

Programmation système

TP 8.a – Un shell élémentaire

Le but de ce TP est d'implémenter un shell qui permet de lancer des commandes en avant-plan ou en arrière-plan, de rediriger les entrées/sorties, et de chaîner des processus avec des tubes (en : *pipes*).

Exercice 1: Mise en route

1. Vous ne partez pas de zéro. On vous fournit du code pour l'analyse syntaxique de la ligne de commande et la gestion des structures de données internes du shell. Commencez par vous familiariser un peu avec ce code :
 - `main.c` contient un squelette de la boucle principale du shell (à modifier).
 - `joblist.ch` permet de gérer la liste des tâches (en : *jobs*) et les structures associées.L'analyse syntaxique est réalisée avec Flex et Bison dans `lexer.l` et `parser.y`. Il n'est pas nécessaire de comprendre ces fichiers en détail car la fonction `newJobFromCmdLine()` (définie dans `joblist.c`) s'occupe déjà d'appeler l'analyseur pour une ligne de commande donnée.
2. Pour l'instant, le shell se contente d'afficher, pour chaque ligne de commande saisie, la structure de données interne qui la représente. Pour vous familiariser avec cette représentation, saisissez quelques commandes simples, avec et sans arguments, puis des commandes plus complexes avec lancement en arrière-plan (&), redirection des entrées/sorties (<, >, >>), et tubes (|).
3. On vous fournit également un script de test. Il ne couvre pas tous les problèmes possibles, mais il vous donnera un petit retour sur votre progrès tout au long du développement. Pensez à le lancer régulièrement, soit avec `make test`, soit en l'appelant directement avec `./test ./mysh`. Vérifiez que :
 - Initialement, tous les tests échouent.
 - Tous les tests passent si vous remplacez `./mysh` par un shell compatible POSIX, i.e., si vous lancez `./test sh`, où `sh` peut être, par exemple, `bash`, `dash`, `ksh`, `yash`, ou `zsh --nomultios`. Si certains tests échouent avec un shell POSIX, alors le script ne fonctionne pas correctement sur votre machine. Dans ce cas, il faut peut-être installer des paquets manquants. (Demandez à votre chargé de TP si vous ne parvenez pas à trouver lesquels.)

Exercice 2: Commandes simples

Commencez par le cas le plus courant : faites en sorte que l'on puisse exécuter des commandes comprenant un seul processus en avant-plan. Par exemple, on doit pouvoir lancer les commandes suivantes :

```
$ ls
...
$ cp /etc/passwd passwd-copy
$ ps -o pid,ppid,cmd --forest
...
```

Veillez à ce que le shell attende la terminaison ou suspension du processus exécuté avant de reprendre la main. Par exemple, si vous lancez `sleep 3`, l'invite de commande ne devrait apparaître qu'après trois secondes. Vous aurez besoin des appels système `fork()`, `exec*()`, et `waitpid()` (ou `wait()`).

Exercice 3: Lancement en arrière-plan (&)

Ajoutez maintenant la possibilité de lancer des commandes en arrière-plan. C'est très facile à faire, puisqu'il suffit que le shell reprenne immédiatement la main, sans attendre la fin du processus enfant. Par exemple, la séquence de commandes

```
$ sh -c "sleep 10; echo Background" &
$ echo Foreground
```

devrait d'abord afficher "Foreground", et seulement après "Background" (à moins qu'il ne vous faille plus de dix secondes pour taper la deuxième commande).

Remarque : Dans l'exemple ci-dessus, la séquence de commandes "sleep 10; echo Background" est interprétée par le sous-shell sh. Pour votre shell, c'est une chaîne de caractères comme une autre.

Exercice 4: Commandes intégrées

Si vous avez correctement fait l'exercice 2, la commande pwd devrait maintenant vous afficher le répertoire courant de votre shell. Essayez de le changer avec la commande cd. Vous verrez que cela ne fonctionne pas. En effet, le programme /usr/bin/cd n'existe pas, et ce pour une bonne raison. Comprenez-vous pourquoi ?

1. Afin de pouvoir utiliser cd, vous devez l'implémenter en tant que commande intégrée (en: *builtin*) au shell. Servez-vous de l'appel système chdir() pour cela. Il peut être utile de définir une fonction de la forme

```
int builtinCD(int argc, char **argv);
```

pour représenter le "main()" de la commande cd. Le même schéma peut être suivi pour toutes les autres commandes intégrées que vous aurez à implémenter.

Petit bonus : Dans un shell POSIX, cd sans arguments va dans le répertoire personnel de l'utilisateur. Si vous voulez ajouter cette fonctionnalité, utilisez getenv() pour récupérer la variable d'environnement HOME (supposée contenir le chemin du répertoire personnel).

2. Rajoutez la commande exit, qui elle aussi doit forcément être implémentée comme une commande intégrée. Elle termine l'exécution du shell avec le code de retour donné en argument. Par exemple, exit 42 doit terminer le shell et renvoyer la valeur 42 à son processus parent. Si aucun argument n'est donné, la valeur renvoyée à la place est :
 - soit le code de retour de la dernière commande exécutée par le shell, si celle-ci s'est terminée de façon normale (i.e., volontairement),
 - soit 128 + SIGxxx, si la dernière commande exécutée par le shell a été terminée par le signal numéro SIGxxx,
 - soit 0, si le shell n'a exécuté aucune commande.

Exercice 5: Redirection des entrées/sorties (<, >, >>)

Maintenant, on veut aussi pouvoir rediriger l'entrée et la sortie standard d'une commande vers des fichiers. La notation <input signifie que la commande à exécuter doit lire dans le fichier input, tandis que >output signifie que la commande doit écrire dans le fichier output, en écrasant son contenu éventuel. Pour ajouter à la fin du fichier au lieu de l'écraser, la syntaxe est >>output. Après avoir implémenté ces fonctionnalités, votre shell devrait, par exemple, permettre l'utilisation suivante :

```
$ printf "Bonjour?" >file
$ printf "Hello, " >file
$ printf "world!\n" >>file
$ tr a-z A-Z <file
HELLO, WORLD!
```

Vous aurez besoin des appels système open(), dup2(), et close().

Exercice 6: Pipelines (|)

Il ne vous reste plus qu'à ajouter les tubes pour obtenir un vrai shell Unix. Le but est de pouvoir exécuter des commandes composées de plusieurs programmes chaînés par des tubes, comme par exemple :

```
$ echo "!olleh" | rev | tr a-z A-Z
HELLO!
```

Pour ce faire, vous devez créer un processus pour chaque programme du pipeline. Dans l'exemple ci-dessus, il faut donc créer trois processus (un pour `echo`, un pour `rev`, et un pour `tr`). De plus, chaque paire de processus consécutifs doit être connectée par un tube, de sorte que la sortie standard du processus "gauche" soit redirigée vers le côté écriture du tube, et l'entrée standard du processus "droit" vers le côté lecture. Dans l'exemple, il faut donc créer deux tubes (un pour `echo` et `rev`, l'autre pour `rev` et `tr`).

Concernant les liens de parenté entre les différents processus, plusieurs solutions sont possibles, mais la plus simple est probablement de faire en sorte que le shell soit le parent de tous les processus du pipeline. Pour cela, essayez de généraliser le code que vous avez déjà écrit, pour passer de la création d'un seul processus enfant à celle d'un nombre arbitraire d'enfants, et intercalez la création d'un tube entre celle de deux enfants. En plus des appels système déjà utilisés, vous aurez besoin de `pipe()`.

Il faudra aussi généraliser d'autres fonctionnalités implémentées dans les exercices précédents :

- Attendre la terminaison d'un pipeline : si vous lancez

```
sleep 3 | true
```

alors l'invite de commande ne devrait apparaître qu'après trois secondes.

- Lancer un pipeline en arrière-plan : la séquence de commandes

```
$ sh -c "sleep 10; echo Background" | cat &
$ echo Foreground
```

devrait d'abord afficher "Foreground", et seulement après "Background".

- Rediriger l'entrée du premier processus d'un pipeline, et la sortie du dernier :

```
$ printf "siort\nxued\nnu\n" >file1
$ tr a-z A-Z <file1 | rev | tac >file2
$ cat file2
UN
DEUX
TROIS
```

De manière plus générale, il est possible d'appliquer des redirections à tout processus au sein d'un pipeline. On pourrait donc aussi rediriger les entrées/sorties de `rev` dans l'exemple ci-dessus, même si cela ne serait pas très utile. Les opérateurs `<`, `>`, `>>` ont alors priorité sur l'opérateur `|`, i.e., une redirection est plus forte qu'un tube.

- Le code de retour d'un pipeline est celui de son dernier processus : si vous exécutez les commandes


```
$ sh -c 'exit 40' | sh -c 'exit 41' | sh -c 'exit 42'
$ exit
```

alors le code de retour de votre shell devrait être 42.

S'il vous reste du temps:

Exercice bonus 1: Historique et autocomplétion

Vous en avez assez de devoir constamment retaper les mêmes commandes ? C'est bien plus agréable quand on peut remonter dans l'historique des commandes avec , et autocompléter les chemins

avec . La bibliothèque GNU Readline (utilisée par Bash) permet de faire cela très facilement. Afin de pouvoir l'utiliser, il faut que les bons paquets soient installés sur votre système, que votre code inclue les en-têtes `readline/readline.h` et `readline/history.h`, et que vous passiez l'option `-lreadline` à votre compilateur. Voir le manuel : <https://tiswww.case.edu/php/chet/readline/readline.html#SEC24>

Reimplémentez la fonction `readCmdLine()` dans `main.c` de manière à ce qu'elle utilise `readline()` au lieu de `getline()`. Il faudra également appeler `add_history()` pour mettre à jour l'historique de Readline, ainsi que réaffecter la variable `rl_outstream` pour que l'invite de commande soit affichée sur l'erreur standard plutôt que sur la sortie standard.

Remarque : Readline ne libère pas toute sa mémoire, et ce fait est évidemment détecté par Valgrind. Si vous utilisez Valgrind (ce qui est toujours une bonne idée), il peut être pratique de pouvoir basculer entre les deux implémentations de `readCmdLine()` à l'aide de directives de compilation conditionnelle (e.g., `#define READLINE ... #ifdef READLINE ... #else ... #endif`).

Exercice bonus 2: Redirection de l'erreur standard (2>, 2>>)

Ajoutez les opérateurs `2>` et `2>>` permettant de rediriger l'erreur standard vers un fichier (en mode écrasement et mode ajout, respectivement). Exemple d'utilisation :

```
$ cp 2>file
$ cat file
cp: missing file operand
Try 'cp --help' for more information.
```

Pour ce faire, il vous faudra adapter l'analyseur syntaxique dans `lexer.l` et `parser.y`, la structure `Proc` et ses fonctions associées dans le module `joblist.[ch]`, et bien entendu le code que vous avez écrit pour effectuer les redirections dans l'exercice 5.

Si vous voulez aller (un peu) plus loin, vous pouvez éviter la duplication de code suggérée ci-dessus en généralisant les opérateurs `<`, `>`, `>>` de sorte à ce qu'ils puissent être optionnellement précédés d'un numéro de descripteur de fichier `n`. Dans ce cas, l'effet de `n<file` est d'ouvrir le fichier `file` en mode lecture sur le descripteur `n`, et, par conséquent, la notation `<file` devient simplement une abréviation de `0<file`. De la même manière, `n>file` et `n>>file` ouvrent `file` sur le descripteur `n` en mode écriture (écrasant et ajoutant, respectivement), et ainsi `>file` et `>>file` deviennent des abréviations de `1>file` et `1>>file`. Cela correspond précisément au comportement spécifié dans la norme POSIX :

https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_07.