WISH : UN PETIT SHELL UNIX

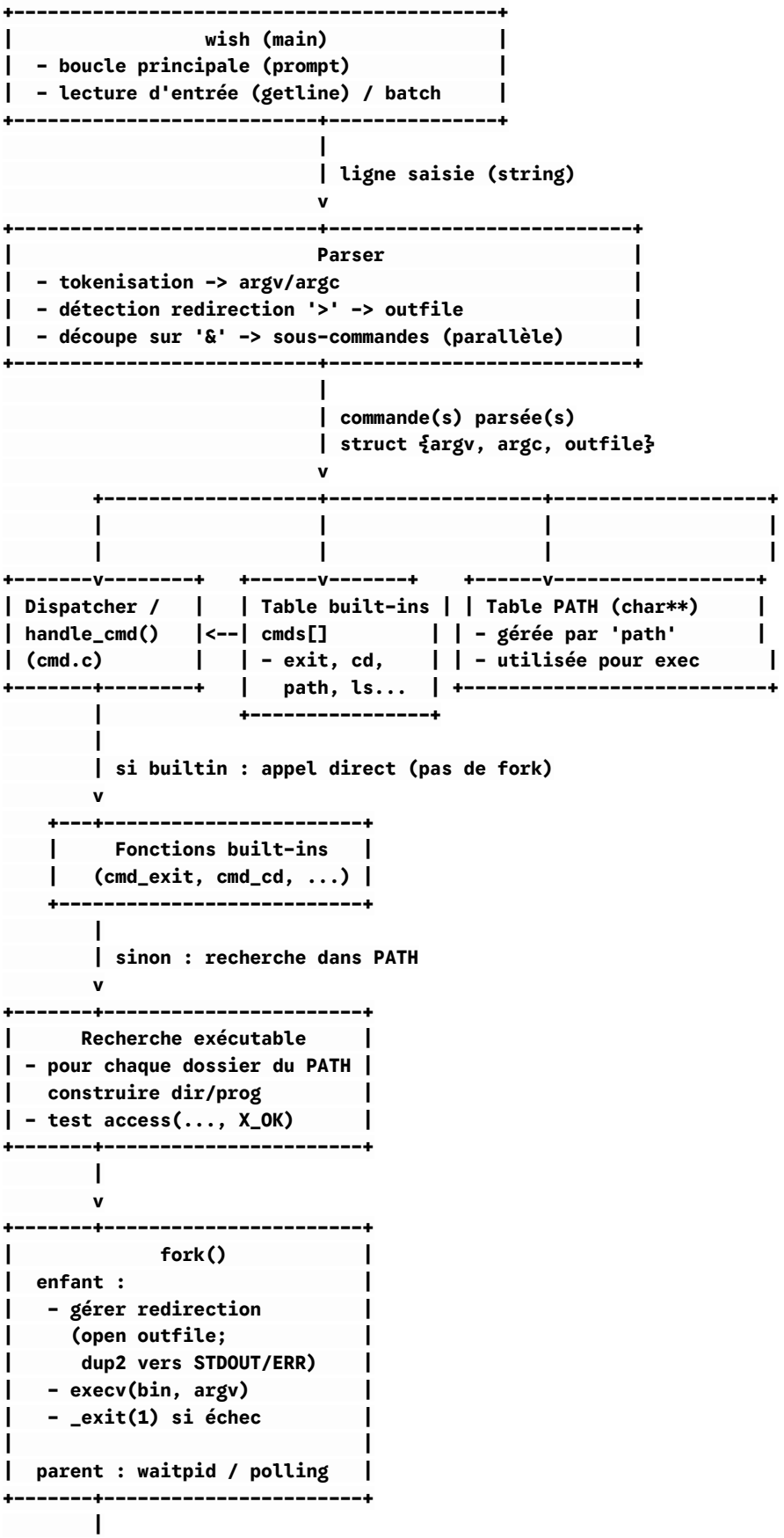
Résumé du projet

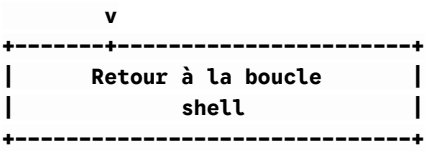
Objectif général :
Il nous est demandé de construire un interpréteur de commandes Unix minimal nommé wish. L'objectif est de mieux maîtriser l'environnement Linux, la gestion de processus (création, exécution, attente) et les mécanismes essentiels d'un shell (parsing des lignes, recherche d'exécutables via un PATH, commandes intégrées). Le shell doit permettre d'exécuter des programmes externes et d'implémenter quelques commandes internes (built-ins) essentielles.

Deux modes d'exécution sont attendus :
- Mode interactif : lancer ./wish sans arguments affiche un prompt "wish> " et attend des commandes saisies par l'utilisateur.
- Mode batch : lancer ./wish batch.txt fait lire et exécuter les commandes dans le fichier, sans afficher le prompt. Le projet exige de gérer EOF proprement (exit(0)) et de traiter les erreurs de fichier de batch (exit(1) si mauvais usage ou fichier illisible).

Fonctionnalités requises :
- Les commandes exit, cd et path ne sont pas lancés par fork/exec mais exécutés directement par le shell.
- Redirection de sortie : supporter la syntaxe "commande ... > fichier" qui redirige stdout et stderr vers le fichier nommé (création/troncation si nécessaire). La syntaxe doit être simple : une seule redirection à droite, un seul fichier ; les cas multiples sont des erreurs. (On ne teste pas forcément la redirection sur les built-ins.)
- Exécution parallèle : supporter la séparation des commandes par '&' pour lancer plusieurs commandes en parallèle (démarrer tous les processus puis attendre la fin de tous).
- Gestion stricte des erreurs : le projet impose un seul et unique message d'erreur, à savoir la chaîne "An error has occurred\n" écrite sur STDERR, à afficher pour toute erreur détectée par le shell (erreurs de parsing/syntaxe, échecs des built-ins, erreurs d'ouverture de fichier en batch, etc.). Pour certaines erreurs fatales (ex. invocation du programme avec trop d'arguments), le shell doit quitter avec exit(1).

Diagramme des parties du code





Explications

Le code est segmenté en deux fichiers *main.c* et *cmd.c* :

- *main.c* prend en charge les entrées :
  - le point d'entrée `main` traite les arguments de lancement du programme, on en a maximum 1 qui est le chemin d'un fichier source
  - ensuite les lignes de la source (soit ligne de commande interactive soit fichier) sont lues et passées une à une à la fonction `parse`
  - `parse` traite la ligne caractère par caractère et retourne tableau de tokens, un token est soit une suite de caractère non vide soit & ou >. Globalement `parse` supprime les caractères vides et isole les tokens. ex: `arg1 arg2&arg3 arg4> arg5&` retourne un tableau avec les tokens `arg1`, `arg2`, &, `arg3`, `arg4`, >, `arg5` et &
  - les tokens sont ensuite consommés dans `main` jusqu'à trouver un &, dans ce cas si la syntaxe de la commande est correcte (> suivi d'un seul argument, au moins un argument,...) la commande est envoyée à la fonction `handle_cmd` de *cmd.c*
- *cmd.c* prend en charge les commandes individuelles :
  - quand `handle_cmd` est appelée par *main.c*, on vérifie si la commande est built-in ou non en comparant au tableau des commandes built-in `cmds`. Si c'est le cas on lance la commande built-in `cmd_*` associée, sinon on cherche dans le path si il existe un exécutable et on le lance dans un processus enfant avec `fork` et `execv`.

Algorithmes utilisés

Parse : passer d'une chaîne brute à une représentation structurée (sous-commandes, argv, redirection). C'est une des parties les plus importantes car la moindre erreur de tokenisation peut entraîner des crashes si tous les cas ne sont pas testés et que des paramètres illégaux sont passés à execv.

```
function parse(line):
        # count number of tokens
        n_tokens = 0
        for character in line:
                ignore whitespaces
                if character == ‘>’ or character == ‘&’:
                        n_tokens += 1
                if character != whitespace:
                        n_tokens += 1
                        skip characters until character is &, > or whitespace

        # populate token table
        table = [ ] of max size n_token
        for character in line:
                ignore whitespaces
                if character == ‘>’ or character ==’&’:
                        write character to table
                if character != whitespace:
                        word = store characters until character is &, > or whitespace
                        write word to table

        write NULL to table
        return table
```

```
+-------------------------------------------+
|              Ligne d’entrée               |
|        echo hello > out.txt & ls          |
+-------------------------------------------+
```

1) Nettoyage
  - retirer '\n'
  - enlever espaces inutiles

2) Scan 1 : compter les tokens
  - ignorer espaces
  - '>' et '&' sont des tokens séparés
  → n = 6

3) Allocation
- argv = tableau de n+1 pointeurs
- argv[n] = NULL

4) Scan 2 : extraire les tokens
- si '>' ou '&' → token d'un caractère
- sinon → lire jusqu'à espace / '>' / '&'
- chaque token est alloué dynamiquement

5) Résultat final (n = 6)
argv = **["echo", "hello", ">", "out.txt", "&", "ls", NULL]**

handle_cmd : décide si la commande est built-in (appel direct) ou externe (recherche PATH → fork/exec).

```
function handle_cmd(args, path, outfile):
        name = args[0]

        # built-in command
        for each { cmd_name, cmd_func } in cmds[]:
                if name == cmd_name:
                        cmd_func(args)

        # external command: search in path table
        bin_path = null
        for path in path_table:
                bin_path = path/name
                if exist(bin_path)
                        break
        if bin_path == null:
                exit

        pid = fork()
        if pid == 0:  # child
                if outfile:
                        redirect stdout -> outfile
                        redirect stderr -> outfile
                execv(bin_path, args)
        else: #parent
                waitpid(pid)
```

Tests réalisés et état du code

Le code a été testé manuellement sur les cas de base de chaque opération pendant le développement et testé en parallèle avec les tests automatiques fournis. Le programme implémente toutes les fonctionnalités prévues et passe tous les tests.

```
cialson@OperatingSystems:~/Documents/IMT/Operating_Systems/Projet/processes-shell$ bash test-wish.sh
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
```

Le code a été adapté pour omettre l'erreur quand une ligne commence par &, le code quitte silencieusement à la place. C'était pour accepter le test 16, qui ne renvoie pas d'erreur à la ligne avec uniquement &, ça nous a un peu interpellé car ce n'est pas vraiment en accord avec le reste du programme (qui renvoie une erreur dès que la syntaxe est incorrecte). Cependant il suffit de remettre l'envoie d'erreur avant de passer à la ligne suivante pour corriger.

Le code est largement commenté, aéré et segmenté en blocs pour faciliter la lecture. Quelques macros sont utilisées pour réduire les répétitions. Certaines commandes préprocesseur sont utilisées pour les tests sur ma machine (dont la distribution linux n'est pas POSIX et ne place pas les programmes sous /bin) mais ces segments restent clairs.

La partie pour le traitement des lignes de caractère (notamment avec parse) est assez verbeuse et intriquée, elle pourrait gagner en clarté avec un style plus caractère par caractère et ainsi éviter les boucles imbriquées.

Conclusion

Le projet se prêtait bien pour itérer sur un programme simple et le complexifier en ajoutant à chaque fois des fonctionnalités. L'architecture était assez simple et n'a demandé que quelques réécritures seulement pour augmenter la clarté.

Les principales difficultés rencontrées dans le projet ont été d'obtenir un parsing robuste et la gestion de la mémoire. En effet, gérer les espaces multiples, opérateurs collés (cmd>out), fin de ligne sans espace, et garantir qu'on n'oublie pas le dernier token a plusieurs fois été des sources d'erreurs. Le parsing est aussi assez peu modulaire et ajouter des nouveaux tokens demande de vérifier les nouveaux cas à chaque ligne.

De plus, concernant la gestion mémoire, écrire du code qui alloue le bon espace mémoire et nettoie correctement les allocations partielles lors d'erreurs (malloc qui échoue) demande de la rigueur et est très verbeux, cela demande parfois des restructurations (notamment avec les macros) pour rendre le code digeste.

Enfin, l'un d'entre nous n'avait pas beaucoup programmé en C avant et a donc mis pas mal de temps avant de bien s'adapter à la syntaxe et aux règles particulières, notamment pour la gestion de la mémoire.

Ce projet nous a donc apporté une vraie mise en pratique des concepts systèmes. En effet, nous avons abordé les concepts de la lecture robuste d'entrée, la tokenisation et le parsing, ainsi que la gestion des processus avec fork/execv/waitpid. Il nous a aussi forcés à être rigoureux sur la gestion mémoire et le traitement des erreurs (vérifier retours de malloc/open/fork et libérer proprement).