

# Rapport Mini-shell

---

Le rapport ci dessous décrit nos avancées, notre travail réalisé, les détails techniques de notre mini-shell et nos problèmes rencontrés.

## 1- Architecture globale du projet.

Notre interpréteur de commande est capable d'exécuter des commandes sous trois formes.

### 1-1 - Exécutables sous trois formes

- **Commandes dans des exécutables séparées:** Nous avons choisi, que les commandes telles que `pwd`, `cd`, `mkdir` sont exécutées sous ce mode car ce sont des commandes propres à l'interpréteur de commandes. Ces commandes sont dans des fichiers séparées, ces fichiers contiennent tous des `main` qui sont exécutées dans le `makefile` afin de générer des fichiers exécutables (`.exe`) qui sont exécutés dans le `minishell`.
- **Commandes intégrées dans un seule exécutables:** toutes ces commandes sont exécutables sous ce mode. Elles sont placées dans un fichier `interne.c` ne contenant pas des `main`. Ces fonctions sont appelées directement depuis le fichier `minishell.c`.
- **Extensible par librairie :** Dans ce mode d'exécution, nous générons la librairie depuis le `makefile` via le fichier `lib.c`. Les fonctions disponibles et exécutables depuis la librairie sont les fonctions `help` et `grep` (fonction détaillée dans la partie commande usuelle).

### 1-2 - Description des fichiers réalisées:

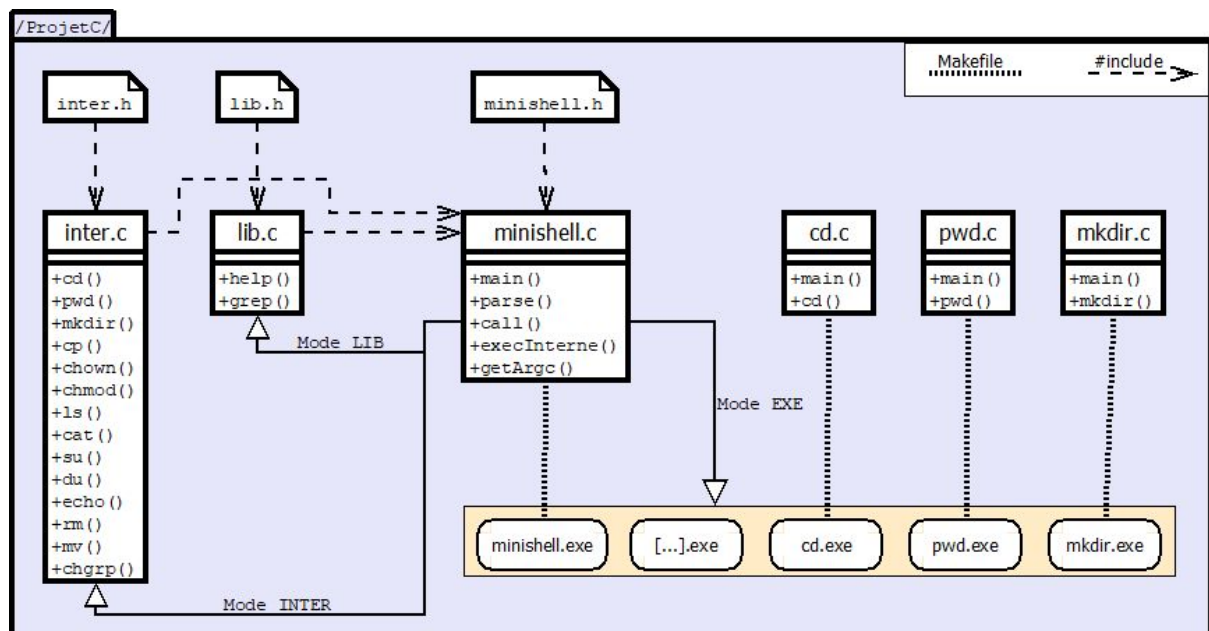
**minishell.c**: fichier principale du projet: il traite les entrées de commandes par un utilisateur dans le minishell dans son intégralité.

**interne.c**: fichier utilisée pour l'exécution interne des commandes il contient les commandes telles que `cd`, `pwd`, `mkdir`, `cp`, `chmod`, `chown`, `chgrp`, `ls`, `cat`, `su`, `du`, `echo`, `rm`, `mv`.

**lib.c**: fichier utilisée pour l'exécution des commandes dans la librairie.

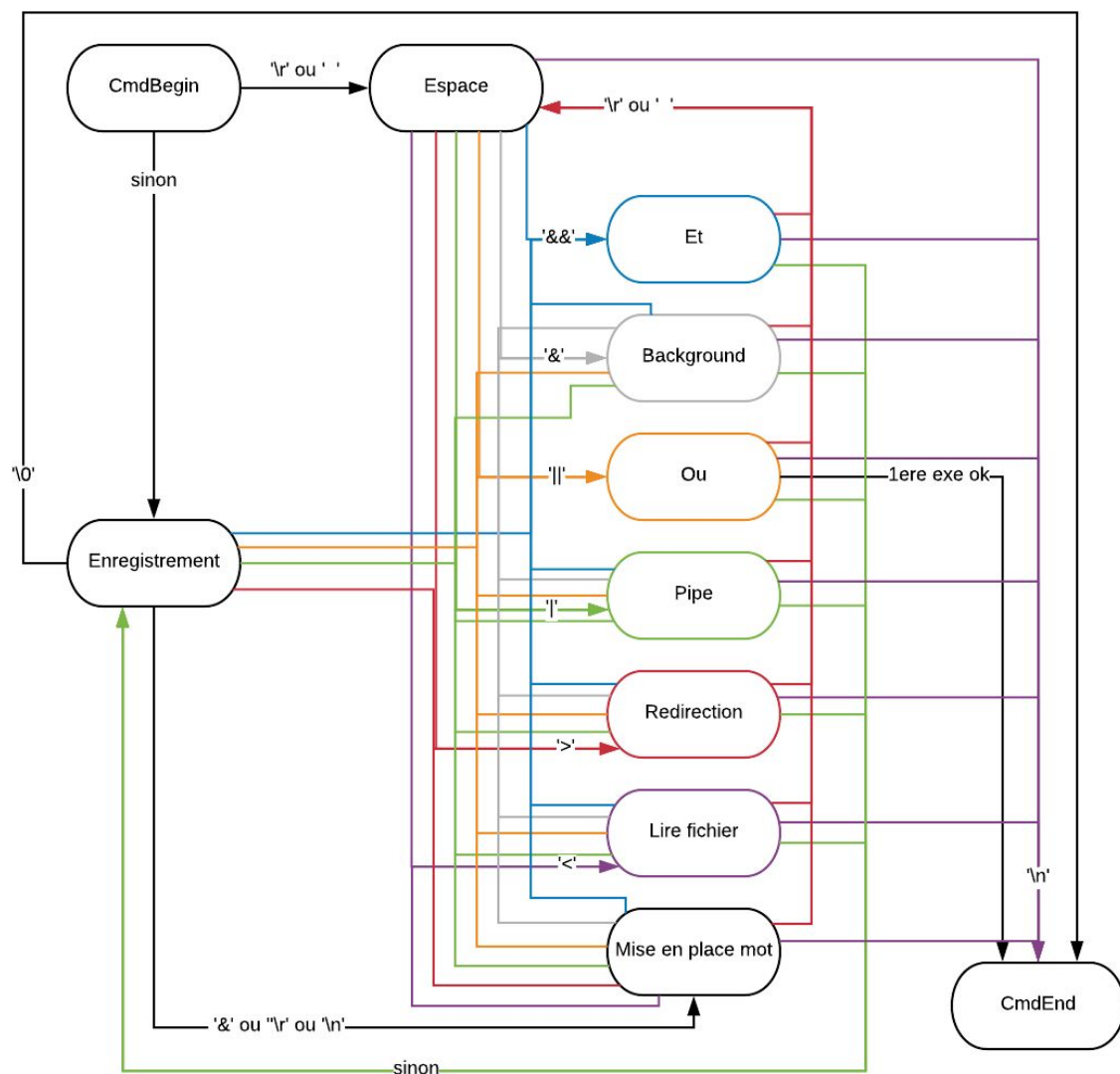
**utils.c**: fichier utilitaire qui contient des fonctions qui nous ont été utiles pour la réalisation du projet. Ces fonctions sont: `fileExist(..)` et

### 1-3 - Diagramme récapitulatif de l'architecture du projet.



## 2 - Partie interpréteur de commande

L'interprétation de la commande tapée par l'utilisateur se fait de la manière suivante. La commande est reçue dans un tableau de caractères (que l'on peut appeler string). Cette string est parcourue, caractère par caractère, et le traitement effectué à chaque fois diffère selon le caractère courant et l'état courant dans lequel on se trouve. Le diagramme d'état-transition suivant décrit l'intégralité de "l'automate" de traitement de la commande.



Ce diagramme reprend en fait la fonction `parse` du `minishell.c`. Voici le traitement effectué dans les états principaux:

- **cmdBegin**: on se met au debut de la commande.
- **Mise en place mot** : le mot courant (entre deux espaces) est enregistré dans un second string temporaire. A la mise en place, ce string est copié dans un tableau de string, `argv`.
- **enregistrement des mots**: On va enregistrer les paramètres de la commande dans une variable `param`, ensuite on va copier ses valeurs dans le `argv`.
- **Redirections**: on va récupérer le nombre de fichiers qui est donné dans la commande et on va rediriger la sortie standard dans ce fichier en utilisant les pipe.
- **Espace** : permet de ne pas ajouter les espaces dans le `argv`.
- **Ou**: effectue les commandes dans l'ordre, tant qu'il n'y a pas eu une bonne execution.
- **Et**: la 1ere commande est effectuée, puis le reste de la commande suit le même traitement. Il est donc possible de mettre plusieurs ET dans la même commande.
- **Background**: On va créer un processus fils ou on va exécuter notre commande sans attendre que le processus fils finisse, le processus père va finir son exécution.

- **Lire le fichier:** la redirection de l'entrée standard, on va créer un fichier ou l'on va mettre les valeurs que l'utilisateur entre et on va mettre ce fichier comme argv 1 de la commande demandée.

## 3 - Partie commandes usuelles

Toutes les fonctions ont les mêmes signatures: `nom(int argc, char argv[])`

Dans cette partie, vous trouverez un résumé du travail effectué dans l'élaboration des commandes ainsi qu'une explication sur les fonctions et appels systèmes utilisés (cf rubrique utilisation spécifique). Les fonctions qui nous ont le plus posées de problèmes sont listées ci dessous:

### 2-1 - cp: copie de fichiers.

#### Cas géré et gestion des erreurs:

On peut copier et coller un dossier en indiquant un nom différent ou le même nom:

- Si la source n'existe pas: on renvoi une erreur.
- Si la destination est un dossier: le nom du fichier copié sera celui du fichier initial.  
On peut donner un nouveau nom au fichier copié.

#### Utilisation de commandes spécifiques:

Utilisation de la fonction `snprintf()` si le chemin source est un dossier. En effet, si c'est le cas, `snprintf` concatène le chemin avec le nom du fichier à copier.

```
snprintf(outfilename, "%s/%s", argv[2], basename(argv[1]));
```

**Exemple :** `cp test.c ~/` : le fichier s'appeller "test.c"

### 2-2 - ls: lister les fichier du répertoire courant ou d'un dossier entrée en paramètre de la fonction.

#### Cas géré et gestion des erreurs:

- Pas d'affichage des fichiers cachés.
- Coloration des dossiers et des fichiers exécutables.
- Si le dossier que l'utilisateur veut lister n'existe pas: on renvoi une erreur.

Option `-l`.

#### Utilisation de commandes spécifiques:

Utilisation de la fonction `stat(..)` qui permet d'avoir des informations spécifiques sur les fichiers. Stat retourne une structure par exemple `st_uid` retourne l'uid du dossier.

```
pwd = getpwuid(s.st_uid);
```

## 2-3 - `chmod`: attribuer des droits.

Cas géré et gestion des erreurs:

- Conversion du nombre entrée en octal (pour rajouter le 0 qui manque, car l'utilisateur rentre 777 or en octal c'est 0777).

```
int octalPermissionString = strtol(argv[1], (char**)NULL, 8);
```

Pour ce faire nous avons utilisé la fonction `strol` qui convertit la chaîne en base 8.

Si la destination est un dossier: le nom du fichier copié sera celui du fichier initial.  
On peut donner un nouveau nom au fichier copié.

Utilisation de commandes spécifiques:

Utilisation de la fonction `snprintf()` si le chemin source est un dossier. En effet, si c'est le cas, `snprintf` concatène le chemin avec le nom du fichier à copier.

```
sprintf(outfilename, "%s/%s", argv[2], basename(argv[1]));
```

Exemple : `cp test.c ~/` : le fichier s'appellera "test.c"

## 2-4 - `chown`: modifier le propriétaire d'un fichier.

Cas géré et gestion des erreurs:

- Vérification du fichier et de l'utilisateur pour savoir s'ils existent.
- Vérification de l'utilisateur.
- Gestion de l'erreur si l'utilisateur n'est pas en super utilisateur. On lui demande ainsi de passer en mode sudo (cf fonction `su`).

Utilisation de commandes spécifiques:

Fonction `getpwnam` qui récupère l'uid de l'utilisateur.

```
pwd = getpwnam(argv[1]);
```

## 2-5- du: afficher la taille d'un fichier ou d'un répertoire.

### Cas géré et gestion des erreurs:

- Si aucun paramètre n'est entré en paramètre, la commande renvoie la taille du répertoire courant
- Utilise la fonction `getFileSize` déclarée et définie dans la librairie `utils.h`
- Gestion de l'erreur où le fichier n'existe pas, et où l'utilisateur n'a pas les droits du fichier dont on souhaite avoir la taille.

### Utilisation de commandes spécifiques:

Fonction `getcwd` qui récupère le répertoire courant.

```
getcwd(cwd, sizeof(cwd));
```

Fonction `access` qui vérifie les droits de l'utilisateur sur un fichier.

```
if(access(argv[i], F_OK) != -1){
```

## 2-6 - mv: déplace un fichier dans un répertoire, ou le renomme.

### Cas géré et gestion des erreurs:

- Si le 2ème paramètre n'est pas un répertoire, le fichier pris en 1er paramètre est renommé.  
Exemple : Soit `DOC` un dossier inexistant et `fic` un fichier existant, la commande : `mv fic DOC` va renommer `fic`. Le fichier `fic` aura dorénavant le nom : `DOC`.
- Utilise les fonctions C de gestion de fichier, e.g : `fopen`, `fclose`, `rename`, `remove`..

### Utilisation de commandes spécifiques:

Fonction `getcwd` qui récupère le répertoire courant.

```
sprintf(outfilename, "%s/%s", argv[2], basename(argv[1]));  
//si le chemin dest est un répertoire: on prend pour nom le nom du fichier à copier
```

## 2-7- Quelques mots sur `mkdir`, `pwd`:

**mkdir:** dans cette fonction il est possible de créer plusieurs dossiers simultanément: `mkdir dossier1, dossier2`.

**pwd:** dans cette fonction si l'utilisateur n'entre aucun argument, il est ramené à son home.  
Pour se faire nous avons utilisé la fonction `getcwd` qui récupère le répertoire courant de

l'utilisateur. Cette fonction n'est pas tout à fait opérationnelle car elle agit sur le processus courant et pas sur les autres.

```
getcwd(cwd, sizeof(cwd))
```

## 2-5 -Fonctions supplémentaires ajoutées `help`, `grep`:

**help:** Nous avons réalisé cette fonction afin de présenter un manuel explicatif de chaque fonction à l'utilisateur. Différence avec le `help` du shell : n'affiche pas toutes les commandes quand elle n'est pas suivie d'arguments. Elle renvoie, à la place, un message priant l'utilisateur d'entrer un paramètre.

**grep:** Nous avons réalisé cette fonction pour tester plus amplement nos redirections. Cette fonction permet juste de chercher un mot dans un fichier.

## 4 - Problèmes rencontrés

**Partie interface:** Au rendu de l'interface, nous n'avions pas tout à fait compris de quoi il s'agissait. Nous pensions que toutes les fonctions devaient être exécutées sous les trois modes. Or, ce n'est que les fonctions propres à l'interpréteur de commande qui sont exécutables en mode exécutable.

**Partie commande :**

Fonction SU :

En ce qui concerne la fonction `su`, nous avons effectué un petit schéma global des étapes à suivre pour avoir le résultat de la commande `su` sur le shell, le voici :

*/\*Structure du code*

- 1.s'il y a un paramètre après `su`:
  - 1.Récupérer ce paramètre.
  - 2.Accéder au fichier `/etc/shadow`
  - 3.Chercher le paramètre entré par l'utilisateur dans le fichier (c'est un identifiant à priori)
    - 1.S'il n'existe pas, on affiche au lecteur un msg d'erreur
    - 2.S'il existe :
      - 1.On récupère le mdp crypté associé à cet identifiant( pas sûre de comment le faire)
      - 2.On demande à l'utilisateur le mot de passe.
        - 1.S'il est le même que le mdp récupéré dans le fichier `/etc/shadow`:
          1. On change de session sur le terminal
        - 2.S'il n'est pas le même :
          - 1.On affiche un msg d'erreur à l'utilisateur "Mot de passe incorrect"
  - 2.On fait passer l'utilisateur en mode superutilisateur (cf à la commande `sudo su`)

Le problème avec les étapes illustrées ci-dessus, notamment la toute première condition,est qu'il nous faut un algorithme de décryptage afin de récupérer les mots de passe stockés

dans le fichier `/etc/shadow`. Par ailleurs, nous n'avons pas les permissions d'accès à ce genre de fichier fortement sécurisés, et par conséquent, il n'y a pas de moyen de tester notre programme.

Nous nous sommes alors contentés de faire un appel système à la fonction 'sudo su' dans notre programme, passant directement au cas où le 'su' n'aurait pas de paramètres. Ainsi, l'utilisateur passe en super utilisateur seulement, il ne peut passer sur la session d'un autre utilisateur via notre su.