Rapport du Travail Pratique 1

(TP1)

Par

Charles Langlois et François Poitras

Rapport présenté à

M. Stefan Monnier

Dans le cadre du cours de

Systèmes d'exploitation (IFT2245)

27 janvier 2016

Université de Montréal

Table des matières

1 Fonctionnement du shell

Le programme démarre en affichant le répertoire de travail courant, qui correspond au répertoire où le programme *ch* est situé. À partir de ce point, toutes les commandes disponibles qui sont définies dans la variable d'environnement *PATH* peuvent êtres éxecutées, avec leurs arguments respectifs. Il est aussi possible de rediriger la sortie ou l'entrée d'une commande en utilisant le symbole '>' ou '<', comme dans la plupart des terminaux sous Linux. De la même façon, l'usager peut utliser des *pipes* avec le symbole 'l' et enchainer plusieurs commandes avec des point-virgule pour séparer les instructions.

Il est possible d'utiliser une astérisque pour faire référence à tous les fichiers normaux dans le répertoire courant. Les fichiers normaux sont ceux qui ne commencent par par un point et ceux qui ne sont pas '.' ou '..'. Par exemple, la commande *echo* * affichera le nom de tous les fichiers du répertoire courant. Entre deux entrées, l'utilisateur peut quitter le programme en utilisant la commande *quit* ou encore en utilisant CTRL-D.

1.1 Traitement de l'entrée

Notre shell commence par éliminer tous les espace au début de la commande, s'il y en a. Ensuite, on lit charactère par charactère dans un buffer jusqu'à ce qu'un caractère spécial soit atteint. La fonction *strmem* permet de détecter si une chaîne comporte un charactère spécial. Ces charactères sont le retour à la ligne, les opérateurs de redirection, de pipe, un espace, un point-virgule ou la fin du fichier (*EOF*). Lorsqu'un de ces évènements se produit, on regarde avec un *switch* quel évènement s'est produit et on ajuste les sorties et entrées en conséquence. Par exemple, dans le cas où on a un '>', la sortie est un fichier spécifié et l'entrée est *stdin*. Si la commande n'est pas *quit* et que la syntaxe de la commande semble valide, la fonction *execCommand* est appelée avec les arguments qui correspondent à ce qui a été entré et les *file descriptors* correctement ajustés. À partir de ce point, deux scénarios peuvent se produire. Soit l'utilisateur a entré une commande externe, soit il a entré une commande interne. Une fois le travail effectué, la mémoire allouée par les appels à *strdup* est libérée et le programme attends la prochaine commande.

1.1.1 Commandes externes

L'argument *args* passé à la fonction *execCommand* est un tableau de chaînes de charactères. Autrement dit, un *char***. *args[0]* correspond a la commande entrée. Il faut donc vérifier le contenu de cette case. En particulier, il faut vérifier s'il y a un contenu. Si il y a un contenu, on vérifie s'il s'agit de 'cd'. Si oui, la fonction du même nom est appelée et gère le changement de dossier. Nous sommes maintenants arrivés au cas où l'utilisateur cherche à exécuter un programme externe. Le processus doit donc être dupliqué à l'aide de *fork()*.

Par la suite, une vérification du *PID* doit être faite pour permettre de savoir si le code suivant est exécuté dans le processus enfant ou parent. Si on est dans l'enfant, on vérifie les *file descriptors* passés en argument et on appelle *execvp* pour rouler le programme. L'utilisation particulière de cette commande permet d'éviter de gèrer la variable d'environnement *PATH*, en raison du suffixe 'p'. De plus, la famille

de fonctions *exec* qui comporte un 'v' en suffixe demande de recevoir un *char*** en guise d'arguments aux programmes externes. Comme c'est ce que nous passons en argument à notre fonction, nous n'avons aucun traitement particulier à faire. Dans le cas où l'exécution est dans le processus parent, il s'agit simplement d'attendre que le processus enfant termine et de retourner le status de l'enfant, s'il y a lieu.

1.1.2 Commandes internes

Par manque de temps, les seules commandes internes qui sont gérées par notre shell sont 'cd' et 'quit'. La deuxième ayant été expliquée plus haut, nous allons ici décrire le comportement de la première. 'cd' se comporte comme nous sommes habitués, à l'exception de la gestion du ' '. En effet, il est possible de faire 'cd' et toutes ses variations définies dans la plupart des terminaux Linux, mais les variations comme *cd /machin* ne sont pas gérées. Il est possible d'utiliser *cd -* pour revenir au dernier dossier visité. Évidemment, si il n'y a pas de dossier précédent, la commande va se comporter comme si l'utilisateur voulait accéder à un dossier nommé '-'.

1.2 Gestion des erreurs

Les messages d'erreurs sont des citations inspirées de Margaret Tatcher. Même si cela peut paraître ridicule, les messages d'erreurs sont quand même descriptifs du problème qui c'est produit. On arrive sans problème à comprendre ce qui c'est passé. Par exemple, si on entre une commande qui n'existe pas, le shell nous répondra «There's no such thing as <command>. Only families and individuals.» où <command> sera le nom de la commande entrée. D'autres messages sont générés si l'utilisateur tente d'accèder à un endroit sur le disque où il n'a pas accès, ou encore si la commande entrée est trop grande.

2 Problèmes rencontrés

Puisque nous parlons de programmation en C, l'un des principaux problèmes est bien évidemment la gestion manuelle de la mémoire. Les *segmentation fault* furent nombreux au cours du développement et parfois subtils à règler. En particulier, l'expansion d'arguments a posé problème car il fallait trouver comment remplacer en mémoire l'astérisque par tous les fichiers normaux présents dans le répertoire courant. Le remplacement direct n'était pas une possibilité, car la longueur de l'astérisque est évidemment inférieure à la longueur des noms des fichiers et un remplacement direct cause un *buffer overflow*. Pour règler le problème, nous avons utilisé *realloc*, en tenant compte du nombre de fichiers présents dans le répertoire.

L'autre problème concerne notre connaissance des fonctions POSIX. En effet, comme nous n'étions pas habitués à ces fonctions, nous avons été ralentis par le fait de devoir nous documenter sur leur fonctionnement. Dans les premières versions de notre code, nous ne connaissions pas l'existence de toutes les fonctions de la famille *exec* et nous avions commencé à développer des fonctions pour *parser* le contenu de la variable d'environnement *PATH*. Nous avons compris comment utiliser les fonctions de gestions de *file descriptors* d'une façon similaire, en consultant les pages de manuel des fonctions appropriées. Nous

avons aussi eu recours à StackOverflow pour la compréhension des fonctions de gestion des répertoires. Ces fonctions se sont toutefois avérées très simples à utiliser.