Rapport de projet

FERSTLER Amélie, GOUVILLE Tom

Projet RS : Tesh

Sommaire

1 Choix techniques

1.1 Fonctionnement général

L'architecture générale de tesh repose sur une boucle qui lit l'entrée standard ligne par ligne, jusqu'à atteindre un `EOF`. Chaque ligne est ensuite découpée en tokens (tableau de `char*` se terminant par `NULL`) par la fonction `parse` [1]. Les tokens sont analysés une première fois par la fonction `command_scheduler` qui va se charger de déterminer la présence d'un `&` à la fin de la commande et de la lancer où non en arrière-plan. Si la commande contient un `&`, un nouveau processus est créé avant la poursuite de l'analyse des tokens dans le processus fils (sinon, on la fait dans le processus père). Les tokens sont analysés par la fonction `command_runner`, qui lit chaque symbole de la commande, la découpe en sous-tableaux de tokens correspondant aux appels de programmes et les exécute en fonction des séparateurs de commandes (`||`, `&&`, `|`, ...). Enfin, les commandes sont exécutées avec `launch_process`.

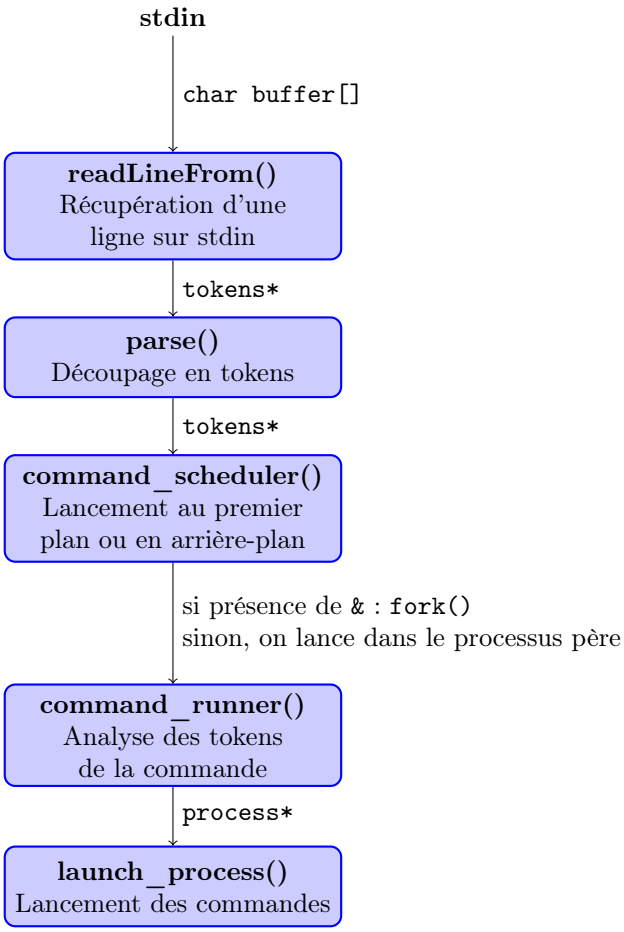

FIGURE 1 – Schéma de fonctionnement de tesh

1.2 Lancement des processus

Le lancement des processus ainsi que les *pipes* sont réalisés au moyen de la struct `process`, qui permet une utilisation de plus haut niveau des processus. Cela permet, dans le reste du code, de lancer et de faire des *pipes* entre processus sans se soucier de l'implémentation. On peut, par exemple, créer un processus dont l'entrée est reliée à la sortie d'un autre processus avec `process* p2 = piped_process(p1)`. La fonction va automatiquement relier les *files descriptors* correspondants. Les processus peuvent être lancés pour écrire dans le terminal avec `lauch_and_print`, dans un tube avec `launch_and_pipe` ou dans un fichier avec `pipe_to_file`.

1. Une description plus précise du fonctionnement et de l'utilisation des fonctions se trouve dans les fichiers `.h` du projet.

1.3 Interprétation des commandes avec `command_runner`

L'interprétation des commandes est une fonction qui a été implémentée de manière itérative.
La première étape pour rendre `command_runner` simple et fonctionnelle a été de gérer la présence de `;` dans la liste des tokens que l'on a auparavant instanciée. En effet, ces `;` permettent de séparer les commandes entre elles. Ils servent donc à informer quand une commande est terminée et qu'on peut l'exécuter.
Pour pouvoir exécuter une commande, il est nécessaire de disposer de tous ses éléments. C'est pourquoi une liste de tokens annexe, `theCommand`, a été mise en place pour stocker au fur et à mesure les tokens que l'on rencontre jusqu'à ce qu'un `;` soit rencontré. Ainsi, on peut exécuter dans un processus la commande, la vider puis continuer à explorer le reste des tokens parsés et à les stocker dans `theCommand`.

Une fois que les commandes simples peuvent être exécutées, il est possible d'enchaîner des commandes grâce aux instructions `&&` et `||`. Dans ce cas, il faut non seulement exécuter la commande précédente, mais également vérifier le statut du processus dans lequel elle a été exécutée. Selon ce statut, plusieurs scénarii peuvent s'enclencher :

- la commande suivante peut être exécutée ; dans ce cas on vide la liste et on recommence à regrouper les tokens ;
- la commande suivante ne peut pas être exécutée ; dans ce cas on met à jour un booléen qui indique qu'il ne faut pas regrouper les prochains tokens jusqu'à une autre instruction intéressante (par exemple `;` ;
- la commande suivante ne peut pas être exécutée et l'arrêt sur erreur est mis en place ; dans ce cas tesh s'arrête tout simplement.

Ensuite, la fonction doit également se charger de prendre en compte les redirections de commande, comme `<` ou `»`. Une nouvelle fonction d'exécution des commandes a été crée pour ces cas-là, `pipe_to_file`. Elle permet d'ouvrir le fichier sur lequel la commande est passée, d'exécuter la commande et d'enregistrer sa sortie dans le fichier. Après l'exécution de la commande, l'analyse des tokens reprend un token plus loin, puisque le fichier utilisé est spécifié après la redirection concernée.

La dernière étape dans la construction de `command_runner` est de détecter la présence des commandes *built-in* comme `cd` et `fg`. C'est une vérification qui doit être faite au tout début de l'analyse des tokens. Dans ce cas, pour chacune de ces instructions, les fonctions qui les exécute sont appelées.

1.4 Commandes en arrière plan

Pour les commandes en arrière plan, nous avons fait le choix de lancer toute la ligne précédant un `&` en arrière-plan. Pour cela, nous créons un nouveau processus qui va se charger d'exécuter les différentes commandes de la ligne séquentiellement. On récupère le pid de ce processus, que l'on affiche dans le shell sous la forme `[pid]`. Ce pid est aussi stocké dans une file de pid qui permet à la commande `fg` de retrouver un des pid de processus en arrière-plan ou de vérifier que le pid entré par l'utilisateur appartient bien à tesh.

1.5 Tests

Les tests réalisés pour le projet sont inspirés des tests automatiques pour leur contenu et des tests du pipeline (fourni avec le template du projet) pour leur fonctionnement. Les tests sont donc exécutés à l'aide d'un script bash et certains utilisent la bibliothèque Python pty[1] pour lancer tests dans un terminal virtuel. Ces tests ont étés ajoutés dans le pipeline git pour assurer une validation du code à chaque push sur le git.

Des tests unitaires supplémentaires avec la bibliothèque snow[2] ont été commencés mais ils ont finalement été abandonnés.

2 Problèmes rencontrés et solutions

2.1 Lire des commandes depuis un pipe ou une redirection

L'un des premiers problèmes rencontrés a été de lire les entrées lors d'un *pipe* sur tesh ou d'une redirection avec un fichier de script. La première technique retenue a été d'utiliser la fonction `getline`. Nous avions le comportement attendu sur les *pipes* mais pas sur les redirections (par exemple `./tesh < ti`). `getline` retournait parfois plusieurs fois une même

ligne. Pour résoudre ce problème, nous avons utilisé la fonction `read` et développé une nouvelle fonction pour remplacer `getline` : `char* readLineFrom(int fd)`, qui retourne une ligne du descripteur de fichier passé en argument, ou `NULL` si un `EOF` est rencontré.

2.2 PID consécutifs avec les commandes en background

Le seul problème que nous n'avons pas pu résoudre est celui d'avoir deux pid consécutifs lors de lancements de commandes en arrière-plan. En effet, l'architecture choisie pour tesh nous contraint à interpréter les commandes lancées avec `&` dans un processus séparé. Ainsi, lorsqu'on lance une commande en arrière plan, on a deux créations de processus (un pour interpréter la commande et un pour lancer le `exec`), ce qui empêche de lancer deux commandes avec des pid consécutifs, et donc de passer les tests vérifiant cette condition.

Nous n'avons pas trouvé de parade à ce problème qui n'implique pas de changer la structure du projet et, ayant terminé par l'implémentation des processus en arrière-plan, nous n'avons pas eu le temps de refaire l'interprétation des commandes.

2.3 Enchaînements de `&&` et `||`

Nous avons eu quelques problèmes lorsqu'il s'agissait d'enchaîner des commandes avec des `&&` ou des `||`. En effet, si les premières instructions étaient exécutées correctement, la fonction `command_runner` sautait une partie du reste des tokens à analyser. Le résultat final affiché n'était donc pas du tout celui escompté.
Pour remédier à ce problème, nous avions tout d'abord introduit une variable `isSkipped` pour enregistrer la nécessité ou non de passer des commandes. Une fois que l'analyseur de tokens tombait sur certaines commandes, comme par exemple un `;`, ce booléen redevenait faux et l'analyse pouvait reprendre. Mais pour les commandes `&&` et `||`, ce n'était pas suffisant puisque l'exécution de la commande suivante dépendait de si la commande précédente était un succès ou non (ou pouvait passer la commande suivant le `&&` seulement si la commande d'avant avait fonctionné). Une deuxième variable `isSuccessful`, qui notait si l'exécution de la commande précédente était une réussite ou non, a été ajoutée. La vérification de ce booléen permet, le cas échéant, de rendre `isSkipped` faux et ainsi de reprendre l'analyse et l'exécution de la commande suivante. Ainsi, plus aucune commande qui ne soit nécessaire n'est passée. Il est désormais possible d'enchaîner des `&&` et des `||`.

3 Temps passé sur le projet

Tableau récapitulatif des heures passées sur le projet.

Tâche	**Amélie F.**	**Tom G.**
Conception	5h	6h
Implémentation	9h	8h
Tests	0h	3h
Rapport	2h	2h
Total	**16h**	**19h**

Références

[1] pty - python documentation. `https://docs.python.org/3/library/pty.html`.

[2] *mortie/snow, github*. `https://github.com/mortie/snow`.