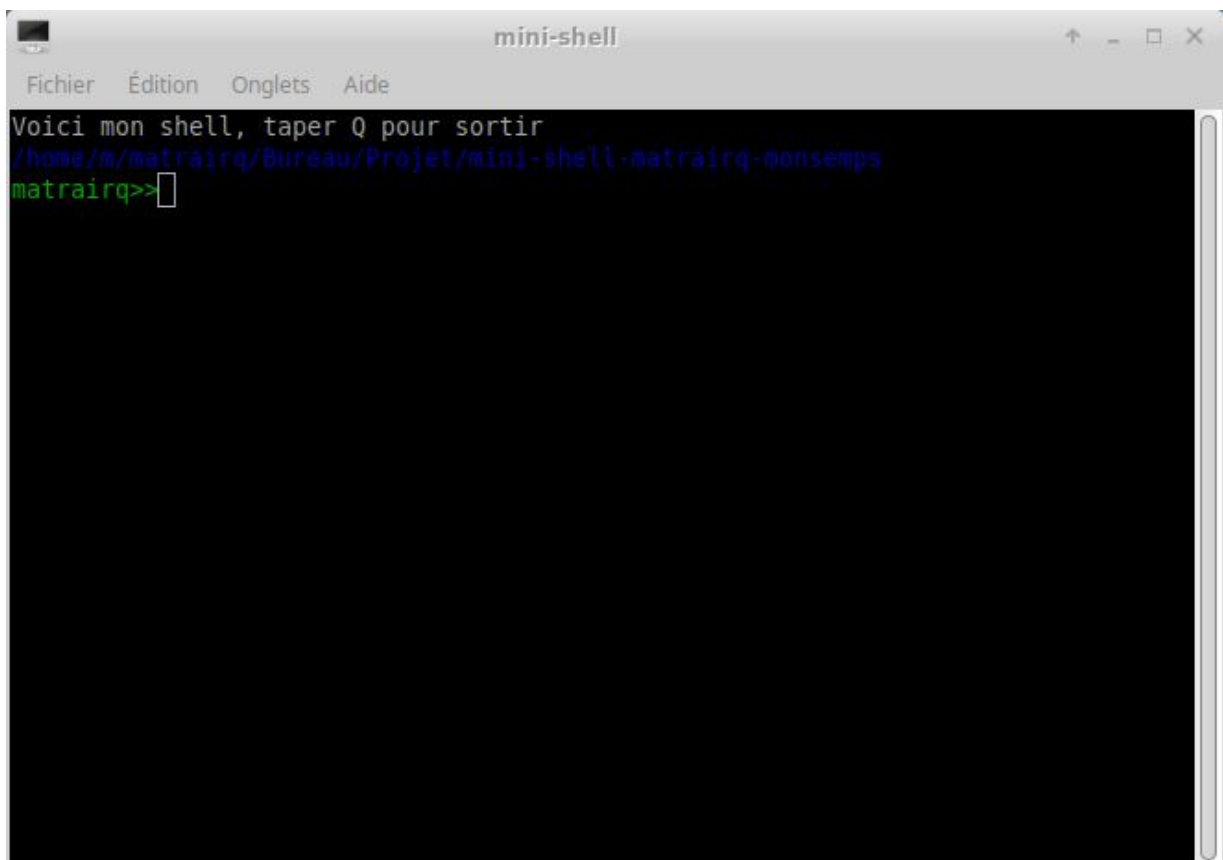


# Projet Mini-Shell

## Rapport de présentation

A screenshot of a graphical user interface for a 'mini-shell' application. The window has a title bar with the text 'mini-shell' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with the items 'Fichier', 'Édition', 'Onglets', and 'Aide'. The main content area has a black background and displays the following text in a monospaced font: 'Voici mon shell, taper Q pour sortir' in white, followed by the file path '/home/m/matrairq/Bureau/Projet/mini-shell-matrairq-monsemp' in blue, and the prompt 'matrairq>' in green with a white cursor. A vertical scrollbar is visible on the right side of the text area.

```
mini-shell
Fichier  Édition  Onglets  Aide
Voici mon shell, taper Q pour sortir
/home/m/matrairq/Bureau/Projet/mini-shell-matrairq-monsemp
matrairq>
```

# **Sommaire**

## **I/ Introduction**

## **II/ Déroulement du projet**

## **III/ Conception**

## **IV/ Annexe**

## **I/ Introduction**

Nous avons réalisé ce projet dans le cadre de notre matière *Introduction au Système*. Nous allons vous présenter notre réflexion, nos choix de conception, ainsi que les problèmes rencontrés pour ce projet. Il s'agit de programmer, dans le langage C, un mini-shell avec quelques fonctionnalités de base.

## **II/ Déroulement du projet**

Pour réaliser ce projet, nous avons utilisé la structure de base donnée à la fin du cours, nous avons réalisé la fonction *Lire\_commande* qui permet de lire une commande ou l'enchaînement de commandes, ainsi que ses arguments. Puis, nous avons implémenté les commandes internes de base (*cd*, *echo*, *exit*, *getenv* et *setenv*).

Nous avons ensuite passé notre projet en programmation modulaire, nous avons également créé un fichier Makefile, pour avoir une meilleure visibilité sur l'ensemble du projet et une compilation plus simple.

Ensuite, nous avons implémenté la fonctionnalité de l'enchaînement de commandes en récursif, ce qui permet d'enchaîner des commandes lorsqu'elles sont séparées d'un ";", ainsi que des opérateurs && et ||.

Après cela, nous avons mis la possibilité d'exécuter des commandes en background avec le caractère "&", ainsi que l'interception de signaux "Ctrl+C" et "Ctrl+\".

Enfin nous avons codé en script sh la mise en fenêtre de notre mini-shell.

### **III/ Conception**

Pour la fonction *Lire\_commande*, nous parcourons notre chaîne de caractère (*cmd*) rentrée par l'utilisateur, et dès que nous rencontrons un caractère espace, nous remplissons le tableau de chaîne de caractère (*com*) avec un tableau dynamiquement alloué de la taille de la commande, ou argument ou caractère “;”.

Pour les commandes internes, nous testons le premier élément de notre tableau de commande (*com*) pour savoir s'il correspond à une commande interne (*cd*, *echo*, *setenv*,...) et nous l'exécutons.

Si la commande n'est pas une commande interne, on considère que c'est une commande externe, alors nous créons un processus fils qui va exécuter la commande. Et nous vérifions que le statut du processus c'est bien déroulé sinon on affiche un message d'erreur.

Pour rendre notre projet plus évolutif, et faciliter le développement du programme nous avons mis notre programme sous forme de module, nous avons donc notre module *mon\_shell* qui fait appel au module *Lire\_commande* et *exec\_command\_rec*. Le module *exec\_command\_rec* quand à lui fait appel au module *commande\_interne* et *commande\_externe*.

Pour pouvoir exécuter plusieurs commandes à la suite grâce au séparateur “;”, ou encore les opérateurs “&&” et “||”, nous avons décidé de faire notre module d'exécution de commande en récursif. Dès qu'un séparateur “;”, “&&” ou “||” est présent dans notre tableau *com*, on exécute la commande avant le séparateur, et on rappelle notre fonction sans le séparateur et la commande qui vient d'être exécuté.

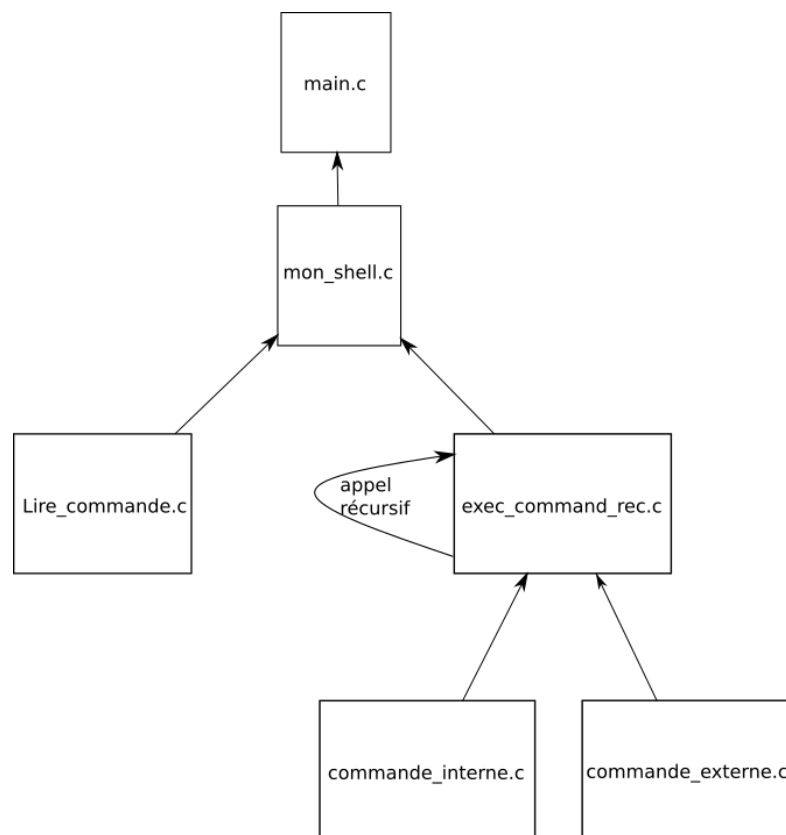


Schéma de l'arborescence du programme

Ensuite pour gérer la mise en background d'une commande, nous testons si le dernier caractère est le caractère "&", si oui notre programme (le père du processus fils qui exécute la commande) n'attend pas la mort de son fils. Par conséquent, notre programme doit vérifier, lorsqu'il crée un autre fils pour exécuter une autre commande, qu'il attend bien la fin (la mort) du dernier processus créé.

Pour intercepter les signaux "Ctrl+C" et "Ctrl+AltGr+\" sans quitter l'exécution du programme, nous avons désactivé l'interception des signaux en question dans notre programme, puis nous les avons réactivé dans les processus fils.

Nous avons ensuite essayé de faire la redirection par *tube* (*pipe*). Pour cela nous vérifions si le caractère "|" est présent après une commande. Si oui nous affectons la valeur 1 à une variable *pipe\_sortie* pour indiquer qu'un tube doit être créé dans la fonction *commande\_externe*. Puis la fonction *commande\_externe*, crée un *tube*, ferme la sortie standard et ouvre notre *tube* dans la sortie du processus lorsque la variable passée en paramètre (*pipe\_sortie*) est égale à 1. De même pour l'entrée standard avec la variable *pipe\_entree*. Au rappel de la fonction *exec\_command\_rec*, *pipe\_entree* prend la valeur de *pipe\_sortie*. Malheureusement, nos *tubes* ne fonctionnent pas (manque de temps).

## **IV/Annexe**

### **Fonctionnalités traitées :**

<b>Fonctionnalité</b>	<b>Réalisée (Oui/Non)</b>	<b>Nom de fonction, nom de fichier</b>
Commandes internes	Oui	commande_interne.c
Redirection d'E/S	Non	
Tubes	Non	
Enchaînement de commandes	Oui	exec_command_rec.c
Mise en background	Oui	commande_externe.c
Interception de signaux	Oui	commande_externe.c
Autre : Enchaînement ( && et    )	Oui	exec_command_rec.c

### **Compilation du projet :**

- par cible Make : make

### **Exécution du projet :**

Commande : **./mode\_fenetre.sh**

ou **./mini-shell** (au cas ou)