



TELECOM NANCY

RAPPORT DE PROJET SHELL

Octobre 2021

Implémentation d'un shell en C : tesh

GROUPE 27

Étudiants :

Clément BAILLY-GRANDVAUX

Céline ZHANG

Numéro Étudiant :

32021255

32024925

Encadrant du projet :

Lucas NUSSBAUM



Remerciements

Nous tenons à remercier les tests de blancs de Monsieur Lucas NUSSBAUM qui nous ont permis de corriger une partie du code et de vérifier la conformité. Nous tenons également à remercier Lucas VALENTIN pour son aide sur quelques *debugs* du code, notamment sur l'ouverture de *file descriptor*, ses explications et la relecture du rapport. Nous nous remercions pour l'entraide durant ce projet, l'implication de chacun et les résultats obtenus suite aux efforts fournis.

Résumé

Le projet concerne l'écriture en langage C d'un shell réduit appelé **tesh**, qui s'apparente aux classiques **bash**, **tcsh**, **zsh**, **dash**. Le travail a pour objectif d'exploiter fichiers, de manipuler des processus avec l'API POSIX et donc de se familiariser avec ces outils. Le résultat du projet est un exécutable nommé **tesh**, il peut faire l'ensemble des commandes basiques d'un **shell** classique, notamment le traitement et le lancement des commandes (avec des conditions, des pipes, des redirections de fichiers), **tesh** peut se lancer en mode interactif ou non-interactif, en arrière plan, afficher les erreurs avec **-s**, éditer avec la **readline** (**-r**). Les principales difficultés étaient la prise en compte de multiples conditions (**&&**, **||**), l'exécution de commandes, et redirection vers multiples fichiers, la gestion des affichages pour les tests.

Table des matières

Introduction	3
1 Conception et implémentation	3
1.1 Affichage du prompt	3
1.2 Récupération de la ligne de commandes	3
1.3 Traitement des instructions	3
1.3.1 Découpage : <code>decouper</code>	4
1.3.2 Réordonner : <code>reorder</code>	4
1.4 Exécution des commandes	4
1.4.1 Création de <i>file descriptor</i> : <code>create_fd</code>	5
1.4.2 Exécution de fonctions : <code>run</code>	5
1.4.3 La pipe : <code> </code>	6
1.4.4 Les conditions : <code>&&</code> et <code> </code>	6
1.4.5 Le background : <code>&</code>	6
1.4.6 Le default : <code>;</code> ou fin	6
1.5 Diverses fonctionnalités	6
1.5.1 Mode non-interactif	6
1.5.2 Sortie sur erreur : <code>-e</code>	7
1.5.3 Readline	7
2 Difficultés rencontrées	7
2.1 Redirection, conditions et <i>file descriptor</i>	7
2.2 Tests	7
2.3 Amélioration possible de <code>fg</code>	7
Conclusion	7
Annexes	8
Mentions légales	8
Les déclarations sur l'honneur de non-plagiat	9
Cahier des Charges	11
Répartition du temps de travail	12
Structure du projet	12
Dépôt Gitlab	12
Le code source : <code>tesh.c</code>	12
Stratégie	12

Introduction

Dans le cadre de notre module RS, il est demandé de réaliser un projet d'implémentation en langage C d'un **shell**, nommé **tesh**. L'objectif de ce travail sera d'exploiter les outils API POSIX, de manipuler les fichiers et les processus pour réaliser un tel shell. Le produit final sera un exécutable ayant les fonctionnalités listées dans le cahier des charges tableau 1, le détail se trouve en Annexes figure 1. Des jeux de tests automatiques sont régulièrement, environ chaque semaine lancés afin de vérifier la conformité des résultats obtenus. Les travaux sont régulièrement envoyé sur le dépôt **gitlab**¹ de l'école **TELECOM Nancy** qui permet un suivi simplifié de l'avancement du projet.

1 Conception et implémentation

La plupart des fonctions utilisées dans l'implémentation des fonctionnalités décrites ci-dessous proviennent de la bibliothèque standard de C. Les headers utilisées sont : **stdio**, **stdlib**, **unistd**, **signal**, **stdbool**, **string**, **limits**, **sys/types**, **sys/wait**, **sys/stat**, **pwd**, **fcntl**, **getopt**, **dlfcn**.

1.1 Affichage du prompt

Pour afficher le *prompt* dès l'exécution de **tesh**, il suffit de récupérer :

- le nom de l'utilisateur **username** par la fonction **getlogin** ;
- le nom de l'hôte² **hostname** par la fonction **gethostname** avec en paramètres une variable où stocker le nom de l'ordinateur et la taille maximale³ de la chaîne possible ;
- le *path* du répertoire courant en utilisant la fonction **getcwd** avec en paramètres une variable **path** pour stocker le chemin et la taille maximale de la chaîne possible.

Ensuite, il ne reste qu'à concatener les variables de cette manière : **username@hostname:path**⁴.

1.2 Récupération de la ligne de commandes

La ligne de commandes tapée en entrée, juste après le *prompt*, est récupérée par la fonction **fgets**⁵ ayant pour paramètres une variable **entree_raw**⁶ pour stocker la chaîne récupérée, la taille du buffer⁷ et la source à laquelle récupérer, en particulier le **stdin**. Le tableau **entree_raw** est recopié dans un autre tableau de même taille, également alloué dynamiquement, afin de faciliter le traitement d'ordre de priorité qui suit.

1.3 Traitement des instructions

Le tableau de caractères, autrement dit la chaîne de caractères, sera découper en sous-chaînes dont le séparateur est l'espace, et sera rangé un autre tableau **entree** par la fonction implémentée à cet effet **decouper**. Après le découpage, il faudra distinguer les commandes à exécuter, les paramètres et les symboles particuliers redirections (<, >, », |), de conditions (&&, ||), d'exécution parallèle (;), en arrière plan (&), et ensuite réordonner les instructions afin de respecter les priorités d'exécution.

1. <https://gitlab.telecomnancy.univ-lorraine.fr/Clement.Bailly-Grandvaux/rs2021-zhang-bailly-grandvaux>

2. ici ordinateur

3. définie : **HOST_NAME_MAX**

4. se terminant par un espace

5. de la bibliothèque standard **stdio**

6. tableau de **char**, représente une chaîne de caractères, alloué dynamiquement de taille **BUFFER_LENGTH**

7. **BUFFER_LENGTH** = 4096

La fonction **reorder** a été implémentée pour cela. Pour faciliter le traitement, il est nécessaire de procéder morceau par morceau, le but étant de lire une commande et un symbole. Par exemple : `ls | grep a | grep b`, il faudrait traiter "`ls |`" ensemble, puis "`grep a |`", et enfin "`grep b`". De cette manière, la fonction peut être exécuter et sa sortie traitée de la manière convenable pour la prochaine commande. C'est une sorte d'étape de 'parsing' pour découper l'exécution des instructions.

1.3.1 Découpage : decouper

La fonction **decouper**, de paramètres `char* entree`, `char** sortie`, de retour `int i` renvoyant le nombre d'élément, remplace dans la chaîne d'entrée **entree** les espaces par des caractères de fin `'\0'`, des sous-chaînes sont ainsi créées. Elles sont rangées au fur et à mesure dans un tableau **sortie** en incrémentant `i`. En réalité, ce sont les adresses des premiers caractères de la chaîne qui y sont rangés. Le tableau **sortie** se termine par un pointeur `NULL` lorsqu'il rencontre un caractère saut de ligne `'\n'`, donc c'est la fin de toute la ligne de commandes.

1.3.2 Réordonner : reorder

La fonction **search** a en paramètres deux `char** base` et `end` qui représentent le début et la fin du bloc à traiter, un `int* spe_i` qui permet de stocker le numéro du symbole spécial (`;`, `|`, `&&`, `||`, `&`), le retour de **search** est un `char** next`, qui donne la position du symbole spécial. Le numéro du symbole permet de retrouver le symbole via un tableau **special_string** déclaré en global statique. La fonction **reorder** prend en paramètres un `char** base` et `char** next`, **base** correspond au curseur du traitement, et **next** la position du symbole. Le but est de balayer tout le tableau de chaîne de caractères, en utilisant la **base**, pour retrouver un symbole redirection type chevron (`<`, `>`, `»`). Dès qu'un symbole est détecter :

- si le symbole est `>` ou `»`, les deux éléments précédents sont stockés dans deux pointeurs `char* redirect_to` et `redirect_to_file`, qui sont respectivement le symbole de redirection et le fichier destination. Leur emplacement initial sont mis à `NULL` et on incrémente de 2 le nombre d'éléments retirés `nb_supprime`;
- si le symbole est `<`, les deux éléments précédents sont stockés dans deux pointeurs `char* redirect_from` et `redirect_from_file`, qui sont respectivement le symbole de redirection et le fichier source. Leur emplacement initial sont mis à `NULL` et on incrémente de 2 le nombre d'éléments retiré `nb_supprime`;

Ce procédé est itéré jusqu'à la fin, lorsque la **base** tape sur la fin **next**. Jusque là, toutes les redirections ont été retirées, seules sont gardées les derniers `>` ou `»` et `<` ainsi que leur fichier correspondant car lors de l'exécution les autres sont 'ignorés'. Pour replacer les symboles et les fichiers, la première étape est déplacé tous éléments vers le début sans laisser de trou (regroupement vers le début en respectant l'ordre). Puis, à la deuxième étape, tous les éléments sont déplacés, en conservant l'ordre, du nombre de décalage nécessaire pour insérer les symboles et les fichiers restant. Ce nombre est **decalage**, il vaut 2 lorsqu'on a une redirection, il vaut 4 lorsqu'on a soit `>`, soit `»`, et `<`. S'il vaut 2, il faut insérer le symbole dans le premier emplacement libre, puis le fichier juste après. S'il vaut 4, il faut insérer **redirect_to** et son fichier, puis **redirect_from** et son fichier, dans cet ordre dans les emplacements libres successivement. Un exemple pour la ligne de commande "`< a cat > b > c`" se trouve dans le tableau 1.

1.4 Exécution des commandes

Une fois que la ligne de commandes a bien été pré-traitée. L'exécution de celle-ci devient assez simple. Un curseur `char** base` qui pointe initialement vers le premier token (élément), un stop

Step	base[0]	base[1]	base[2]	base[3]	base[4]	base[5]	base[6]	to	from
0	<	a	cat	>	b	>	c		
1	NULL	NULL	cat	>	b	>	c		< a
2	NULL	NULL	cat	NULL	NULL	>	c	> b	< a
3	NULL	NULL	cat	NULL	NULL	NULL	NULL	> c	< a
4	cat	NULL	NULL	NULL	NULL	NULL	NULL	> c	< a
5	NULL	NULL	NULL	NULL	cat	NULL	NULL	> c	< a
6	>	c	<	a	cat	NULL	NULL	> c	< a

TABLE 1 – Exemple d'application de fonction `reorder`

`next` de même type qui pointe initialement vers le premier caractère spécial (`;`, `|`, `&&`, `||`, `&`), une fin `end` de même type qui pointe vers le dernier élément, la fin du tableau, sont placés.

La procédure est la suivante :

- Retirer le caractère spécial en mettant `NULL` ;
- Aller dans le `case` qui correspond au caractère spécial
- Selon le caractère spécial, détaillé ci-après, les fonctions `create_fd` et `run` sont appelés, cela va créer les *files descriptors* nécessaires à l'exécution de la commande, et selon les options données à `run`, il mettra le résultat dans un *file descriptor* `fd` ou dans la sortie standard `stdout` ;
- le curseur `base` est mis à jour sur le token après le caractère spécial ;
- le pointeur du caractère spécial est mis à jour sur le suivant.

Ce procédé est itéré jusqu'à la fin de la ligne de commandes, donc quand `base` croise `end` ou jusqu'à son interruption par un code d'erreur.

1.4.1 Création de *file descriptor* : `create_fd`

Comme le nom l'indique, cette fonction permet de créer un descripteur de fichier, elle prend en paramètre le pointeur de la `base` et le pointeur de la variable `last_out` qui sauve la dernière sortie de la dernière commande réalisé, le retour est un descripteur de fichier `fd`. Cette fonction crée un descripteur de fichier `fd` en cas d'existence d'un élément de redirection entre `base` et la première commande à exécuter. Grâce à la fonction `reorder` si le premier élément n'est pas un chevron, alors rien n'a besoin d'être créé, le retour est le `stdout` par défaut.

S'il existe au moins un chevron :

- `base[0]` est `>` ou `>>` : récupération du chemin du fichier, création s'il n'existe pas, ouverture pour écriture, écriture par écrasement pour `>` ou écriture par ajout, remplacement de `fd` et incrémentation de 2 de la `base` ;
- `base[0]` est `<` : si la dernière sortie est le `stdin` alors fermeture, dans tous les cas récupération du chemin du fichier, création du descripteur de fichier, ouverture pour lecture, remplacement de `last_out` par ce descripteur de fichier et incrémentation de 2 de la `base`.

1.4.2 Exécution de fonctions : `run`

Cette fonction permet l'exécution des commandes, elle a en paramètres le fichier de la fonction (commande), les arguments, les descripteurs de fichier en entrée `input` et sortie `out`, un pointeur `pid_t* child_pid_adr` où sera mis le `pid` du processus fils créer pour l'exécution de la commande. S'il n'y a pas de descripteur de fichier créer, c'est-à-dire `out=-1`, un `pipe` est créé entre `fd[0]` et `fd[1]`. Ensuite, un processus fils est créé dont le `pid` est stocké dans `*child_pid_adr` (`child_pid`).

Dans le processus fils :

- Si l'entrée `input` n'est pas le `stdin` alors `input` est dupliqué dans `stdin` et est fermé.
- Si l'entrée du `pipe` n'est pas le `stdout` alors il est dupliqué dans celui-ci et est fermé.
- Si la sortie du `pipe` n'est pas le `stdout` alors le fermer.
- Exécution de la commande avec la fonction `execvp`, elle prend le fichier et les arguments.

Dans le père :

- Si l'entrée `input` n'est pas le `stdin` alors il est fermé.
- Si l'entrée du `pipe` n'est pas le `stdout` alors il est fermé.
- Retour du descripteur de la sortie de l'exécution.

1.4.3 La pipe : |

Le commande et ses arguments se lancent avec `run`, s'il y a un `file descriptor` `fd`, il sera dans les paramètres de la fonction, le dernier résultat `last_out` (par défaut le `stdin` est mis en `input` de la commande, le `pid` du processus d'exécution est gardé. Après l'exécution, s'il y a un descripteur dans `fd`, le résultat de sortie de la fonction y est et le `last_out` doit être nul. Sinon, le `last_out` est le résultat de la commande prêt à passer en `input` à la commande suivante. On n'attend pas le retour du fils.

1.4.4 Les conditions : && et ||

S'il y a un `fd` à créer, il est fait et est passé dans l'exécution `run`. On attend le retour avec `waitpid` pour attendre et récupérer le code de retour du processus fils. Et selon la condition, on met `run_next` à `true` ou à `false`, ceci indique si, la commande après le symbole sera lancée ou non pour la prochaine itération.

1.4.5 Le background : &

Les commandes suivies de `&` sont lancées en arrière plan. De la même manière, le descripteur de fichier est créé si nécessaire, on exécute la commande. On n'attend pas le retour, le `pid` du processus fils qui exécute la commande est stocké dans le tableau des `pids` de processus en `background`, `pid_tab` et il est affiché sur la console. Une fonction `fg` a été implémentée dans le but de récupérer les commandes lancées en arrière plan, on donne le `pid` du fils qu'on veut récupérer, et la fonction fouille dans le tableau, récupère son retour et affiche le résultat de l'exécution ainsi que son `pid` et le code de retour. Si aucun argument est donné, la fonction attend le retour d'un processus (le premier qui termine).

1.4.6 Le default : ; ou fin

Dans ce cas, la commande est exécutée comme pour le cas précédent, mais on attend le retour de la commande et le `last_out` est mis à `stdin` par défaut.

1.5 Diverses fonctionnalités

1.5.1 Mode non-interactif

Le mode non-interactif correspond au non affichage du prompt. Il est automatique activé quand le programme détecte qu'il n'est pas dans un terminal (avec `isatty`) ou qu'on lui a donné un script en paramètre.

1.5.2 Sortie sur erreur : -e

La sortie sur erreur est implémenté en regardant à chaque `waitpid` si le statut est bon. Si il ne l'est pas on arrête l'exécution du programme. L'option est activé avec le paramètre `-e`.

1.5.3 Readline

Readline est une librairie qui simplifie la vie de l'utilisateur pour écrire des lignes de commande. On l'a implémenté son utilisation avec `dlopen` qui va importer la librairie. Puis avec `dlsym` on récupère les fonctions utile à readline. Après cela on doit remplacer `fgets` par `readline` et lui donner le prompt. L'option est activé avec le paramètre `-r`.

2 Difficultés rencontrées

2.1 Redirection, conditions et *file descriptor*

Au début, nous avons eu des soucis sur les tests de redirections et ceux sur les conditions, notamment pour les multiples redirections et les suites de conditions `&& ||`. Pour résoudre le problème nous avons dû revoir l'ensemble du code pour réordonner les commandes et l'exécution. De plus, la création des descripteurs de fichiers a posé problème, notamment d'une part pour les oublies d'argument `O_TRUNC` ou `O_APPEND`, les droits par exemple et d'autre part la fermeture des descripteurs.

2.2 Tests

Lors des derniers tests blancs, nous avons eu trois tests qui ne passaient pas, `anchor2`, `fd_leak1`, `readline1`. Ces mêmes tests passaient lors des deux précédentes sessions. La dernière modification des tests blancs, le 10 décembre 2021, a donné ces résultats, nous n'avons pas pu retrouver le soucis par manque de temps et d'information. Malheureusement, nous n'avons pas pensé à noter nos propres tests, tout au long du projet, après chaque finalisation d'une fonctionnalité nous la testons avec une dizaine de commandes. C'est de cette manière que nous avons décider de réordonner la ligne de commandes avant de la traiter. Par exemple : `ls -l | grep a < file`, il y a deux entrées, avec la fonction `reorder` nous obtenons `ls -l | < file grep a` et par notre méthode d'exécution, ce sera le contenu du file qui sera l'entrée de `grep`.

2.3 Amélioration possible de fg

La fonction `fg` fonctionne assez bien pour les tests simples et les tests blancs, cependant elle peut causer des erreurs si par exemple on exécute : `fg > a`. Des améliorations sont possibles, cela nécessite de revoir l'exécution des commandes.

Conclusion

Dans l'ensemble, nous avons bien réalisé ce qui est demandé dans le cahier des charges. Les fonctionnalités implémentées fonctionnent bien, malgré les quelques soucis sur le dernier test blanc, mais de notre côté, sur ces mêmes tests, nous n'avons pas identifié de problème. La partie la plus compliquée était certainement le *parsing* de la ligne de commandes et l'exécution des commandes, beaucoup de scénarios devait être pris en compte, mais nous avons assez bien réussi cela, c'est l'une partie dont nous en sommes la plus fière, notamment savoir que notre shell est capable de lancer `> a ls -la` nous récompense du temps et du travail que nous y avons consacré.

Annexes

Mentions légales

Ce projet n'est pas destiné à un usage commercial, ainsi, les images présentés, notamment les images de tests, d'applications ou de gestion de projet, ne sont pas destinées à la publication.

Cependant, le caractère strictement scolaire de ce projet nous autorise à les inclure en accord avec :

- Code civil : articles 7 à 15, article 9 : respect de la vie privée
- Code pénal : articles 226-1 à 226-7 : atteinte à la vie privée
- Code de procédure civile : articles 484 à 492-1 : procédure de référé
- Loi n78-17 du 6 janvier 1978 : Informatique et libertés, Article 38

Cette version⁸ du rapport a été finalisée le 12 Décembre 2021.

8. version 1

Les déclarations sur l'honneur de non-plagiat

Déclaration sur l'honneur de non-plagiat

Je, soussignée,

Nom, prénom : ZHANG, Céline

Étudiant ingénieur inscrit en 2e année à TELECOM Nancy

Numéro de carte étudiante : 32024925

Année universitaire : 2021 - 2022

Auteur, en collaboration avec Clément BAILLY-GRANDVAUX du rapport :

Implémentation d'un shell en C : tesh

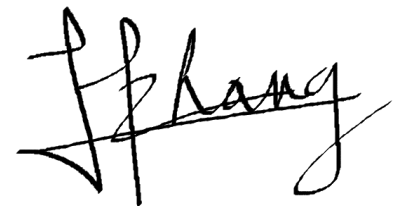
Je déclare sur l'honneur que ce rapport est le fruit d'un travail personnel et que je n'ai ni contre-fait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier, texte ou code informatique, afin de la faire passer pour mienne.

Je certifie donc que le travail rendu est un travail original et que les sources utilisées, notamment pour les formulations, les idées, les documentations, les raisonnements, les analyses, les schémas ou autres créations ont été mentionnées conformément aux usages en vigueur.

Je suis consciente que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université et qu'il peut être sévèrement sanctionné.

Fait à Nancy, le 12/12/2021

Signature de l'étudiant :

A handwritten signature in black ink, appearing to read 'C Zhang', with a stylized flourish at the end.

Déclaration sur l'honneur de non-plagiat

Je, soussigné,

Nom, prénom : BAILLY-GRANDVAUX, Clément

Étudiant ingénieur inscrit en 2e année à TELECOM Nancy

Numéro de carte étudiante : 32021255

Année universitaire : 2021 - 2022

Auteur, en collaboration avec ZHANG Céline du rapport :

Implémentation d'un shell en C : tesh

Je déclare sur l'honneur que ce rapport est le fruit d'un travail personnel et que je n'ai ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier, texte ou code informatique, afin de la faire passer pour mienne.

Je certifie donc que le travail rendu est un travail original et que les sources utilisées, notamment pour les formulations, les idées, les documentations, les raisonnements, les analyses, les schémas ou autres créations ont été mentionnées conformément aux usages en vigueur.

Je suis conscient que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université et qu'il peut être sévèrement sanctionné.

Fait à Nancy, le 12/12/2021

Signature de l'étudiant :

Bailly - C

Cahier des Charges

L'objectif général du projet est de réaliser un interpréteur de commandes, appelé **tesh**, très fortement inspiré des shells Unix classiques comme **sh**, **dash**, **bash**, **tcsh**, **zsh**.

Fonctionnement de base. Une fois lancé, le shell attend que l'utilisateur entre une commande, puis l'exécute, attend sa fin, et attend la commande suivante. Pour simplifier l'implémentation, les paramètres des commandes ne peuvent pas contenir d'espaces (il est donc inutile de gérer des cas comme `./monprogramme "premier argument" 'deuxieme argument'`), et les séparateurs de commandes ou caractères de redirections (`>`, `>>`, `|`, `&&` etc) sont forcément sur des *mots* distincts (inutile de gérer `./monprog>sortie`, ce sera toujours écrit `./monprog > sortie`).

Commande interne : cd. La commande `cd` doit être traitée de manière particulière, car, si elle était exécutée dans un processus fils, son effet serait limité à ce processus fils, et n'aurait pas d'effet sur la commande suivante. Il faut donc l'implémenter comme une commande interne du shell (on parle de `built-in`).

Affichage d'une invite de commande (*prompt*). En mode interactif, le shell affiche un *prompt* de la forme `"USER@HOSTNAME:REPCOURANT$ "`. `USER` est le nom de l'utilisateur courant, `HOSTNAME` est l'hostname de la machine (cf `gethostname(2)`), et `REPCOURANT` est le répertoire courant.

Enchaînement conditionnel de commandes. Comme dans `bash` : `cmd1 ; cmd2` doit exécuter `cmd1` puis `cmd2`. `cmd1 && cmd2` doit exécuter `cmd1`, puis `cmd2` seulement si `cmd1` a terminé avec le code 0. `cmd1 || cmd2` doit exécuter `cmd1`, puis `cmd2` seulement si `cmd1` a terminé avec un code différent de 0.

Redirections d'entrée et de sortie. Comme dans `bash` : `cmd > fichier` doit exécuter `cmd` en redirigeant sa sortie vers `fichier`; `cmd >> fichier` doit exécuter `cmd` en ajoutant sa sortie à la fin de `fichier`; `cmd < fichier` doit exécuter `cmd` en utilisant `fichier` comme entrée; `cmd1 | cmd2` doit exécuter `cmd1` et `cmd2` en redirigeant la sortie de `cmd1` vers l'entrée de `cmd2` (pensez à généraliser : `cmd1 < fichier1 | cmd2 | cmd3 >> fichier2`).

Mode interactif et non-interactif. **tesh** doit aussi pouvoir s'exécuter en mode non-interactif, soit si un fichier contenant les commandes à exécuter est passé en paramètre (`./tesh monscripttesh`), soit si l'entrée standard de **tesh** n'est pas un terminal (à tester avec `isatty(3)`). Quand **tesh** s'exécute en mode non-interactif, alors il n'affiche pas de *prompt*.

Sortie sur erreur (-e). Quand l'option `-e` est passée à **tesh**, alors **tesh** s'arrête dès qu'une commande termine avec un code de retour différent de 0 (c'est particulièrement utile en mode non-interactif).

Lancement de commandes en arrière plan. Si une commande est suivie du séparateur de commandes `&`, alors elle doit être lancée en arrière plan. Le shell doit simplement afficher son PID, sous la forme `[pid]` (par exemple `[42]`). Ensuite, une commande interne `fg` doit permettre de ramener la commande au premier plan et d'en attendre la fin. Si `fg` est appelé sans paramètre, alors le shell attend la fin d'un des processus en arrière plan. Si un PID est passé en paramètre, alors le shell attend la fin de ce processus. Dans les deux cas, le shell doit afficher le PID du processus en arrière plan qui a terminé, et son code de retour, sous la forme `[pid->retcode]` (par exemple `[42->2]`, si le processus 42 a terminé avec le code de retour 2).

Édition de la ligne de commande du shell avec *readline*. Si l'option `-r` est passée au shell, alors le shell doit utiliser la bibliothèque *readline* pour permettre l'édition interactive de la ligne de commande, et la gestion de l'historique.

Chargement dynamique de la bibliothèque *readline*. Au lieu de lier le programme avec la bibliothèque *readline*, utilisez `dlopen` pour charger dynamiquement cette bibliothèque.

FIGURE 1 – Le cahier des charges complet

Répartition du temps de travail

Thèmes	Clément Bailly-Grandvaux	Céline Zhang
Conception	5h	2h
Implémentation	15h	18h
Tests	6h	4h
Rédaction	2h	14h
Total	28h	38h

TABLE 2 – Répartition temps de travail

Structure du projet

Dépôt Gitlab

Le dépôt `gitlab` contient :

- `.gitignore` : contient les noms de répertoire ou fichier à ignorer lors du *push*.
- `.gitlab-ci.yml` : contient les instructions pour mettre à jour les versions et initialiser le dépôt.
- `AUTHORS` : contient les auteurs du projet.
- `Makefile` : contient les instructions pour construire l'exécutable (compiler le code de `tesh.c` avec les options nécessaires).
- `README.md` : contient les indications d'utilisation et les divers commentaires sur le projet.
- `tesh.c` : contient le code source du shell `tesh`.

Le code source : `tesh.c`

L'organisation du code est assez simple, elle a trois parties :

- `import` et `define` : où se trouve les imports et les définitions de constantes.
- déclarations de variables et fonctions : où se les variables globales et l'implémentation implémentation de l'ensemble des fonctions utilisées.
- `main` : le corps donnant les instructions d'exécution des fonctions.

Stratégie

Les étapes stratégiques du shell sont :

- l'affichage du *prompt* ;
- la récupération de la ligne de commandes entrée par l'utilisateur ;
- le traitement de l'instruction (notamment identifier les symboles, les commandes et ordonner les priorités) ;
- l'exécution des commandes ;

Table des figures

1	Le cahier des charges complet	11
---	---	----

Liste des tableaux

1	Exemple d'application de fonction <code>reorder</code>	5
2	Répartition temps de travail	12