Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TP n°4 : Réalisation d'un mini-*shell*

Ce TP consiste à réaliser un mini-*shell* pour Unix.

1 Introduction

Un système d'exploitation fournit à ses utilisateurs une interface de programmation comprenant des fonctions de création de processus, de manipulation des E/S, de travail avec les fichiers, etc. Ces *appels système* peuvent être faits dans un programme C quelconque (en utilisant les fonctions de bibliothèque standard, p.ex. `fork`, `open`, `dup`, ...) ou alors en ligne de commande. Dans le deuxième cas, un interprête de langage de commande qui transforme une commande tapée sous forme textuelle en un ou plusieurs appels système. Dans le système Unix, l'interprète du langage de commande est appelé un *shell*. Des exemples de shell sont `tcsh`, `bash`, `ksh`, etc.

2 Le langage de commande (rappels et compléments)

Une commande est une suite de mots séparés par un ou plusieurs espaces. Le premier mot d'une commande est le nom de la commande à exécuter et les mots suivants sont les arguments de la commande. *Chaque commande doit s'exécuter dans un processus autonome, fils du processus shell. Le shell doit attendre la fin de l'exécution d'une commande.* Les appels systèmes `wait` et `waitpid` permettent de réaliser cette attente et de récupérer la valeur de retour de la commande (`status`).

Une séquence est une suite de commandes séparées par le délimiteur "|" ; la sortie standard d'une commande doit alors être connectée à l'entrée standard de la commande suivante. Une telle connexion s'appelle en Unix un tube. La valeur de retour d'une séquence est la valeur de retour de la dernière commande de la séquence.

L'entrée ou la sortie d'une commande (ou l'entrée de la première commande d'une séquence ou la sortie de la dernière commande d'une séquence) peuvent être redirigées vers des fichiers. On utilise pour cela les notations usuelles d'Unix :

— `< toto` : redirige l'entrée standard vers le fichier `toto`
— `> lulu` : redirige la sortie standard vers le fichier `lulu`

Voici quelques exemple de séquences de commandes, avec ou sans redirection :

```
ls -a
ls -a >toto
ls jacques | grep en
ls < fichier1 | grep en > fichier2
```

3 Travail à réaliser

Le but du mini-projet est de réaliser un interprète pour un langage de commande simplifié. L'objectif est d'une part de comprendre la structure d'un shell et d'autre part d'apprendre à utiliser quelques appels systèmes importants, typiquement ceux qui concernent la gestion des processus, les tubes et la redéfinition des fichiers standards d'entrée et de sortie.

3.1 Analyse des lignes frappées au clavier

La procédure de lecture d'une ligne et son analyse vous sont fournies (fichiers `readcmd.h` et `readcmd.c`). Un programme `shell.c` est également joint pour vous aider à comprendre ce qui est renvoyé par `readcmd()`.

La fonction `readcmd()` renvoie un pointeur vers une structure `struct cmdline` dont les champs sont les suivants :

- `char *err` : message d'erreur à afficher, null sinon
- `char * in` : nom du fichier pour rediriger l'entrée, null si pas de redirection
- `char *out` : nom du fichier pour rediriger la sortie, null sinon
- `char *** seq` : une commande est un tableau de mots (`char **`) dont le dernier terme est un pointeur null ; une séquence est un tableau de commandes (`char ***`) dont le dernier terme est un pointeur null.

La complexité de la structure n'est qu'apparente ; vous vous apercevrez lors de la réalisation du programme qu'elle est très bien adaptée, tout particulièrement pour les appels à `execvp`.

3.2 Organisation du travail

Il vous est demandé de programmé un shell qui peut interpréter les commandes écrites avec le langage défini dans 2. Vous devez passer par/réaliser successivement les étapes suivantes :

1. *Compréhension de la structure cmdline et des résultats de readcmd.*
   Lisez attentivement les fichiers fournis et appropriez vous le code. N'hésitez pas à poser des questions sur la logique ou sur le langage C.
2. *Commande pour terminer le shell*
   Pour cette étape, vous devez modifier le code fournis de telle manière à implémenter une commande `quit` qui termine proprement votre shell. Vous pourrez valider cette première étape avec le fichier de test `test01.txt` (voir section suivante).
3. *Interprétation de commande simple*
   Pour cette étape, vous devez utiliser vos connaissances sur les fonctions de création de processus, notamment la famille de la commande `exec`, que nous avons vu en TP1. Se référer aux points techniques à la fin du sujet et aux compléments fournis.
4. *Commande avec redirection d'entrée ou de sortie*
   Pour cette étape, vous devez utiliser vos connaissances sur les fonctions de redirection d'entrée/sortie, notamment `dup` et `dup2`.

5. *Gestion des erreurs*

   Tout au long de l'implémentation, vous devrez gérer correctement les erreurs en affichant un message adapté sur la sortie d'erreur. Par exemple, si l'utilisateur demande à votre shell d'exécuter une commande innexistante (e.g., `dvfefa`), celui-ci affichera le message d'erreur : `dvfefa: command not found` ou si l'utilisateur redirige la sortie standard sur un fichier dont il ne dispose pas des droits d'écriture, votre shell affichera un message du type `monFichier.txt: Permission denied`.

6. *Séquence de commandes composée de deux commandes reliées par un tube*

   Pour cette étape, vous devez utiliser vos connaissances sur la gestion de tubes, notamment `pipe`.

7. *Séquence de commandes composée de plusieurs commandes et plusieurs tubes*

   Pour cette étape vous réutiliserez les points traités précédemment.

8. *Exécution de commandes en arrière-plan*

   Lorsqu'une commande est terminée par le caractère &, elle s'exécute en tâche de fond, c'est à dire que le shell crée le processus destiné à exécuter la commande, mais n'attend pas sa terminaison.

   Remarque. La détection du caractère & dans une ligne de commande implique une modification de la fonction readcmd() et de la structure cmdline.

9. *Gestion des zombis*

   Une tâche lancée en arrière plan ne doit pas être attendue par le shell. Néanmoins, le shell doit ramasser les processus terminés pour éviter la prolifération de zombis. Pour cela implémenter un traitant de `SIGCHLD`. Utiliser les options suivantes de la primitive waitpid : `waitpid(-1, &status, WNOHANG|WUNTRACED)`

4 Tests

En parallèle de l'implémentation de votre shell, il vous faudra écrire des fichiers de tests pour valider la bonne implémentation de celui-ci. Pour ce faire, nous vous avons mis à disposition un script perl (`sdriver.pl` [1]).

4.1 Fonctionnement du script de test

Le script de test lit un fichier texte contenant les instructions à envoyer à votre shell (commandes et/ou signaux). Le fichier de test devra contenir un header décrivant brièvement l'objet du test puis une série d'instructions permettant de réaliser ce test (une instruction par ligne). Le script vous permet soit d'exécuter des commandes classiques (e.g., `echo titi > toto.txt`) soit d'intéragir avec votre shell avec des instuctions spécifiques :

— Les instructions `TSTP`, `INT`, `QUIT`, et `KILL`, vous permettent d'envoyer respectivement les signaux SIGTSTP, SIGINT, SIGQUIT et SIGKILL à votre shell.

— L'instruction `CLOSE` pour envoyer `EOF` à votre shell (équivalent à un CTRL+D)

1. Script emprunté à R. E. Bryant, D. O'Hallaron. *Computer Systems : a Programmer's Perspective*, Prentice Hall, 2003.

— L'instuction `WAIT` permet d'attendre la fin d'exécution de votre shell
— L'inscription `SLEEP <n>` permet d'espacer deux instuctions de `<n>` secondes.

Pour testers votre shell avec les instructions contenues dans le fichier `test01.txt`, il vous faut exécuter `sdriver.pl` avec les paramètres suivants : `./sdriver.pl -t test01.txt -s ./shell`

4.2 Exemples de fichiers de test

A titre d'exemples, nous vous avons fournis 4 fichiers de test :
— `test01.txt` permet de tester l'implémentation de la commande quit. Il contient la commande `quit` suivie de l'instruction `WAIT` permettant de tester que le shell a bien terminé son exécution
— `test02.txt` permet de tester l'exécution d'une commande sans paramètre. Il fait appel à la commande `ls`.
— `test03.txt` permet de tester l'exécution d'une succession de commandes sans paramètre. Il exécute successivement les commandes `ls`, `echo`, et `ls`.
— `test04.txt` permet de vérifier que le shell ne crée pas de zombies. Il exécute successivement les commandes `ls`, `echo`, `ls`, `echo`, et `ps`.

5 Pour aller plus loin

Inutile d'implémenter les fonctionnalités de cette section si celles de la section 3 ne sont pas entièrement implémentées et fonctionnelles. Il vaut mieux aller moins loin dans le TP et avoir un shell fonctionnel que l'inverse. Nous vous proposons de rajouter les fonctionnalités suivantes (à traiter dans l'ordre) :

1. *Changer l'état du processus en premier plan*

   Implémenter la gestion de `control-c` et `control-z` pour qu'ils envoient respectivement un signal SIGINT et un signal SIGTSTP au(x) processus de premier plan.

2. *Commande integrée `jobs`*

   Les commandes exécutées en arrière-plan s'appellent des jobs. Un job peut être désigné par le PID du processus qui l'exécute (exemple 14567) ou par son numéro de job précédé de % (exemple %3). Les numéros de jobs sont des entiers positifs, attribués à partir de 1.

   Implémenter la commande `jobs` qui donne la liste des commandes lancées.

3. *Agir sur les commandes en arrière plan*

   Implémenter les commandes `fg`, `bg` et `stop` qui agissent sur les jobs d'un shell et respectivement mettent un job en premier plan, en arrière plan ou l'arrêtent.

4. *Ajouter la possibilité d'utiliser le tilde, l'étoile et les variables d'environnements*

   Pour cette étape, vous devrez utiliser les primitives `wordexp` et `wordfree` qui vous permettront de remplacer, entre autres, le tilde, l'étoile et les variables d'environnements avant l'exécution des commandes.

6 Présentation des résultats

La présentation des résultats se fera de la manière suivante. Dans un premier temps, nous vous demanderons de téléverser les fichiers sources sur moodle. Puis, vers la fin du semestre, vous nous présenterez votre travail lors d'une soutenance.

6.1 Rendu des sources

Chaque binôme rendra le code source *commenté* des programmes réalisés et les fichier de test realisés. Nous vous demandons également de rédiger un bref compte-rendu présentant : (a) les principales réalisations et (b) une description des tests effectués.

Les différents fichiers seront rendus sous forme d'archive `tar.gz` ayant pour nom `NOM1-NOM2.tar.gz`. L'archive devra être structurée de la manière suivante :

— un dossier `src` contenant le code source
— un dossier `tests` contenant vos tests unitaires
— un makefile permettant de compiler votre programme
— votre rapport en PDF

6.2 Soutenance

Une démonstration des réalisations sera organisée en fin de semestre. Nous vous demanderons de préparer une liste de commandes pour tester chaque fonctionnalité de votre shell. Les détails liés à cette soutenance vous seront communiqués en temps utile.