|  |  |  |  |
| --- | --- | --- | --- |
|  | 20/03/2018 |  | |
|  | |  | |
| Système d’exploitation avancé  *Projet : MiniSHell « my\_sh »* | | | |
|  |  | |  |
|  | Victor Lambs & Laurent Freyss | | |



Système d’exploitation avancé

Projet : MiniSHell « my\_sh »

"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity. "

**-- Dennis Ritchie, inventeur du**[**langage C**](https://fr.wikipedia.org/wiki/Langage_C)**et codéveloppeur de**[**Unix**](https://fr.wikipedia.org/wiki/Unix)**, 1941/2011**

## Objectifs :

L’objectif de ce projet et de réaliser -dans une proportion simplifiée- l’implémentation d’un interpréteur de commande similaire à bash. Au lancement ce dernier doit pouvoir afficher un prompt attendant la saisie d’une commande ou d’un sous-ensemble de commandes.

**Prompt>** ls –a ; who

**Prompt>** ls –a | grep toto

**Prompt>** date

Table des matières

[Objectifs : 1](#_Toc510514159)

[1. Introduction 3](#_Toc510514160)

[A) Code source 3](#_Toc510514161)

[B) Exécution 3](#_Toc510514162)

[2. Fonctionnalités réalisées 4](#_Toc510514163)

[A) Fonctions métiers 4](#_Toc510514164)

[B) Exigences techniques 4](#_Toc510514165)

[3. Architectures 5](#_Toc510514166)

[A) Généralités 5](#_Toc510514167)

[B) Arbre 5](#_Toc510514168)

[a. Séparation 5](#_Toc510514169)

[a. Opérateur de contrôle 6](#_Toc510514170)

[b. Redirection de flux 6](#_Toc510514171)

[C) Exécutions 7](#_Toc510514172)

[a. Commandes 7](#_Toc510514173)

[b. Opérateurs 8](#_Toc510514174)

# Introduction

## Code source

Notre code source est disponible sur github au lien suivant :

<https://github.com/lfreyss/ShellProject>

La branche active et fonctionnelle est la branche : « gitfix ».

## Documentations

### Doxygen

Un rapport de documentation Doxygen a été généré pour notre projet Shell. Les commentaires Doxygen pour les fonctions ont été écrit dans les fichiers h afin d’alléger le contenu des fichiers c.

Il est possible de consulter le rapport Doxygen en ouvrant le fichier /doc/html/index.html

### GCOV

Un rapport de couverture de code GCOV a été généré pour notre projet Shell.

Il est possible de consulter le rapport GCOV en ouvrant le fichier /gcov/report/index.html

Il est possible de consulter un manuel avec la commande suivante : « man ./shellMan »

## Exécution

Il est possible d’exécuter notre shell de deux manières différentes.

En réalisant au préalable un « make », il est possible de lancer comme tout autre programme c en exécutant la commande suivante :

« ./shell.c »

Il est également possible de lancer le programme en mode batch avec l’option -c

**Exemple :** ./shell –c « ls –al | grep toto »

# Fonctionnalités réalisées

## Fonctions métiers

**FM01 –** Le binaire est capable d’exécuter une commande simple (ie : ls –l ; ps ; who)

**FM02** – Le binaire est capable d’exécuter un sous-ensemble de plusieurs commandes de sorte à prendre en compte :

* Les opérateurs de contrôle : && et ||
* Les redirections de flux simples : |, >, <, >>, <<
* L’exécution en arrière-plan : &

**FM03** – L’exécution des commandes internes (fonctionnalités built-in) suivantes :

* cd - Permettant de se déplacer au sein d’une arborescence de fichier.
* pwd – Affichant la valeur de la variable contenant le chemin du répertoire courant.
* exit – Permettant de quitter l’interpréteur.
* echo – Permettant d’afficher du texte sur la sortie standard.

**FM04** - La persistance des commandes saisie dans un fichier (historique)

**FO01** – La réalisation d’un mode batch (ie : ./my\_shell –c « ls –al | grep toto »)

**FO03** – La prise en charge d’alias

## Exigences techniques

**CT01** – La compilation du projet doit se faire via un Makefile.

**CT02** – La définitions des structures doit se faire dans un fichier typedef.h.

**CT03** – La définition des méthodes protoype (.h) & implémentation (.c) doit se faire de manière séparée autant que faire se peut.

**CT04** – Le code produit doit être documenté.

**CT05** – La gestion des erreurs doit se faire via « les mécanismes proposés par errno ».

**CTO01** – La documentation du code générée via l’utilitaire doxygen.

**CTO02** – Le code est soumis à un contrôle de couverture via l’utilitaire gcov.

**CTO03** – Une page de manuel Linux est rédigée pour détailler l’exécution du shell.

# Architectures

## Généralités

Notre Shell s’exécute dans une boucle infinie s’arrêtant uniquement quand l’utilisateur saisie la commande « exit ». Cette boucle aura pour but pour chacune de ses itérations :

* D’afficher en premier lieu l’emplacement du dossier courant
* Puis d’attendre la saisie de l’utilisateur
* Logger la saisie de l’utilisateur dans un fichier d’historique
* Ensuite construire un arbre binaire de commande
* Lire et exécuter les commandes de l’arbre
* Recommencer si la commande n’est pas « exit »

Comme annoncé, pour la réalisation de notre Shell et afin de réaliser la fonctionnalité **FM02**, notre architecture se repose sur un arbre binaire. Chaque nœud de notre arbre correspond à un opérateur (De contrôle ou de redirection de flux) tandis que chaque feuille de notre arbre correspond à un opérande, c’est-à-dire, à une commande et potentiellement ses options/arguments.

Il s’agira ensuite de lire notre arbre en partant de la feuille la plus à gauche à celle la plus à droite.

## Arbre

### Séparation

Lors du traitement de la saisie utilisateur, il faut séparer les commandes qui sont censé s’exécuter l’un après l’autre. Ces commandes sont séparées par le caractère « ; ».

Chacune des suites de commande correspondent à un arbre.

**Exemple** : ls -al | grep bin && echo ok >> res.txt || echo nok >> res.txt ; who > jesuis.txt

Arbre 1 Arbre 2

### Opérateur de contrôle

Ensuite pour chaque suite de commande, on effectue un nouveau travail de séparation. En effet, ici on cherche à séparer la suite de commande par chaque opérateur de contrôle (&& et ||) qu’elle contient. En séparant la suite de commande ainsi, on aura un tableau alternant constamment un opérand avec un opérateur.

**Exemple :** **[** ls -al | grep bin **] [** && **] [** echo ok >> res.txt **] [** || **] [** echo nok >> res.txt **]**

Une fois en possession du tableau de chaine de caractère séparé par les opérateurs de contrôle, on construit notre arbre en parcourant le tableau de la fin vers le début.

Si la chaine de caractère est une commande

🡪 On crée un nouveau nœud et la commande devient la feuille droite du nœud

Sinon

🡪 L’opérateur devient le nœud, et le reste du tableau devient la feuille gauche du nœud

🡪 On utilise la récursivité pour construire le noeud de la feuille gauche si la valeur de la feuille gauche possède encore un opérateur de controle

**Exemple : [** ls -al | grep bin **] [** && **] [** echo ok >> res.txt **] [** || **] [** echo nok >> res.txt **]**

1ere itération – ConstructionTree 2nd itération – Récursivité ConstructionTree

### Redirection de flux

Lorsqu’une feuille ne possède plus d’opérateur de contrôle, notre algorithme vérifie si la feuille n’est qu’une seule commande ou bien s’il s’agit d’une suite de commande intercalé par des redirections de flux. Pour cela, nous séparons la valeur de la feuille par caractère de redirection de flux ( |, < ,<<,>,>>) afin d’obtenir comme avant une tableau de chaine de caractère.

Ensuite, on effectue le même traitement que pour les opérateurs de contrôle.

**Exemple :** **[** ls -al **]**[ | **][** grep bin **]**

Création d’un nœud avec un flux de redirection Arbre final

## Exécutions

Afin d’exécuter l’arbre, on le parcourt de gauche à droite. On commence par exécuter la feuille la plus à gauche, puis on remonte vers l’opérateur supérieur. Selon l’opérateur, on exécute ou non la feuille de droite puis on remonte jusqu’à atteindre l’opérateur racine.

Nous fonctionnons avec un fichier **out** et un fichier **in**. Le résultat des commandes est systématiquement écrit dans le fichier **out**. Puis selon l’opérateur, les traitements différents comme démontré ci-dessous. Les changements d’entrée et de sortie sont réalisés à l’aide de la fonctionnalité **dup** de la librairie *unistd.h.*

### Commandes

#### Commande de base

Les commandes simples de type « ls ; who ; ps » sont exécuté via la fonctionnalité **execvp** provenant de la librairie[*unistd.h*](file:///\\usr\include\unistd.h). Pour se faire, notre processus crée un processus fils via un fork afin de lancer **execvp**. Le père attend ensuite la fin du processus fils avant de continuer son exécution.

Si le fichier in est vide, alors la commande s’exécutera sur l’entrée standard, sinon on utilisera les données se trouvant dans le fichier in. Le résultat de la commande est écrit dans le fichier out. Puis à la fin du traitement, le contenu du fichier out est affiché sur la sortie standard et son contenu est effacé.

#### Commande Build-in

Les commandes build-in telle que « echo ; cd ; pwd » fonctionnent avec le même système. Cependant nous n’utilisons pas ici la fonctionnalité **execvp**.

Nous nous servons de la fonctionnalité :

* + - **Printf()** provenant de la librairie *stdio.h* pour la commande « echo »
    - **Chdir()** provenant de la librairie *unistd.h* pour la commande « cd »
    - **Getcwd()** provenant de la librairie *unistd.h* pour la commande « pwd »

La commande « exit » est particulière. Celle-ci est directement traité dans la boucle infinie, si la saisie entrée par l’utilisateur est « exit », alors on sort de la boucle et le programme se termine.

#### Fonctionnalité alias

Un alias permet de donner un nom (alias) à une suite de commande afin majoritairement de raccourcir son écriture.

**Exemple :** ls -al | wc -l **🡪** alias countLine = ls -al | wc -l

Lorsque l’alias « countLine» sera saisie, le programme exécutera la commande « ls -al | wc -l »

Deux étapes sont nécessaires pour la réalisation d’un alias :

* + La sauvegarde de l’alias dans une HashMap telle que [AliasName][Commande]
  + La vérification de l’existence de l’alias dans la HashMap et le cas échéant, le remplacement de l’alias par la commande correspondante

### Opérateurs

#### Opérateur |

L’opérateur « | » permet d’exécuter une seconde commande sur le résultat de la première.

**Exemple :** ls -al | wc -l

Pour ce faire, en prenant l’exemple ci-dessus, on exécute la commande ls -al, son résultat est stocké dans le fichier **out**, puis quand on arrive à l’opérateur « | », on copie le contenu du fichier **out** dans le fichier **in**. Dès lors, la commande wc -l utilisera comme entrée non pas l’entré standard mais le fichier **in** car celui-ci n’est plus vide. Le résultat de la seconde commande est toujours redirigé vers le fichier **out**. Et l’on vide le fichier **in** afin que l’entrée standard soit à nouveau celle par défaut. S’il n’y a pas d’autre commande à réaliser, alors on affiche le contenu du fichier **out** sur la sortie standard.

#### Opérateur > et >>

Les opérateurs « > et >> » sont des opérateurs de redirection de flux. Ils permettent d’écrire le résultat de la commande dans un fichier. L’opérateur « > » remplace le contenu du fichier sans demander confirmation, l’opérateur « >> » écrit le résultat de la commande à la suite du contenu du fichier.

**Exemple :** ls -al > res.txt ; pwd >> res.txt

Pour ce faire, en prenant les exemples ci-dessus, on exécute la commande et l’on stocke son résultat dans le fichier **out**. En arrivant sur l’opérateur, dans le premier cas, on ouvre le fichier portant le nom res.txt et l’on écrit le contenu à partir de l’offset 0, on réécrit donc sur le contenu existant (ou non). Pour le second cas, on ouvre le fichier à partir de l’offset du dernier caractère.

Bien évidemment, à la fin du traitement dans ces deux cas présents, on ne réécrit pas le contenu du fichier **out** dans la sortie standard.

#### Opérateur < et <<

Les opérateurs « < et << » sont également des opérateurs de redirection de flux. Le premier permet de changer l’entrée standard sur lequel s’exécute la commande. Le second permet une saisie sur l’entrée standard de plusieurs lignes sur lesquelles s’exécutera la commande.

**Exemple :** grep test < res.txt ; grep test << fin

Pour ce faire, en prenant les exemples ci-dessus, le fonctionnement diffère légèrement. Si en parcourant l’arbre, un des deux opérateurs est détecté, alors dans le premier cas, on copie le contenu du fichier ciblé, ici res.txt, dans notre fichier **in**. Ensuite, on lance la commande « grep test », elle s’exécutera sur le fichier **in** car celui-ci ne sera pas vide et le résultat sera toujours stocké dans le fichier **out**. Puis on efface le contenu du fichier **in** et on affiche le contenu du fichier **out** sur la sortie standard.

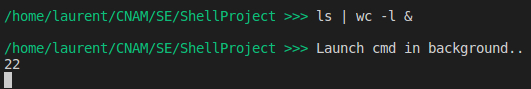
Pour le second opérateur, on lit simplement la saisie de l’utilisateur sur l’entrée standard et on l’ajoute au fichier **in** tant que la saisie de l’utilisateur n’est pas égale à « fin ». Ensuite même déroulement que pour l’opérateur « < », on lance la commande « grep test », elle s’exécutera sur le fichier **in** car celui-ci ne sera pas vide et le résultat sera toujours stocké dans le fichier **out**. Puis on efface le contenu du fichier **in** et on affiche le contenu du fichier **out** sur la sortie standard.

#### Opérateur &

L’opérateur « & » permet l’exécution d’une commande en arrière-plan.

**Exemple :** ls | wc -l &

Pour ce faire, en prenant l’exemple ci-dessus, le fonctionnement est sensiblement le même que le lancement standard d’une commande, seulement le processus père n’attend pas la fin du processus fils pour continuer.



On peut voir sur la capture d’écran ci-dessous que notre processus père n’a pas été bloqué par le processus fils. En effet, lorsque le processus père à lancer le processus fils, il a continué son exécution et a donc reboucler, réaffichant le chemin courant et l’attente de la saisie. Entre temps, le processus fils a fini son exécution et affiche donc son résultat dans la sortie standard, décalant l’icône d’attente de saisie.