Travail pratique #1 - IFT-2245

Christophe Apollon-Roy (920403)
et
Amélie Lacombe Robillard (20016735)

February 10, 2018

Introduction

Dans le cadre de ce travail pratique, nous avons dû implémenter une ligne de commande similaire au Shell dans le langage C. Pour se faire, nous avons écrit un programme qui lit dans un premier temps le texte entré par l'usager, le décompose en commandes et arguments, puis invoque les appels systèmes adéquats. Dans ce rapport, nous présenterons les principales problématiques rencontrées au fil de l'implémentation ainsi que les solutions que nous avons adoptées.

Les problèmes discutés seront, dans l'ordre : la décomposition du texte fournit en entrée, la répartition des responsabilités du Shell, la priorité des opérations et le raffinement de la décomposition (parsing) du texte. Si la donnée du travail a été plutôt simple à comprendre, se familiariser avec la programmation système de style POSIX ainsi que lire la documentation des librairies C associées (unistd.h, sys/wait.h) a pris un certain temps, de même que configurer un environnement linux pour tester notre programme.

1. Décomposition du texte en commandes et arguments

Ayant peu d'expérience de programmation en C, il n'a pas toujours été facile d'écrire du code s'exécutant comme nous le souhaitions. Ce problème relève davantage de notre inexpérience que de la complexité de la donnée du travail. Bien qu'il ait été plutôt facile de comprendre conceptuellement comment le texte lu en entrée doit être décomposé en commandes et arguments, écrire le code adéquat s'est révélé plus ardu que prévu, à cause notamment de la façon dont les strings sont représentées en C, soit comme tableaux de caractères et pointeurs, et des problèmes de gestion de la mémoire manuelle qui en découlent. À cette difficulté s'est ajouté le problème de manipulation des strings, et nous

avons d'abord dû nous documenter sur la libraire string.h, ses fonctions et leur usage.

Tout au long du travail, nous nous sommes heurtés à plusieurs petits problèmes qui auraient sans doute semblés évidents à un programmeur plus expérimenté. Par exemple lorsqu'une ligne de texte est lue avec la fonction *getLine()*, il reste un charactère de fin de ligne '\n' invisible qui doit être enlevé manuellement, sans quoi il peut devenir une source d'erreur. Nous avons également souvent fait face au message d'erreur "Segmentation fault" signalant une mauvaise manipulation de pointeurs ou un accès de mémoire au-delà des limites d'un tableau, mais dont la cause n'était pas toujours facile à identifier rapidement. Manipuler un tableau de string (et donc un tableau de pointeurs) s'est révélé tout particulièrement délicat.

Coder en C nous a également forcé à réfléchir et prendre des décisions pour des détails que nous avions l'habitude de prendre pour acquis, comme l'initialisation d'un tableau dont la taille n'est pas prédéterminée. Pour contourner ce problème, nous avons choisis dans certains cas de précalculer le nombre d'éléments du tableau avant son initialisation, avant de réellement copier ces éléments dans celui-ci. Cette solution a un coût peu significatif au niveau de l'efficacité de notre programme, et limite l'espace mémoire gaspillé. Dans d'autres cas, la solution a été de déterminer une taille maximum au nombre d'éléments que peut contenir le tableau.

2. Division des responsabilités entre le Shell et les appels système

Si avant de débuter ce travail nous étions déjà familiers avec le Shell en tant qu'usagers, nous en ignorions le fonctionnement interne, et tout particulièrement la façon dont les responsabilités sont divisées entre le Shell lui-même et les appels systèmes. Or, pour implémenter correctement notre Shell, il nous a d'abord été nécessaire de bien comprendre cette répartition des tâches : quelles tâches sont prises en charge avant qu'un appel système soit exécuté, et selon quelle logique le Shell accomplit ces tâches.

S'il est évident que les premières étapes du traitement du texte entré par l'usager (ex. : décomposition en commande et arguments) est prise en charge par le Shell, ce n'est pas aussi évident pour la gestion de l'environnement et la résolution de variables. Ainsi, notre premier essai d'ajout de variables à l'environnement appelait d'abord la fonction *setenv()* pour ajouter une variable à l'environnement, puis appelait *exec()* dans le processus enfant ayant hérité de cet environnement en espérant que d'éventuelles occurrences de variables seraient résolues automatiquement par l'appel système. Ce n'est qu'en testant cette implémentation que nous avons découvert que la résolution de variables

dans les arguments prend place dans le Shell lui-même, avant que *exec()* ne soit appelé. Nous avons ainsi dû ajuster notre code pour détecter les variables dans les arguments et appeler *getenv()* pour récupérer leur valeur.

Subséquemment, nous avons ainsi choisi de rejeter l'usage des fonctions *execle()* et *execve()* puisque le passage de l'environnement comme paramètre à *exec()* ne nous apportait aucun avantage apparent, de même que nous avons préféré *execv*()* à *execl*()* puisqu'il était plus facile de concaténer les arguments dans un vecteur. Nous avons également découvert par essai-erreur que la commande *cd* est un cas particulier, devant être implémenté par un appel à *chdir()* plutôt qu'*exec()* et affectant directement le parent, et que son statut de terminaison de processus retourné par *wait(&waitError)* gardait la même valeur malgré l'échec ou la réussite de son exécution. Pour détecter une éventuelle erreur lors de l'exécution de *cd*, notre code vérifie donc plutôt la valeur de la variable *errno*.

3. Imbrication et concaténation des opérations for, && et ||

Ayant abordé le travail linéairement en suivant l'ordre des consignes, nous avons complété le point 4. (&& et ||) avec une implémentation qui semblait fonctionnelle lorsque testée sur des exemples simples, mais dont la logique de la priorité des opérations était incomplète et erronée, menant à des résultats incorrects ou à des commandes entières non-exécutées.

D'abord, notre première implémentation du && et || ne tenait compte que du statut de terminaison du processus précédent pour déterminer si la prochaine commande devait être exécutée, ce qui fonctionnait correctement tant qu'un seul type d'opérateur était utilisé. Par exemple, dans la ligne "cat nofile && echo test || ls" seule la commande "cat nofile" était exécutée (plutôt que "cat nofile" suivi de "ls"). Pour que la logique de notre implémentation soit cohérente avec l'évaluation d'une équation booléenne, nous avons dû ajouter à la logique du && et || la possibilité de sauter une commande pour aller exécuter directement la suivante. C'est le cas lorsque la première partie de l'équation avant un || est vraie, ou lorsque la première partie avant un && est fausse.

Ensuite, des tests plus complexes nous ont permis de découvrir que des opérateur && et || dans un for, ou des for imbriqués n'étaient pas gérés, car le code assigné au traitement du for était séparé de celui s'occupant du && et du || et ne s'exécutait pas récursivement. Pour corriger ce problème, nous avons dû réorganiser complètement la structure de notre code afin de regrouper le traitement du for avec les opérateurs && et ||. Pour se faire, nous avons créé une fonction récursive (*bigBoyParser()*) responsable de décomposer une ligne de texte en commandes et arguments et d'en éliminer les &&, || et for.

Chaque appel à cette fonction résout un élément : soit un &&, un ||, un for, une assignation de variables ou une exécution de commande, et elle s'appelle récursivement au besoin. Cette correction a grandement amélioré la qualité de notre code, le rendant à la fois plus simple et général.

4. Raffinement de la décomposition du texte

Finalement, nous avons dû raffiner nos fonctions de décomposition du texte pour prendre en charge correctement certains des exemples rencontrés dans le fichier de correction rendu disponible avant la remise (correction.sh). D'abord, notre implémentation naïve de résolution de variables ne reconnaissait comme variable qu'un string commençant par $ et délimité avant et après par un espace. Cela ne permettait pas de résoudre des cas tels que $MAN:$VERSION:$LS. Pour corriger cela, nous avons écrit une nouvelle fonction lisant le nom d'une variable n'importe où dans un string à partir du caractère $ et jusqu'à ce qu'un caractère non permis dans un nom de variable soit rencontré, puis rebâti le string en remplaçant le nom de variable par sa valeur. Cette fonction est appelée itérativement jusqu'à ce que toutes les variables aient été résolues. Ensuite, nous avons également dû réorganiser notre code afin que l'assignation de valeur à un flag dans l'argument d'une commande (ex. : ls –color=auto -lh) ne soit pas interprétée comme une assignation de variable, bien qu'un symbole d'égalité soit présent.

Conclusion

En conclusion, ce travail a été une belle opportunité de mettre en pratique la théorie des systèmes POSIX telle que vue en classe en gérant plusieurs processus grâce aux appels systèmes *fork()*, *exec()* et *wait()*, en plus de nous aider à parfaire nos connaissances du langage C. Ce travail a été d'une complexité très raisonnable, et s'est déroulé presque sans embûches, si ce n'est du temps passé au débogage des erreurs de pointeurs et de mémoire qui a ralenti notre progression. Si ce travail était à recommencer, réfléchir davantage à la structure du code avant de commencer l'implémentation pourrait nous sauver le temps passé à restructurer. Finalement, les apprentissages réalisés dans ce projet nous ont menés à jeter un regard différent sur le Shell tel que nous l'utilisions quotidiennement, ayant acquis une compréhension plus approfondie de son fonctionnement.