Micro-projet μ-shell

Généralités

Le but de ce TP est de programmer en C un interpréteur de commande simplifié pour Unix. L'interpréteur de commande (souvent appelé *shell* en anglais) est le programme qui est lancé lorsque vous ouvrez un nouveau terminal. Il permet entre-autres d'exécuter les programmes disponibles sur votre système, par exemple [1] :

```
1 $ ls -l -a | less
2 $ cat /etc/hosts > test.txt
3 $ emacs &
4 $ emacs & ls
```

Dans le cas de la première commande, par exemple, le *shell* découpe la commande `ls | less` en trois morceaux :

— la commande `ls` ;
— l'opérateur de tuyau (*pipe*) `|` ;
— la commande `less`.

Une fois reconnu que la commande contient un tuyau et deux noms d'exécutable, les fonctions systèmes comme `pipe()`, `fork()` et `execv()` sont utilisées pour exécuter ce qu'a demandé l'utilisateur.

On vous demande un minimum de travail d'analyse : c'est à vous de choisir comment organiser votre programme pour qu'il soit le plus simple et le plus élégant possible. Néanmoins, il est fortement suggéré de suivre l'ordre d'implémentation suggéré par le sujet.

En cas de difficultés de compréhension du sujet, n'hésitez pas à *observer comment le shell que vous utilisez se comporte* dans un cas similaire.

Remarques importantes

— **Il est indispensable de gérer correctement les cas d'erreur** (commande introuvable, erreur de syntaxe, etc.) ; c'est souvent la partie la plus difficile de chaque question !
— **Faites une sauvegarde complète de votre code au moins à chaque fois que vous avez fini une question.** [2] Nous préférons un programme incomplet mais fonctionnel à un programme ne compilant pas ou cassé.
— Dans une commande shell, le nombre d'espaces ne compte pas.

Contraintes

— **Les fonction `system` et `popen` sont strictement interdites pour tout ce TP** (car elles font appel au shell).

1. Dans la suite de ce document, le symbole « $ » des exemples correspond à l'*invite de commande* (prompt) du shell. Il s'agit d'une simple chaîne de caractères (variant suivant les interpréteurs utilisés) qui marque le début d'une ligne dans laquelle l'utilisateur peut taper des commandes.
2. Ou mieux, utilisez un système de gestion de versions type Git.

— Les fonctions `scanf` et `fscanf` sont interdites également. Utilisez `fgets` et `strtok`. (La fonction `gets` est interdite pour raison de sécurité (voir cours 3).)
— Les fuites mémoires seront pénalisées (vérifiez avec Valgrind).
— Les erreurs détectées par Valgrind seront pénalisées (variable non initialisée, lecture invalide, etc.).
— Faites un programme robuste : toute erreur de segmentation impactera fortement votre note finale.
— Suivez les règles de propreté du code données en début de module et présentes sur le moodle.

Rendu du projet

— Vous devez envoyer via Moodle une archive tar.gz contenant :
  — votre code source ;
  — un fichier texte appelé README expliquant rapidement les questions traitées et non traitées ;
— un fichier Makefile permettant de compiler votre logiciel.

Notation

La partie 1 (fonctions de base) comptera pour environ 10 points, la partie 2 (fonctions avancées) pour 6 points, et la propreté et la qualité du code comptera pour 4 points.

1 Fonctions de base

1.1 Éxecution de programmes sans arguments au premier plan

Commencez par gérer les commandes simples, sans argument :

```
1 $ ls
2 Makefile
3 projet.tex
4 projet.pdf
5 $ who
6 endy    tty7         2018-10-15 18:49 (:0)
```

Votre programme doit quitter lorsque l'utilisateur tape la commande `exit`. De plus, vous devez afficher un message d'erreur avec `perror()` si `execvp()` ne fonctionne pas (par exemple, si la commande n'existe pas). On ne gère pas les variables d'environnement, vous pouvez donc utiliser `execvp()` (et non pas `execvpe()`).

Attention :
— comme dans un shell normal, vous devez attendre la fin de l'exécution d'un programme avant de re-afficher l'invite de commande (*prompt*) « $ » ;
— certaines commandes simples sont des *builtins* (voir la section 1.3), c'est à dire que ce ne sont pas des programmes mais des commandes directes du shell. C'est notamment souvent le cas de `cd`, `which` et `kill` ; il est donc normal que vous ne puissiez pas les exécuter (testez votre shell avec d'autres commandes, par exemple `ls`). Nous ne nous en occuperons pas dans cette question.

Fonctions utiles `fork()`, `execvp()`, `strcmp()`, `fgets()`, `waitpid()`, `malloc()`, `free()`,`perror()`,`fflush()`

1.2 Exécution de programmes avec des arguments

On souhaite maintenant pouvoir donner des arguments aux programmes lancés. Par exemple :

```
$ ls -l -a
total 32K
drwxr-xr-x  4 endy endy 4,0K oct.  15 16:49 ./
drwxr-xr-x 12 endy endy 4,0K oct.  15 11:54 ../
drwxr-xr-x  2 endy endy 4,0K oct.  15 16:25 code/
-rwxr-xr-x  1 endy endy   72 oct.  15 16:42 Makefile*
-rw-r--r--  1 endy endy  11K oct.  15 16:49 tp-minishell.tex
$ who -a
           démarrage système 2018-09-13 10:47
IDENTIFIANT   tty1         2018-09-13 10:48          1016 id=tty1
endy  + tty7         2018-09-13 10:48  ancie   1548 (:0)
      niveau d execution 5 2018-09-13 10:49
      pts/0     2018-10-15 16:39         24178 id=ts/0  term=0 sortie=0
      pts/2     2018-10-09 11:27         26328 id=ts/2  term=0 sortie=0
...
```

Pour implémenter ce comportement, vous pouvez utiliser la fonction `strtok()` qui permet de découper une chaine de caractères en fonction de *délimiteurs*, ici des espaces.

Fonctions utiles `strtok()`

1.3 Builtin cd

Un *builtin* est une commande du shell qui est directement interprétée par celui-ci (contrairement aux commandes qui sont des exécutables). Ce sont généralement des commandes très simples pour lesquelles lancer un nouveau processus (via un `fork()`, etc.) serait inapproprié. Implémentez les commande `cd` et `pwd` afin de pouvoir changer de répertoire et de connaitre le repertoire courant. Exemple :

```
$ pwd
/home/mouret
$ cd /
$ ls
bin/   dev/  home/    lib/    lost+found/  mnt/  opt/
boot/  etc/  initrd/  local/  media/       net/
$ pwd
/
```

Fonctions utiles `chdir()`,`getcwd()`

N'oubliez pas de faire une copie de sauvegarde de votre travail avant de poursuivre !

2 Fonctions avancées

Les fonctions suivantes nécessitent un peu de réflexion de votre part et une analyse syntaxique plus poussée. En améliorant votre analyseur syntaxique, pensez à gerer correctement les espaces : dans la plupart des cas, le nombre d'espace séparant les commandes n'est pas significatif.

Les trois questions suivantes sont largement indépendantes, vous pouvez les traiter dans l'ordre de votre choix.

2.1 Commandes en arrière plan

Si la commande est suivie d'un &, alors elle se lance en arrière plan :

```
1 $ who &
2 $ ls -a&
```

On ne demande pas de gérer les commandes du type `who & ls`.

2.2 Tuyaux

Implémentez maintenant la gestion de tuyaux. Pour des raisons de simplicité, on se restreindra au cas où seulement deux commandes sont séparées par un tuyau (par exemple, on ne gérera pas `ls|grep etu|less`. Exemples :

```
1 $ ls -la | less
2 $ cat /etc/resolv.conf | tee log
```

Fonctions utiles `pipe(),dup2(),close()`

2.3 Redirection de stdout

L'opérateur > permet de rediriger la sortie d'un programme vers un fichier. Implémentez le pour le cas simple : « commande > fichier ». Par exemple :

```
1 $ echo "test" > test.txt
2 $ cat test.txt
3 test
```

Fonctions utiles `dup2(),fopen(),freopen()`

3 Bonus

3.1 Chaîne de commande

Implémentez la possibilité d'enchainer les commandes en les séparant par un point-virgule.

```
1 $ ls -a; echo "ok"; /bin/cat /etc/hosts
```

3.2 Chaîne de tuyaux

Implémentez la possibilité d'enchainer les tuyaux.

```
1 $ ls -a /etc|grep cron|sort
```