

# Projet Shell Unix

Rapport de Projet UE OS

Systemes d'Exploitation

Léo Thomas - Mathis Da Cunha

# Table des matières

<b>1</b>	<b>Résumé des objectifs</b>	<b>2</b>
<b>2</b>	<b>Diagramme des composants et explication des éléments</b>	<b>2</b>
2.1	Diagramme . . . . .	2
2.2	Explication des composants . . . . .	2
<b>3</b>	<b>Algorithmes</b>	<b>4</b>
3.1	Algorithme 1 : Exécution d'une unique commande . . . . .	4
3.2	Algorithme 2 : Lancement de l'exécutable . . . . .	5
3.3	Algorithme 3 : Main . . . . .	6
<b>4</b>	<b>Tests</b>	<b>7</b>

# 1 Résumé des objectifs

Ce projet a pour objectif de concevoir et implémenter un interpréteur minimaliste de type shell, capable de traiter des commandes internes, d'exécuter des programmes externes, de gérer les redirections, et de maintenir un comportement robuste face aux erreurs. L'enjeu principal est d'obtenir une architecture claire et modulaire, permettant d'assurer un fonctionnement fiable tout en facilitant l'analyse, les tests et l'évolution du code. L'application doit rester simple mais conforme aux comportements attendus d'un shell basique, afin de constituer un socle pédagogique solide pour comprendre la gestion de processus, l'analyse d'entrées utilisateur et la manipulation des chemins d'exécution.

## 2 Diagramme des composants et explication des éléments

### 2.1 Diagramme

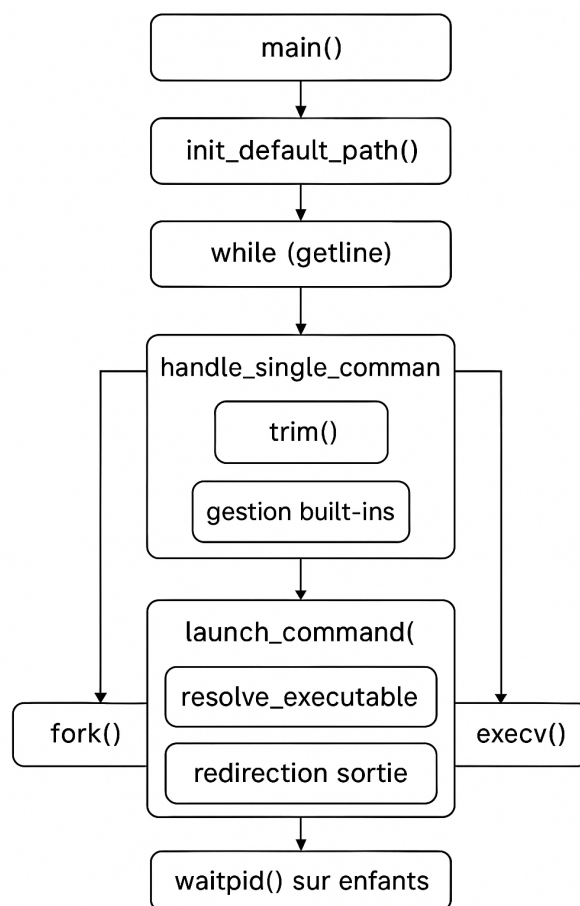


FIGURE 1 – Diagramme des composants

### 2.2 Explication des composants

Ce shell repose sur une série de fonctions et de composants modulaires qui assurent un traitement linéaire et contrôle de chaque commande.

**error()** : Affiche un message d'erreur standardisé via STDERR. Sert de point central pour tous les cas d'erreurs détectés afin de garantir une sortie cohérente.

**init\_default\_path()** : Initialise la liste des chemins avec un seul répertoire par défaut : /bin. Permet de garantir qu'un shell nouvellement lancé sait où chercher les exécutables.

**free\_paths()** : Libère proprement la mémoire allouée à la liste des chemins. Évite les fuites mémoire et est appelé lors du changement de PATH.

**set\_path(newpaths, n)** : Remplace complètement la liste de chemins actuelle par une nouvelle liste. Chaque chemin est dupliqué en mémoire et géré indépendamment. Si n vaut 0, alors PATH est vide et aucune commande externe ne pourra être exécutée.

**trim(s)** : Supprime les espaces, tabs et retours ligne au début et à la fin d'une chaîne. Retourne un pointeur vers l'intérieur de la chaîne d'origine sans allouer de nouvelle mémoire. Garantit une analyse propre des commandes.

**tokenize\_whitespace(cmd, out\_argc)** : Découpe une commande en tokens séparés par espaces/tabs/newlines. Renvoie un tableau dynamique terminé par NULL. Permet de construire argv pour exec.

**free\_argv(argv)** : Libère le tableau alloué par la fonction précédente.

**resolve\_executable(cmd)** : Parcourt les chemins déclarés dans PATH pour construire un chemin complet vers l'exécutable cmd. Vérifie les permissions d'exécution via access. Renvoie le chemin valide ou NULL.

**launch\_command(argv, argc, outfile)** : Exécute un programme externe. Crée un processus fils via fork, gère la redirection si nécessaire, remplace le processus via execv, et retourne le PID. Le parent récupère le PID tandis que l'enfant s'occupe de l'exécution.

**handle\_single\_command(command, out\_pid)** : Traite une commande unique sans parallèle. Gère la détection et vérification de la redirection, la tokenisation, et l'exécution des built-in (exit, cd, path). Si ce n'est pas un built-in, lance un programme externe via launch\_command. Retourne 0 pour un built-in, 1 pour un programme externe, ou -1 si erreur.

**main()** : Point d'entrée du shell. Gère l'interactivité ou le mode script, lit les lignes, découpe par & pour l'exécution parallèle, traite chaque commande avec handle\_single\_command, stocke les PIDs, attend leur terminaison, puis recommence. Le shell tourne en boucle jusqu'au EOF ou exit.

## 3 Algorithmes

### 3.1 Algorithme 1 : Exécution d'une unique commande

---

**Algorithm 1** Traitement d'une ligne de commande avec support parallèle

---

```

0: procedure HANDLELINE(ligne)
0:   segments  $\leftarrow$  Diviser ligne par '&'
0:   pids  $\leftarrow$  [] {Tableau des PIDs des processus enfants}
0:   for chaque segment dans segments do
0:     segment  $\leftarrow$  Trim(segment)
0:     if segment est vide then
0:       Afficher erreur
0:       return
0:     end if
0:     (argv, redirection)  $\leftarrow$  ParseCmd(segment)
0:     if échec du parsing then
0:       Afficher erreur
0:       return
0:     end if
0:     if argv[0] est "exit" then
0:       if argc  $\neq$  1 then
0:         Afficher erreur
0:       end if
0:       Exit(0)
0:     else if argv[0] est "cd" then
0:       if argc  $\neq$  2 then
0:         Afficher erreur
0:       else
0:         if chdir(argv[1]) échoue then
0:           Afficher erreur
0:         end if
0:       end if
0:     else if argv[0] est "path" then
0:       SetPath(argv, argc)
0:     else
0:       pid  $\leftarrow$  RunExec(argv, redirection)
0:       if pid > 0 then
0:         Ajouter pid à pids
0:       end if
0:     end if
0:     Libérer mémoire de argv et redirection
0:   end for
0:   for chaque pid dans pids do
0:     waitpid(pid, NULL, 0)
0:   end for
0: end procedure=0

```

---

### 3.2 Algorithme 2 : Lancement de l'exécutable

---

**Algorithm 2** launch\_command(argv, argc, outfile)

---

```
1: if argc = 0 then
2:   return -1
3: end if
4: cmd ← argv[0]
5: fullpath ← resolve_executable(cmd)
6: if fullpath = NULL then
7:   error()
8:   return -1
9: end if
10: pid ← fork()
11: if pid < 0 then
12:   error()
13:   return -1
14: end if
15: if pid = 0 then
    {processus fils}
16:   if outfile existe then
17:     ouvrir outfile en écriture
18:     rediriger stdout et stderr vers outfile
19:   end if
20:   execv(fullpath, argv)
21:   error()
22:   terminer le processus
23: end if
24: return pid {processus parent} = 0
```

---

### 3.3 Algorithme 3 : Main

---

**Algorithm 3** main

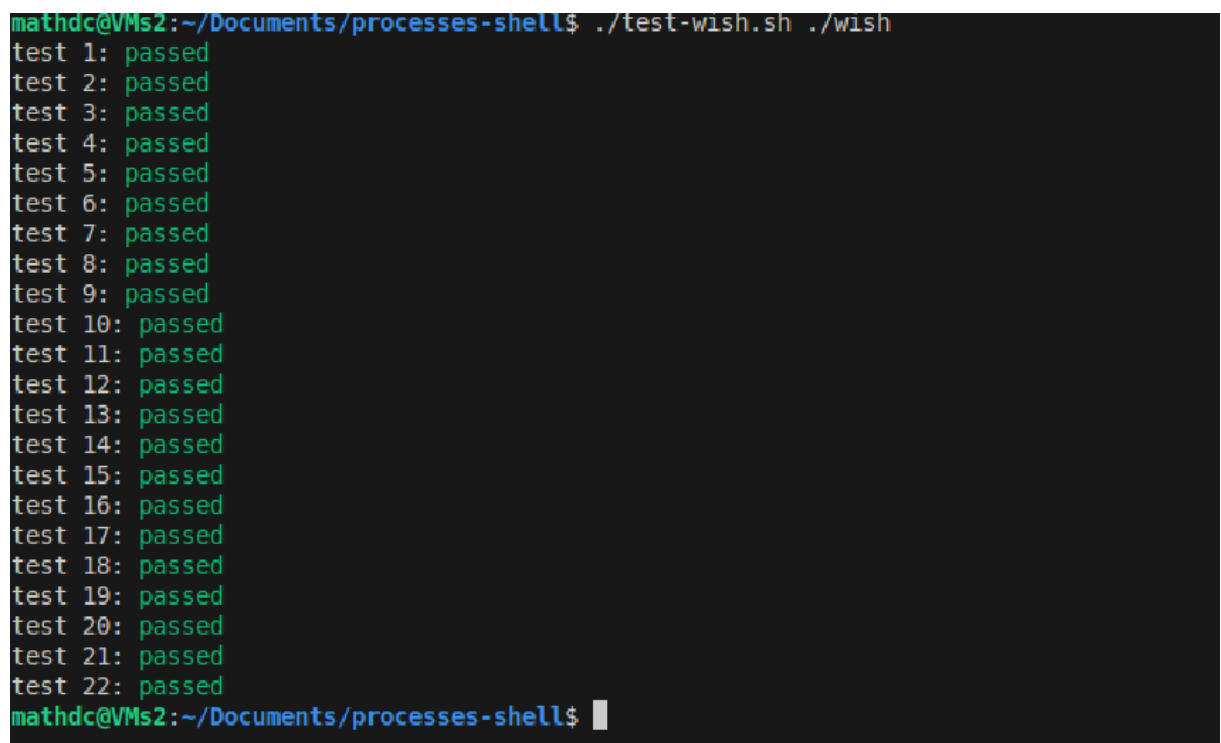
---

```
1: déterminer mode interactif ou fichier script
2: initialiser PATH par défaut
3: while vrai do
4:   if interactif then
5:     afficher prompt
6:   end if
7:   lire ligne
8:   if EOF then
9:     quitter
10:  end if
11:  tline ← trim(ligne)
12:  if tline est vide then
13:  end if
14:  découper tline par ' ' en sous-commandes
15:  liste_pids ← liste vide
16:  for chaque sous-commande cmd do
17:    if cmd est vide then
18:      error()
19:      continuer
20:    end if
21:    rc, pid ← handle_single_command(cmd)
22:    if pid > 0 then
23:      ajouter pid à liste_pids
24:    end if
25:  end for
26:  for chaque pid dans liste_pids do
27:    attendre la fin du processus
28:  end for
29: end while=0
```

---

## 4 Tests

Le code a été testé en validant chaque fonctionnalité les unes après les autres.



```
mathdc@VMs2:~/Documents/processes-shell$ ./test-wish.sh ./wish
test 1: passed
test 2: passed
test 3: passed
test 4: passed
test 5: passed
test 6: passed
test 7: passed
test 8: passed
test 9: passed
test 10: passed
test 11: passed
test 12: passed
test 13: passed
test 14: passed
test 15: passed
test 16: passed
test 17: passed
test 18: passed
test 19: passed
test 20: passed
test 21: passed
test 22: passed
mathdc@VMs2:~/Documents/processes-shell$
```

FIGURE 2 – Tests unitaires avec `./test-wish.sh`

## Conclusion

La réalisation de ce projet de mini-shell Unix, *wish*, a permis de comprendre en profondeur le fonctionnement interne d'un interpréteur de commandes. En développant une boucle interactive capable de lire, analyser et exécuter des commandes, nous avons manipulé directement les mécanismes fondamentaux de gestion des processus, ainsi que la gestion des erreurs et des flux standards.

La mise en œuvre des commandes internes a permis de distinguer clairement ce qui relève de la logique du shell lui-même de ce qui doit être délégué au système sous forme de nouveaux processus.

Le support du mode batch, de la redirection de sortie et de l'exécution parallèle a ajouté des dimensions supplémentaires, notamment au niveau de l'analyse syntaxique, de la robustesse du programme et de la gestion simultanée de multiples processus indépendants. Ces fonctionnalités plus avancées ont permis de construire un shell plus complet et plus proche des comportements réels observés dans des interpréteurs comme *bash*.

Au-delà des aspects techniques, ce projet a nécessité une attention soutenue à la détection des erreurs, au traitement des cas limites et à la structuration modulaire du code. Il a également mis en évidence l'importance d'un test approfondi et varié pour garantir la fiabilité d'un programme aussi central qu'un shell.

En somme, la construction de *wish* a constitué une excellente introduction pratique à la programmation système sous Unix. Elle a permis d'acquérir une maîtrise concrète



des appels systèmes, de la gestion des processus et des principaux mécanismes qui sous-tendent le fonctionnement des shells, ouvrant la voie à des développements futurs plus avancés dans le domaine des systèmes et des interfaces en ligne de commande.